PhD Dissertation

International Doctorate School in Information and Communication Technologies

DISI - University of Trento

Greedy Feature Selection in Tree Kernel Spaces

Daniele Pighin

Advisor:

Prof. Marcello Federico

FBK, Human Language Technologies

Co-Advisor:

Prof. Alessandro Moschitti

University of Trento, DISI

April 2010

Abstract

Tree Kernel functions are powerful tools for solving different classes of problems requiring large amounts of structured information. Combined with accurate learning algorithms, such as Support Vector Machines, they allow us to directly encode rich syntactic data in our learning problems without requiring an explicit feature mapping function or deep specific domain knowledge.

However, as other very high dimensional kernel families, they come with two major drawbacks: first, the computational complexity induced by the dual representation makes them unpractical for very large datasets or for situations where very fast classifiers are necessary, e.g. real time systems or web applications; second, their implicit nature somehow limits their scientific appeal, as the implicit models that we learn cannot cast new light on the studied problems.

As a possible solution to these two problems, this Thesis presents an approach to feature selection for tree kernel functions in the context of Support Vector learning, based on a greedy exploration of the fragment space. Features are selected according to a gradient norm preservation criterion, i.e. we select the heaviest features that account for a large percentage of the gradient norm, and are explicitly modeled and represented. The result of the feature extraction process is a data structure that can be used to decode *the input structured data, i.e. to explicitly describe a tree in terms of its more relevant fragments.*

We present theoretical insights that justify the adopted strategy and detail the algorithms and data structures used to explore the feature space and store the most relevant features. Experiments on three different multi-class NLP tasks and data sets, namely question classification, relation extraction and semantic role labeling, confirm the theoretical findings and show that the decoding process can produce very fast and accurate linear classifiers, along with the explicit representation of the most relevant structured features identified for each class.

Keywords: Machine learning, Supervised learning, Kernel methods, Tree kernel functions, Model reverse engineering, Feature selection

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Context

The last decades have seen a massive shift of attention from the so-called *knowledge based* approaches to Natural Language Processing (NLP) in favour of *corpus based* or *statistical* approaches to the analysis of language. In the former, linguists and domain experts would hard-code the rules and knowledge necessary to complete a task, wheras in the latter a system learns how to perform a task by means of rules inferred from text corpora in which the target phoenomena are instantiated. The research in Statistical NLP is indeed devoted to the development and exploitation of Machine Learning (ML) models and techniques for NLP applications.

Among discriminative machine learning algorithms, Support Vector Machines (SVMs) have seen a very widespread application across diverse learning tasks and domains, and they are at the heart of many state-of-the-art models and systems. They are indeed very appealing for four main reasons: 1) solid theoretical foundations, that allow us to estimate a lower bound on the error based on the empirical error, measured on the training set, and the corpus size; 2) robustness to irrelevant features; 3) an optimization problem that can be solved efficiently; and 4) outperforming accuracy and generalization capabilities, thanks to the large margin learning bias.

In combination with SVMs, Kernel functions have been proven very useful to implicitly represent data in high dimensional spaces for NLP systems, e.g. [Kudo and Matsumoto, 2003, Cumby and Roth, 2003, Culotta and Sorensen, 2004, Toutanova et al., 2004, Shen et al., 2003, Kudo et al., 2005].

An especially interesting class of kernel functions for statistical NLP are the so-called Tree Kernels (TK). A TK is a convolution kernel [Haussler, 1999] defined over pairs of trees. Convolution kernels are functions that measure the similarity between structured object pairs in terms of the number of substructures that they share. Each substructure is a feature in the convolution kernel space, and can be univocally associated with a component in a very-high dimensional space. The number and type of features is specific to each kernel function.

By using a TK, for example, it is possible to directly encode rich, structured syntactic data into a learning problem without the need for manual feature design, as the kernel function will automatically evaluate the similarity between two parses as a measure of their overlap.

For all these reasons, the combination of a robust learning algorithm, such as Support Vector Machines, with the flexibility and ease of use of a tree kernel function is an effective and interesting way to explore new tasks and domains, where the knowledge about the relevant features can be inadequate or insufficient, e.g. [Diab et al., 2008], or in those contexts where a massive amount of syntactic information is needed, e.g. [Collins and Duffy, 2002] and [Moschitti et al., 2008].

1.2 The Problem

Generality and implicitness, the key advantages of high dimensional kernels, are also the cause of their main drawbacks.

Generality comes at a computational cost, which does not make high

dimensional kernels very practical to deal with extremely large data sets (due to excessively long training time), or to cope with problems where classification speed is a must, e.g. when fast response times are required or large sets of unlabeled documents have to be classified. Concerning tree kernels even the most efficient algorithms e.g. [Moschitti, 2006b, Zhang et al., 2006], suffer from the burden imposed by the dual formulation of the problem, that makes them much less performant than conventional linear classifiers working in the primal space.

Concerning implicitness, high dimensional kernel spaces allow us to model very complex problems more easily and with smaller injections of domain knowledge, but on the other hand we cannot directly observe the most relevant features, which could provide useful insights towards a deeper understanding of the studied problems. In this respect, it is undeniable that corpus based approaches and especially kernel methods are not as informative as knowledge based methods when trying to explain *why* some model performs better than others. Exploring the feature space of tree kernel functions, that can cope with large amounts of rich syntactic data, would indeed be a very promising way to discover new, relevant structured features.

Complexity and implicitness make the adoption of tree kernels less attractive for a number of possible users, like those who would be interested in performant solutions for real-world tasks and applications, such as industries and IT companies, or those that would prefer approaches that can advance our understanding of linguistic processes, such as linguists, cognitivists or anyone interested in improving available models by means of error analysis and feature inspection.

Complexity-wise, feature selection techniques can offer a solution in many important cases. Still, even though very effective models exist for kernel families defined over $\mathbb{R}^N$, such as polynomial or gaussian kernels (e.g. [Cao et al., 2007], [Aksu et al., 2008] and [Guyon et al., 2002]), the

approaches that focus on, or can cope with, the rich space generated by a convolution kernel are few and isolated [Kudo and Matsumoto, 2003, Suzuki and Isozaki, 2005]. As for the implicitness of the result, to our knowledge there are no previous works that directly try to address this problem for high dimensional kernels.

1.3 Proposed solution

This thesis describes a methodology to employ feature selection in a very high dimensional kernel space as a possible solution to both problems. In particular, it will focus on the kernel space generated by TK functions in the context of a Support Vector Machine (SVM) learning framework.

The SVM optimizer is an effective device to select the most relevant examples (the support vectors) and to obtain a feature selection side-effect. Indeed, the weights expressed by the gradient of the SVM's separating hyperplane implicitly establish a ranking between features in the kernel space. This property has been exploited in feature selection models based on approximations or transformations of the gradient, e.g. [Rakotomamonjy, 2003], [Weston et al., 2003], [Guyon et al., 2002] or [Kudo and Matsumoto, 2003].

Tree kernels generate a huge feature space, in which each distinct tree sub-structure is mapped onto a different dimension. In this situation it is impossible to enumerate and rank all the features in the space. The only possibility is to start generating features in order of relevance, starting from the most relevant, and define a criterion to terminate the exploration.

We mine the TK feature space encoded by an SVM model and discard all the features that do not contribute relevantly to gradient norm of the separating hyperplane. As a result, we are able to select just a very small number of features (i.e. a few hundreds or thousands instead of billions) and

still retain a large fraction of the gradient norm. As a consequence, we also preserve a large fraction of the margin of the original model, and therefore its accuracy. The relevant fragments are explicitly represented and stored in a convenient data structure that we can then use to *decode* the data of the original problem. We call decoding the process by which the input trees are projected onto an explicit, lower-dimensional space where each component accounts for the presence of a relevant feature. The decoded data can then be used to carry out fast learning and classification in the projected space.

The data structure that we use to store the relevant fragment can actually be considered as a graphical representation of a set of explicit algorithms to extract structured features from the input data. In this respect, the expressivity of the rules that we can induce is only limited by the expressivity of the target kernel space, and the kind of rules that can be produced is a combination of the structured input data and the characteristics of the kernel. This kind of representation allows us to actually unleash all the potential for automatic feature discovery of tree kernels, generating and weighing relevant features in the huge fragment space. We select the most relevant structured features and encode them as linear attributes in a traditional attribute-value representation, thus combining the advantages of both representations.

The suggested line of research poses modeling and computational challenges, collocating itself in the largely unexplored research field of feature selection for convolution kernels and in an area of interest between:

- machine learning, since the feature selection technique moves from statistical learning theory and offers interesting solutions that may be employed in fields other than natural language processing, or for other classes of structural kernels;

- data mining, since the algorithms and the data structure that we employ are heavily influenced by previous work in this field, e.g. [Zaki, 2002, Pei et al., 2001];
- computational linguistics, since by proposing an approach that can make (part of) the tree kernel space explicit we hope to offer the community a valuable technique for discovering and engineering new structured features for a wide class of problems.

A note on related publications.

Parts of this work have already been peer-reviwed by the scientific community.

In [Pighin and Moschitti, 2009a], we presented an earlier version of our feature selection framework based on the SIMLE_MINER(·) algorithm (discussed in Sec. 4.5.2), and applied it to a semantic role labeling benchmark. In that context, we also considered the very demanding task of boundary classification for semantic role labeling, including 1,000,000 training instances. We showed that the LOpt (Sec. 4.1.2) and Split (Sec. 4.1.3) architectures can result in very accurate and fast learning and classification cycles.

In [Pighin and Moschitti, 2009b], we mostly focused on the explicit representation of the tree kernel feature space, by tackling the question classification task with the LOpt architecture (4.1.2) and the BOUNDED_MINER(·) algorithm (Sec. 4.5.3). We demonstrated that feature selection in the TK space is a very effective way to automatically engineer relevant structured features.

The theoretical framework, outlined in Section 4.3, and the latest version of the mining algorithm, GREEDY_MINER(·) (Sec. 4.5.4), are currently under review.

1.4 Innovative Aspects

The thesis presents the following main points of novelty:

A theoretical framework for feature selection in very high dimensional feature spaces. We link the gradient norm to the margin of a classifier in the kernel space, showing that small changes in the gradient norm have a limited effect on the

margin and therefore on the error rate of the classifier (Lemma 4.3.1). We show how the peculiarities of the TK space make it possible to discard an exponentially large number of features while preserving most of the gradient norm (Lemma 4.3.4). The combination of these two findings establishes the basis of our feature selection technique. To our best knowledge, this is the first attempt to feature selection in TK spaces that clearly establishes a link between the empirical model and the theory, thus providing a solid starting point for the exploration of the feature space of other structural kernel families.

Insights about the inner working of TK functions. Due to TK functions implicit formulation, the nature of the feature space they generate and the behaviour of individual features in these spaces is by and large obscure. We clearly break down the process by which TK functions generate their rich feature space, and provide insights about the kind of information that different kernel functions can represent (Sec. 4.4).

A greedy strategy to mine the TK feature space. We describe an algorithm for the exploration of the TK space (Alg. 4.4) that can efficiently select the most relevant features in the high dimensional tree kernel space. Supported by our theoretical claims, the algorithm can implement a very aggressive selection strategy. The gradient norm in the TK space is employed to guide the selection process and to estimate the relevance of individual fragments. The space is explored in a small-to-large fashion, as according to the kernel definition smaller fragments have more chances of being highly relevant (Sec. 4.5).

Efficient data structure and algorithms for fragment indexing. We introduce a data structure that can conveniently store several hundreds of thousands of frag-

ments, and design algorithms for fragment indexing and matching that have linear complexity with respect to the number of nodes of the input trees (Sec. 4.6).

An explicit representation of the fragment space. Our data structures store explicit representations of the most relevant fragments. This allows us to exploit the feature-discovery capabilities of tree kernel functions in very fast linear classifiers, by projecting the input data onto a lower dimensional space where only the most relevant fragments are accounted for. The fragments that we isolate can be a valuable tool in the hands of linguists and domain experts to gain insights on the problems at study.

Three architectures for exploiting feature selection in TK spaces. We present three different architectures that stress different aspects of the feature selection methodology (Sec. 4.1): the link with the theoretical framework (MLin, Sec. 4.1.1), classification accuracy (LOpt, Sec. 4.1.2) and training time efficiency (Split, Sec.4.1.3).

A general framework for feature selection in high dimensional spaces. Even though we focus on a specific class of kernel functions, the framework that we introduce is general enough to be easily extended to include other families of kernel functions.

1.5 Structure of the Thesis

The rest of the document is structured as follows.

Chapter 2 introduces notations and concepts that will be used throughout the discussion, namely support vector machines, kernel functions, tree kernel function and feature selection techniques. Chapter 3 presents the most

relevant results in the previous work concerning the use of tree kernels for NLP and feature selection techniques for kernel learning. Chapter 4 details the solution that we advocate, in terms of theoretical insights, algorithms and data structures. Chapter 5 presents the setup and the results of an extensive empirical evaluation on three very different benchmarks: question classification, relation extraction and semantic role labeling. Finally, Chapter 6 draws the conclusions and hints possible directions for future work.

Chapter 2

Preliminary Concepts

In this chapter, we introduce terminology and concepts that will be used throughout the rest of the discussion. The chapter is structured as follows: Section 2.1 explains the problem of classification and introduces linear classifiers; Section 2.1.1 describes maximum margin classifiers and support vector machines; Section 2.1.3 shows how the *kernel trick* can allow a linear classifier to cope with non linearly separable problems; Section 2.2 explains tree kernel functions in more detail, and presents a selection of relevant TK families; finally, Section 2.3 outlines the basic concepts behind feature selection.

2.1 Linear Classifiers

The problem of classification consists of learning how to partition elements of some set $\mathcal{O}$ into a finite number of classes C. As an example, we may want to assign the most appropriate topical label to some news (e.g. politics, economics, sports or technology), or, given a collection of X-rays lung scans, we may be interested in recognizing the cases showing evidence of tumoral forms. If the problem only involves two classes, i.e. $|C| = 2$, the classifier is called a binary classifier. If $|C| > 2$, then it is referred to as a multi-class classifier.

Classification is very conveniently handled as a supervised learning problem, where a learning algorithm learns an approximation g of the function $f : \mathcal{O} \to C$ that assigns the proper class to any object $o \in \mathcal{O}$, based on the observations provided by a training set $T \subset \mathcal{O} \times C$, in which training points $o_i \in \mathcal{O}$ are paired with their correct label $f(o_i) \in C$. Learning is a generalization process, since the learner must be capable of abstracting from individual traits of the training data in order to be able to cope with examples never seen before, i.e. the test data $E \subset \mathcal{O}$. Here, an important assumption is that the examples that make up the training and test data are independent and identically distributed (iid), meaning that they are sampled from a fixed, yet possibly unkown, distribution independently from each other.

For complex objects, a mapping function $\phi : \mathcal{O} \to \mathbb{R}^N$ can provide a so-called *feature based representation* of the objects $o_i \in \mathcal{O}$ as vectors $\boldsymbol{x}_i \in \mathbb{R}^N$, where the scalar product can be used as a measure of pairwise similarity. In this case, learning a classifier requires to estimate a function $g : \mathbb{R}^N \to \mathbb{R}$ that can separate the examples belonging to different classes. The mapping function $\phi(\cdot)$ summarizes the process of feature design, a relevant aspect of classifier design that requires efforts, expertise and domain knowledge in order to find a convenient representation for the (potentially complex and structured) objects of $\mathcal{O}$ in $\mathbb{R}^N$.

Given a set of training points T, it is generally possible to find more than one function that can separate them. Figure 2.1 gives a graphical example by showing a few of the infinite functions that could separate the two classes of points in a simple 2-dimensional classification problem. Since when learning a classifier we do not have any information about the test data, we cannot decide which of these functions is preferable. All we can do is minimizing the so called empirical risk, i.e. the error on the training set:

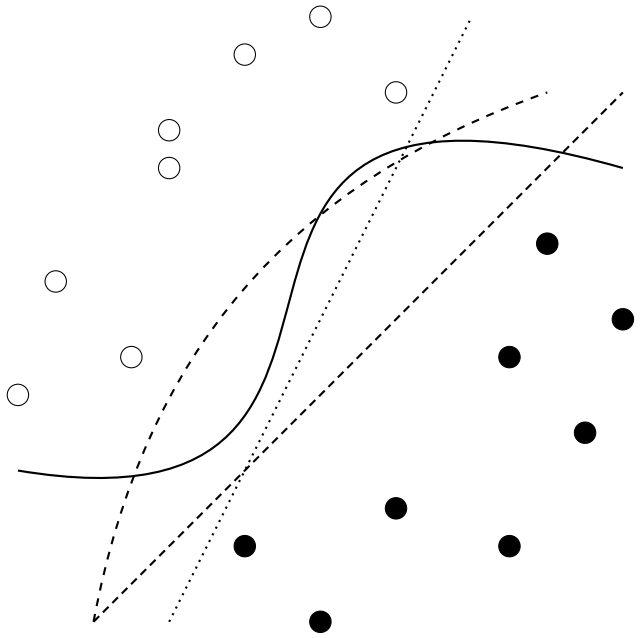


Figure 2.1: **Separating boundaries for a binary classification problem.**

$$R_{emp}(g) = \frac{1}{|T|} \sum_{i=1}^{|T|} \frac{1}{2} |f(x_i) - g(x_i)| \ ,$$

but it does not give us information about the risk, i.e. the error on the test data, and therefore it is not a useful tool for comparing different choices of g.

Intuitively, we can imagine that very complex functions would be better at outlining the boundaries of class distributions. On the other hand, such carefully tailored boundaries would increase the risk of over-fitting the training data, i.e. of learning a classifier which performs very well on the training data but not as well on a training sample having a distribution even slightly different.

Statistical learning theory [Vapnik, 1998] (SLT) uses the the Vapnik-Chervonenkis (VC) dimension of a class of functions $\mathcal{F}$ as a measure of the trade-off between its capacity to separate a set of data points and its generalization capability. The VC dimension of $\mathcal{F}$ is defined as the

maximum number of points that can be shattered by $\mathcal{F}$. $\mathcal{F}$ is said to shatter a set of points P iff, $\forall P_1, P_2 \subset P \mid P_1 \oplus P_2 = P$, [1] $\exists f \in \mathcal{F}$ so that P_1 and P_2 are separated by f, i.e. if there is at least one function in the family that can be used to define a boundary between any partition of the points.

A collection of related results shows that it is very important to consider classes of functions that have just enough capacity to separate the training points [Schölkopf and Smola, 2001]. Interestingly enough, knowing the VC dimension of a decision function also allows us to estimate an upper bound on the risk of the classifier, i.e. the error on any possible selection of test points, as explained by the following theorem [Vapnik, 1998]:

Theorem 2.1.1. *Let h be the VC dimension of a class of functions $\mathcal{F}$. Then, with probability $1 - \delta$, the risk $R(f)$ of a classifier $f \in \mathcal{F}$ on a test set of ℓ examples is bounded by:*

$$R(g) = R_{emp}(g) + \sqrt{\frac{h(\log \frac{2\ell}{h} + 1) - log\frac{\delta}{4}}{\ell}} \,. \tag{2.1}$$

The structural risk minimization (SRM) principle is a straightforward consequence of these results: when designing a classifier, the decision function should be selected so as to

- minimize the empirical risk, and
- belong to a family with the lowest possible VC dimension.

If we can satisfy these two properties, we minimize Equation 2.1 and identify the function(s) with the lowest bound on the test error.

Linear functions, which have a low VC dimension, are hence interesting candidates for the definition of the boundary if the training data are linearly separable.

[1] $X = \{P_1, P_2\}$ is a partition of P.

Algorithm 2.1 LEARN_PERCEPTRON($T, D, alpha$)

main
$b \leftarrow 0, \boldsymbol{w} \leftarrow \mathbf{0}^N$
for $d \in \{1, \ldots, D\}$
do
  for each $\langle y_i, \boldsymbol{x}_i \rangle \in T$
  do
    $\Delta = \alpha(y_i - \text{sgn}(\boldsymbol{w} \cdot \boldsymbol{x}_i + b))$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \Delta \boldsymbol{x}_i$
    $b \leftarrow b + \Delta$

The decision function of a linear classifier is a hyperplane in $\mathbb{R}^N$, i.e.:

$$g(\boldsymbol{x}) = \text{sgn}(\boldsymbol{w} \cdot \boldsymbol{x} + b) \ , \tag{2.2}$$

where $\boldsymbol{x}$ is a point to classify and $\boldsymbol{w}$ and b are called the gradient and the bias of the hyperplane, respectively. A set of points $x_1, \ldots, x_\ell$ are linearly separable if $\exists \gamma \in \mathbb{R}^+, \boldsymbol{w} \in \mathbb{R}^N, b \in \mathbb{R}$ so that $\forall i = 1, \ldots, \ell$, it holds that $y_i (\boldsymbol{w} \cdot \boldsymbol{x}_i + b) \geq \gamma$. Here, $y_i \in \mathbb{R}$ is the label associated to the point $\boldsymbol{x}_i$, that marks it as belonging (or not) to the target class. The minimum distance between two points in different classes along the direction of $\boldsymbol{w}$, γ, is called the margin of the classifier.

Different learners use different algorithms to estimate the weight vector $\boldsymbol{w}$ and the bias b, generally resulting in different boundaries for the same data and, as a consequence, in different margins. A very simple algorithm for learning a linear classifier is the perceptron [Rosenblatt, 1958]. The learning process consists of one or more iterations over the training points. Whenever the perceptron misclassifies an example, the components of the gradient are updated accordingly. The learning algorithm is shown in Algorithm 2.1, where:

$T = \bigcup_{i=1}^{\ell} \langle y_i, \boldsymbol{x}_i \rangle$ is the training set, $y_i \in \{0, 1\}$,

$0 < \alpha \leq 1$ is the learning rate of the perceptron, and

D is the number of intended iterations.

2.1.1 Maximum Margin and Support Vector Machines

Among the class of linear functions, an especially interesting family is that of maximum margin hyperplanes, i.e. the hyperplanes that are maximally distant from the examples of the two training classes. As we have seen before, this property is expressed by the margin γ of the hyperplane. Indeed, statistical learning theory shows that maximum margin hyperplanes have a lower VC dimension than other hyperplanes. As a consequence, they show better generalization performance than any other linear functions.

As shown by the following theorem, the margin of the hyperplane is in fact inversely proportional to the bound on the risk [Bartlett and Shawe-Taylor, 1998]:

Theorem 2.1.2. *Let*

$$\mathcal{C} = \{\boldsymbol{x} \to \boldsymbol{w} \cdot \boldsymbol{x} : \|\boldsymbol{w}\| \leq 1, \|\boldsymbol{x}\| \leq R\}$$

be the class of real-valued functions defined in a ball of radius R in $\mathbb{R}^N$. Then there is a constant k such that for any classifier $h = \mathrm{sgn}(c) \in \mathrm{sgn}(\mathcal{C})$, for any sample of ℓ randomly selected examples, if all the ℓ examples are separated with margin γ, i.e. $|\boldsymbol{w} \cdot \boldsymbol{x}| \geq \gamma$, then with probability $1 - \delta$ the error over the sample is bounded by

$$\frac{k}{\ell}\left(\frac{R^2}{\gamma^2}\log^2\ell + \log\frac{1}{\delta}\right) .$$

Furthermore, if b examples are separated with margin less than γ, then with probability $1 - \delta$ the error on the ℓ examples is less than

$$\frac{b}{\ell} + \sqrt{\frac{k}{\ell}\left(\frac{R^2}{\gamma^2}\log^2\ell + \log\frac{1}{\delta}\right)} .$$

A Support Vector Machine [Boser et al., 1992] (SVM) is a learning ma-

chine that implements the structural risk minimization principle by forcing margin maximization when learning a linear solution to a classification task [2].

Given a set of training examples $T = \{\langle \boldsymbol{x}_1, y_1 \rangle, \ldots, \langle \boldsymbol{x}_\ell, y_\ell \rangle\}$, with $y_i \in \{+1, -1\}$, the optimization problem solved by the SVM optimizer is

$$\begin{aligned} \text{Maximize: } & \frac{1}{2}\|\boldsymbol{w}\|^2 \\ \text{Subject to: } & y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i - b) \geq 1 \ \forall i = 1, \ldots, \ell \ , \end{aligned} \tag{2.3}$$

where the space is implicitly normalized so that the closest points are at distance 1 from the hyperplane. This is generally referred to as the primal optimization problem.

By introducting Lagrange multipliers $\alpha_i \geq 0$, the previous conditions can be rewritten as the Lagrangian:

$$L(\boldsymbol{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{i=1}^{\ell} \alpha_i \left(y_i \left(\boldsymbol{w} \cdot \boldsymbol{x}_i - b\right) - 1\right) \tag{2.4}$$

where $\boldsymbol{w}$ and b are the primal variables, while $\boldsymbol{\alpha}$ is the dual variable. Solving the problem requires to find a saddle point of the Lagrangian, by minimizing L for the primal variables and maximizing it for the dual variables. By deriving for $\boldsymbol{w}$ and b we obtain that

[2] SVMs can also be used for regression, but this aspect falls outside the scope of this thesis.

$$\boldsymbol{w}^* = \sum_{i=1}^{\ell} \alpha_i y_i \boldsymbol{x}_i \tag{2.5}$$

$$\sum_{i=1}^{\ell} \alpha_i y_i = 0 \ . \tag{2.6}$$

If we substitute 2.5 and 2.6 in 2.4 we can derive the dual form of the optimization problem, where the only variable is the dual variable $\boldsymbol{\alpha}$:

$$\begin{aligned} \text{Maximize: } & W(\boldsymbol{\alpha}) = \sum_{i=1}^{\ell} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \boldsymbol{x}_i \cdot \boldsymbol{x}_j \\ \text{Subject to: } & \sum_{i=1}^{\ell} \alpha_i y_i = 0 \ , \ \alpha_i \geq 0 \ . \end{aligned} \tag{2.7}$$

Practically, only a few α_i will be greater than zero. The corresponding $\boldsymbol{x}_i$ are called the support vectors of the decision function, and lie exactly on the margin, i.e. they satisfy $y_i (\boldsymbol{w} \cdot \boldsymbol{x}_i + b) = 1$.

Figure 2.2 shows a simple two dimensional classification problem and the maximum margin hyperplane that separates the two classes. The gradient $\boldsymbol{w}$ is normal to the separating hyperplane, and the margin measures $\gamma = \frac{2}{\|\boldsymbol{w}\|}$. The bias b is the distance, along the direction of w, of the hyperplane from the origin.

2.1.2 Soft-margin SVMs

The constraints of the SVM optimization problem require that all the points are correctly separated by the hyperplane. This very strong condition, called hard margin, would make the SVM not applicable to a wide range of problems where some examples are mislabeled, i.e. they lie on the wrong

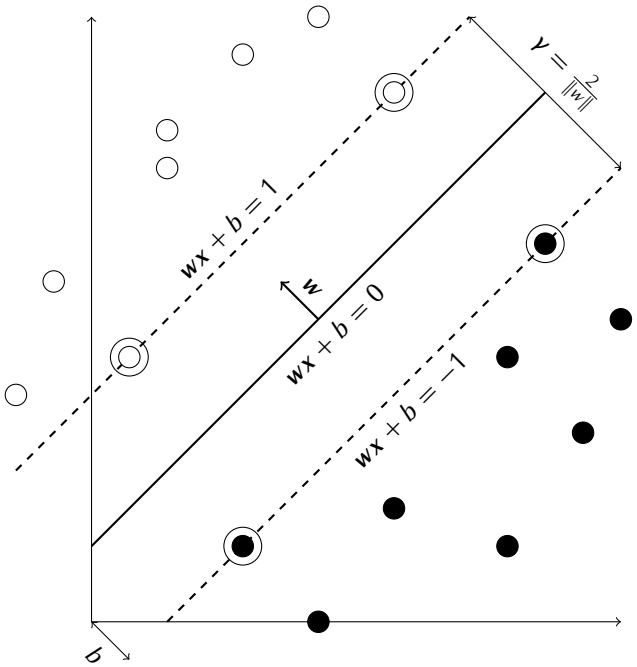


Figure 2.2: **Maximum margin classification.**

side of the hypothetical boundary.

Soft margin SVMs [Cortes and Vapnik, 1995] extend the range of applicability of SVMs by learning a hyperplane that allows for some examples within the margin, while still trying to maximize inter-class distance. This result is obtained by including in the optimization problem slack variables ξ_i that allow a training point $\boldsymbol{x}_i$ to fall also within the margin, i.e.:

$$\begin{aligned} \text{Maximize: } & \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{\ell}\boldsymbol{\xi}_i \\ \text{Subject to: } & y_i(\boldsymbol{w}\cdot\boldsymbol{x}_i - b) \geq 1 - \xi_i \\ & 0 \leq \alpha_i \leq C\ , \end{aligned}$$

where the costant $C > 0$ accounts for the trade-off between classification errors, i.e. examples within the margin, and margin maximization.

It should also be noted that the optimization problems of the hard and

soft margin SVM fall under the category of quadratic problems (QP), for which very efficient solvers exist [Nocedal and Wright, 2000].

2.1.3 Kernel Machines

By combining together i) the optimization of QP, ii) the low VC dimension of the large margin classifier, and iii) the ability to cope with mislabeled data, thanks to the soft margin formulation, an SVM is an efficient, accurate and robust solution which is very attractive for learning linear classification problems. By applying the so-called kernel trick [Aizerman et al., 1964], as explained below, these interesting features can be exploited also to tackle classification problems that require a more complex boundary to be separated.

If we substitute (2.5) in (2.2), we obtain the decision function of the SVM for a test point $\boldsymbol{x}$, i.e.:

$$\begin{aligned} g(\boldsymbol{x}) &= \operatorname{sgn}\left(\boldsymbol{w} \cdot \boldsymbol{x} + b\right) \\ &= \operatorname{sgn}\left(\left(\sum_{i=1}^{\ell} \alpha_i y_i \boldsymbol{x}_i\right) \boldsymbol{x} + b\right) \\ &= \operatorname{sgn}\left(\left(\sum_{i=1}^{\ell} \alpha_i y_i \boldsymbol{x}_i \cdot \boldsymbol{x}\right) + b\right) , \end{aligned} \tag{2.8}$$

and we can observe that the result only depends on the dot product between pairs of points rather than on the individual points. Similarly, also the optimization problem in 2.7 depends on the dot product.

Since both training and classification do not depend on individual test points, the inner product in all the equations can be replaced with a function $k : \mathbb{R}^n \to \mathbb{R}$, so that $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i \cdot \boldsymbol{x}_j$. More generally, for any set of input

objects $\mathcal{O}$ we can define a function $k : \mathcal{O} \times \mathcal{O} \to \mathcal{H}$ so that:

$$k(o_i, o_j) = \phi(o_i) \cdot \phi(o_j) = \boldsymbol{x}_i \cdot \boldsymbol{x}_j \quad , \quad \boldsymbol{x}_i, \boldsymbol{x}_j \in \mathcal{H}^N \ , \tag{2.9}$$

i.e. a function that evaluates the inner product in some high-dimensional space $\mathcal{H}^N$ by representing the input objects $o_i, o_j \in \mathcal{O}$ via a mapping function $\phi : \mathcal{O} \to \mathcal{H}^N$, where $\mathcal{H} = \mathbb{R}$ or $\mathcal{H} = \mathbb{C}$.

As an example, let us consider the polynomial kernel of degree d, which is defined as

$$K(\boldsymbol{a}, \boldsymbol{b}) = (\boldsymbol{a} \cdot \boldsymbol{b} + 1)^d \ . \tag{2.10}$$

If $d = 2$ and $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^2$, then we can write (2.10) explicitly as:

$$\begin{aligned} K(\boldsymbol{a}, \boldsymbol{b}) =& (a_1 b_1 + a_2 b_2 + 1)^2 \\ =& a_1^2 b_1^2 + a_2^2 b_2^2 + 1 + 2a_1 b_1 + 2a_2 b_2 + 2a_1 b_1 a_2 b_2 \\ =& [a_1^2, a_2^2, 1, \sqrt{2}a_1 a_2, \sqrt{2}a_1, \sqrt{2}a_2] \cdot [b_1^2, b_2^2, 1, \sqrt{2}b_1 b_2, \sqrt{2}b_1, \sqrt{2}b_2] \\ =& \phi(\boldsymbol{a}) \cdot \phi(\boldsymbol{b}) \ , \end{aligned} \tag{2.11}$$

and observe that the $\phi(\cdot)$ maps a vector onto a space where also all the conjunctions of features having length up to d are represented, i.e. $a_1 a_2$ and $b_1 b_2$.

Using the kernel trick, i.e. replacing dot products with a kernel function, we can rewrite the decision function of the SVM as:

$$c(o) = \operatorname{sgn}\left(\sum_{i=1}^{\ell} \alpha_i y_i k(o_i, o) + b \right) \ . \tag{2.12}$$

If the mapping is appropriate, we can expect our objects to be mapped onto a space with enough dimensions to make the problem linearly separable. As an example, consider the set of points in Figure 2.3, which are not linearly separable in the original space (left). By applying the mapping ϕ induced

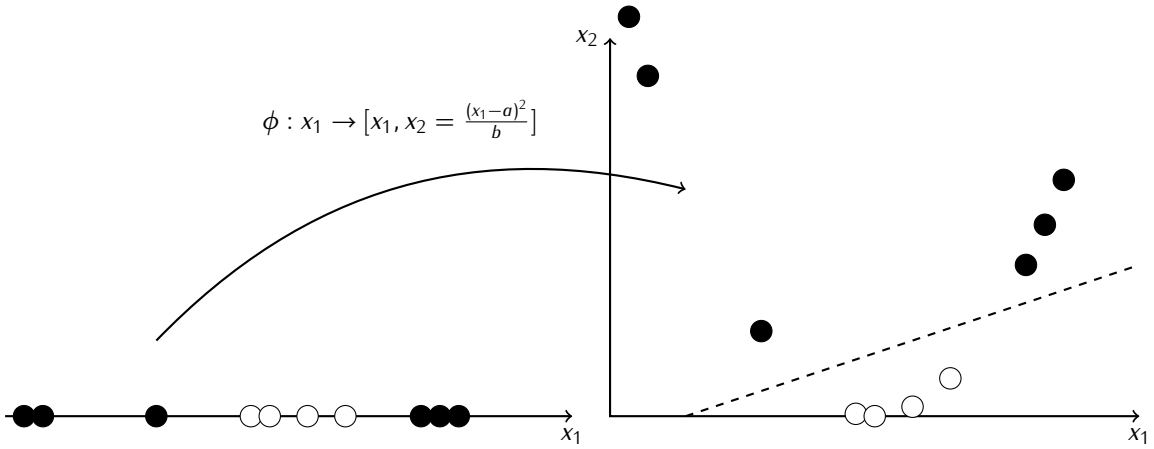


Figure 2.3: **Kernel functions and linear separability** - The points are not linearly separable in the original 1-dimensional space (left), but they are separable in the higher dimensional space induced by the mapping $\phi(\cdot)$ (right).

by a kernel function, we can represent them in a higher dimensional space where a linear separation is possible.

The condition to apply the kernel trick is that k must be equivalent to a dot product in some high dimensional space. According to Mercer's theorem [Mercer, 1909], for real valued functions the equivalence holds if k is continuous, symmetric and positive semidefinite, but other theorems can be used to demonstrate that a function is a valid kernel also in different cases [Schölkopf and Smola, 2001].

As a side effect of these conditions, if $c > 0$ and $k_1, k_2 : \mathcal{O} \times \mathcal{O} \rightarrow \mathcal{H}$ are valid kernel functions, then in all the following cases k is a valid kernel too:

$$
\begin{aligned}
k(o_i, o_j) &= c k_1(o_i, o_j) \\
k(o_i, o_j) &= c + k_1(o_i, o_j) \\
k(o_i, o_j) &= k_1(o_i, o_j) + k_2(o_i, o_j) \\
k(o_i, o_j) &= k_1(o_i, o_j) \cdot k_2(o_i, o_j)
\end{aligned}
$$

Interestingly, the definition of a kernel function k does not require the

corresponding mapping $\phi_k(\cdot)$ to be explicit, as it suffices to demonstrate that such mapping exists by satisfying Mercer's theorem or equivalent conditions. This allows us to evaluate pairwise similarity in very high dimensional spaces using very compact and implicit definitions.

It should be noted that the kernel trick is not a peculiarity of support vector learning, as it can be applied to any learning algorithm for which both the training and the decision function can be expressed in terms of dot products. Learning algorithms that can be reformulated to exploit the kernel trick are generally referred to as kernel machines. For example, also the perceptron algorithm can be rewritten in terms of dot products, which can then be replaced by a kernel function [Freund and Schapire, 1999].

2.2 Tree Kernel Functions

For the scope of this thesis, we focus on a specific class of kernel functions that can directly estimate pairwise similarity between trees, the so-called Tree Kernel (TK) functions. Before describing TKs in more detail, it is convenient to introduce notation and terminology that will be used throughout the rest of the discussion.

Formally, a tree is a simple, connected and undirected graph. As such, a tree t is defined by the a pair $\langle N_t, E_t \rangle$, where N_t is the set of vertices, or nodes, of t, and E_t the set of edges. A tree is rooted if one node has been designated as the root, in which case the edges have a natural orientation, towards or away from the root. In a rooted tree, the parent of a node is the node connected to it on the path to the root; every node except the root has a unique parent. A child of a node v is a node of which v is parent. A leaf (or terminal node) is a node without children. Conversely, a node with at least a child is called internal. A node whose all children are leaves is called preterminal. An ordered tree is a rooted tree for which an ordering

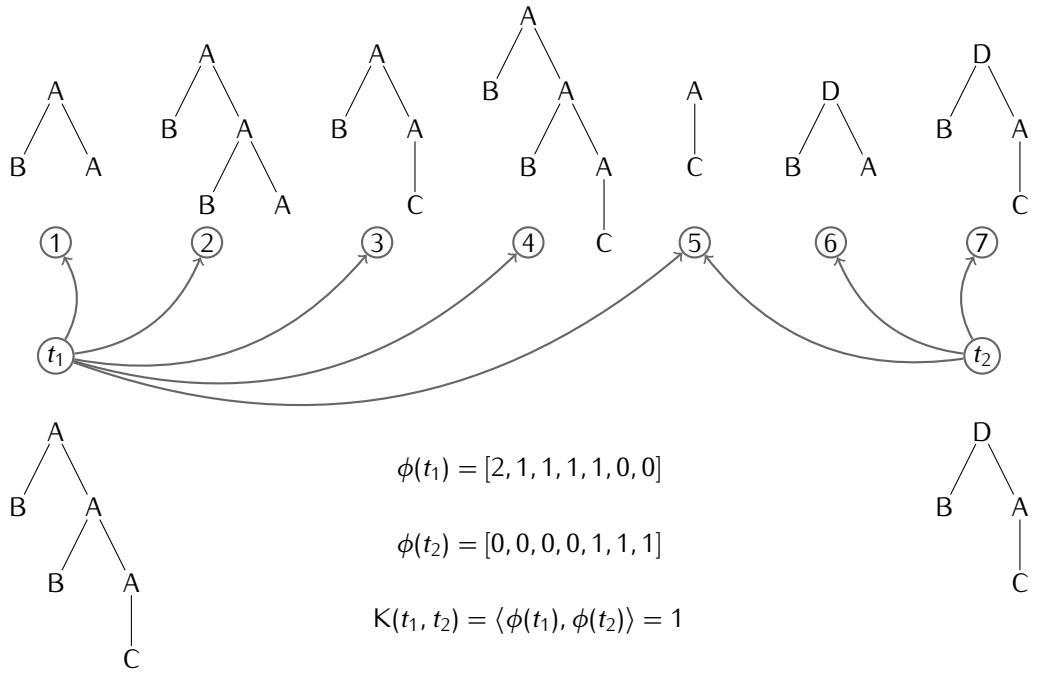


Figure 2.4: **Fragment space** - The fragment space generated by two trees, and the resulting kernel product as evaluated by a tree kernel function.

is specified for the children of each node. In the rest of the discussion, the word *tree* will always be used to refer to a rooted and ordered tree.

A tree kernel is a convolution kernel [Haussler, 1999] defined over tree pairs, i.e. a kernel that evaluates the similarity between two trees by estimating the degree of their overlap. The overlap is estimated by counting the number of substructures, or fragments shared between the two trees. It does so by establishing an implicit mapping $\phi(\cdot)$ that associates different fragments with different dimensions in a high-dimensional space.

Basically, each tree t is mapped onto a vector $\boldsymbol{x} = [x^{(1)}, \ldots, x^{(N)}]$, whose attributes $x^{(i)}$ account for the occurrences within t of the fragment f_i, i.e. the fragment mapped onto the i^{th} dimension of the N-dimensional kernel space, and the kernel product is equivalent to the scalar product between pairs of such vectors, as exemplified in Figure 2.4. Here, the tree labeled t_1, on the left, contains the five fragments labeled 1-5, while the tree on

the right, t_2, contains the fragments labeled 5-7. Since the two trees only share the fragment labeled 5, the kernel product evaluates to 1.

Actually, each fragment can also be weighted according to one or more decay factors that penalize larger substructures. Decay factors are introduced to compensate for the intrinsic dependence between a large fragment and the smaller fragments it contains. For example, if we consider the fragment labeled as 4 in Figure 2.4 we can observe that it is a super-structure of fragments 1, 2, 3 and 5, which are already accounted for. In turn, fragment 3 can be expressed as a combination of 1 and 5.

Different kernel functions (e.g. [Collins and Duffy, 2002, Kashima and Koyanagi, 2002, Viswanathan and Smola, 2003, Moschitti, 2006b]) result in different constraints to the construction of fragments, that affect the topology and number of substructures that can be observed in a tree. More precisely, each kernel function defines implicitly: i) constraints about the topology of admissible fragments; ii) rules to generate the fragments encoded in a tree; iii) weights to be assigned to each fragment depending on how it is generated. All these aspects will be explained in more detail in the next sections and chapters.

The rest of this section details two kernel families that are especially interesting for computational linguistics, as they can effectively model problems involving constituency and dependency parsed data. In Section 3.1, other tree kernels and their applications to natural language processing will be discussed.

2.2.1 The Syntactic Tree Kernel

The Syntactic Tree Kernel (STK) [Collins and Duffy, 2001, Collins and Duffy, 2002] relies on a fragment definition that does not allow to break production rules, that is: if any child of a node is included in a fragment, then also all the other children have to. As such, it is especially indicated

for tasks involving constituency parsed texts as it allows to directly employ rich syntactic data in the learning algorithm.

Let $\mathcal{F} = \{f_1, f_2, \ldots, f_{|\mathcal{F}|}\}$ be an explicit representation of all the fragments encoded by the training data, i.e. its fragment space. Let $\chi_i(n)$ be an indicator function[3], equal to 1 if the target fragment f_i is rooted at node n, and equal to 0 otherwise. The STK function over t_1 and t_2 is defined as

$$STK(t_1, t_2) = \sum_{n_1 \in N_1} \sum_{n_2 \in N_2} \Delta(n_1, n_2), \tag{2.13}$$

where N_1 and N_2 are the sets of nodes in t_1 and t_2, respectively and

$$\Delta(n_1, n_2) = \sum_{i=1}^{|\mathcal{F}|} \chi_i(n_1)\chi_i(n_2). \tag{2.14}$$

The Δ function counts the number of subtrees rooted in n_1 and n_2 and can be evaluated as:

1. if the productions at n_1 and n_2 are different, then $\Delta(n_1, n_2) = 0$;

2. if the productions at n_1 and n_2 are the same, and n_1 and n_2 have only leaf children (they are pre-terminal symbols), then $\Delta(n_1, n_2) = \lambda$;

3. if the productions at n_1 and n_2 are the same, and n_1 and n_2 are not pre-terminals then

$$\Delta(n_1, n_2) = \lambda \prod_{j=1}^{l(c_{n_1})} (1 + \Delta(c_{n_1}^j, c_{n_2}^j)), \tag{2.15}$$

[3]We will consider it as a weighting function.

where $l(c_{n_1})$ is the number of children of n_1, c_n^j is the j-th child of node n, and λ is a decay factor penalizing larger structures.

2.2.2 The Partial Tree Kernel

The Partial Tree Kernel (PTK) [Moschitti, 2006a] defines a more general class of fragments, allowing any connected substructure of a tree to be considered as a valid fragment. Unlike the STK, it does not require that two nodes have the same productions in order to contribute to the kernel product. This feature makes it more appropriate to deal, for example, with dependency parsed text.

The evaluation of the common fragments rooted in two nodes n_1 and n_2 involves the evaluation of all the possible subsequences of the children of both nodes, and considers all the identical subsequences. As an example, let $n_1 = (S(DT)(JJ)(N))$ and $n_2 = (S(DT)(N))$. Even though the productions of the two nodes are different, we can observe that there is one children sequence of length 2 that is shared across n_1 and n_2, i.e. $[DT, NN]$. As a consequence, the two nodes also share two children sequences of length 1, i.e. $[DT]$ and $[NN]$. This process is no different than applying a sequence kernel [Lodhi et al., 2002] to the nodes children.

More formally, let Z_i be an index sequence associated with the ordered child sequence c_i of the node n_i. Let $Z_i[k]$ be the k-th element of Z, and $Z_i[-1]$ a notation for its last element. For example, if $n = (A(B)(C)(D))$, two of its possible index sequences would be $Z = [0, 2]$ (selecting nodes B and D) or $Z = [2]$ (selecting node D).

Let $l(Z_i)$ be the length of Z_i. Zimilarly to the STK, the PTK can be evaluated as:

$$PTK(t_1, t_2) = \sum_{n_1 \in N_1} \sum_{n_2 \in N_2} \Delta(n_1, n_2) \ , \tag{2.16}$$

but in this case the Δ function is defined as:

$$\Delta(n_1, n_2) = \mu \left(\lambda^2 + \sum_{Z_1, Z_2 | l(Z_1) = l(Z_2)} \lambda^{d(Z_1)+d(Z_2)} \prod_{i=1}^{l(Z_1)} \Delta(c_1^{Z_1[i]}, c_2^{Z_2[i]}) \right) \tag{2.17}$$

where

$$d(Z_i) = \begin{cases} 1 & \text{, if } l(Z_i) = 0 \\ Z_i[-1] - Z_i[1] + 1 & \text{, else.} \end{cases} \tag{2.18}$$

accounts for the length of the sequence Z_i in terms of the difference between the last and the first element in the sequence, plus 1. Thus, for example:

$$d([2,3,4,5]) = d([2,5]) = d([2,4,5]) = 5 - 2 + 1 = 4 \ ,$$

and

$$d([2]) = 2 - 2 + 1 = 1 \ .$$

The PTK makes use of two decay factors: μ, which accounts for the depth of the fragment, and λ, which accounts for the number of nodes in the fragment.

It should be noted that the set of fragments generated by the PTK is a superset of those generated by the STK. In the general case, the same fragments will be assigned different weights by the two kernels. This is a consequence of the different decay factors, and of the utterly different dimensionality of the induced spaces.

2.2.3 Tree Kernel Normalization

The output of tree kernel functions is generally normalized in the interval $[0, 1]$. Since the norm of a tree t can be evaluated as:

$$\|t\|_{\mathsf{TK}} = \sqrt{\phi(t) \cdot \phi(t)} = \sqrt{\mathsf{TK}(t, t)}\,, \tag{2.19}$$

where $\phi(\cdot)$ is the explicit mapping of a generic kernel function TK, to normalize $\mathsf{TK}(t_i, t_j)$ it is sufficient to replace it with:

$$\begin{aligned}
\widetilde{\mathsf{TK}}(t_i, t_j) &= \mathsf{TK}\left(\frac{t_i}{\|t_i\|}, \frac{t_j}{\|t_j\|}\right) \\
&= \frac{\mathsf{TK}(t_i, t_j)}{\|t_i\| \cdot \|t_j\|} \\
&= \frac{\phi(t_i)}{\sqrt{\mathsf{TK}(t_i, t_i)}} \cdot \frac{\phi(t_j)}{\sqrt{\mathsf{TK}(t_j, t_j)}} \\
&= \frac{\phi(t_i)}{\sqrt{\phi(t_i) \cdot \phi(t_i)}} \cdot \frac{\phi(t_j)}{\sqrt{\phi(t_j) \cdot \phi(t_j)}}\,.
\end{aligned} \tag{2.20}$$

2.3 Feature Selection Techniques

The problem of variable, or feature, selection arises in almost any research field, from gene microarray data analysis to image recognition and text categorization, where common machine learning problems are characterized by the necessity to cope with very large data sets, typically described by high-dimensional vectors in some dot product space.

Feature selection is the name given to a set of techniques commonly used to improve the quality of the models learned with machine learning methods. Depending on the context, it can aim to alleviate the effect of the curse of dimensionality [Bellman, 1961], enhance the generalization capabilities of the learning algorithm, improve the efficiency of the learning

process or make the models more easily interpretable. A very interesting and comprehensive survey on feature selection is carried out in [Guyon and Elisseeff, 2003].

As explained in Section 2.1.3, when using kernel functions we generally do not know explicitly all (if any) of the attributes that will represent the objects in the kernel space. Instead, a mapping function $\phi(\cdot)$ projects an example in some implicit feature space, generally very high if not infinite-dimensional. Given the very high dimensionality of kernel spaces, feature selection is a critical task for the realization of compact, accurate and efficient predictors. Feature selection strategies are typically divided into three main categories:

filters, where features are selected independently of the learning algorithm. Features are filtered based on some measure suggested by the data, such as the correlation between features and labels (e.g. mutual information);

wrappers, in which the learning algorithm is used as a black box to search the space of feature subsets. The learning machine is trained on different subsets of features. Then, the accuracy of the resulting model is evaluated and used to focus the search;

embedded methods, that incorporate the search of the feature subsets into the optimization problem of the learning algorithm. A common strategy is to minimize the cost function of the learner while enforcing some constraints on the dimensionality of the input space.

Filter methods are very generic, yet the kind of induction used by the filter may be utterly different from the one employed by the learning machine and introduce a new source of bias in the learning process.

In this respect, the main advantage of wrapper methods is that the same inductive method is responsible for both the evaluation of the relevance

of features and the learning, and no further bias is introduced. On the other hand, wrappers are computationally very expensive, and for very large feature spaces only rough searches (generally involving greedy algorithms) can realistically be performed.

Embedded methods share the virtues of wrapper methods, with the further advantage that the optimization problem can be refined in many subtle ways. The main disadvantages of this approach are the complexity of the implementation and the general impossibility to decouple the feature selection model from the embedding learning machine.

In Section 3.2 we will discuss a selection of interesting feature selection approaches in the context of support vector and high-dimensional kernel learning.

Chapter 3

Related Work

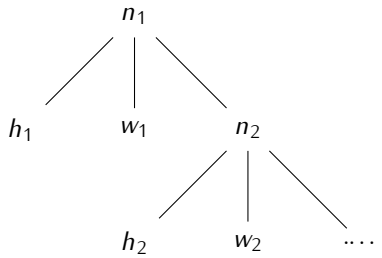
This chapter presents a selection of relevant work concerning the tree kernels and feature selection approaches for support vector machines and kernel methods.

In particular, in Section 3.1, we will focus our attention on several interesting applications of TKs that show how they have been successfully applied to a wide range of different tasks. These applications demonstrate the flexibility of the tool and its importance as a solution for all those situations where the clues about the relevant features are not enough to define accurate explicit models. These works motivate the interest towards effective feature selection strategies, and especially towards ways of making the most relevant fragments observable.

Concerning feature selection, in Section 3.2 we present an overview of feature selection techniques in the context of kernel methods and support vector learning. Due to the breadth of the topic and the vast amount of literature on this topic, we will only consider work that is in some way related to the approach that we propose.

3.1 Tree Kernels for Natural Language Processing

Seminal works for TK learning are [Collins and Duffy, 2001, Collins and Duffy, 2002], where the authors define the STK (see Section 2.2.1) and apply it to the task of parse reranking, in conjunction with the voted perceptron algorithm of [Freund and Schapire, 1999]. They also define a variant of the algorithm, the *Tagging Kernel*, employed for labeling tasks where a sentence S can be described as a sequence of states $S = [n_1, n_2, \ldots, n_{|s|}$, with each state n_i being a pair $\langle w_i, h_i \rangle$. Here, w_i is the i-th word in the sentence and h_i the associated tag. The tagging kernel, defined over pairs of state sequences, is equivalent to the evaluation of the STK on trees where each state n_i is a node whose children are h_i, w_i and the next state in the sequence n_{i+1}, e.g.



This is an interesting example of the flexibility of tree kernels, that due to their generality can be used to abstract a wide range of more specific problems and to prototype effective working solutions.

The PTK is introduced in [Moschitti, 2006a], where it is applied to the task of question classification. The paper clearly shows how the PTK can cope with dependency parsed data, whereas the constraints of the STK do not allow it the necessary flexibility to deal with the task. The kernels are also compared on a semantic role labeling benchmark defined over syntactic parse trees, where the STK shows far superior performance. Since it also generates all the STK fragments, the accuracy of the PTK is never noticeably inferior when applied to constituency parsed data. The loss

in accuracy of the PTK can mostly be ascribed to the extra fragments generated by the PTK, possibly overfitting the training data, and to the dimensionality of the fragment space.

In [Moschitti et al., 2007], the PTK is employed to build a tree-kernel driven model for question answering. Sequences (with gaps) of words or POS tags, which could be modeled using string kernels [Lodhi et al., 2002, Cancedda et al., 2003], are here evaluated by a PTK on pairs of ad-hoc engineered trees. A fake syntax is used as a container for the sequences of words/POS tags, and to allow the computation of the tree kernel.

In [Zhang and Lee, 2003], the authors describe a variant of the STK that also assigns a weight to terminal nodes, whereas the STK would not consider them independently of their pre-terminal parents. This allows the kernel to fall back to a bag of words (BOW) model in the cases where there is no syntactic overlap between two trees, i.e. the only contribution comes from the leaves. They also introduce a second decay factor that accounts for the depth of the trees, similarly to the PTK. The resulting kernel is applied to the task of coarse grained question classification.

In [Moschitti et al., 2008], the STK is used in conjunction with a polynomial kernel on an assessed set of attribute-value features for semantic role labeling. The classifiers that also employ tree kernels show an improvement over the explicit features, thus suggesting that the tree kernels are discovering new attributes which are relevant for the task and are not encoded by the linear features. Still, these features (and the feature classes they stand for) are unknown as they are only represented implicitly in the kernel space. The TK is also exploited to carry out fast feature-prototyping, by engineering artificial trees that encode the relation between a predicate and a set of candidate arguments in a reranking model for semantic annotations. In this case, the structures are designed so as to exploit the features of the STK.

In [Diab et al., 2008], TKs are used to tackle the problem of semantic role labeling for Arabic. Unlike the English language, where a set of relevant lexical and syntactic features for the task have been identified and commonly exploited [Gildea and Jurafsky, 2002, Pradhan et al., 2005, Xue and Palmer, 2004], this kind of linguistic knowledge is not available for the Arabic language. The STK is therefore used to automatically discover relevant features by only relying on the information encoded in parse trees. The results show that TKs are valuable tools for tackling in an effective way tasks where there is not enough knowledge to explicitly design a set of relevant features, but there is high availability of rich syntactic data.

[Kazama and Torisawa, 2005] describe an interesting algorithm to speed up TK evaluation. This algorithm looks for node pairs in which the rooted-subtrees share many substructures (malicious nodes) and applies a transformation to the trees rooted in such nodes to make the kernel computation faster. The results show a several-hundred-fold speed increase with respect to the basic implementation.

[Shen et al., 2003] define a lexicalized tree kernel based on the structured features generated by a Lexicalized Tree Adjoining Grammar (LTAG) and apply it to the task of parse reranking. The subtrees induced by the kernel are built using the set of elementary trees as defined by the LTAG, and the STK of [Collins and Duffy, 2002] is extended so as to increase the relevance of lexical features.

In [Zelenko et al., 2003], two kernels over syntactic shallow parser structures are devised for the extraction of linguistic relations, e.g. person-affiliation. To measure the similarity between two nodes, the contiguous string kernel and the sparse string kernel are used. [Culotta and Sorensen, 2004] generalize the approach by defining a kernel over dependency parsed sentences that provides a matching function for node pairs. Other examples of tree kernels for relation extraction include [Zhang et al., 2006], [Reichartz

et al., 2009] and [Nguyen et al., 2009].

3.2 Feature Selection for Support Vector Learning

As SVMs and kernel methods are very popular learning frameworks, they have also been studied in great detail with respect to feature selection issues, and many interesting approaches have been proposed. Most of the literature concentrates on polynomial and Gaussian kernels, and this may have two main justifications:

- these families of kernels have shown to be very general. They have a very broad field of application, and have successfully been applied to many domains, thus attracting the interest of different communities;
- other kernel families, such as convolution kernels, generate very high dimensional spaces to which traditional feature selection approaches may not be easily extended. Furthermore, as the resulting spaces cannot be traced back to a set of linear features previously extracted, convolution kernels have an inherent abstract quality that complicates the interpretation of the outcome of feature selection.

In the context of support vector learning, since results in statistical learning theory clearly link the gradient of the separating hyperplane to the margin on the risk [Vapnik, 1998, Schölkopf and Smola, 2001], most of the approaches try to remove as many features as possible while limiting the effect on the gradient.

A very popular approach to feature selection for linear problems and support vector machines is called Recursive Feature Elimination (RFE) [Guyon et al., 2002]. Basically, it is an embedded method that, after each training iteration, ranks features based on their weight magnitude, i.e. the associated gradient component, and selects out the (set of) feature(s) with the

smallest magnitude. The claim is that, by removing such features, the gradient norm, and hence the classifier's accuracy, is largely preserved. The authors also propose an extension to the non linear case, but its application requires that features in the primal space are explicitly represented, i.e. it can only work with kernels defined over $\mathbb{R}^N$.

In [Aksu et al., 2008] the authors observe that the theoretical assumption behind RFE is verified in the case of linear and polynomial cases, where it is possible to demonstrate that the norm of the gradient is monotonically increasing with the dimensionality of the space, but it does not hold in general. As an example, they claim to have empirical evidence (even though no theoretical proof) that for a Gaussian kernel the gradient norm can increase or decrease when removing features. As an alternative, they propose a method called Margin-based Feature Elimination (MFE) that directly enforces margin maximization after each feature selection step, in an iterative approach similar to RFE.

A study on several alternative embedded approaches to SVM feature selection is carried out in [Rakotomamonjy, 2003]. The author compares three strategies based on different criteria: the gradient norm, the radius/margin bound and the span estimate. He concludes that the approach based on the gradient norm criterion performs consistenly well across different data sets, and could be the most indicated for practical applications. It is interesting to observe that his gradient based approach is equivalent to RFE in the case of linear kernels.

In [Neumann et al., 2005], an embedded approach to select features using linear and non linear (polynomial and Gaussian) SVMs is detailed. For the former case, a combination of ℓ_0, ℓ_1 and ℓ_2-norm penalty terms is combined to achieve good feature selection and classification. For the latter, the authors introduce the appropriate indicator functions in the optimization problem, so that the features can be selected in the (explicit) input space

rather than in the (implicit) kernel space.

[Weston et al., 2003] exploit SVM as a feature selection device by considering the zero-norm of the gradient in the optimization problem of a linear SVM. As a result, the gradient can be used to project the most relevant features of the input vectors. The resulting features can then be used to train a traditional SVM. In the paper, which mostly discusses a computational-friendly approximation of the zero-norm optimization problem, the authors observe that their method does not generalize to non-linear kernels for which the mapping function cannot be explicitly represented.

All the work discussed so far addresses the problem of feature selection in the linear space, before considering the mapping implied by the kernel function. Conversely, the following approaches try to select the most relevant features in the high dimensional kernel space.

[Cao et al., 2007] present a general approach to feature selection in the kernel space based on the idea of building an orthogonal basis set in the kernel space. They provide theoretical proof that, even for infinite dimensional spaces, it is possible to identify a finite dimensional basis set that is a good approximation of the real one, based on the assumptions that training and test examples are drawn from the same distribution. The process of finding a basis set only depends on the input points and the kernel function, and therefore the basis set can be used to carry out learning and classification using any kernel machine. Feature weighting is carried out via a kernelized extension of the Relief [Kira and Rendell, 1992] method. The approach, which never makes the kernel space explicit, is general enough to be applied to any kernel function. In the paper, experiments are carried out on radial basis and sigmoid kernels.

Concerning convolution kernels, the most simple way to carry out feature selection would simply require to consider structures which have a limited size. This approach, which is also suggested in [Cancedda et al.,

2003] and [Collins and Duffy, 2001] for sequence and tree kernels, respectively, is motivated by two considerations: 1) large structures are very unfrequent, and therefore generally not relevant for classification; 2) convolution kernels include decay factors that make the contribution of large structures marginal. However, as also pointed out in [Suzuki and Isozaki, 2005], though, such methods inhibit the most interesting aspect of convolution kernels: their potentiality to generate large structured features that would not be represented otherwise. These large structures should at least have a chance to contribute their relevance to the learning problem.

The idea of an explicit representation of a kernel feature space to build a fast and accurate SVM is explored in [Kudo and Matsumoto, 2003]. The work focuses on polynomial kernels and relies on a rewriting of the kernel function that allows to shift most of the computational burden from the classifier onto the learner. This leads to a linear representation of the kernel space in which feature combinations are explicitly expanded, resulting in a very fast classifier. An extension of the PrefixSpan algorithm [Pei et al., 2001] is used to efficiently mine the features in the kernel space. The authors also discuss an approximation of their method for polynomial kernels of high degree, whose explicit representation cannot easily be dealt with. They also hint that a similar approach may be possible for tree kernels, by efficiently enumerating the effective fragments encoded in the support vectors.

In [Suzuki and Isozaki, 2005], the authors present a feature selection method for convolution kernels based on the statistical relevance of the features encoded in the data. The proposed methodology applies to convolution kernels and concentrates on efficiently mining the kernel space. The kernel function is extended to embed substructure mining and techniques for the evaluation of statistical significance. To assess the relevance of a structure (i.e. a partial sequence or a tree fragment), the χ^2 of its distri-

bution within the two classes is evaluated. A threshold is set to filter out all the structures with a low χ^2. The mining strategy, based on [Pei et al., 2001], considers structures of increasing size. An upperbound on the χ^2 of larger structures is the key ingredient to contain the complexity of the mining algorithm.

A very recent paper [Rieck et al., 2010] discusses a feature selection technique for tree kernels called Approximate Tree Kernel (ATK). The main idea behind ATK is to speed up TK evaluation for very large trees (e.g. HTML or XML documents) by only considering fragments rooted in nodes with certain labels. The authors redefine the optimization problem by forcing a limit to the number of node types (symbols) in which a fragments can be rooted.[1] Experiments are carried out on on question classification and spam detection. In both cases, accuracy is comparable with a standard TK, even if only a very small number of symbols (between five and ten) are retained. On question classification, training and test time are reduced by a factor of 1.7 and 1.8, respectively. The improvement is more noticeable on the larger spam detection benchmark, on which training and classification are approximately thirteen times as fast. Space complexity of TK evaluation is also considerably reduced. The approach is very interesting in terms of feature selection, and it also provides some interesting insights concerning relevant features in the kernel space. On the other hand, its benefits are mostly exploited in those cases in which a small fraction of the symbols of the grammar are relevant for the task. In fact, optimization and classification still rely on the dual representation. This aspect may limit its application to very complex syntactic tasks, such as relation extraction or argument boundary detection for semantic role labeling.

[1] They also present results in the context of unsupervised learning, in which case the selection of symbols is based on a bound on the expected time complexity of ATK evaluation.

Chapter 4

Mining Fragments Efficiently

At a very high level, the feature projection process that we propose consists of three main tasks:

- We exploit the target kernel function in the original, high dimensional space in combination with the SVM optimizer to carry out a first step of example selection and select the most relevant example points, i.e. the support vectors. This step is called *Kernel Space Learning* (KSL), since learning occurs in the space of the target kernel function;
- We use a greedy algorithm to explore the fragment space encoded by the support vectors, generate the most relevant fragments and store them into an *index*. We employ a gradient-based approach to decide wich features to retain or discard, and also as a criterion to guide the greedy exploration of the fragment space. Indeed, fragments are selected based on their contribution to the norm of the gradient of the model learnt during KSL. This stage is called *Kernel Space Mining* (KSM);
- The index is used to *decode* the input structured data, i.e. the trees in the dataset of the TK learning problem, and to represent them as vectors in a linear space. This step is called *Linear Space Generation* (LSG);

These three main building blocks can be combined in different ways, resulting in different architectures for tackling different learning problems or stress different properties of the feature selection methodology, as explained in Section 4.1.

Gradient-based approaches to feature selection (see Section 3.2) exploit the idea that a good variable selection strategy should have a limited effect on the geometry of the separating hyperplane, i.e. on the gradient. The contribution of each variable to the norm of the gradient is used to establish a ranking between features (or feature sets) and hence to discard the least relevant ones.

The component of the gradient associated with each feature can be calculated as a linear combination of the weights, estimated by the learning algorithm optimizer for the training points, with the values assumed by the feature in each examples. For a linear classification problem having ℓ training examples $\boldsymbol{x}_i \in \mathbb{R}^N$, the absolute value of the j-th component $w^{(j)}$ of the gradient $\boldsymbol{w} = [w^{(1)}, w^{(2)}, \ldots, w^{(N)}]$ has value:

$$w^{(j)} = \sum_{i=0}^{\ell} \alpha_i y_i x_i^{(j)} \ . \tag{4.1}$$

Generalizing this criterion to any kernel function K is straightforward if we assume that $\boldsymbol{x}_i$ is the result of the application of the mapping function ϕ_K to the input object o_i. Still, in order to apply (4.1) it must be possible to isolate the value that the feature mapping function projects on each dimension j for any given object o_i, i.e. $x_i^{(j)}$. In other words, it must be possible to weigh individual components of the kernel space. How this value can be calculated in the case of the STK and PTK functions will be explained in Section 4.2.

The exploration of the huge tree fragment space is a very challenging task in terms of temporal and spatial complexity. Efficient algorithms based

on solid theoretical assumptions and compact data structures are necessary pre-requisites for a feature selection approach that should be both computationally reasonable, by possibly improving the efficiency of learning and classification, and preserve the accuracy of the rich tree kernel function.

The last three sections of this chapter deal with these aspects of the problem: Section 4.3 presents theoretical results that provide a formal justification to the criterion employed for the greedy exploration of the fragment space; Section 4.5 describes the algorithms used to enumerate the fragments and to explore the fragment space; finally, Section 4.6 discusses the datastructure that is used to store (during KSM) and access (during LSG) the mined fragments conveniently.

4.1 Architectural Configurations

In this section we describe three architectures for feature selection in TK spaces. In our experiments (Chapter 5), the three models will be employed to assess different properties of our linearization technique. The first architecture, MLin (Sec. 4.1.1), is a valuable tool to empirically support our theoretical claims (discussed in Section 4.3). The second architecture, LOpt (Sec. 4.1.2), can produce very accurate and efficient linear classifiers, that alleviate the burden of TK classification. The third architecture, Split (Sec. 4.1.3), can be used to reduce learning time on large data sets, while retaining most of the accuracy of non-linearized TK models.

4.1.1 Model Linearization (MLin)

The first architecture that we present is called *Model Linearization* (MLin), depicted in Figure 4.1. In the diagram, black boxes stand for training activities; light gray boxes stand for testing activities; arrows indicate data

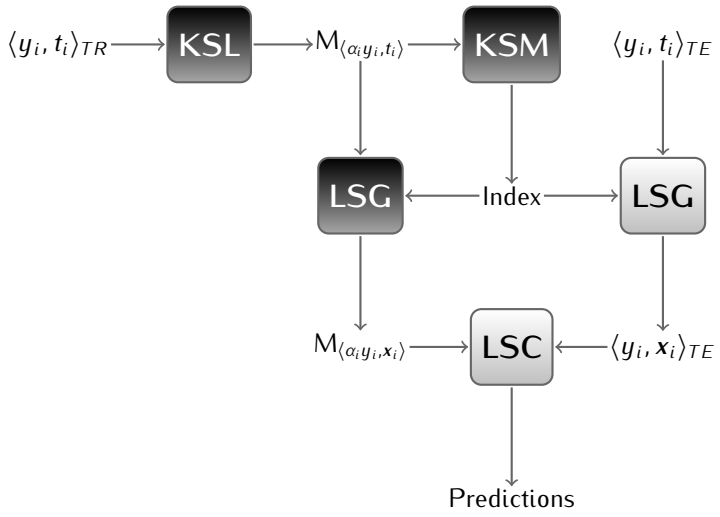


Figure 4.1: **Architectural overview of an MLin classifier.**

flowing between activities of the process. With respect to the figure, training an MLin binary classifier involves the following steps:

1. KSL: all the available training data is used to learn an STK model. Training data consists of label/tree pairs, $\langle y_i, t_i \rangle_{TR}$. In the model $M_{\langle \alpha_i y_i, t_i \rangle}$, each support vector t_i is associated with its estimated weight and label $\alpha_i y_i$;

2. KSM: the model M is mined and the most relevant fragments are stored in an index;

3. LSG: by means of the index, the support vectors are decoded, i.e. represented as vectors in the linear space. A linear model $M_{\langle \alpha_i y_i, \boldsymbol{x}_i \rangle}$ is built by combining:

   - the linearized support vectors, $\boldsymbol{x}_i$;
   - their labels, y_i;
   - the weights estimated for them in the TK space, α_i.

Concerning test activities, they are:

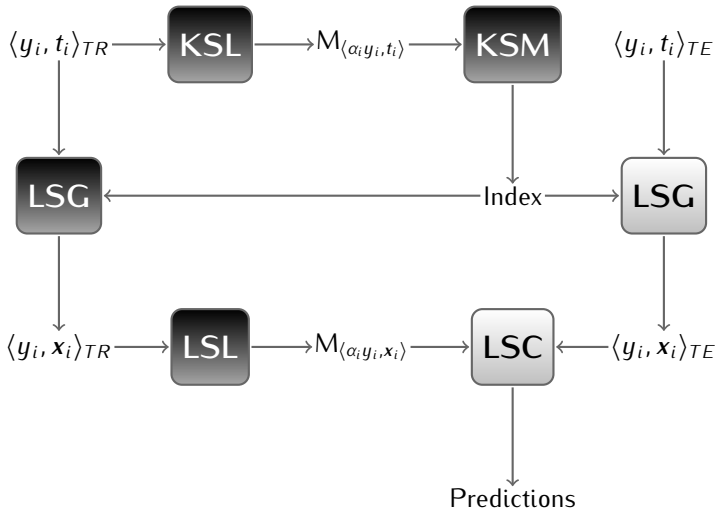


Figure 4.2: **Architectural overview of an LOpt classifier.**

1. LSG: test data, consisting of label/tree pairs $\langle y_i, t_i \rangle_{TE}$, is projected onto a lower dimensional space, resulting in $\langle y_i, \boldsymbol{x}_i \rangle_{TE}$;

2. LSC: we use the linearized model $M_{\langle \alpha_i y_i, \boldsymbol{x}_i \rangle}$ to classify $\langle y_i, \boldsymbol{x}_i \rangle_{TE}$.

MLin is a very simple architecture, in which we reuse the support vectors and their weights, as estimated by the learner in the TK space, to carry out classification in the target linear space.

Since the weights are estimated for a space that is utterly different from the projected linear space, we would not expect this approach to result in very accurate classifiers. Nonetheless, studying this kind of classifier is interesting to assess the aftermaths of feature selection on the original kernel space. The experiments that exploit this property are detailed in Section 5.2.

4.1.2 Linear Space Optimization (LOpt)

With the Linear Space Optimization (LOpt) architecture, we learn a new model in the target low dimensional space. The diagram for this architecture is shown in Figure 4.2. The differences with MLin are the following: 1) we

linearize all the available training data; 2) the model learnt during KSL is solely employed for feature mining. Training LOpt involves:

1. KSL: we use the structured data to learn an STK model (same as MLin);
2. KSM: the model is mined to collect the most relevant fragments into an index (same as MLin);
3. LSG: all the available training data $\langle y_i, t_i \rangle_{TR}$ are linearized, resulting in $\langle y_i, \boldsymbol{x}_i \rangle_{TR}$. Every input tree is now represented as a vector in a linear space;
4. LSL: the linearized data is used to learn a new model $M_{\langle \alpha_i y_i, \boldsymbol{x}_i \rangle}$ in the lower dimensional space.

As for testing, the LOpt and MLin configurations are just the same: the structured test data are linearized, and classified with the linear model.

Unlike MLin, the support vectors retained in the linear model will be generally different from those used during KSM, and the new SVM-learned weights will be optimal with respect to the target low-dimensional space. In LOpt, the model learnt during KSL is only exploited for fragment mining. LSG is applied to all the available training and test data, and LSL is carried out on the linearized training data $\langle y_i, \boldsymbol{x}_i \rangle_{TR}$ to obtain the linear model $M_{\langle \alpha_i y_i, \boldsymbol{x}_i \rangle}$, which in turn is used to classify the decoded test data.

This kind of configuration can produce very fast and accurate linear classifiers, a property that will be exploited in Section 5.3 to assess the accuracy of linearized classifiers.

4.1.3 Split KSL, Linear Space Optimization (Split)

The good accuracy achieved with cascades of SVMs [Graf et al., 2004] suggests that support vectors that are collected from locally learned models

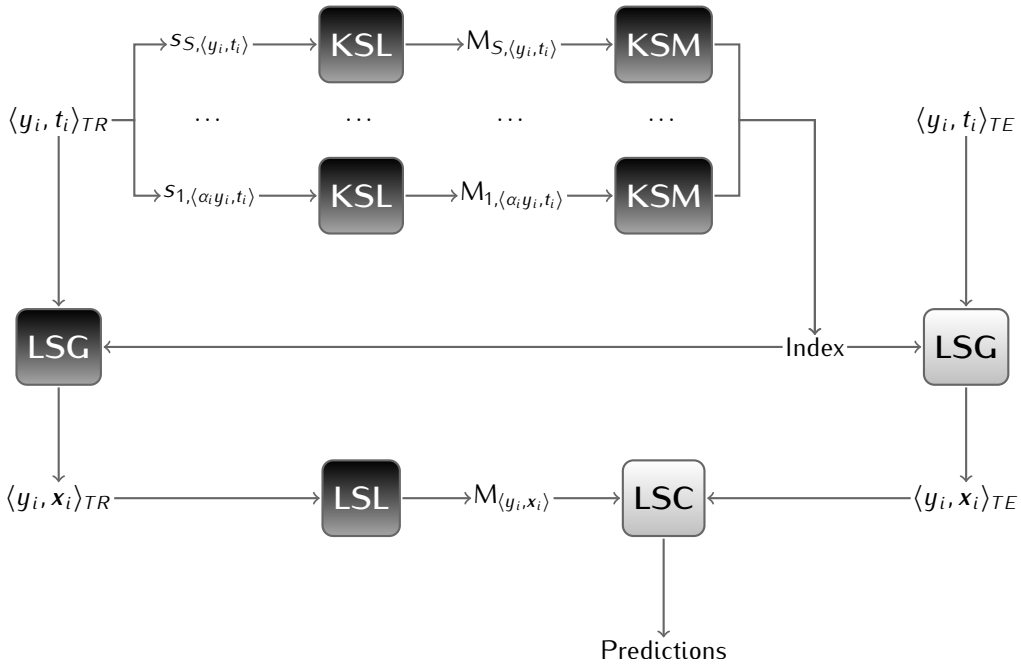


Figure 4.3: **Architectural overview of a Split classifier.**

encode many of the relevant features retained by global models.

The Split architecture is an extension of LOpt in which this property is exploited to improve the efficiency of learning from large datasets. We partition training data into S smaller sets, learn S models and mine fragments from each of them. The fragments mined from all the models are then used for LSG. As shown in Figure 4.3, training a Split classifier involves:

1. KSL: S tree kernel models $M_{1,\langle \alpha_i y_i, t_i \rangle}$, ..., $M_{S,\langle \alpha_i y_i, t_i \rangle}$ are learned independently on S splits of the available trainin data;

2. KSM: each model is mined. The relevant fragments collected from all the models are collected in a unique index. The index contains features that were observed only in one of the models, as well as features appearing in more than one;

3. LSG: all the available training data is linearized using the fragments stored in the index (same as LOpt);

4. LSL: the linearized training data are used to learn an optimized model in the low dimensional space (same as LOpt).

Since SVM training time is approximately quadratic in the number of examples, with the Split configuration we expect to achieve considerable efficiency improvements when estimating support vector weights in the TK space. According to statistical learning theory, being trained on smaller subsets of the available data, these models will be less robust with respect to the minimization of the empirical risk [Vapnik, 1998]. Nonetheless, since the weights are only needed to establish a coarse ranking among fragments, we can accept to rely on sub-optimal solutions. In Section 5.5 we will show that Split classifiers can indeed result in much faster learning cycles. Provided that the data set is large enough, the efficiency improvement can come at little or no cost in terms of accuracy.

4.2 Relevance Estimation

This section explains how we can calculate feature weights in the rich space generated by the STK and PTK functions.

4.2.1 STK Fragments

Eq. 2.14 shows that Δ counts the shared fragments rooted in n_1 and n_2 in the form of scalar product, as evaluated by Eq. 2.13. However, when λ is used in Δ as in Eq. 2.15, it changes the weight of the product $\chi_i(n_1)\chi_i(n_2)$ according to the topology of the fragment. As λ multiplies Δ in each recursion step, we may be induced to assume that the weight of a fragment is λ^d, where d is the depth of the fragment.

On the contrary, we should consider that the kernel product holds between pairs of fragments. Therefore, the term λ^d is contributed by one

fragment for $\lambda^{d/2}$, and by the other for $\lambda^{d/2}$. Furthermore, the exponent of the decay factor does not depend on the depth of the fragment d, but rather on the number of nodes with children that it contains, $s(f)$. It follows that the real weight of an individual fragment is $\lambda^{s(f)/2}$. With the following theorem, we prove that the correct exponent of λ is the number of fragment nodes that have at least one child, divided by 2: [1]

Theorem 4.2.1. *Let T and f be a tree and one of its STK fragments, respectively. The weight of f accounted by STK is $\lambda^{\frac{s(f)}{2}}$, where $s(f) = |\{n \in T : l_f(n) > 0\}|$ is the number of nodes that have active productions in the fragment (i.e. at least one child) and $l_f(n)$ is the number of children of n in f.*

Proof. The thesis can be proven by induction on the depth d of f. The base case is f of depth 1. Fragments of depth 1 are matched by step 2 of $\Delta(n_1, n_2)$ computation, which assigns a value $\lambda = \chi_i(n_1)\chi_i(n_2)$ (where $f_i = f$ are the minimal fragments rooted in n_i), independent of the number of children. Since $\chi(n_1) = \chi(n_2)$, it follows that the weight of f is $\lambda^{1/2}$, due to $n_1 = n_2$.

Suppose that the thesis is valid for depth d and let us consider a fragment f of depth $d+1$, rooted in r. Without loss of generality, we can assume that f is in the set of the fragments rooted in n_1 and n_2, as evaluated by Eq. 2.15. It follows that the production rules associated with n_1 and n_2 are identical to the production rule in r. Let us consider $M = \{i \in \{1, .., l(n_1)\} : l(c_r^i) > 0\}$, i.e. the set of child indices of r which have at least one child. For $j \in M$, c_r^i has a production shared by $c_{n_1}^j$ and $c_{n_1}^j$. Conversely, for $j \notin M$, there is no match and $\Delta(c_{n_1}^j, c_{n_2}^j) = 0$.

The resulting product is $\lambda \prod_{j \in M} \Delta(c_{n_1}^j, c_{n_2}^j)$, where the term 1 in $(1 +$

[1] In [Collins and Duffy, 2002], there is a short note about the correct value of the weight of lambda for each product components (i.e. pairs of fragments), and also in [Zhang et al., 2006] there are hints in this direction.

$\Delta(c^j_{n_1}, c^j_{n_2})$) is not considered since it accounts for those cases in which there are no common productions in the children, i.e. $c^j_{n1} \neq c^j_{n2} \forall j \in M$.

We can now substitute $\Delta(c^j_{n_1}, c^j_{n_2})$ with the weight of the subtree t_j of f rooted in c^j_r (and extended until its leaves), which is $\lambda^{s(t_j)}$ by inductive hypothesis (since t_j has depth lower than d). Thus, the weight of f is $\lambda \prod_{j \in M} \lambda^{s(t_j)} = \lambda^{1+\sum_{j \in M} s(t_j)}$, where $\sum_{j \in M} s(t_j)$ is the number of nodes in f's subtrees rooted in r's children and having at least one child; by adding 1, for the contribution of r, we obtain $s(f)$. Finally, we have $\lambda^{s(f)} = \chi_i(n_1)\chi_i(n_2)$, which satisfies our thesis: $\chi_i(n_1) = \chi_i(n_2) = \lambda^{\frac{s(f)}{2}}$. □

In the light of this result, we can use the definition of a TK function to project a tree t onto a linear space, by recognizing that t can be represented as a vector $\boldsymbol{x}_i = [x_i^{(1)}, \ldots, x_i^{(N)}]$ whose attributes are the count of the occurrences of each fragment weighed with respect to the dacay factor λ.

For a normalized STK, the value of the j-th attribute of the example $\boldsymbol{x_i}$ is therefore:

$$x_i^{(j)} = \frac{t_{i,j}\lambda^{\frac{s(f_j)}{2}}}{\|\boldsymbol{x}_i\|} = \frac{t_{i,j}\lambda^{\frac{s(f_j)}{2}}}{\sqrt{\sum_{k=1}^{N} t_{i,k}^2 \lambda^{s(f_k)}}} \tag{4.2}$$

where $t_{i,j}$ is the number of occurrences in the tree t_i of the fragment f_j, associated with the j-th dimension of the feature space. It follows that the components of $\boldsymbol{w}$ (see Eq. 2.5) can be rewritten as:

$$w^{(j)} = \sum_{i=1}^{\ell} \alpha_i y_i x_i^{(j)} = \sum_{i=1}^{\ell} \frac{\alpha_i y_i t_{i,j}\lambda^{\frac{s(f_j)}{2}}}{\sqrt{\sum_{k=1}^{N} t_{i,k}^2 \lambda^{s(f_k)}}} \, . \tag{4.3}$$

4.2.2 PTK Fragments

With a similar reasoning, it is also possible to calculate the weight of PTK fragments.

The reader should recall that in (2.18) we used the symbol Z_n to represent an index sequence of the children of a node n. For the way it is constructed, Z_n has at most $l(c_n)$ elements, where c_n is the ordered set of n's children and the operator $l(\cdot)$ calculates its length.

The values of Z_n can range from 1 to $l(c_n)$, and it holds that $k < k' \Rightarrow Z[k] < Z[k']$. The quantity $d(Z_n)$ for an index sequence Z_n was defined as the difference between the last and the first value in the sequence, plus 1, i.e. $d(Z_n) = Z_n[-1] - Z_n[1] + 1$.

If we decouple the contribution of the two fragments in (2.17), we obtain that the cumulative relevance of a PTK fragment can be measured with:

$$w^{(j)} = \sum_{i=1}^{\ell} \alpha_i y_i x_i^{(j)} = \sum_{i=1}^{\ell} \frac{\alpha_i y_i t_{i,j} \mu^{\frac{n(f_j)}{2}} \lambda^{D(f_j)}}{\sqrt{\sum_{a=1}^{N} t_{i,k}^2 \mu^{n(f_k)} \lambda^{2D(f_k)}}} \,. \tag{4.4}$$

where $n(f) = |N_f|$ is the number of nodes in f, and $D(f)$ is defined as

$$D(f) = \sum_{n \in N_f} d(Z_n) \,,$$

i.e. it is the cumulative length of the node sequences Z_n evaluated for each node in the fragment. Since it depends on the set of node expansions carried out on the original tree, the value of Z_n for any node in f cannot be derived by observing its surface form. Therefore, for PTK fragments we also need to store the value of $D(f)$ and update it at every expansion.

4.3 Theoretical Justification

In order to provide a theoretical background to our feature selection technique and to develop effective algorithms, we want to relate our approach to statistical learning and, in particular, support vector classification theory. Since we select features with respect to their weight $w^{(j)}$, we can use

Theorem 2.1.2 that establishes a general bound for margin-based classifiers.

According to the theorem, if $\mathcal{X}$ is separated with a margin γ by a linear classifier, then the error has a bound depending on γ. A feature selection algorithm that wants to preserve the accuracy of the original space should not affect the margin.

Since we would like to exploit the availability of the initial gradient $\boldsymbol{w}$ derived by the application of SVMs, it makes sense to try to quantify the percentage of γ reduction after feature selection, which we indicate by ρ. We found out that γ is linked to the reduction of $\|\boldsymbol{w}\|$, as illustrated by the next lemma.

Lemma 4.3.1. *Let $\mathcal{X}$ be a set of points in a vector space and $\boldsymbol{w}$ be the gradient vector which separates them with a margin γ. If the selection decreases $\|\boldsymbol{w}\|$ by a rate ρ, then the resulting hyperplane separates $\mathcal{X}$ by a margin larger than $\gamma_{in} = \gamma - \rho R\|\boldsymbol{w}\|$.*

Proof. Let $\boldsymbol{w} = \boldsymbol{w}_{in} + \boldsymbol{w}_{out}$, where $\boldsymbol{w}_{in}$ and $\boldsymbol{w}_{out} \in \mathbb{R}^N$ are constituted by the components of $\boldsymbol{w}$ that are selected in and out, respectively, and have zero values in the remaining positions. By hypothesis, $|\boldsymbol{w} \cdot \boldsymbol{x}| \geq \gamma$. Without loss of generality, we can consider just the case $\boldsymbol{w} \cdot \boldsymbol{x} \geq \gamma$, and write

$$\begin{aligned} \boldsymbol{w} \cdot \boldsymbol{x} &= \boldsymbol{w}_{in} \cdot \boldsymbol{x} + \boldsymbol{w}_{out} \cdot \boldsymbol{x} \geq \gamma \\ &\Rightarrow \\ \boldsymbol{w}_{in} \cdot \boldsymbol{x} &\geq \gamma - \boldsymbol{w}_{out} \cdot \boldsymbol{x} \\ &\geq \gamma - |\boldsymbol{w}_{out} \cdot \boldsymbol{x}| \\ &\geq \gamma - \|\boldsymbol{w}_{out}\| \times \|\boldsymbol{x}\| \ , \end{aligned} \tag{4.5}$$

from Cauchy-Schwarz inequality. The margin associated with $\boldsymbol{w}_{in}$, i.e. γ_{in}, is therefore

$$\gamma_{in} \geq \gamma - \|\boldsymbol{w}_{out}\| \times \|\boldsymbol{x}\| \geq \gamma - \|\boldsymbol{w}_{out}\|R = \gamma - \rho R\|\boldsymbol{w}\| \ . \tag{4.6}$$

□

Remark 4.3.2. *The lemma suggests that, even in case of very aggressive feature selection, if a small percentage ρ of $\|\boldsymbol{w}\|$ is lost, the margin reduction is small. Consequently, through Theorem 2.1.2, we can conclude that the accuracy of the model is by and large preserved.*

Remark 4.3.3. *We prefer to show the lemma in the more general form, but if we use normalized $\boldsymbol{x}$ and classifiers with $\|\boldsymbol{w}\| \leq 1$, then $\gamma_{in} = \gamma - \|\boldsymbol{w}\|\rho > \gamma - \rho$.*

A note on the relation between the gradient and the margin.

The reader should not be confused by the fact that, for a linear classifier, the norm of the gradient $\|\boldsymbol{w}\|$ is inversely proportional to the margin, γ: $\gamma = \frac{2}{\|\boldsymbol{w}\|}$ (see Fig. 2.2). In that context, we are learning an optimal separation for the two classes: the best separation maximizes the margin, hence minimizing the gradient norm.

In the context of Lemma 4.3.1, the point of view is completely different. We are transforming the input space by ignoring several dimensions. We show that if we select these dimensions so as to have a small effect on the gradient norm, the margin is only slightly affected. In fact, we are inducing a trasformation on the initial space that has a limited (and measurable) effect on our ability to tell the classes apart. This ensures that the features that we are retaining are those that encode the relevant information.

The last result that we present justifies our selection approach, as it demonstrates that most of the gradient norm is concentrated in relatively few features, with respect to the huge space induced by tree kernels. The selection of these few features allows us to preserve most of the norm and the margin.

Lemma 4.3.4. *Let $\boldsymbol{w}$ be a linear separator of a set of points $\mathcal{X}$, where each $\boldsymbol{x}_i \in \mathcal{X}$ is an explicit vector representation of a tree in the space induced by STK, and let v be the maximum size (number of active productions) in any tree. Then, if we select fragments with size greater than η, it holds*

that

$$\|\boldsymbol{w}_{out}\| \leq \frac{\nu}{\gamma^2}\sqrt{\frac{(\lambda\nu)^{\eta} - (\lambda\nu)^{\nu}}{1-\lambda\nu}} \,. \tag{4.7}$$

Proof. By applying simple norm properties,

$$\|\boldsymbol{w}_{out}\| = \left\|\sum_{i=1}^{\ell} \alpha_i y_i \boldsymbol{x}_{out_i}\right\| \leq \sum_{i=1}^{\ell} \|\alpha_i y_i \boldsymbol{x}_{out_i}\| = \sum_{i=1}^{\ell} \alpha_i \|\boldsymbol{x}_{out_i}\| \,. \tag{4.8}$$

To evaluate the latter, we first re-organize the summation in Eq. 4.2 by summing on fragments of different size, obtaining:

$$\|\boldsymbol{x}_i\|^2 = \sum_{k=1}^{\nu} \sum_{j:s(f_j)=k} \frac{t_{i,j}^2 \lambda^{s(f_j)}}{\sum_{k=1}^{N} t_{i,k}^2 \lambda^{s(f_k)}} \,. \tag{4.9}$$

Since a fragment f_j can be at maximum rooted in ν nodes, then $t_{i,j} \leq \nu$. Moreover, for not extremely small λ, it holds that $\sum_{k=1}^{N} t_{i,k}^2 \lambda^{s(f_k)} > 1$ (e.g. for $\lambda > 1/\nu$).

By using ν^k as an upper bound for the number of trees having size k, we obtain

$$\|\boldsymbol{x}_i\| < \sqrt{\sum_{k=1}^{\nu} \nu^2 \lambda^k \nu^k} = \sqrt{\sum_{k=1}^{\nu} \nu^2 (\nu\lambda)^k} = \sqrt{\nu^2 \frac{1-\mu^{\nu}}{1-\mu}} \,, \tag{4.10}$$

where we have assumed that $\mu = \lambda\nu < 1$ (by using a small enough λ) and applied geometric series summation. If we assume that our algorithm selects out (i.e. discards) fragments with size $s(f) > \eta$, we can write $\|\boldsymbol{x}_{out_i}\| <$

$\sqrt{\nu^2 \frac{\mu^\eta - \mu^\nu}{1-\mu}}$. It follows that

$$\|\boldsymbol{w}_{out}\| < \sum_{i=1}^{\ell} \alpha_i \sqrt{\nu^2 \frac{\mu^\eta - \mu^\nu}{1-\mu}} \,. \tag{4.11}$$

In case of hard-margin SVMs, we have $\sum_{i=1}^{\ell} \alpha_i = 1/\gamma^2$. It follows that

$$\|\boldsymbol{w}_{out}\| < \frac{\nu}{\gamma^2} \sqrt{\frac{\mu^\eta - \mu^\nu}{1-\mu}} = \frac{\nu}{\gamma^2} \sqrt{\frac{(\lambda\nu)^\eta - (\lambda\nu)^\nu}{1-\lambda\nu}} \,. \tag{4.12}$$

□

Remark 4.3.5. *The lemma shows that for an enough large η and $\lambda < 1/\nu$, $\|\boldsymbol{w}_{out}\|$ can be very small, even though it includes an exponential number of features, i.e. all the subtrees whose size ranges from η to ν. Therefore, according to Lemma 4.3.1 and Theorem 2.1.2, we can discard an exponential number of features with a limited loss in accuracy.*

Remark 4.3.6. *Regarding the proposed norm bound, we observe that ν^k is a coarse overestimation of the the real number of fragments having size k rooted in the nodes t. In case of soft-margin SVMs, we can bound α_i with the value of the trade-off parameter C.*

4.4 Generating Fragments

As already mentioned in Section 2.2, a fragment f is a substructure of some tree t. A fragment is rooted in a node $n \in N_t$ (the set of nodes of t), and it comprises a set of nodes $N_f \subseteq N_t$, with the constraint that the resulting graph must be connected in the original tree.

In the remainder, let $E_f \subseteq N_f \subset N_t$ be the set of expandable nodes of f. Expandable nodes are nodes that are leaves with respect to the fragment

(they have no active production in the fragment) but that are not leaves with respect to the original tree t. For example, if $t = (A(B(b))(C(c)))$ and $f = (A(B(b)(C)))$, then $E_f = \{C\}$, since C is the only node of f that has children in t but that has no children in f.

The fragments encoded in a tree t can be enumerated by combining two atomic operations:

FRAG(n) (*base fragment generation*), that builds the smallest fragment rooted in $n \in N_t$, and

EXPAND(f) (*fragment expansion*), that builds the set of fragments that span one more level of the tree. A fragment expansion consists of one or more *node expansions*, in which one or more children of a node $n \in E_f$ are included in the fragment.

The actual implementation of the two operations depends on the target kernel function and on the kind of fragments that it can generate.

4.4.1 STK Fragments

According to the definition of the STK (see Section 2.2.1), fragments that span a single level of the tree, i.e. isolated nodes, do not contribute to the evaluation of the kernel function. Therefore, the smallest possible fragment must contain at least one node and some of its children. However, the STK does not allow us to break production rules. Therefore, if a fragment includes any of the children of a node n, then it must also include all their siblings. It follows that the minimal fragments that can be generated according to the definition of the STK are those that encompass a node and all its direct children, i.e. all the fragments that describe a production rule of the grammar. It follows that the number of base fragments in a tree t is the number of internal nodes, i.e. I_t.

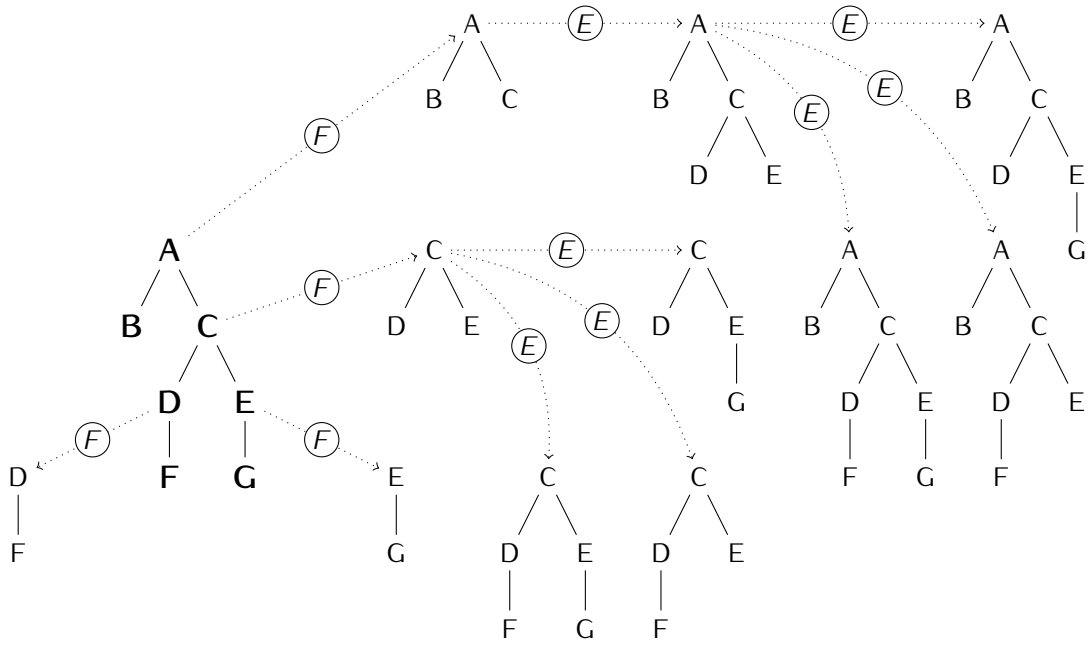


Figure 4.4: **Recursive enumeration of the STK fragments encoded in a tree.** – The fragments are generated by combining the two atomic operations FRAG(·) (F) and EXPAND(·) (E).

As for the EXPAND(f) operation, the number of fragments it generates depends on the number of leaves in f that have active productions in t, i.e. $|E_f|$. The definition of the STK forces us to include all the children of a node whenever we include at least one of its children. It follows that for any expandable node in a fragment there is only one possible expansion, i.e. the one in which all its children are included. Since we must consider all the possible combinations of nodes to expand, the complexity of this operation for the STK is:

$$C(\text{EXPAND\_STK}(f)) = \sum_{i=1}^{|E_f|} \binom{|E_f|}{i} \tag{4.13}$$

Figure 4.4 shows how the FRAG(·) and EXPAND(·) operations can be combined to generate all the STK fragments encoded in a tree.

4.4.2 PTK Fragments

The PTK provides a very general definition of fragment. Indeed, every connected subset of a tree is a valid fragment according to the PTK definition (see Section 2.2.2) . As a consequence, generating PTK fragments is an intrinsecally complex operation.

Since even a single node is a valid fragment, in a tree having $|N_t|$ nodes there are exactly $|N_t|$ base fragments.

Concerning the EXPAND$(\cdot)$ operation, it should be considered that expanding a node n is a combinatorial operation. For any node $n \in E_f$ having $|c_n|$ children we obtain $\sum_{i=i}^{|c_n|} \binom{|c_n|}{i}$ different fragments, i.e. one fragment for every combination of n's children. If we consider all the nodes that can be expanded in f, i.e. E_f, then the number of generated fragments is:

$$C(\text{EXPAND\_PTK}(f)) = \prod_{n_j \in E_f} \sum_{i=1}^{|c_{n_j}|} \binom{|c_{n_j}|}{i} \tag{4.14}$$

Figure 4.5 represents graphically the process of generating all the PTK fragments in a tree by combining the FRAG$(\cdot)$ and EXPAND$(\cdot)$ operations. First, the five base fragments are generated. Then, they are recursively expanded to generate the larger fragments. As we can see, even a very small tree like the one in figure can generate a conspicuous number of fragments.

4.5 Algorithms for Fragment Mining

We call *fragment mining* the process by which the fragment space encoded by a set of tree is explored, and the most relevant fragments are stored in an index. Since we use SVM estimated weights to assess the relevance of fragments, henceforth we will assume that every input tree has an associated weight, i.e. the set of tree is an SVM model M.

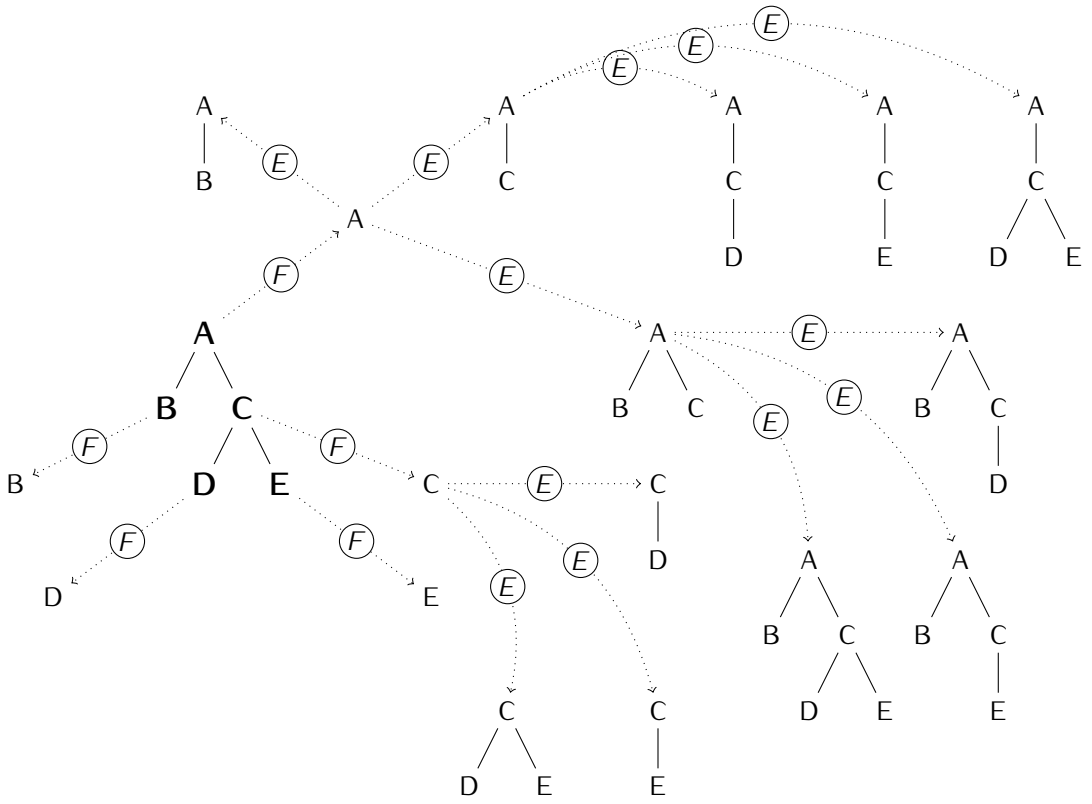


Figure 4.5: **Recursive enumeration of the PTK fragments encoded in a tree.** - The fragments are generated by combining the two atomic operations FRAG(·) (F) and EXPAND(·) (E).

The algorithms described in the following pages make use of several operators that is convenient to define in advance:

COPY(x), which makes a shallow copy of an object x;

MKINDEX(), which creates a new index to store the relevant features. The actual implementation of the index will be detailed in Section 4.6;

UPDATE(i,f), which updates the index i with the fragment f. The index keeps track of the cumulative relevance of a fragment, i.e. all the instances of the same fragment across all the input trees. For simplicity, it will be assumed that the fragment contains all the required information to calculate its individual relevance.

We will start defining a very naive approach to fragment mining, going through the steps that led us to the formulation of the greedy mining strategy currently employed in our model.

4.5.1 Naive Fragment Space Generation

Algorithm 4.1 FULL_MINER(*model*)

```
global result
main
 result ← MKINDEX()
 for each tree ∈ model
    do { for each node ∈ NODES(tree)
         do MINE(FRAG(node))
 return (result)

procedure MINE(frag)
 UPDATE(result, frag)
 { for each fragment ∈ EXPAND(frag)
   do MINE(fragment)
```

The most naive approach to fragment mining would be the generation of the complete fragment space encoded by the model. The FULL_MINER(·) procedure, listed in Algorithm 4.1, shows how the basic operators FRAG(·) and EXPAND(·) can be combined to achieve this goal. All the nodes of all the trees in the model are traversed, and the MINE(·) procedure is invoked on the base fragment generated from each node. The MINE(·) procedure first updates the index by calling the UPDATE(·) operator; then it generates the expansions of the input fragment by means of the EXPAND(·) operator; finally, it recursively invokes itself on the newly generated fragments.

This solution has the advantage of generating the complete fragment space, but its very high computational complexity is a major limitation. Indeed, even very small sets of real world trees can encode billions of fragments. Explicitly generating and storing all of them implies a computational burden that is not possible to handle within reasonable time and spatial boundaries.

4.5.2 Fragment-size Constrained Generation

Since we are interested in identifying the most relevant fragments, we could consider (4.3) and (4.4) and find out which are the factors that mostly influence the relevance of a fragment.

The only exponential term is the decay factor λ, and in both cases the exponent is a function of the number of nodes included in the fragment. Let us concentrate on the STK (i.e Eq. 4.3), and try to understand how the number of nodes, i.e. the size of a fragment, affects its relevance. For simplicity, we can assume that a fragment appears either zero or one times in each tree, and that for all the support vectors, i.e. $\alpha_i \neq 0$, it holds that $\alpha_i = \alpha$ and $\|t_i\| = T$. The relevance of a fragment can then be expressed as

$$w^{(j)} = \frac{\alpha\lambda^{\frac{s(f_j)}{2}}|c_j|}{T} ,$$

where $c_j = \sum_{i|y_i=1} y_i - \sum_{i|y_i=-1} y_i$ is the difference between the number of positive and negative support vectors in which f_j appears, and is a measure of the correlation of the fragment with one of the two classes. If we consider two fragments f_a and f_b, with $s(f_a) = k$ and $s(f_b) = k + \beta$, and force the equality between the relevance of the two fragments we obtain that:

$$\begin{aligned} w^{(a)} &= w^{(b)} \\ &\Rightarrow \lambda^{k/2}|c_a| = \lambda^{(k+\beta)/2}|c_b| \\ &\Rightarrow |c_b| = \frac{1}{\lambda^{\beta/2}}|c_a| , \end{aligned} \tag{4.15}$$

i.e. if f_b includes β nodes more than f_a, then in order for the two fragments to have the same relevance, $|c_b|$ must be larger than $|c_a|$ by a factor $\frac{1}{\lambda^{\beta/2}}$. To give a practical example, assuming the default value of $\lambda = 0.4$ [Collins and Duffy, 2002], if $\beta = 5$ then $|c_b/c_a| = 9.88$, i.e. if f_b includes 5 more nodes than f_a then it must be approximately ten times more correlated with one of the two classes in order to have the same relevance. If $\beta = 10$ (f_b includes ten more nodes than f_a), then $|c_b/c_a| = 97.66$, i.e. its correlation must be almost a hundred times as much.

These figures suggest that a first direction to explore for reducing the complexity of the mining process is to force the algorithm to consider only fragments which include a given number of nodes, i.e. to avoid generating too large fragments whose relevance would probably be very low. Similarly to what proposed in [Collins and Duffy, 2001], we can control the size of the fragments we generate in two ways, by limiting their maximum depth and the number of nodes included with every expand operation.

The resulting procedure, called SIMPLE_MINER(·), is shown in Algorithm 4.2.

Algorithm 4.2 SIMPLE_MINER($model, maxexp, maxdepth$)

```
global result
main
 result ← MKINDEX()
 for each tree ∈ model
   do { for each node ∈ NODES(tree)
        do MINE(FRAG(node), maxexp, maxdepth, 0)
 return (result)

procedure MINE(frag, depth)
 UPDATE(result, frag)
 if depth < maxdepth
   then { for each fragment ∈ EXPAND(frag, maxexp)
          do MINE(fragment, maxexp, maxdepth, depth + 1)
```

The two extra parameters, *maxexp* and *maxdepth*, control respectively, i) the maximum number of nodes included during each fragment expansion operation, and ii) the maximum depth of the generated fragments. Here, the EXPAND(·) operator (see Section 4.4) is overloaded to expand at most *maxexp* nodes in a fragment.

In the light of the considerations about fragment size in the previous paragraphs, we can assume that in the general case this very simple approach will generate a set including the most relevant fragments. On the other hand, the selection of the *maxexp* and *maxdepth* parameters is critical in order not to exclude possibly heavy larger fragments and not to include irrelevant ones. Furthermore, this kind of approach clearly limits the ability of the kernel to generate larger structured features.

4.5.3 Fragment-number Constrained Generation

An alternative approach would be to directly enforce a limit on the number of fragments that we want to collect. Suppose that we are interested in mining the L most relevant fragments. As shown in the previous paragraphs

and by Equation 4.15, small fragments are generally more relevant than large ones. Therefore, we can keep generating fragments in a small-to-large fashion, with the difference that we collect all the fragments with the same depth from all the input trees.

Let B_L be the set of the L best fragments generated for depth values up to d. Let f be the least relevant of the fragments in B_L. If we expand the fragments in B_L having depth d, we obtain a set of fragments that have depth equal to $d+1$. Let us call this set F. If F contains at least one fragment more relevant than f, then:

- we update the set: $B_L \leftarrow B_L \cup F$;
- we sort it based on the relevance of the fragments;
- we keep the L most relevant fragments. At least one of the L elements of B_L has depth $d+1$;
- we continue iterating by expanding these fragments, and generate the fragments with depth $d+2$.

If no fragment in F is more relevant than f, we can stop. In fact, it is very unlikely that fragments with depth $d+2$ will be more relevant than those generated for depths $1, \ldots, d+1$.

The algorithm implementing this search strategy, which is shown in Algorithm 4.3, is called BOUNDED_MINER($\cdot$). Similarly to Algorithm 4.2, the parameter *maxexp* is used to control the maximum number of expansion produced by EXPAND($\cdot$). The L parameter is the number of fragments that we want to collect. The procedure BASE_FRAGS($\cdot$) generates all the base fragments encoded in a model and stores them in a new fragment index, whereas BEST($\cdot$) sorts the fragments in an index according to their cumulative relevance and discards those ranked lower than L. The **for each** loop

Algorithm 4.3 BOUNDED_MINER($model, maxexp, L$)

```
main
 result ← BASE_FRAGS(model)
 prev ← COPY(result)
 best_pr ← BEST(result, L)
 while true
   do { next ← ∅
        for each f ∈ prev
          do { if f ∈ best_pr
               then { E_f = EXPAND(f, maxexp)
                      for each frag ∈ E_f
                        do { next ← next ∪ {frag}
                             UPDATE(result, frag) }
        best ← BEST(result, L)
        if not CHANGED()
          then break
        prev ← next
        best_pr ← best }
 return (best_pr)

procedure BASE_FRAGS(model)
 result ← MKINDEX()
 for each t ∈ model
   do { for each n ∈ N_t
          do UPDATE(result, FRAG(n)) }
 return (result)
```

generates all the expansions of the best-L fragments collected at the previous step, and stores them in the index. After each iteration, the CHANGED$(\cdot)$ operator verifies if the fragment ranked L is still the same and if its relevance has not changed, which is the stop condition of the loop.

This algorithm is an improvement over SIMPLE_MINER$(\cdot)$, because it can generate fragments of any depth d, assuming that at depth $d-1$ at least one fragment made it to the set of the best-L. On the other hand, it still cannot generate fragments where more than $maxexp$ nodes are expanded at the same time. Furthermore, sorting all the fragments based on their relevance after each iteration is a very costly procedure when the number of indexed fragments is very large.

4.5.4 Greedy Generation

The last algorithm that we present aims at solving these two limitations, by eliminating the need for the $maxexp$ parameter and by including fragments in the index based on their relevance rather than based on their ranking. We want to generate f expansions including $k+1$ new nodes only if at least one of the expansions of k nodes is considered relevant. The value of k is called the *width factor* of the expansion. Concerning the criterion used to decide which fragments are relevant, we need to set a threshold value to compare against. The solution that we adopt is to assess the relevance H of the most relevant fragment in the model, and to consider relevant only the features whose weight is at least $\sigma = H/L$, where L is a parameter of the algorithm. As we will show briefly, the value of H can be linked to the gradient norm after feature selection.

To excatly determine H, we should first generate the whole fragment space, and then calculate the maximum among the fragment weights. Since this approach is unpractical, we need to find an approximator $\widetilde{H}$ for H. We decide to approximate H with the relevance of the best base fragment, i.e.

the heaviest fragment among those generate by the FRAG(·) operator. The choice is motivated as follows.

In Eq. 4.3, we can identify a term $T_i = \alpha_i y_i / \|t_i\|$ that is the same for all the fragments in the tree t_i. For $0 < \lambda \leq 1$, if $f_j \in E_{f_k}$, i.e it is an expansion of f_k, then from our definition of fragment expansion it follows that $\lambda^{\frac{s(f_j)}{2}} < \lambda^{\frac{s(f_k)}{2}}$. It can also be observed that $t_{i,j} \leq t_{i,k}$. Indeed, if $t_{i,k}$ is a subset of $t_{i,j}$, then it will occur at least as many times as its expansion $t_{i,k}$, possibly occurring as a seed fragment for different expansions in other parts of the tree as well. Therefore, for every two fragments $f_{i,j}, f_{i,k}$ coming from the same tree t_i, we can conclude that $x_i^{(j)} < x_i^{(k)}$ $\forall f_{i,j} \in E_{f_{i,k}}$. In other words, for each tree in the model, base fragments are the most relevant. This fact and the discussion about fragment size carried out in 4.5.2 suggest that there is a high probability that $\widetilde{H}$ is the correct approximation fo H. As empirical evidence in support of this conjecture, we report that in all our experiments we have never observed a counterexample.

The value of H can be linked to the fraction of norm that we lose with feature selection, i.e. ρ (see Section 4.3). Let us define the quantity $\sigma = \frac{H}{L}$, where L is a parameter of the algorithm, and assume that we want to select only the fragments f_j so that $w^{(j)} \geq \sigma$, i.e. σ is the relevance of the less relevant fragment that we will consider. Let N be the number of selected features. If we assume that all the selected features are as relevant as the least relevant fragment, i.e. σ, we obtain the following lower bound for $\|w_{in}\|$, i.e. $\|w_{in}\| \leq \sqrt{N\sigma^2} = \sigma\sqrt{N}$. Similarly, if we assume that all the fragments have the same relevance as the best fragment, i.e. H, we can derive an upper bound $\|w_{in}\|$, and conclude that the norm of the gradient after feature selection will be

$$\sigma\sqrt{N} \leq \|w_{in}\| \leq H\sqrt{N} \,. \tag{4.16}$$

This result can be exploited to link the the values of H and L to the gradient

norm after feature selection. In fact,

$$\|\boldsymbol{w}_{in}\| = (1-\rho)\|\boldsymbol{w}\| \geq \sigma\sqrt{N} = \frac{H}{L}\sqrt{N} \tag{4.17}$$

that tells us that norm after feature selection can be expressed as a function of H, L and N.

Algorithm 4.4 GREEDY_MINER($model, L$)

```
main
B ← BASE_FRAGS(model)
H̃ ← REL(BEST(B))
σ ← H̃/L
D_prev ← FILTER(B, σ)
UPDATE(result, D_prev)
while D_prev ≠ ∅
    do { D_next ← ∅
         τ ← 1/*widthfactor*/
         W_prev ← D_prev
         while W_prev ≠ ∅
             do { W_next ← ∅
                  for each f ∈ W_prev
                      do { E_f ← EXPAND(f, τ)
                           F ← FILTER(E_f, σ)
                           if F ≠ ∅
                               then { W_next ← W_next ∪ {f}
                                      D_next ← D_next ∪ F
                                      UPDATE(result, F) }
                  τ ← τ + 1
                  W_prev ← W_next }
         D_prev ← D_next }
return (result)
```

If we combine all these elements, we obtain a new algorithm for the exploration of the fragment space that we call GREEDY_MINER($\cdot$), which is shown in Algorithm 4.4. We generate all the base fragments and calculate the values of $\widetilde{H}$ and σ. We then apply the FILTER($\cdot$) operator to the set F, which removes from the set all the fragments whose cumulative score

is less than σ. To improve the efficiency of the algorithm, the FILTER(·) algorithm also removes all the fragments which appear less than three times, which are very unlikely to be ever observed in the test set. The fragments whose relevance is above the threshold are added to the index *result*, and are considered for further expansion. The inner and outer **while** loops are responsible for growing fragments in width and height, respectively.

In the algorithm, $\mathcal{D}_{prev}$ is the set of fragments expanded at the previous depth level and that have the required relevance, while $\mathcal{D}_{next}$ is the set of fragments that will have to be expanded at the next level. Similarly, $\mathcal{W}_{prev}$ stores the fragment that must be expanded with the current *width factor* τ, that controls the maximum number of nodes to be included in a new fragment. $\mathcal{W}_{next}$ is used to collect the fragments that will be expanded with a larger width factor. A fragment f that generates no relevant expansions for a width factor τ will not be considered for expansions of width $\tau + 1$. Relevant expansions of f, generated for width factors smaller than τ, will still be considered for expansions at the next depth level.

The inner loop terminates when none of the $\mathcal{W}_{prev}$ fragments can generate a relevant expansion for a given width factor, i.e. when $\mathcal{W}_{next}$ ends up being an empty set. Similarly, the outer loop ends if no fragments in $\mathcal{D}_{prev}$ have generated at least one relevant expansion, i.e. when $\mathcal{D}_{next}$ is empty.

Unlike the algorithms defined in the previous sections, GREEDY_MINER(·) works according to the theoretical framework established in Section 4.3, as the criterion used to select the fragment can be linked to the norm of the gradient after feature selection. Furthermore, since there are no hard-coded limitations to the number of expanded nodes or to the maximum depth of generated fragments, in theory it could generate fragments of any size, provided that smaller fragments have sufficient relevance. It is also very efficient, since it implements a very aggressive search strategy that builds larger expansions of a fragment only if the smaller ones are interesting.

This kind of approach is in line with the considerations about fragment size presented in the previous paragraphs, which suggest that an expansion of a fragment is very unlikely to be more relevant than the fragment itself.

4.6 Fragment Indexing

One of the critical issues for fragment mining is the definition of a data structure that can store compactly and efficiently a large number of fragments.

At first, an attempt was made to employ a Direct Acyclic Graph (DAG) by referring to the algorithms described in [Aiolli et al., 2006] where the structure is used to store compactly all the trees a TK model. This kind of approach has soon shown some limitations in three main areas:

Memory: every subtree (i.e. a node along with all its descendants, up to the leaves) is only represented once. This property makes it a compact structure for the representation of subtrees, while it is not as convenient in the case of arbitrary tree subsets;

Insertion: insertion is a costly operation, as it requires sorting the nodes of each fragment in reverse fan-out order;

Lookup: searching for the fragments encoded in a tree would require to generate all the fragments in the tree, which is an operation with exponential complexity.

The first two problems, that affect time and space complexity of the training stages of the process, would not be a real concern: the amount of available memory can always be increased, and the additional time required to populate the index would be largely shaded by the learning time of the TK function. On the other hand, if we want to realize a fast, linear classifier, the

efficiency of the decoding process is absolutely critical. In this respect, it is necessary to devise a data structure whose decoding performance degrades nicely for growing numbers of indexed fragments.

4.6.1 The *FragTree* Data Structure

The adopted solution is called a *FragTree*. Its design is based on the following idea: the nodes in a graph can be used to describe the set of expansions that define a fragment, starting from its root. This property can be exploited when *decoding* a tree t, i.e. when querying the index to retrieve the list of fragments contained in t. Instead of generating all the fragments in t and trying to match them in the index[2], we can apply the expansions in the index to the nodes of t, and check if they result in some indexed fragment.

In a FragTree, each path in the graph can then be univocally associated with a fragment. As an example, consider the fragment

(A (B) (C (d))).

The fragment can be obtained by applying the following algorithm:

- ☐ The root of the fragment is labeled A;
- ◯ Expand the first node observed at the previous level by generating two children. This operation can be represented with the pair $(0, 2)$, where the first number is the relative offset of the expanded node, and the second is the number of nodes resulting from the expansion;
- ☐ The two resulting nodes are B and C;
- ◯ Expand the second node at the previous level (i.e., C) and generate one child. Again, this operation can be described with the pair $(1, 1)$: expand the second node (offset $= 1$) and obtain one child;

[2]That would have exponential complexity, assuming lookup time of a fragment in the index to be constant.

- The resulting node is labeled d.

The sequence of these operations, i.e. $[A] \rightarrow (0:2) \rightarrow [B, C] \rightarrow (1:1) \rightarrow [d]$, completely describes the fragment and can be represented as a path in a graph whose nodes are of two kinds:

- nodes that list the label sequences encountered at some level in a tree, and
- nodes that describe node expansion operations.

In a FragTree, these different kinds of information are accounted for by two different classes of nodes, called *label* and *production* nodes, respectively. Together, label and production nodes can be used to describe the structure of a fragment without ambiguity.

As an example, consider the FragTree in Figure 4.6, that describes all the PTK fragments of the tree *(A (B (c) (d)) (C (h)))* rooted in *A*. Here, production nodes are represented as circles, whereas label nodes are represented as squared blocks and are given a unique numeric identifier (id). In a label node, a special character ('#', in the example) is used to separate node labels originating from different parents. Each path from the root of the FragTree to any label node identifies the surface form of an observed fragment, i.e. it is possible to establish a bijective correspondence between nodes in the FragTree and fragments. For example, the path from the root to the node with $id = 8$ describes the fragment *(A (B) (C))*, whereas the nodes with $id = 11$ and $id = 12$ correspond to the fragments *(A (B (c)) (C (h)))* and *(A (B (c)) (C (k)))*, respectively. This kind of structure can be effectively exploited to reduce decoding complexity in the general case, as will be shown shortly.

Even if they are not shown in figure, label nodes also store a pointer to a so-called *data* node that collects statistics about the occurrencies of the fragment in the data: the number of positive and negative support

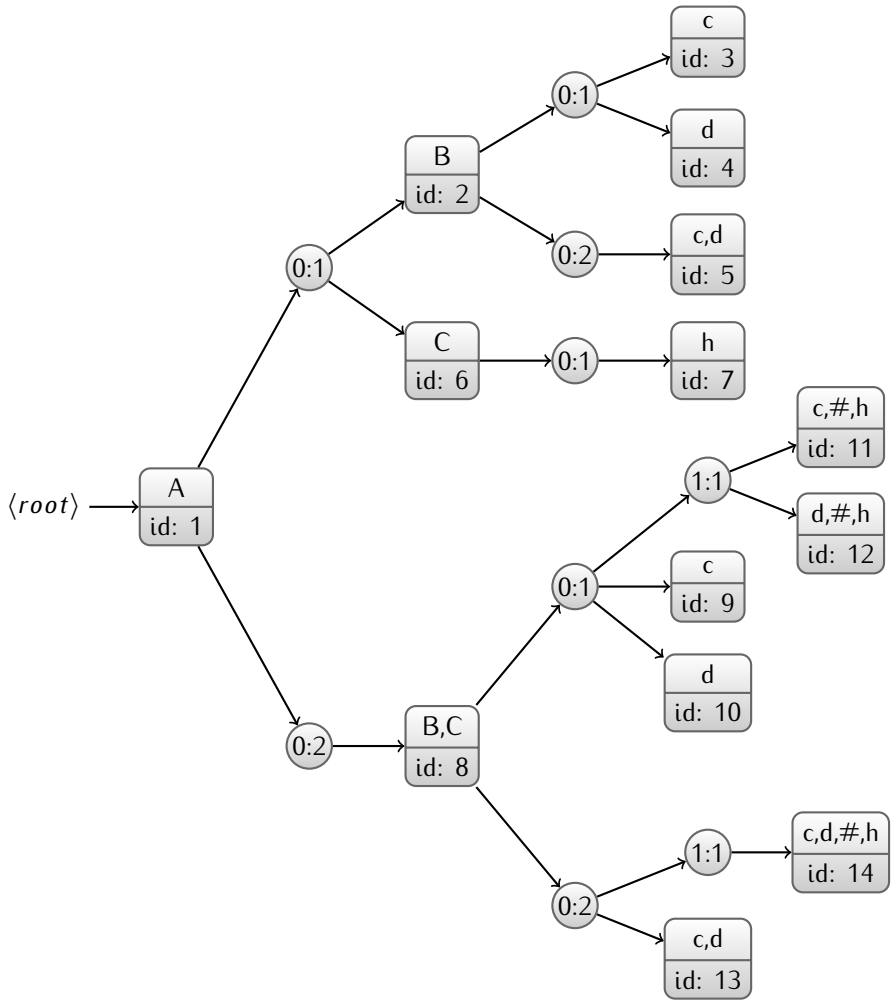


Figure 4.6: **Exemplification of a FragTree** - A FragTree encoding all the PTK fragments in the tree $(A(B(c)(d))(C(h)))$ which are rooted in A.

vectors containing the fragment, and its cumulative relevance. When a fragment is added to the FragTree, if the node describing the fragment already exists (i.e. the same fragment has already been observed before), then the statistics about the new fragment are merged with the existing data, otherwise a new data node is created.

Algorithm 4.5 ENCODE($frag, fragData, idxRoot$)

```
main
 nodes ← [ROOT(frag)]
 PUT(idxRoot, nodes, fragData)

procedure PUT(idxNode, nodes, fragData)
 idxNode ← GET_LAB(idxNode, nodes)
 temp ← idxNode
 next ← [ ]
 for each offset ∈ [0, ..., LEN(nodes) − 1]
   do { children ← CHILDREN(nodes[offset])
        l ← LEN(children)
        if l > 0
          then { idxNode ← GET_PROD(idxNode, offset, l)
                 APPEND(next, children)
 if LEN(next) = 0
   then UPDATE(idxNode, fragData)
   else PUT(idxNode, next, fragData)
```

4.6.2 Tree Encoding

Algorithm 4.5 lists the pseudocode of the ENCODE(·) operation, i.e. the operation that creates the path describing the fragment $frag$ in the FragTree rooted in $idxRoot$. Here, $fragData$ are the statistics associated with $frag$. The description of the algorithm requires the definition of several functions and operators:

[ ] (empty square brackets) return an empty sequence;

x[**y**] accesses the y-th element of the list x;

x{**y**} accesses the element associated with the key y in the hashmap x;

LEN(x) returns the length of the sequence x;

APPEND(x, y) appends all the elements of the list y to the end of the list x;

ROOT(x) returns the root of the tree (or fragment) x;

CHILDREN(x) returns the ordered sequence of children of the tree node x;

GET_LAB(x, y) returns the label node outgoing from x that encodes the node sequence y. If no such node exists, it is created and returned;

GET_PROD(x, y, z) returns the production node outgoing from x that encodes the production rule $y : z$. If no such node exists, it is created;

UPDATE(x, y) uses the statistics in y to update the data node linked by the FragTree node x.

At the beginning of the main procedure, a list containing the root of the fragment is created. Then, the PUT$(\cdot)$ procedure is invoked. The procedure recursively invokes itself to generate all the label nodes that describe the nodes in each level of the fragment. During each recursion, the procedure iteratively builds the production nodes that describe the transition between the nodes encountered in two consecutive levels. PUT$(\cdot)$ first checks if a label node describing the incoming set of nodes *nodes* is already present. If it is not, it is created. The *idxNode* pointer is set to point to this label node. The **for each** loop calculates all the expansion operations that must be sequentially applied to obtain the sequence of nodes at the next level in the tree. For each expansion, a corresponding production node is created, and *idxNode* is updated to point to it. The sequence *next*, i.e. the sequence of nodes at the next level in the tree, is built incrementally. If a node does not have any children, then the iteration is skipped and the value of *idxNode* is unaffected. At the end of the loop, if all the nodes in *nodes* have no children, then *next* will be empty. If this is the case, then the path from the root to *idxNode* completely describes the fragment. Therefore, the data node associated with *idxNode* can be updated with the statistics about the fragment. Otherwise, the PUT$(\cdot)$ procedure is invoked recursively on the FragTree node pointed by *idxNode* and the set of nodes at the next level in fragment *next*.

One fragment $\iff$ One representation. It should be observed that the set of expansions that describe a fragment is generally not unique. As an example, the fragment *(A (B (b)) (C (c)))* can be obtained from the fragment *(A (B) (C))* in two distinct ways: by first expanding the node *B* and then *C*, or vice-versa. In a FragTree, this problem is resolved by adopting a strict left-to-right policy for the description of a node's children in the fragment, which in Algorithm 4.5 is enforced by the **for each** loop. This is enough to ensure that in a FragTree there cannot be more than one representation of the same fragment.

4.6.3 Tree Decoding

Tree decoding is the process by which an input tree is actually represented as a vector, based on the relevant fragments stored in an index. The most naive approach to tree decoding would require to generate all the fragments encoded in a tree and look them up in a dictionary. This solution was employed in a very early version of our model, but due to its exponential complexity it was later discarded in favor of a more efficient strategy that relies on the information stored in a FragTree.

The underlying idea is to combine the information stored in the label and production nodes so that only the fragments that are actually stored in the FragTree are generated. Each time a label node of the FragTree is traversed, the production rules encoded by outgoing production nodes are applied to the sequence of input nodes, i.e. the nodes described by the label node. Only the productions that are *compatible* with the input set of nodes are applied. For example, if one of the input nodes has only one child and the production rule requires to expand two of its children, then the rule is incompatible with the tree. An incompatible production rule encoded by the path *P* allows to cut the search space by discarding all the subsequent branches of the FragTree on that path: if the fragment represented by *P* is not present in the tree, then all its expansions are not either.

Each chain of compatible production rules outgoing from a label node

results in a search direction, and the production rules in each chain are applied in sequence. At each step, if a label node matching the sequence of labels generated by the expansions is found, then information about the matching fragment is added to the search result.

As an example, consider decoding the tree $t = (A(B)(D))$ with the FragTree shown in Figure 4.6. First, a label node matching the root of the tree is searched within the children of the FragTree root, and it is found (id:1). Since node A has 2 children, both production rules outgoing from the label node are compatible, and result in two search paths. If we follow the top path, we are required to expand one of the children of A in t. This results in two fragments: $(A(B))$ and $(A(D))$. The first fragment is matched (id:2) and added to the result, whereas the second does not exist. This search path is now exhausted, since neither $(A(B))$ nor $(A(D))$ can be further expanded. If we follow the bottom search path, we are required to include both children of A in t, and obtain the fragment $(A(B)(D))$. Since there is no label node matching the label set $[B, D]$, search on this path is terminated.

Algorithm 4.6 lists the pseudocode of the DECODE$(\cdot)$ operation, which realizes the decoding process. Explaining the algorithm requires the definition of the following operators, in addition to those already introduced in Section 4.6.2:

NODES(x) returns the set of nodes of tree x, i.e. N_x;

IS_LAB_NODE(x) is verified if the FragTree node x is a label node;

LAB_NODES(x) returns the set of label nodes emanating from node x;

PROD_NODES(x) returns the set of production nodes emanating from the node x;

FOUND(x) updates the results of the decoding process with information

Algorithm 4.6 DECODE($tree, idxRoot$)

```
main
 for each node ∈ NODES(tree)
      do { nodes ← [node]
           labs = LAB_NODES(idxRoot)
           if nodes ∈ labs
             then { idxNode ← labs{nodes}
                    LOOKUP(idxNode, nodes, [ ])
procedure LOOKUP(idxNode, curLevel, nextLevel)
 if IS_LAB_NODE(idxNode)
   then FOUND(idxNode)
   else { labs ← LAB_NODES(idxNode)
          if nextLevel ∈ labs
            then LOOKUP(labs{nextLevel}, nextLevel, [ ])
 for each prod ∈ PROD_NODES(idxNode)
      do { if COMPATIBLE(prod, curLevel)
             then { nextLabCombs ← APPLY(prod, curLevel)
                    for each nextLabs ∈ nextLabCombs
                      do { next ← COPY(nextLevel)
                           APPEND(next, nextLabs)
                           LOOKUP(prod, curLevel, next)
```

about the fragment x appearing in the input tree. Decoding keeps track of how many times each fragment is found within a tree;

COMPATIBLE(x, y) checks whether the production rule encoded by x is compatible with the node set y. For example, if the first element of y were a node with only one child, a production rule like $(0 : 2)$ would not be compatible, as it would require to expand two children of a node having just one child. Conversely, $(0 : 1)$ would be a compatible production rule;

APPLY(x, y) applies the production rule x to the node set y, and returns all the compatible sequences of expandable nodes. This operation is combinatorial. Assume that $x = (0 : 1)$ and $y = [A]$, where A is the root of the subtree *(A (B) (C))*. The production rule requires to expand one

child of A. Since A has 2 children, the inclusion of one of its children can result either in $(A(B))$ or $(A(C))$. Therefore, APPLY$((0:1),[A])$ would return $[B]$ and $[C]$;

COPY(x) returns a copy of the list x.

The main procedure of the algorithm just traverses all the nodes in the tree, wraps them in one-element lists and invokes the LOOKUP$(\cdot)$ procedure on the top-level label nodes with a matching label, if any. The LOOKUP$(\cdot)$ procedure, which actually implements the exploration of the FragTree, requires three parameters. The first parameter, $idxNode$ is the FragTree node currently being investigated. The second parameter, $curLevel$, is a sequence of nodes. If $idxNode$ is a label node, then $curLevel$ is a sequence of nodes whose labels match the sequence encoded by $idxNode$. If $idxNode$ is a production node, then $curLevel$ is the last matched node sequence, i.e. the sequence of nodes matching the label of the first label node p on the path from $idxNode$ to the root of the index. Let P be the path from p to $idxNode$. The third parameter, $nextLevel$, is the sequence of nodes obtained by applying to $curLevel$ the expansions encoded by the production nodes on P. If the length P is zero, i.e. $idxNode$ is a label node, then it follows that $nextLevel = [\,]$.

When the procedure is invoked, if $idxNode$ is a label node, the FOUND$(\cdot)$ operator is used to add the associated fragment to the result. If $idxNode$ is not a label node, then it is a production node and it may have outgoing label nodes; in this case, if a label node matching $nextLevel$ is found, a new search is started from there. In both cases, the node might have outgoing production nodes, and the search continues on all the branches that describe a compatible production.

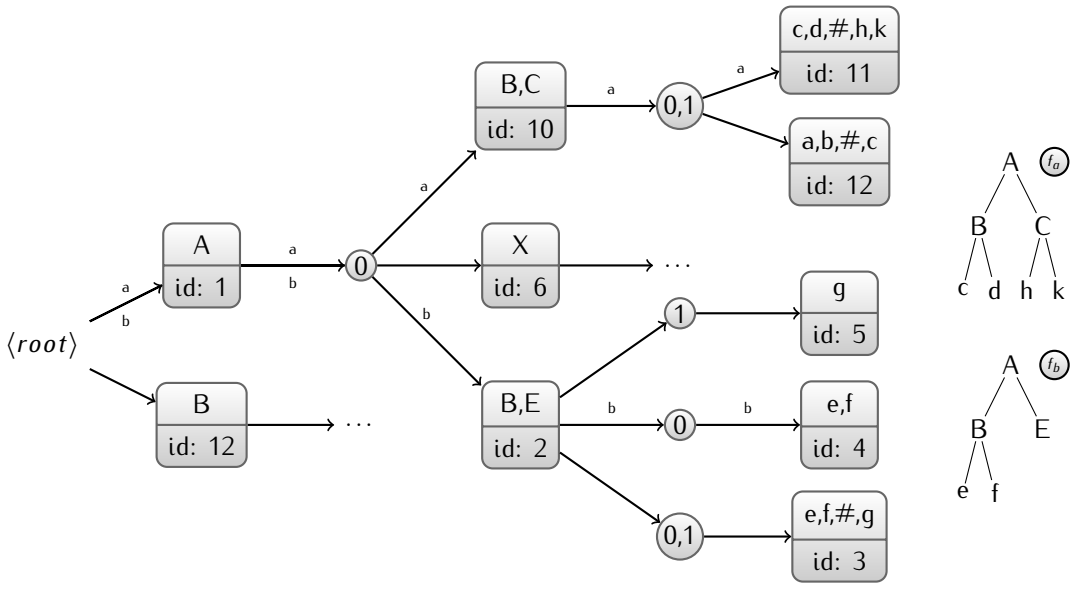


Figure 4.7: **Examplification of an STKTree.**

4.6.4 *STKTree*: a Simplified FragTree for the STK

The FragTree is general enough to represent all the fragments generated by the PTK. For kernel functions that generate a more constrained fragment space, it is possible to re-engineer the data structure and make it more compact and fit for the task.

The *STKTree* is a specialized FragTree that is enough to represent the set of operations necessary to describe STK fragments. The simplification moves along the following lines:

- Expanding a node automatically implies the inclusion of all its children. As a consequence, production nodes do not need to represent the cardinality of each node expansion operation, but only the offset of expanded nodes;
- All the expansions performed at a given depth can be compacted into a single production node, listing the offsets of all the nodes expanded.

An example of STKIndex is shown in Figure 4.7. A production node la-

beled (0, 1) means that the first and the second nodes listed in the incoming label node should be expanded by including all their children in the fragment. As an example, consider all the nodes on the path labeled *a*, i.e. the path connecting the root of the index to the node labeled *c,d,#,h,k* (id: 11), on the top right. From left to right, the path can be read as:

1. The root of the fragment is labeled *A*;
2. Expand the first node (offset 0) at the previous level;
3. The expansion produces two nodes, *B* and *C*;
4. Expand the first and the second (offsets 0 and 1) nodes at the previous level;
5. The expansion produces four nodes: *c* and *d*, which descend from the first node expanded (i.e. *B*), and *h* and *k*, which descend from the second.

This path encodes the fragment labeled as f_a in Figure 4.7. Similarly, the path whose edges are labeled *b* encodes the fragment f_b.

Encoding and decoding for the STKTree are basically the same as for the general case. However, we can exploit the simplified structure of the graphs to streamline the algorithms and make them more compact and efficient. Algorithm 4.7 shows a tailored version of the ENCODE(·) procedure for the STKTree. In this case, the main difference is that we do not generate a new production node for each node expansion, but just one production node that encodes all the expansions that occur at a given depth. The **for each** loop builds the sequence of nodes that constitute the next level in the tree and the corresponding list of offsets. If these sequences are not empty, then a new production node is created and the PUT(·) method is invoked with the new set of nodes, otherwise the statistics about the fragment are updated via the UPDATE(·) operator.

Algorithm 4.7 ENCODE_STK($frag, fragData, idxRoot$)

```
main
 nodes ← [ROOT(frag)]
 PUT(idxRoot, nodes, fragData)

procedure PUT(idxNode, nodes, fragData)
 idxNode ← GET_LAB(idxNode, nodes)
 next ← [ ]
 productions ← [ ]
 for each offset ∈ [0, ..., LEN(nodes) − 1]
   do { children ← CHILDREN(nodes[offset])
        l ← LEN(children)
        if l > 0
          then { APPEND(next, children)
                 APPEND(productions, offset)
 if LEN(next) = 0
   then UPDATE(idxNode, fragData)
   else { idxNode ← GET_PROD(productions)
          PUT(idxNode, next, fragData)
```

Algorithm 4.8 lists the pseudocode for tree decoding in a STKTree. The first difference is that only an internal node of a tree can be the root of a fragment. Therefore, only the internal nodes of the tree are considered in the main procedure. Concerning the LOOKUP(·) procedure, the main difference is that the APPLY(·) operation in this case is not combinatorial, i.e. there is only one possible expansion for any set of nodes, since every time all their children must be included. If *idxNode* is a label node, then we identify the compatible production nodes and initiate as many search paths, if any. If *idxNode* is a production node, we search for an outgoing label node that matches the expected labels (at the next level) and, if found, we continue searching in the direction.

Algorithm 4.8 DECODE_STK($tree, idxRoot$)

```
main
 for each node ∈ INTERNAL_NODES(tree)
      do { nodes ← [node]
           labs = LAB_NODES(idxRoot)
           if nodes ∈ labs
             then { idxNode ← labs{nodes}
                    LOOKUP(idxNode, nodes, [ ]) }
procedure LOOKUP(idxNode, curLevel, nextLevel)
 if IS_LAB_NODE(idxNode)
   then { FOUND(idxNode)
          for each prod ∈ PROD_NODES(idxNode)
            do { if COMPATIBLE(prod, curLevel)
                   then { next ← APPLY(prod, curLevel)
                          LOOKUP(prod, curLevel, next) } }
   else { labs ← LAB_NODES(idxNode)
          if nextLevel ∈ labs
            then LOOKUP(labs{nextLevel}, nextLevel, [ ]) }
```

4.6.5 Learning Architectures and Decoding

With respect to the alternative architectures described in Section 4.1, we should observe that the results of the decoding process are employed differently in the MLin and the LOpt (or Split) architectures.

In the first case (Sec. 4.1.1), since we want to carry out classification using the weights estimated by the SVM, we want the linear representation of the space to be as close as possible to the original frament space. We evaluate Eq. 4.3 (for STK) or Eq. 4.4 (for PTK) for every fragment identified within a tree t_i, and build the corresponding vector $\boldsymbol{x}_i$ using these values. If we imagine to project the whole fragment space, it is equivalent to simply assume that relevant fragments have the correct weight, whereas the weight of irrelevant fragments is set to zero.

Concerning LOpt (Sec. 4.1.2), since the linearized data will be used for a new optimization problem in the projected space, we are only interested

in listing which fragments appear in a tree. In this case, it is sufficient to build the vectors by encoding the number of occurrences of each fragment in the original tree, as exemplified in Figure 2.4.

Chapter 5

Experimental Evaluation

To confirm our theoretical results and to demonstrate the performance of our algorithms, we ran experiments on three very different NLP benchmarks that allow us to stress and evaluate different aspects of the problem. The three tasks are Question Classification, Relation Extraction and Semantic Role Labeling.

In the remainder of this chapter, 5.1 describes the three tasks and the data sets that we employed.

In Section 5.2 we provide evidence that confirms the theoretical framework outlined in Section 4.3. We do so by means of the Model Linearization architecture (MLin, Sec. 4.1.1).

In Section 5.3 we compare the accuracy of the Linear Space Optimization (LOpt, Sec. 4.1.2) model against non-linearized STK classifiers, and show that our feature selection technique achieves comparable results on all the benchmarks.

In Section 5.4 we demonstrate the efficiency of the greedy mining algorithm (Sec. 4.5.4) and the STKTree data structure (Sec. 4.6.4). We empirically analyze time complexity of the kernel space mining (KSM) and linear space generation (LSG) stages of the process (Sec. 4.1).

In Section 5.5 we show how classification in the linear space can be far

more efficient than in the original STK space, and that, for large datasets such as SRL, the Split architecture (Sec. 4.1.3) can be used to improve the efficiency of the learning process, without compromising the final accuracy.

Finally, in Section 5.6 we discuss the capability of the feature selection technique to make explicit the most relevant fragments for each class.

Experimental setup. All the experiments were run on a machine equipped with 4 Intel® Xeon® CPUs clocked at 1.6 GHz and 4 GB of RAM running on a Linux 2.6.9 kernel. As a supervised learning framework we used SVM-Light-TK [1], which extends Thorsten's SVM-Light optimizer [Joachims, 2000] with support for tree kernel functions. The package implements the efficient STK and PTK algorithms described in [Moschitti, 2006a]. In all the experiments, the STK is normalized and evaluated with the default decay factor $\lambda = 0.4$. During LSL, the classifier is trained using a linear kernel.

5.1 Tasks and Datasets

This section describes the tasks that we tackled with our feature extraction framework and provides details about the composition of the datasets.

5.1.1 Question Classification

Given a question, the QC task consists of selecting the most appropriate expected answer type from a given set of possibilities. We adopt the question taxonomy known as *coarse grained*, which has been described in [Zhang and Lee, 2003] and [Li and Roth, 2006], including six non overlapping classes: Abbreviations (ABBR), Descriptions (DESC, e.g. definitions or explanations), Entity (ENTY, e.g. animal, body or color), Human (HUM, e.g. group or individual), Location (LOC, e.g. cities or countries) and Numeric (NUM, e.g. amounts or dates).

We employ TREC 10 QA data [Voorhees, 2001], consisting of 6,000 questions. For each question, we generate the full parse of the sentence and

[1] http://disi.unitn.it/~moschitt/Tree-Kernel.htm

	ABBR	DESC	ENTY	HUM	LOC	NUM
TR_+	85	1,192	1,223	1,201	834	933
TR_-	5,383	4,276	4,245	4,267	4,634	4,535
TE_+	11	106	122	91	80	75
TE_-	474	379	363	394	405	410

Table 5.1: **Question classification dataset** - Number of positive and negative examples in the training (TR) and test (TE) set for the binary classifiers of the QC task.

	1	2	3	4	5	6	7
TR_+	982	272	1,284	160	115	433	217
TR_-	33,555	34,265	33,253	34,377	34,422	34,104	34,320
TE_+	225	91	331	51	19	91	58
TE_-	8,409	8,543	8,303	8,583	8,615	8,543	8,576

Table 5.2: **Relation extraction dataset** - Number of positive and negative examples in the training (TR) and test (TE) set for the binary classifiers of the RE task.

used it to train our models. The automatic parses are obtained with the Stanford parser [Klein and Manning, 2003]. [2] We actually have only 5,953 sentences in our data set due to parsing issues with a few of them.

Since we observe an uneven distribution of positive and negative examples in the standard split of the dataset, we use a balanced random selection to generate our training/test split, containing respectively 5,468 and 485 sentences.Table 5.1 shows, for each class, the number of positive and negative examples in the training and test splits.

The classifiers are arranged in a one vs. all configuration, where each sentence is a positive example for one of the six classes, and negative for the other five. The accuracy of our models is measured as the percentage of correct class assignments.

5.1.2 Relation Extraction

For the relation extraction task we used the newswire and broadcast news domain of the ACE 2004 English corpus [Doddington et al., 2004], consisting of 348 documents. The dataset defines seven types of relations between entity pairs: Physical, Person/Social, Employment/Membership/Subsidiary, Agent-Artifact, Person/Organization Affiliation, Geo-Political Entity Affiliation, and Discourse. In the remainder, these relations will be associated with class labels 1 through 7. The class label 0 is assigned to those examples where none of the seven relations is instantiated. Classifiers are arranged in a One vs. All fashion, where a positive instance for a class is negative for all the others.

The dataset used in the experiments is the same used by [Nguyen et al., 2009]. We consider the entity mentions annotated in each sentence, and for each sentence we generate as many examples as the number of mention pairs compatible with some class definition. For example, the sentence "Bush commented on the agreement between Yahoo! and Microsoft" would generate two examples for Person/Organization, one for the pair "Bush/Yahoo!" and one for "Bush/Microsoft". In this case, both of them would be positive examples for the class labeled 0, i.e. no relation. Each example is parsed with the Stanford parser [Klein and Manning, 2003], where entities are annotated as synthetic nodes which are added between the node dominating the mention and its parent. The result of this process includes 43,171 examples, 34,537 of which are used for training and 8,632 for testing. Table 5.2 shows, for each class, the number of positive and negative examples in the training and test splits.

The accuracy of the multiclass classifier is evaluated as the micro-averaged precision, recall and F_1 measure of the seven relation classifiers, i.e. 1-7. If all the classifiers classify an example as a negative instance,

[2] http://nlp.stanford.edu/software/lex-parser.shtml.

	A0	A1	A2	A3	A4	A5
TR_+	60,707	81,511	19,423	3,313	2,651	68
TR_-	106,966	86,162	148,250	164,360	165,022	167,605
TE_+	2,007	2,791	624	101	60	2
TE_-	3,533	2,749	4,916	5,439	5,480	5,538

Table 5.3: **Semantic role labeling dataset** - Number of positive and negative examples in the training (TR) and test (TE) set for the binary classifiers of the SRL task.

then it is considered as labeled 0.

5.1.3 Semantic Role Labeling

The Semantic Role Labeling (SRL) task requires to identify word sequences that play a semantic role with respect to some predicate word w. As a benchmark, we use PropBank annotations [Palmer et al., 2005] and automatic Charniak parse trees [Charniak, 2000] as provided for the CoNLL 2005 evaluation campaign [Carreras and Màrquez, 2005]. We start from trees annotated with gold information about the position of arguments, and focus on the multi-classification problem of assigning the correct role to each argument. In particular, we will consider argument nodes corresponding to any of the six core roles defined in PropBank, i.e. A0, A1, ..., A5.

We build a dataset by extracting AST_m structured features [Moschitti et al., 2008] for each predicate/argument pair. An AST_m is defined as the minimal tree that covers all and only the words of the predicate node p (i.e. the node that dominates the predicate word) and the argument node a (i.e. the node that dominates the argument words). In the AST_m, p and a are marked so that they can be distinguished from the other nodes. As a training set, we collected all the AST_ms for core roles from all the available training sections, i.e. 2 through 21, for a total of 167,673 examples. For testing, we used 5,540 AST_m similarly extracted from section 24.

Table 5.3 shows, for each class, the number of positive and negative

examples in the training and test splits.

5.2 Fragments and Gradient Norm

The basic assumption of our feature selection technique is that in a TK space it is possible to select out an exponential number of fragments, while retaining a large fraction of the norm of the original gradient, as theorized in Lemma 4.3.4. If we can preserve the gradient norm, we can assume that removing a great number of features has not altered the geometry of the separation problem significantly, i.e. the margin, as shown by Lemma 4.3.1. Therefore, it should be possible to use the hyperplane characterized by $\boldsymbol{w}_{in}$, i.e. the projection of the original hyperplane in the low-dimensional space, to classify the projected data with some degree of accuracy.

To this aim, we can combine the greedy miner algorithm described in Algorithm 4.4 (for aggressive feature selection) with the MLin architecture presented in Section 4.1.1 (for the evaluation of the linearized models). It should be recalled that in this configuration we re-use the weights estimated in the TK space to classify examples in the lower-dimensional linear space.

Figures 5.1, 5.2 and 5.3 show how the gradient norm changes according to the number of fragments that we mine, i.e. according to different values of the threshold factor parameter value L, for each of the QC, RE and SRL classes, respectively. The reader should recall that, in Section 4.3, we used $(1-\rho)$ to designate the fraction of norm that is retained after feature selection.

As we can see, all the plots show a very similar behaviour. By increasing the number of fragments that we mine, first $(1-\rho)$ increases very fast, approximately with the logarithm of the number of fragments, then it stabilizes and remains mostly constant even if we keep adding fragments. This point is approximately between 10^3 and $10^{3.5}$ fragments for QC classes, between

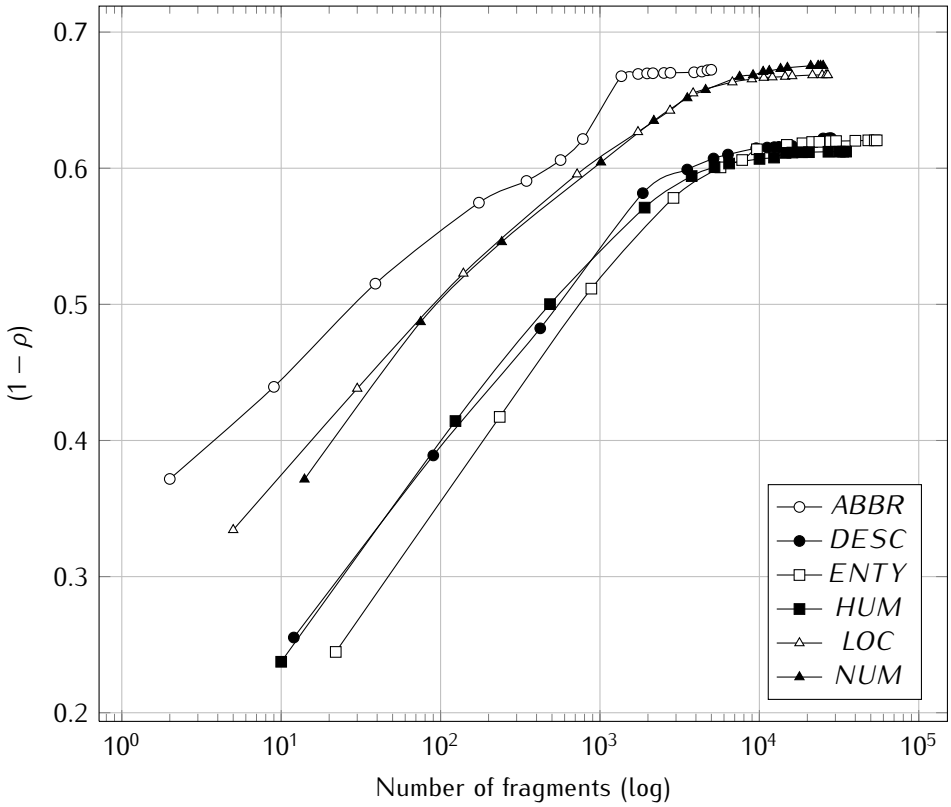


Figure 5.1: **QC: included fragments vs. norm.**

$10^{3.5}$ and 10^4 for RE classes, and between 10^4 and 10^5 for SRL classes.

The plateau point of each class shows some correlation with the number of training instances for each class, and especially with the number of positive points, which is then correlated with the number of support vectors retained in the model. In fact, in Figure 5.1 we can see that it takes more fragments to reach the plateau for DESC, ENTY and HUM, which have approximately 1,000 positive examples in their training sets, than for ABBR that only have 85 positive training points (see Table 5.1). The same behaviour can be observed in Figure 5.2 (classes 1 and 3 vs. 4 and 5) and in Figure 5.3 (classes A0, A1 and A2 vs. A5). As we can expect, classes having very few positive training points, for which we can generally learn compact models with a small number of SVs, require a smaller number of fragments to reach the plateau. As an example, with approximately 150–200

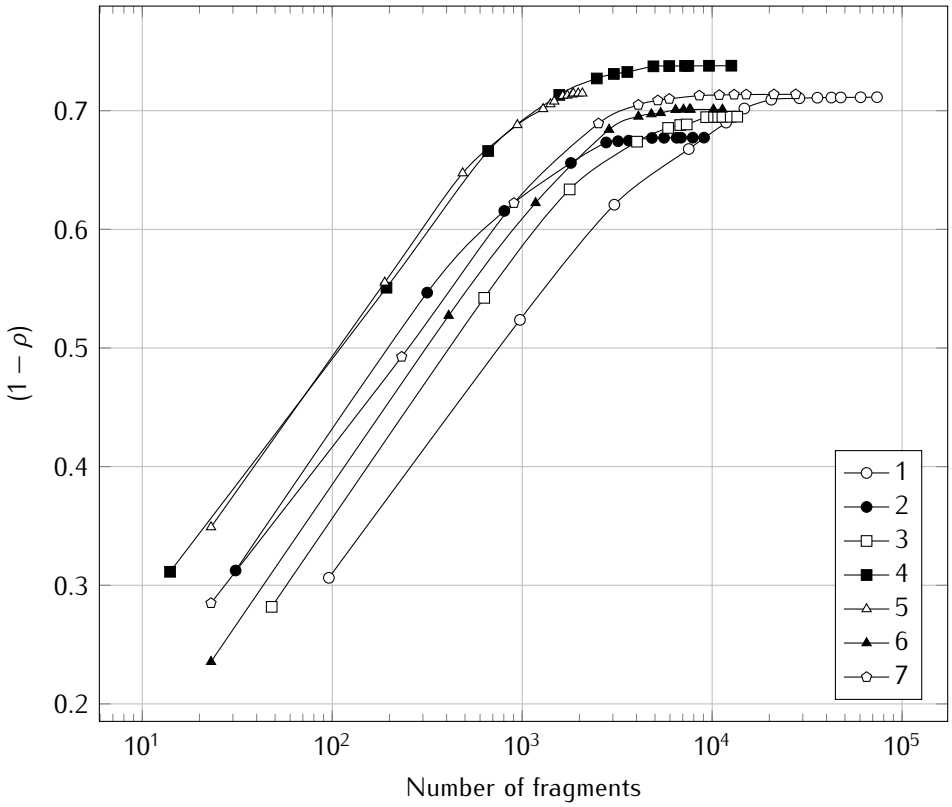


Figure 5.2: **RE: percentage of norm retained after feature selection $(1-\rho)$ vs. number of fragments.**

fragments we can build a gradient $\boldsymbol{w}_{in}$ for the A5 classifier in Figure 5.3 whose norm is approximately 85% of the norm of the original gradient $\boldsymbol{w}$.

Most interestingly, these plots confirm the theoretical result of Lemma 4.3.1, showing that it is actually possible to discard an exponential number of features producing only a limited effect on the gradient norm. To quantify the aggressiveness of the feature selection with an example, the reader should consider that the fragment space encoded in the A1 STK model consists of approximately $10^{25} \sim 10^{35}$ fragments. The 140 most relevant fragments that we select account for approximately 26% of the gradient norm, and with less than 2,000 fragments only half of the norm is lost. By including between 50 and 60 thousand fragments, less than 1/4 of the norm is lost.

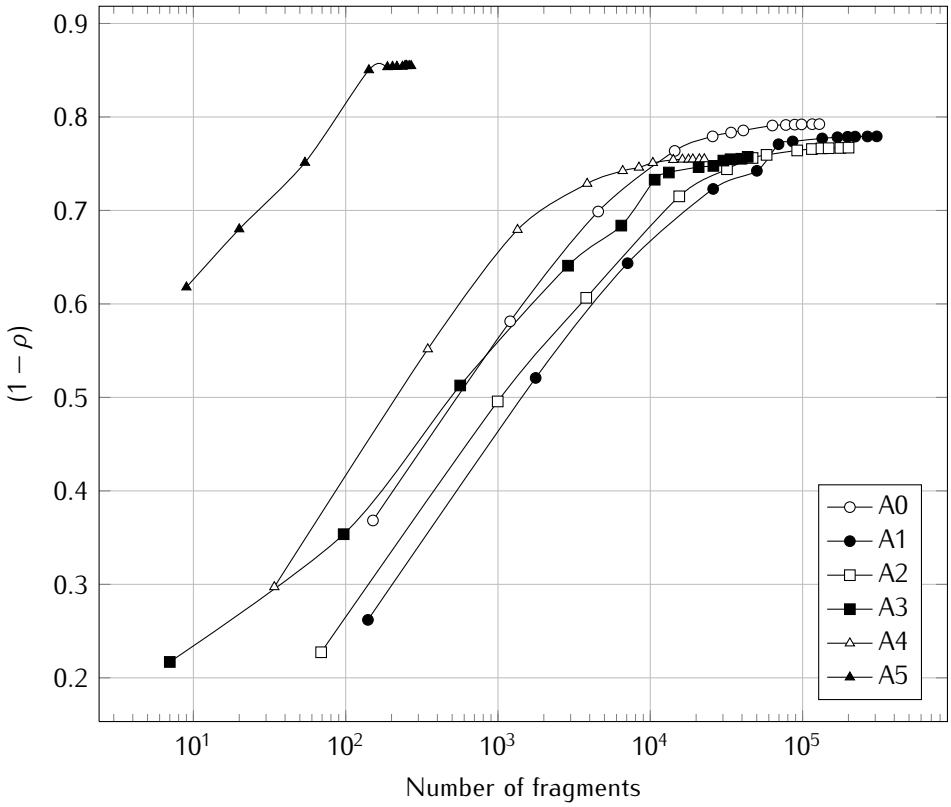


Figure 5.3: **SRL: percentage of norm retained after feature selection $(1-\rho)$ vs. number of fragments.**

The next three figures plot the F_1 measure of MLin with respect to $(1-\rho)$ for each class of the three tasks, respectively QC (Figure 5.4), RE (Figure 5.5) and SRL (Figure 5.6). In spite of per-class and per-task differences, we can generally observe that the accuracy of the linearized models is near zero for lower values of $(1-\rho)$, e.g. $0.2 \sim 0.4$, and tends to grow very fast and reach results that are comparable with non-linearized classifiers, represented as dashed lines. With an appropriate number of fragments, almost every MLin classifier can match the accuracy of STK.

By comparing the three groups of plots, we can observe that the trend of accuracy is especially regular for those classifiers that observed a large number of positive examples during training, e.g. DESC and ENTY for QC, 1 and 3 for RE, and A0, A1 and A2 for SRL. Since the features that

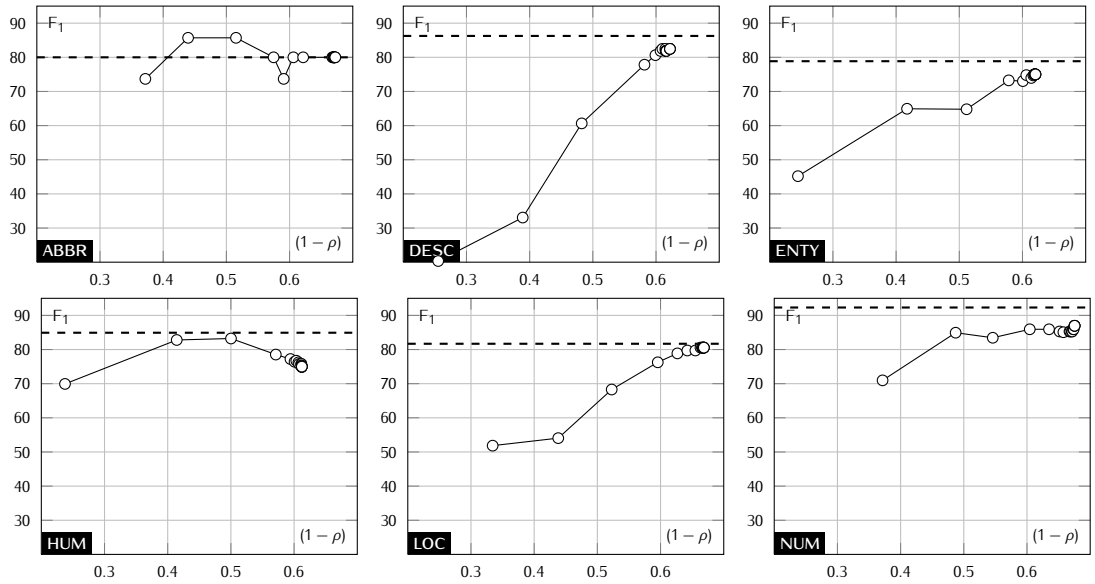


Figure 5.4: **QC: Accuracy of the MLin model vs. norm after feature selection** - A dashed line marks the accuracy of the corresponding STK (i.e. non-linearized) classifier.

we consider while calculating $\|\boldsymbol{w}_{in}\|$ are those with the highest cumulative relevance, potentially relevant fragments are not very likely to be selected if they do not appear with sufficient frequency. This leads to irregular trends, due to bad generalization performance, in classes like A5 (SRL) or ABBR (QC).

The high accuracy values that we can achieve with MLin clearly confirms the findings of Lemma 4.3.1. They show that the less we affect the gradient norm, the more we can retain the geometric properties of the separating hyperplane, and preserve the large margin estimated by the SVM. In fact, we can reuse the weights estimated by the SVM in a space with an exponentially larger number of features and still achieve good classification accuracy.

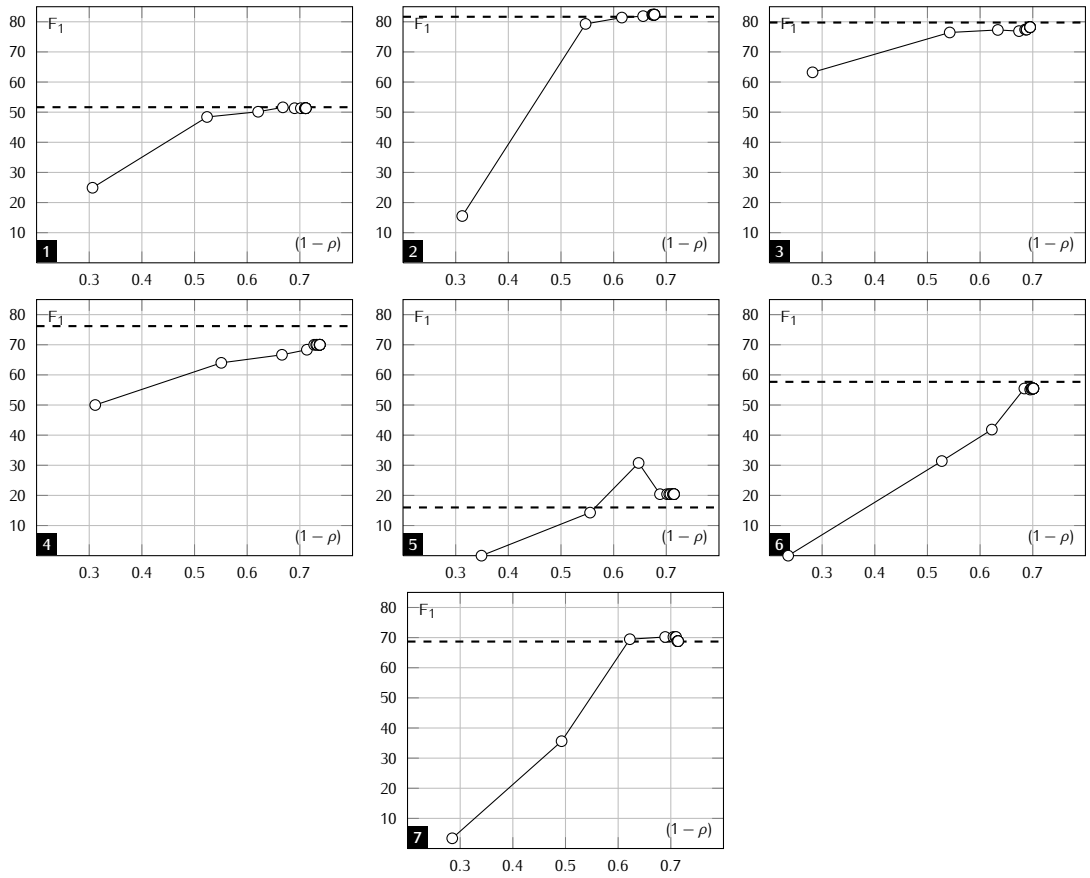


Figure 5.5: **RE: Accuracy of the MLin model vs. norm after feature selection** - A dashed line marks the accuracy of the corresponding STK (i.e. non-linearized) classifier.

5.3 Comparing Accuracy against STK

To assess the potential of the feature selection strategy in terms of accuracy, we consider the greedy miner algorithm (Alg. 4.4) with the LOpt architecture (Sec. 4.1, Fig. 4.2) and compare its accuracy against STK. For each task, a multi-classifier is obtained by combining the binary classifiers in a One-vs-All (OvA) fashion.

Training LOpt for a class c implies: 1) learning an STK model M for c (KSL); 2) mining M with a value L for the threshold factor parameter (KSM); 3) decoding the training and test data (LSG); 4) learning a linear model M' in the projected space using the decoded training data (LSL); 5)

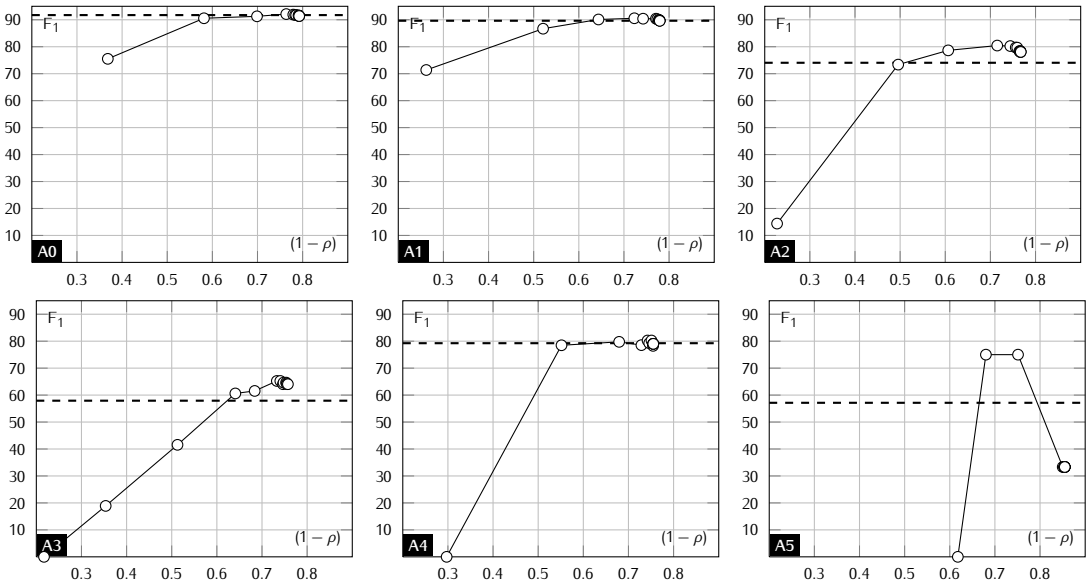


Figure 5.6: **SRL: Accuracy of the MLin model vs. norm after feature selection** - A dashed line marks the accuracy of the corresponding STK (i.e. non-linearized) classifier.

classifying the decoded test data with M' (LSC).

Since binary STK classifiers for QC and RE showed a consistent precision/recall unbalance, we optimize the parameter j of SVM-Light on a per-class basis. j is the cost factor by which training errors on positive examples outweight errors on negative examples [Morik et al., 1999]. Validation is carried out on a development set obtained by sampling 1/7 and 1/6 of the respective training data. The optimal value of the parameter is estimated with a simple hill-climbing algorithm. Per-class estimated values of the parameter are listed in Table A.4a.

The optimized STK models are then employed for KSL, as well as a term of comparison for LOpt. To make the comparison fair, for QC and RE we also optimize j for learning in the linear space. LSL optimization is carried out for each value of L, i.e.:

1. we set a value for L;

2. the STK model is mined;

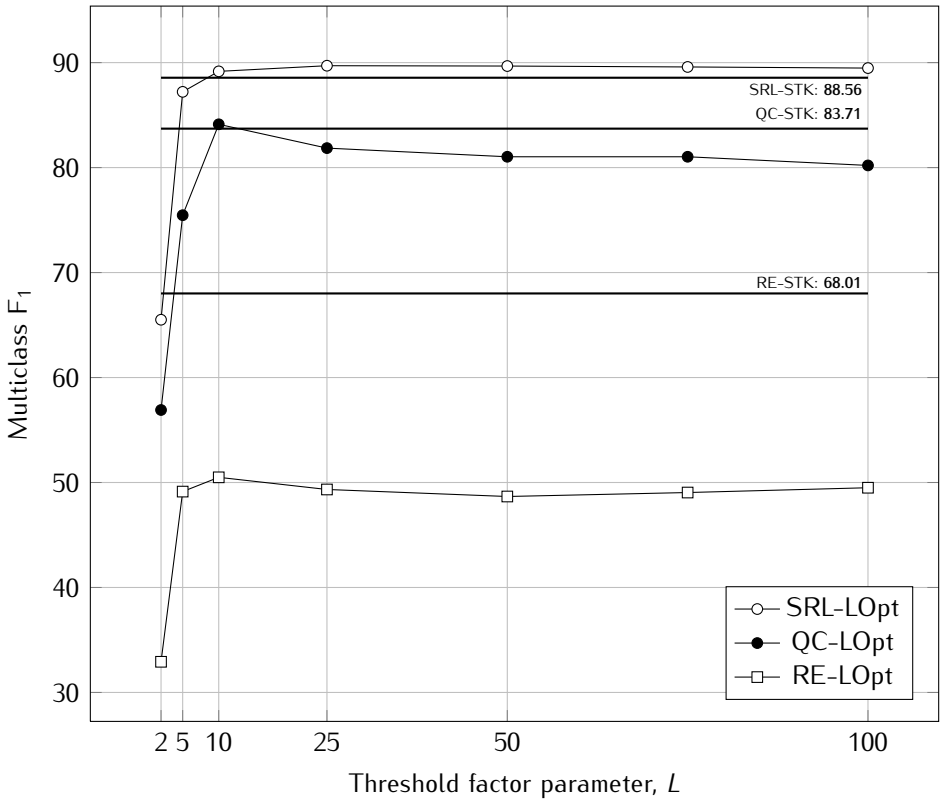


Figure 5.7: **LOpt multiclass accuracy on the different tasks for different values of the threshold factor parameter L.**

3. linear data are generated;

4. the optimal value for j is estimated on these data.

Figure 5.7 plots the multiclass F_1-measure on the three tasks. Multi-classifiers are obtained by OvA-combining LOpt classifiers that used the same value of L during KSM. For reference, the number of fragments mined from each class for different values of L is listed in Tables A.1 (QC), A.2 (RE) and A.3 (SRL).

The three tasks exhibit a very similar behaviour: they are very inaccurate for extremely low values of L (i.e. 2 or 5, corresponding to a linear space with approximately $O(10^1) \sim O(10^2)$ attributes), then peak for $10 \leq L \leq 25$. After that, they stabilize. The thick lines in the plot correspond to the accuracy of STK multi-classifiers for the three tasks, from top to bottom:

	ABBR	DESC	ENTY	HUM	LOC	NUM	**MULTI**
STK	80.00	86.26	**76.86**	**84.92**	81.69	92.31	83.71
LOpt	**89.66**	**87.50**	75.56	84.53	**92.08**	**94.60**	**83.92**

Table 5.4: **QC: F_1-measure of STK vs. LOpt best model.**

SRL, QC and RE.

For SRL, LOpt generally outperforms STK, stabilizing for $L \geq 10$ around F_1 values close to 90. Concerning QC, LOpt improves over STK only for $L = 10$, then stabilizes approximately two F_1 measure points below it (e.g. 81.65 for $L = 50$ vs. 83.71). The RE-LOpt multi-classifier cannot improve over STK, and stabilizes two points below it, i.e. 65.92 for $L = 75$ vs. 68.01.

These results suggest that the feature selection framework is capable of producing linear classifiers whose accuracy is comparable with STK, if appropriate values of the threshold factor parameter L are selected. Interestingly enough, these values are generally very low, i.e. 10 or 25, resulting in very compact linear classifiers.

In the next experiment, for each class we estimate the optimal value of L on the development set. For the SRL task, validation is carried out on the 9,277 argument nodes of PropBank section 23. [3] For QC and RE, after estimating the optimal value for L, we also select the best cost factor in the linear space. The set of optimal parameters for all the binary LOpt classifiers are listed in Table A.4b. The results of the per-class and multi-class evaluation are displayed in tables 5.4, 5.5 and 5.6.

Concerning QC (Table 5.4), LOpt binary classifiers outperform STK on four classes, namely ABBR, DESC, LOC and NUM. In two cases, i.e. ABBR and LOC, the improvement is very consistent: 9.66 points in the former (89.66 vs. 80.00) and 7.39 in the latter (92.08 vs. 81.69). The improvements are very interesting, especially if we consider that the STK classifier is

[3] In the official CoNLL-2005 split, section 24 is used for development and section 23 for testing. We used section 23 for development because we already running experiments using section 24 as a test set.

	1	2	3	4	5	6	7	MULTI
STK	51.61	**81.66**	**79.75**	**76.19**	16.00	57.69	**68.69**	**68.01**
LOpt	**63.96**	67.33	74.14	63.33	**58.82**	**62.83**	61.70	67.08

Table 5.5: **RE: F_1-measure of STK vs. LOpt best model.**

highly balanced and optimized.

ABBR and LOC are the two classes with the smallest number of positive training points (85 and 834), and in these cases the TK may over-estimate the relevance of positive training points, by over-fitting irrelevant features to the data. This is not an issue in the very low dimensional projected space. As for the multiclassifier, we observe that LOpt can preserve the accuracy of STK, i.e. 83.92 vs. 83.71. Unluckily, most of the accuracy gained with the binary classifiers is lost during recombination.

In the case of RE (Table 5.5), we observe that the linearization framework is generally less effective. In fact, LOpt improves over STK on three classes out of 7, namely 1, 5 and 6. The LOpt classifier for class 5 is more than three times as accurate as its STK counterpart, which has a very low F_1. However, the STK classifier is still well parametrized, with a precision of 12.9 and a recall of 21.05. Also in this case, the class has a very small positive training set, and the good improvement is an effect of the reduced tendency to overfit. The classes where LOpt accuracy loss is more evident, e.g. 2 and 4, correspond to heavily lexicalized relations, Person/Social Group and Agent/Artifact. Overall, the LOpt multi-classifier is approximately one F_1 point less accurate than STK, i.e. 67.08 vs. 68.01.

Unlike the QC task, in which different classes of questions often have a different syntactic structure, in RE many relations share a similar syntax but are characterized by different lexical realizations. In many cases, the mentions involve proper nouns, and in order to establish the presence of a relation it is necessary to consider fragments that contain lexical anchors

	A0	A1	A2	A3	A4	A5	**MULTI**
STK	91.55	88.50	72.33	57.34	65.96	**66.67**	87.69
LOpt	**91.77**	**90.09**	**78.63**	**62.67**	**72.16**	**66.67**	**88.90**

Table 5.6: **SRL: F_1-measure of STK vs. LOpt best model.**

for both mentions. This kind of information (the subtree dominating both mentions, up to their leaves) is encoded only by large fragments, which are very unlikely to achieve high cumulative relevance due to data sparsity and the decay factor of the kernel. In this context, the STK has the advantage of considering all the fragments that characterize the two mentions and their syntactic link. These may be discarded by the miner since not very relevant per se. However, they still play a decisive role when observed together.

The results on the SRL benchmark are listed in Table 5.6. All the LOpt classifiers outperform or match (A5) the accuracy of STK. LOpt manages to improve by approximately 1.5 points the accuracy of A1-STK, 90.09 vs. 88.50, whereas for A2, A3 and A4 the improvement is more consistent, by about 5 F_1 points. The A5-LOpt classifier classifies correctly only one of the two positive test examples, just like A5-STK. On the multi-class problem, LOpt improves over STK by more than 1 F_1 point, i.e. 88.90 vs. 87.69, a good achievement considering the high accuracy of STK.

As a whole, the results obtained on the three benchmarks confirm that the LOpt model can result in classifiers whose accuracy is in line with non-linearized STK classifiers. Interestingly, the best values for the threshold factor parameter L are generally very low, the most frequent being 10 and 25 (as shown in Table A.4b and anticipated by 5.7). As we will show in Section 5.5, these classifiers are also very efficient if compared against STK.

5.4 Algorithmic Efficiency

This section focuses on the evaluation of the efficiency of the algorithms involved in the feature selection process. It begins with an empirical analysis of KSM and LSG time complexity. KSM insists on the greedy miner and fragment encoding algorithms, respectively described in Alg. 4.4 and Alg. 4.7. LSG efficiency depends on the complexity of the STK decoding algorithm, listed in Alg. 4.8.

> **A note on implementation.** The STKTree data structure and all the algorithms involved in the linearization process are coded in Python3[4]. This choice is motivated by the need for a fast development platform that would allow us to effectively implement and experiment with different algorithms and data structures. Due to the experimental nature of the software, non-critical parts of the process have not been optimized. We can expect that optimizing the code and re-implementing it in a compiled language like C or C++ would result in further and consistent efficiency gains.

Figure 5.8 plots KSM and LSG time against the number of mined fragments, along with the respective trendlines. The scatters are based on the STKTrees for the SRL task, which contains the highest number of fragments due to the larger dataset. Both algorithms show sub-linear behaviour, respectively $O(x^{0.76})$ and $O(x^{o.17})$, confirming that STKTree scales well with the number of mined fragments.

Concerning KSM, fragment mining and tree encoding are very tightly coupled operations. However, we can observe that the encoding process (Alg. 4.7) traverses a fragment in depth first order. The complexity of this operation is linear with the number of nodes in the fragment, and does not really depend on the size of the STKTree. On the other hand, the more the fragments in the STKTree, the less the chances of having to create new nodes when invoking the GET_LAB$(\cdot)$ and GET_PROD$(\cdot)$ procedures. The asymptotic behaviour of KSM time can therefore be largely ascribed to the

[4] http://www.python.org.

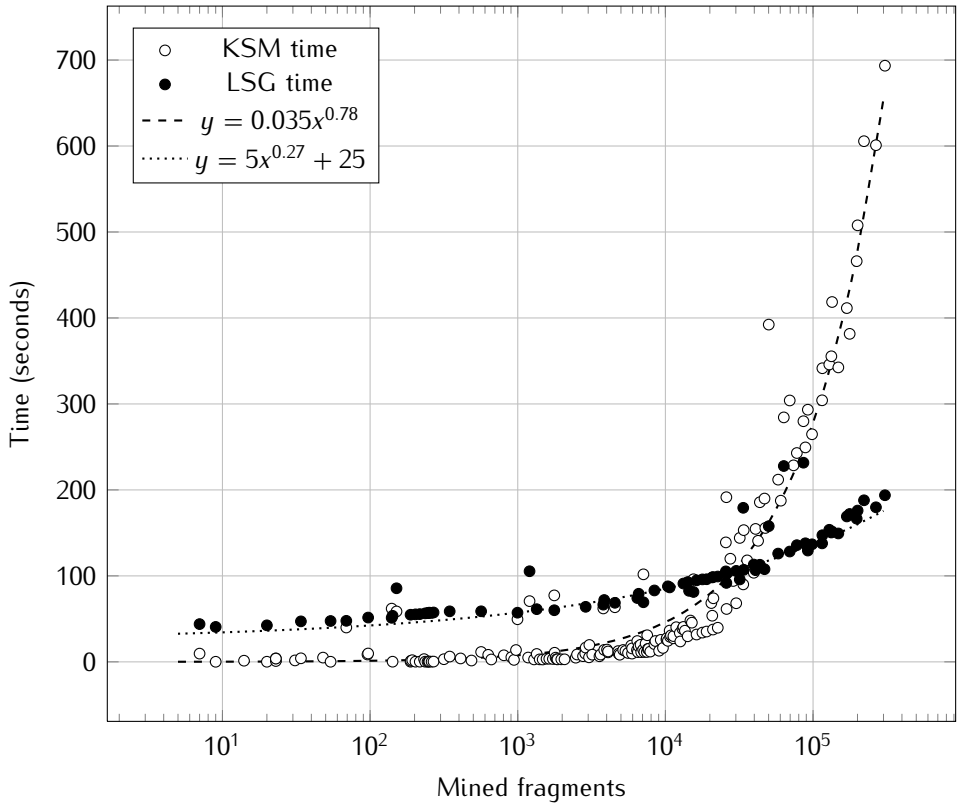


Figure 5.8: **KSM and LSG time vs. number of mined fragments** - For KSM, the plot is obtained by combining data from all binary linearized classifiers of the three tasks. For LSG, SRL data has been employed.

greedy miner[5].

Concerning LSG, the plot shows that the time it takes to decode the data is approximately proportional to the cubic root of the size of the dictionary. This result, which suggests that the STKTree structure is effective in guiding the decoding process, is confirmed by the plots in Fig. 5.9, where we show how the size of the to-be-decoded tree affects LSG time. With respect to Algorithm 4.8, the figure plots time complexity of the LOOKUP(·) operation (that finds the fragments rooted in a given node of the tree) and of the overall DECODE_STK(·) operation (that invokes LOOKUP(·) for all the nodes

[5]Mining complexity is actually a function of L, i.e. the threshold factor parameter, and not of the number of fragments in the STKTree. The latter is, in turn, a function of L and the characteristics of the dataset. The values of L corresponding to different number of fragments for the SRL classifiers are listed in Table A.3.

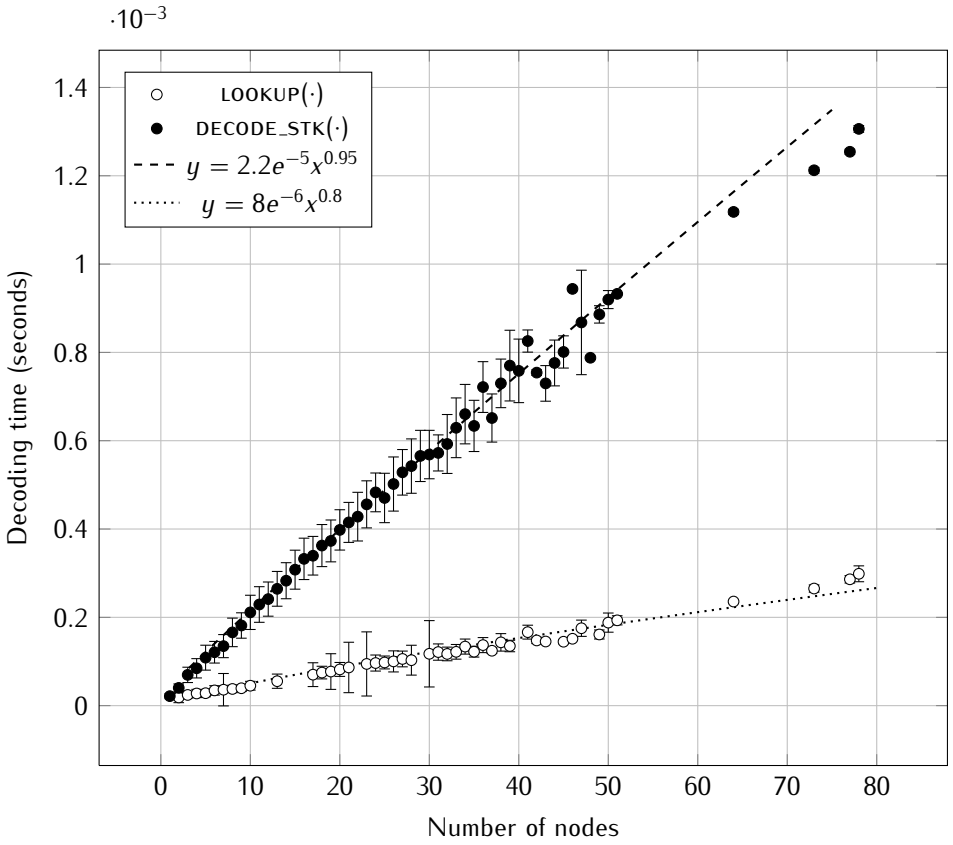


Figure 5.9: **Average decoding time for trees of different size.**

in a tree). The plot is obtained by measuring decoding time for all the trees in the SRL with the STKTree for the A1 class, the index with the highest number of fragments, and shows average and standard deviation for subtrees of the same size.

The scatters can be approximated with sub-linear curves, but they basically show a linear trend. This is a feature of the STKTree, that while decoding allows us to generate only a subset of all the possible fragments. If we could not rely on this information, we would have to generate all the fragments in the tree, and hence expect an exponential complexity. Even assuming to know the size of the maximum expansion performed during mining, let us call it E, and the depth of the largest encoded fragment, D, the complexity of generating all the fragments rooted in a node n would still be

$O(s(n)E^D)$, $s(n)$ being the number of internal nodes in the subtree rooted in n.

5.5 Process Efficiency

In this section we provide an empirical evaluation of the efficiency of our feature selection strategy, i.e. its effect on the training and test time of a classification problem.

When employing LOpt (or Split), classification time is the time necessary to decode the test data (LSG) and to classify it with the linear model (LSC)[6]. Similarly, training time is the sum of the time required to carry out: KSM, during which we learn the tree kernel model; KSM, where the relevant fragments are mined; LSG, consisting of the decoding of the training data; and LSL, during which the linear model is learned.

Figure 5.10 plots LOpt classification time normalized against its STK counterpart, for different values of the threshold factor parameter L. For each task we show the results measured on the slowest of the binary classifiers, namely ENTY for QC, class 1 for RE and A1 for SRL. STK classification time is shown as a thick black line. As we can expect, the efficiency improvement is especially noticeable for the A1 classifier: in this case, the very large model (36,824 support vectors) requires STK to perform a great number of kernel products, making it quite inefficient. LSG time for LOpt is more than compensated by the increased burden of tree kernel evaluation, and also for $L = 1,000$ (when the STKTree indexes approximately $3e^5$ fragments) the A1-LOpt classifier is about 20% faster than STK.

At the other end of the spectrum, the model for ENTY is very small (3,342 SVs), and therefore STK classification is more efficient than LSG+LSC.

[6]We consider the sum of LSG and LSC, even though decoding and classification could be easily pipelined, e.g. after decoding the first test example, we could classify it and decode the first example in parallel.

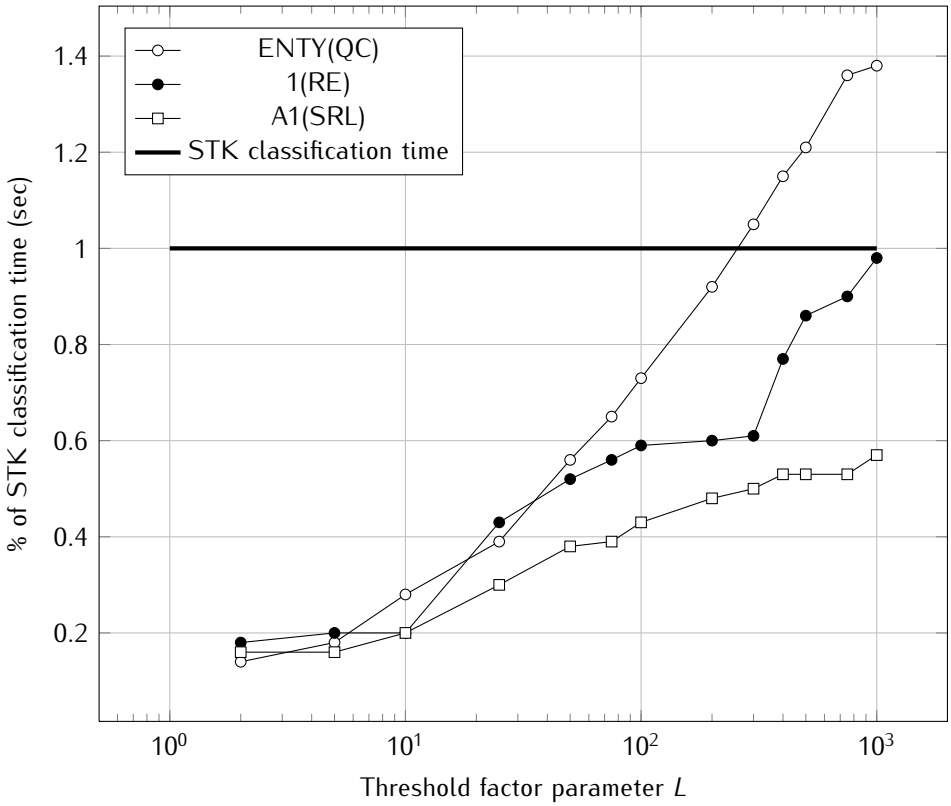


Figure 5.10: **Classification efficiency of LOpt classifiers** - We consider classification time the time required to decode the test set (i.e. LSG time for the test data) and to classify in the linear space. For each task, the slowest STK classifier was selected.

However, by referring to Table A.4b we can observe that the most accurate models for the three binary classifiers are those with $L = 25$. With these parameters, LSC time is approximately 40% of STK time for ENTY (QC) and class 1 (RE) classifiers, and less than 20% in the case of A1 (SRL). We should also note that while LOpt classification time also accounts for I/O time (which is a major player, especially for large values of L since decoding a tree requires to write very long vectors), STK classification time does not.

To analyze learning efficiency, we consider the most data intensive among the three tasks, SRL, and the Split configuration outline in Fig. 4.3. The split configuration is an extension of the LOpt architecture, where the original training data is partitioned into S sets at the time of KSL.

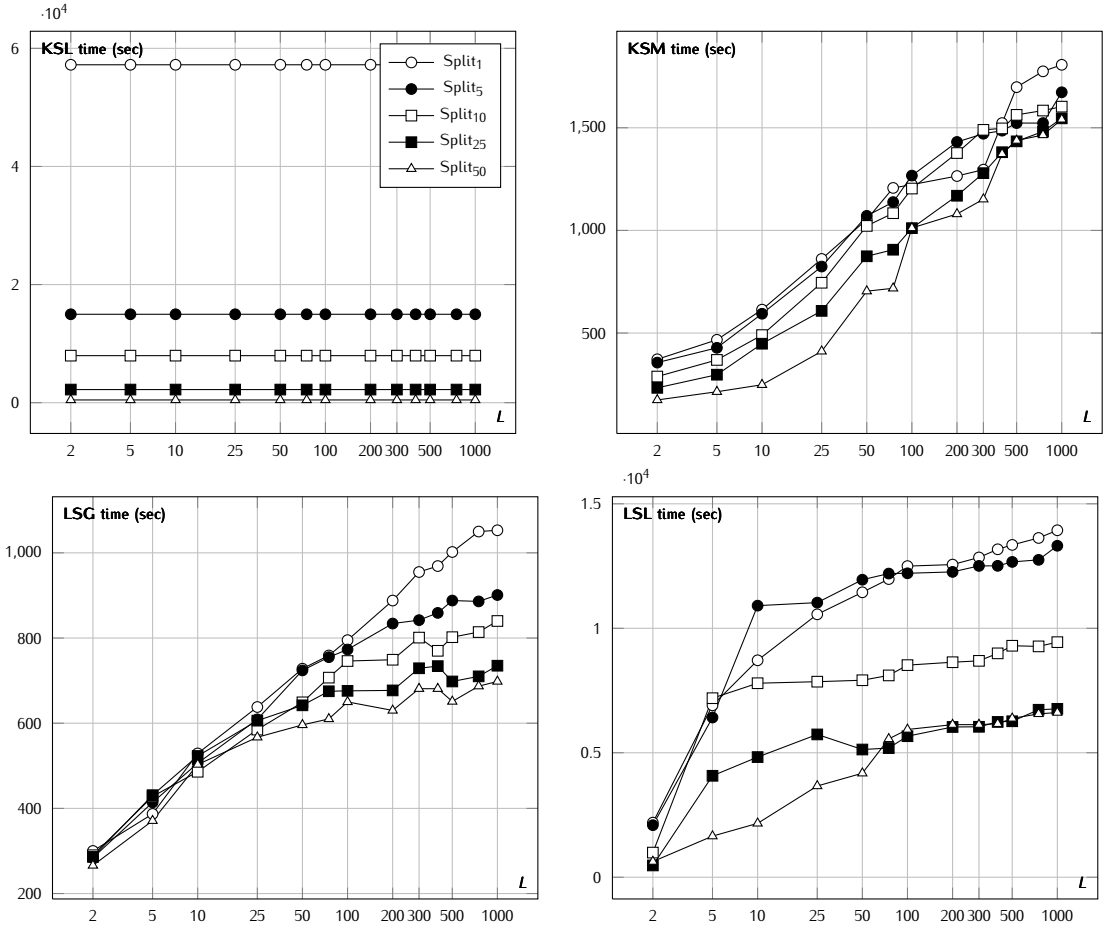


Figure 5.11: **Learning time on the A1 class (SRL) with the Split configuration.**

For each class, we learn S models using STK during KSL, then mine them during KSM, and finally recombine the fragments into a unique STKTree, which is used for LSG. The linearized data is then used for learning a linear model in the lower dimensional space, i.e. LSL. The LOpt architecture is a special case of Split with $S = 1$. Henceforth, let Split_x be a Split model with $S = x$, e.g. Split_1 is an LOpt model.

Figure 5.11 focuses on the most demanding classifier for the SRL task, A1, and plots, for different values of L, time complexity (in seconds) of the 4 components of Split learning: KSL, KSM, LSG (only for training data) and LSL. We consider learning to be carried out on a single CPU, i.e. all the stages of the process are sequential. The top-left plot in Fig. 5.11 shows

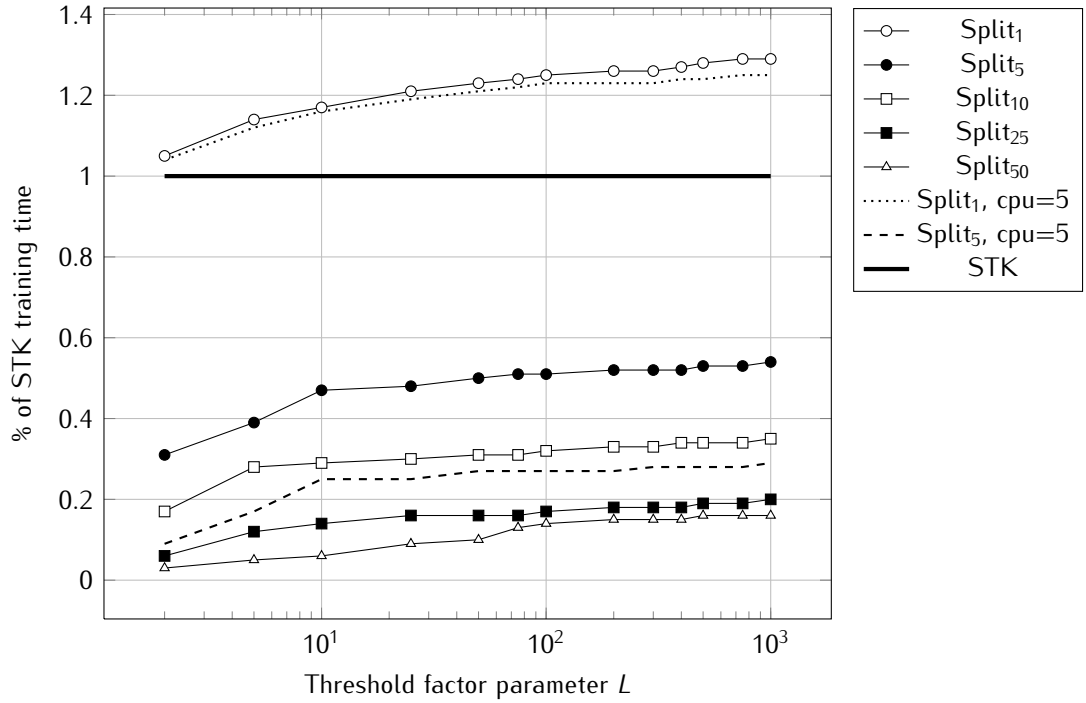


Figure 5.12: **Learning efficiency of the Split architecture vs. STK on the A1 class.**

KSL time. Since KSL does not depend on L, all the plots are constant. As expected, increasing the number of splits can drastically reduce learning time in the kernel space.

Concerning KSM (top-right) and LSG time (bottom-left), they are only marginally influenced by the number of splits. For increasing values of S, we have less SVs in a model, therefore mining each model is generally faster. Furthermore, the final STKTree (obtained by combining the fragments mined from each mode) will tend to include less fragments than when mining a single model. This is explained by the fact that most of the fragments will be duplicated across models, and therefore will be redundant after merging.[7] For the same reason, LSG is faster for Split$_{50}$ than for Split$_1$. Due to the smaller linear space, increasing the number of splits also has a positive effect on LSL, as shown by the bottom-right plot in Fig. 5.11.

[7] See Table A.5 for the size of the A1 STKTree for different values of L and S.

Overall, a high number of splits can considerably reduce KSL, which contributes most of the computational burden, while affecting only marginally the other stages of the process. In Figure 5.12 we plot the cumulative training time of the A1-Split model, i.e. KSL + KSM + LSG + LOpt time, normalized against STK training time (which is the same as KSL time for Split_1. As a reference, KSL learning time has been measured to approximately 50 thousand seconds, i.e. 5 days). Training Split_1 is less efficient than training STK, since the latter does not have the overhead introduced by KSM, LSG and, especially, LSL. The overhead introduced by the linearization framework is approximately 20% for $L = 25$ and 30% for $L = 1,000$. Increasing the number of splits, learning efficiency improves considerably. Training Split_5 is approximately between 70% and 50% less costly than STK, while Split_{50} can be trained in between 1/10 ($L < 10$) and 1/5 ($L = 1,000$) of the time necessary to train STK.

The efficiency of Split can be further improved if we consider parallelization. In fact, several activities of the process can be carried out concurrently: KSL (we can assign each split of the training data to a distinct CPU/core/node to learn the STK model); KSM (each model can be mined indipendently); and LSG (each split can be linearized on a different node). The only stage that cannot be carried out in parallel is the optimization of the global linear model, LSL. The dotted line in Fig. 5.12 marks Split_1 training time when using 5 CPUs. In this case, also KSL cannot be parallelized, since we have only one split. Therefore, increasing the number of CPUs for Split_1 does not introduce a real gain. The situation is very different for Split_5 using 5 CPUs, whose learning time is shown in figure as a dashed line: since we can parallelize KSL, we can drastically reduce the time complexity of the model and make it approximately as efficient as Split_{10} or Split_{25} on a single processing unit.

Faster algorithms in the linear space. Concerning learning time of LOpt and Split

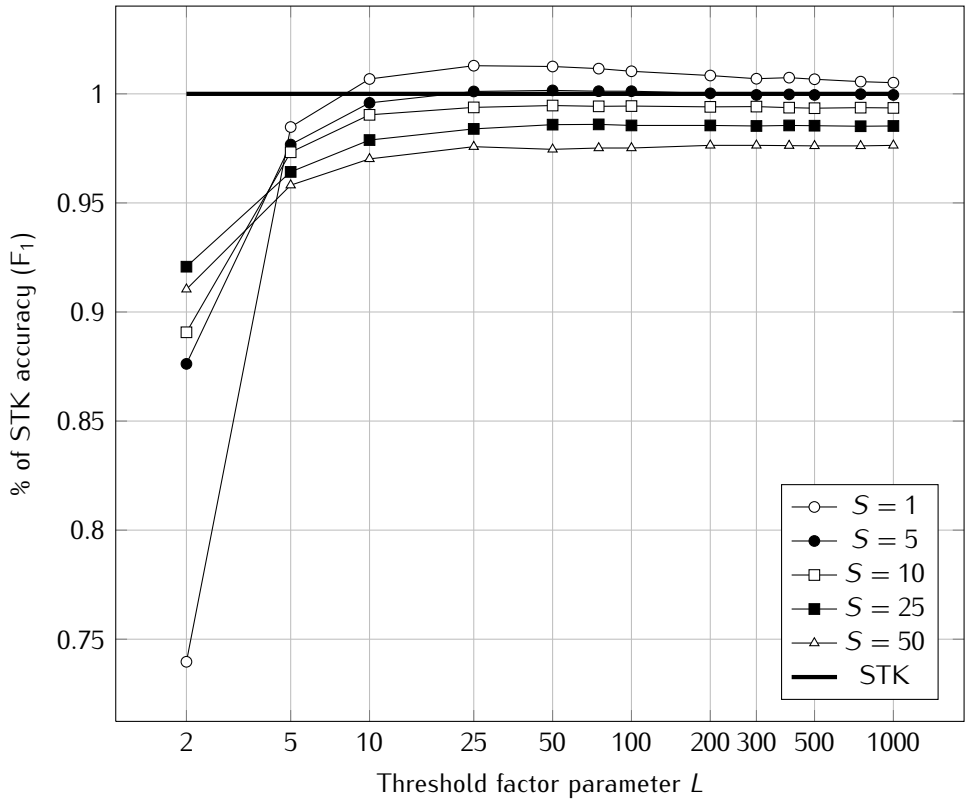


Figure 5.13: **Multi-class accuracy of the Split configuration on SRL** - The plot shows the percentage of STK accuracy achieved by using different number of splits for different values of the threshold factor parameter L.

classifiers, the LSL stage (during which we learn a global optimization in the linear space) is a major bottleneck. The impact of LSL on the overall training time could be easily reduced by employing faster SVM learners than SVMLight, which is not highly optimized for linear spaces. As an example, the LinearSVM optimizer (http://www.linearsvm.com/Linear_SVMs.html) scales linearly with the size of training data and promises very fast learning cycles.

Finally, Figure 5.13 shows the effect of training data splitting on the final accuracy of the multi-class models. The accuracy of Split, for different values of L and S, is normalized with respect to the accuracy of STK, shown as a thick black line. Increasing the number of splits has a negative effect on the accuracy, but the loss of accuracy is almost negligible. While Split_1 (i.e. LOpt) is more accurate than STK, Split_5 is as accurate as STK for $L > 10$, and Split_{50} preserves more than 97.5% of the accuracy of STK,

while being five times more efficient.

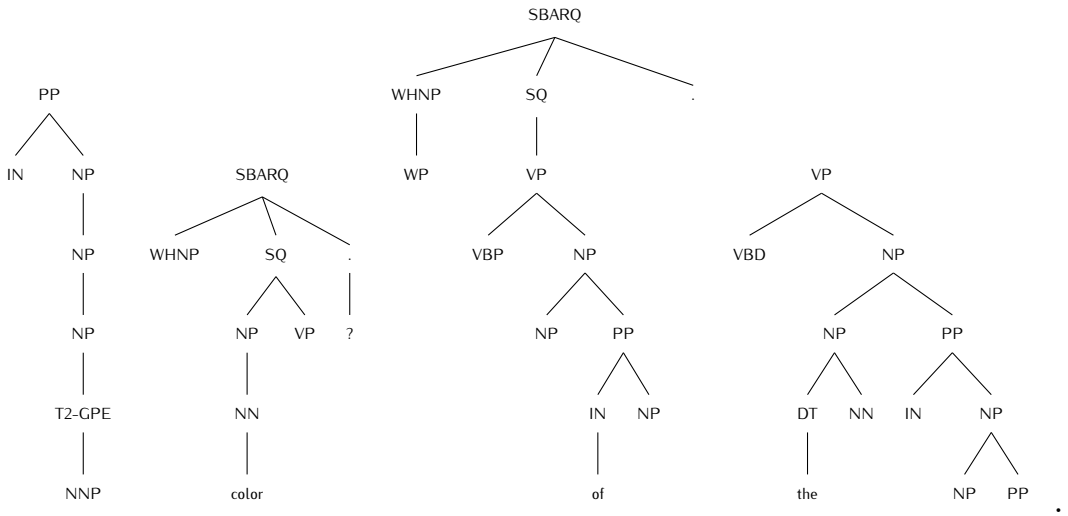
5.6 Making the Fragment Space Explicit

As we pointed out in Chapter 1, one of the key features of our fragment selection strategy is its ability to make the fragment space explicit and observable. The fragments that we obtain can be a valuable instrument to improve our understanding of the problems at study, since they can provide interesting clues about the features that characterize different classes.

A meaningful analysis of the fragments, requiring deep linguistic knowledge and expertise, is out of the scope of this thesis. However, some results in this direction are discussed in our previous work. In [Pighin and Moschitti, 2009b], we carried out a very high-level analysis of the most relevant fragments identified for the QC task, observing how different classes insist on consistently different families of relevant fragments; in [Pighin and Moschitti, 2009a] we observed that a large part of the most relevant fragments for SRL encode the linguististic features described in [Gildea and Jurafsky, 2002] and [Pradhan et al., 2005].

For completeness, in Appendix B we list the 100 most relevant features that were selected for each class of the three investigated tasks. The ranking of the fragments is not based on their weight in the kernel space, but rather on their relevance in the linear space. After learning the LOpt model, we check which fragments are encoded by support vectors in the lower-dimensional space, and we rank them according to their cumulative weight, considering their sign. By doing so, we obtain the fragments that are more strictly correlated with positive examples, i.e. those that mostly characterize the class. The features are collected from the best models optimized for each class, with the parameters listed in tables A.4a and A.4b for the different stages of LOpt.

As a general remark, we can observe that the decay factor is very effective at limiting the relevance of large structures as shown by the fact that most fragments only span one or two levels. Still, some classes (e.g. ENTY and DESC) show a consistent presence of more structured information, and in some cases the greedy miner is capable of generating quite large fragments, as in the following examples:



This is a good piece of evidence in support of our feature generation strategy, that allows large structures to be considered, as opposed to containing complexity by hard-coding a limit to the size of the admissible fragments.

Chapter 6

Conclusions

Tree Kernels are interesting tools that can be effectively employed to tackle problems requiring a large amount of syntactic information, and more generally for all those cases where there are no clear theories or evidence concerning the relevant features that characterize a problem.

This thesis has introduced a novel method for feature selection in tree kernel spaces, combining the power and modeling simplicity of convolution kernels with the speed and clarity of linear models and explicit feature representations. Among the major points points of novelty:

- it provides a theoretical framework for feature selection in convolution kernel spaces. The theory motivates the adoption of very aggressive selection strategies, which are necessary to cope with the exceptional dimensionality of the kernel space;
- it describes a greedy selection strategy that exploits the properties of convolution kernels to project structured data onto a significantly lower dimensional space. After featue selection, the gradient norm and the margin in the original space are largely preserved;
- the projection is effective and efficient. Our model can automatically discover relevant features in a huge space, without the need for explicitly defined feature extraction algorithms;

- the interesting features discovered in the rich tree kernel space are made observable;

- the linearized classifiers sport very good accuracy and faster classification time. With the Split configuration, training time on larger datasets can also be improved.

Our theoretical framework is based on statistical learning theory and properties of support vector learning. We showed that it is possible to discard an exponential number of attributes in the kernel space and lose only a relatively small portion of the norm of the separating hyperplane's gradient. This finding allows us to reduce the dimensionality of the problem by several orders of magnitude while still retaining a large margin, and hence most of the classifier's accuracy, in the original space.

We studied the problem of how to map individual fragments onto a linear space, and how to estimate the contribution of individual fragments to the gradient. We designed an algorithm that finds and explicitly represents the most relevant fragments encoded in a tree kernel space. The gradient component associated with a fragment is used both as a measure of its relevance and to direct the search strategy. As it does not enforce any constraint on the maximum size of the fragments that it generates, it has the potential to selectively generate large structures.

To support our claims, we ran an exhaustive empirical evaluation on three different NLP multi-classification tasks: question classification, relation extraction and semantic role labeling.

We showed that the weights estimated by the SVM in the tree kernel space can be reused to carry out classification in a projection of the input space, where only a few thousands out of billions of attributes are retained. The good accuracy obtained by these classifiers confirms the validity of our theoretical assumptions. The algorithms and the data structures that

we devised show interesting time-complexity properties. Mining the kernel space and generating the linear representation of the structured data exhibit sub-linear behaviour with respect to the number of fragments that we decide to consider. Tree decoding (the process of generating an explicit vector of relevant fragments from a tree) is approximately linear with the size of the tree.

The features selected in the high-dimensional space were reused to learn a new model in the linear space, resulting in classifiers as accurate as those in the original tree kernel space. Interestingly, the accuracy is generally maximized for small values of the parameter that controls the size of the linear space, resulting in very compact models. Such classifiers are very fast, and they can classify the same test data much more efficiently even if we consider the burden introduced by tree decoding. We also demonstrated a technique that, for large datasets, can also improve the efficiency of learning by splitting the learning problem in the kernel space and learning a global model in the linear space. With this technique we can exploit parallel hardware and have learning cycles up to five times as fast, without compromising the final accuracy of the classifiers.

Finally, we showed that our strategy can produce an explicit representation of the most relevant structured features for each class, and presented a selection of the most relevant fragments identified for each class of the three tasks. Every class shows some peculiarities that would require a deeper linguistic analysis to be exploited and understood. We demonstrated that the framework can also be employed for feature discovery in high dimensional spaces, exploiting the capability of the TK to generate and weigh complex structured objects. This evidence confirms that this thesis marks an interesting advance towards effective techniques for automatic feature engineering.

Research Directions

The results of our experiments have shown that our feature selection approach for structural kernel is efficient and accurate. Nonetheless, the model and the algorithms could still be improved in several ways. Most notably, it would be extremely interesting to establish a theoretical correlation between the optimal value of the threshold factor parameter and the classification problem. Assuming that it would be possible, the solution likely depends on several factors, among which the distribution of positive and negative examples, the weight of the most and least relevant fragments, the norm of the original gradient and the number of classes involved. The outcomes of our experiments suggest that a simple correlation between the distribution of the examples and the optimal threshold does not exist. This fact is clearly shown by the best parameters identified for the SRL task.

Even lacking a theoretical framework, it would be interesting fo find an effective algorithm to automatically set the threshold value, without the need to estimate an optimal parameter on a development set. In this way, the linearization framework would be more efficient and less prone to overfitting training data, thus becoming an even more appealing solution. Our results show that fragment/norm curves (e.g. figures 5.1 and 5.2) have a monotonic behaviour that makes them interesting candidates in this respect. As we know that including more fragments than necessary tends to have a negative effect on accuracy, we could increase the value of the threshold parameter until the derivative of the fragment/norm curve is above a fixed threshold.

The linearization framework has been designed to be adaptable to cope with different families of kernel functions. Even though we only experimented wth the STK, basic support for the PTK and for combinations of TK functions is already available in the software and ready to be experimented with. By testing our framework with other kernel families, we will be able

to tackle a lot of other interesting problems and verify the generality of our claims. It would also be interesting to test the model on datasets coming from other disciplines, such as bio-informatics or topologic problems, where TKs and other kinds of structural kernels can find a natural application.

Last but not least, it would be interesting to make the framework even more general by devising strategies to mine classes of relevant fragments rather than fragments. In other words, we would like to define a set of topological and domain-knowledge based rules that can define classes of equivalence between fragments, and then mine the TK space for the most relevant classes rather than for fragment instances. A similar approach would make it possible to learn automatically the abstract feature definitions that characterize a learning problem.

Appendix A

Evaluation Complement

This appendix collects some tables that can be helpful in reading the outcome of the experiments (Chapter 5).

L	ABBR	DESC	ENTY	HUM	LOC	NUM
2	$2 \cdot 10^0$	$1.2 \cdot 10^1$	$2.2 \cdot 10^1$	$1 \cdot 10^1$	$5 \cdot 10^0$	$1.4 \cdot 10^1$
5	$9 \cdot 10^0$	$9 \cdot 10^1$	$2.4 \cdot 10^2$	$1.2 \cdot 10^2$	$3 \cdot 10^1$	$7.5 \cdot 10^1$
10	$3.9 \cdot 10^1$	$4.2 \cdot 10^2$	$8.8 \cdot 10^2$	$4.9 \cdot 10^2$	$1.4 \cdot 10^2$	$2.4 \cdot 10^2$
25	$1.7 \cdot 10^2$	$1.9 \cdot 10^3$	$2.9 \cdot 10^3$	$1.9 \cdot 10^3$	$7.2 \cdot 10^2$	$1 \cdot 10^3$
50	$3.5 \cdot 10^2$	$3.5 \cdot 10^3$	$5.7 \cdot 10^3$	$3.8 \cdot 10^3$	$1.7 \cdot 10^3$	$2.2 \cdot 10^3$
75	$5.7 \cdot 10^2$	$5.2 \cdot 10^3$	$7.8 \cdot 10^3$	$5.2 \cdot 10^3$	$2.8 \cdot 10^3$	$3.5 \cdot 10^3$
100	$7.8 \cdot 10^2$	$6.4 \cdot 10^3$	$9.7 \cdot 10^3$	$6.5 \cdot 10^3$	$3.8 \cdot 10^3$	$4.6 \cdot 10^3$
200	$1.4 \cdot 10^3$	$9.6 \cdot 10^3$	$1.5 \cdot 10^4$	$1 \cdot 10^4$	$6.8 \cdot 10^3$	$7.5 \cdot 10^3$
300	$1.7 \cdot 10^3$	$1.1 \cdot 10^4$	$1.9 \cdot 10^4$	$1.2 \cdot 10^4$	$9 \cdot 10^3$	$9.1 \cdot 10^3$
400	$2 \cdot 10^3$	$1.2 \cdot 10^4$	$2.1 \cdot 10^4$	$1.5 \cdot 10^4$	$1.1 \cdot 10^4$	$1.1 \cdot 10^4$
500	$2.1 \cdot 10^3$	$1.3 \cdot 10^4$	$2.4 \cdot 10^4$	$1.6 \cdot 10^4$	$1.2 \cdot 10^4$	$1.2 \cdot 10^4$
750	$2.5 \cdot 10^3$	$1.5 \cdot 10^4$	$2.8 \cdot 10^4$	$1.9 \cdot 10^4$	$1.5 \cdot 10^4$	$1.4 \cdot 10^4$
1,000	$2.8 \cdot 10^3$	$1.6 \cdot 10^4$	$3 \cdot 10^4$	$2 \cdot 10^4$	$1.6 \cdot 10^4$	$1.5 \cdot 10^4$
2,500	$3.9 \cdot 10^3$	$2.5 \cdot 10^4$	$4 \cdot 10^4$	$2.7 \cdot 10^4$	$2.2 \cdot 10^4$	$2.1 \cdot 10^4$
5,000	$4.4 \cdot 10^3$	$2.7 \cdot 10^4$	$4.8 \cdot 10^4$	$3.2 \cdot 10^4$	$2.5 \cdot 10^4$	$2.3 \cdot 10^4$
7,500	$4.7 \cdot 10^3$	$2.8 \cdot 10^4$	$5.3 \cdot 10^4$	$3.4 \cdot 10^4$	$2.6 \cdot 10^4$	$2.4 \cdot 10^4$
10,000	$5 \cdot 10^3$	$2.8 \cdot 10^4$	$5.4 \cdot 10^4$	$3.5 \cdot 10^4$	$2.7 \cdot 10^4$	$2.5 \cdot 10^4$

Table A.1: **QC: number of fragments mined for different values of the threshold factor parameter.**

L	1	2	3	4	5	6	7
2	$9.6 \cdot 10^1$	$3.1 \cdot 10^1$	$4.8 \cdot 10^1$	$1.4 \cdot 10^1$	$2.3 \cdot 10^1$	$2.3 \cdot 10^1$	$2.3 \cdot 10^1$
5	$9.7 \cdot 10^2$	$3.2 \cdot 10^2$	$6.3 \cdot 10^2$	$1.9 \cdot 10^2$	$1.9 \cdot 10^2$	$4.1 \cdot 10^2$	$2.3 \cdot 10^2$
10	$3.1 \cdot 10^3$	$8.1 \cdot 10^2$	$1.8 \cdot 10^3$	$6.6 \cdot 10^2$	$4.9 \cdot 10^2$	$1.2 \cdot 10^3$	$9.1 \cdot 10^2$
25	$7.5 \cdot 10^3$	$1.8 \cdot 10^3$	$4 \cdot 10^3$	$1.6 \cdot 10^3$	$9.4 \cdot 10^2$	$2.9 \cdot 10^3$	$2.5 \cdot 10^3$
50	$1.2 \cdot 10^4$	$2.8 \cdot 10^3$	$5.9 \cdot 10^3$	$2.5 \cdot 10^3$	$1.3 \cdot 10^3$	$4.1 \cdot 10^3$	$4.1 \cdot 10^3$
75	$1.5 \cdot 10^4$	$3.2 \cdot 10^3$	$6.8 \cdot 10^3$	$3 \cdot 10^3$	$1.4 \cdot 10^3$	$4.8 \cdot 10^3$	$5.2 \cdot 10^3$
100	$2 \cdot 10^4$	$3.6 \cdot 10^3$	$7.3 \cdot 10^3$	$3.6 \cdot 10^3$	$1.5 \cdot 10^3$	$5.4 \cdot 10^3$	$5.9 \cdot 10^3$
200	$2.9 \cdot 10^4$	$4.8 \cdot 10^3$	$9.3 \cdot 10^3$	$4.9 \cdot 10^3$	$1.6 \cdot 10^3$	$6.4 \cdot 10^3$	$8.6 \cdot 10^3$
300	$3.6 \cdot 10^4$	$5.6 \cdot 10^3$	$1 \cdot 10^4$	$5.9 \cdot 10^3$	$1.7 \cdot 10^3$	$7.1 \cdot 10^3$	$1.1 \cdot 10^4$
400	$4.2 \cdot 10^4$	$6.5 \cdot 10^3$	$1.1 \cdot 10^4$	$7.2 \cdot 10^3$	$1.8 \cdot 10^3$	$7.6 \cdot 10^3$	$1.3 \cdot 10^4$
500	$4.8 \cdot 10^4$	$6.9 \cdot 10^3$	$1.1 \cdot 10^4$	$7.5 \cdot 10^3$	$1.9 \cdot 10^3$	$7.7 \cdot 10^3$	$1.5 \cdot 10^4$
750	$6 \cdot 10^4$	$7.9 \cdot 10^3$	$1.3 \cdot 10^4$	$9.6 \cdot 10^3$	$2 \cdot 10^3$	$1 \cdot 10^4$	$2.1 \cdot 10^4$
1,000	$7.4 \cdot 10^4$	$9.1 \cdot 10^3$	$1.4 \cdot 10^4$	$1.3 \cdot 10^4$	$2.1 \cdot 10^3$	$1.1 \cdot 10^4$	$2.8 \cdot 10^4$

Table A.2: **RE: number of fragments mined for different values of the threshold factor parameter.**

L	A0	A1	A2	A3	A4	A5
2	$1.5 \cdot 10^2$	$1.4 \cdot 10^2$	$6.9 \cdot 10^1$	$7 \cdot 10^0$	$3.4 \cdot 10^1$	$9 \cdot 10^0$
5	$1.2 \cdot 10^3$	$1.8 \cdot 10^3$	$1 \cdot 10^3$	$9.7 \cdot 10^1$	$3.5 \cdot 10^2$	$2 \cdot 10^1$
10	$4.5 \cdot 10^3$	$7.1 \cdot 10^3$	$3.8 \cdot 10^3$	$5.7 \cdot 10^2$	$1.3 \cdot 10^3$	$5.4 \cdot 10^1$
25	$1.4 \cdot 10^4$	$2.6 \cdot 10^4$	$1.5 \cdot 10^4$	$2.9 \cdot 10^3$	$3.9 \cdot 10^3$	$1.4 \cdot 10^2$
50	$2.6 \cdot 10^4$	$5 \cdot 10^4$	$3.2 \cdot 10^4$	$6.5 \cdot 10^3$	$6.6 \cdot 10^3$	$1.9 \cdot 10^2$
75	$3.4 \cdot 10^4$	$7 \cdot 10^4$	$4.7 \cdot 10^4$	$1.1 \cdot 10^4$	$8.4 \cdot 10^3$	$2 \cdot 10^2$
100	$4.1 \cdot 10^4$	$8.6 \cdot 10^4$	$5.8 \cdot 10^4$	$1.3 \cdot 10^4$	$1 \cdot 10^4$	$2.2 \cdot 10^2$
200	$6.3 \cdot 10^4$	$1.3 \cdot 10^5$	$9.2 \cdot 10^4$	$2.1 \cdot 10^4$	$1.4 \cdot 10^4$	$2.4 \cdot 10^2$
300	$7.8 \cdot 10^4$	$1.7 \cdot 10^5$	$1.2 \cdot 10^5$	$2.6 \cdot 10^4$	$1.6 \cdot 10^4$	$2.5 \cdot 10^2$
400	$8.9 \cdot 10^4$	$2 \cdot 10^5$	$1.3 \cdot 10^5$	$3 \cdot 10^4$	$1.8 \cdot 10^4$	$2.5 \cdot 10^2$
500	$9.8 \cdot 10^4$	$2.2 \cdot 10^5$	$1.5 \cdot 10^5$	$3.4 \cdot 10^4$	$1.9 \cdot 10^4$	$2.5 \cdot 10^2$
750	$1.2 \cdot 10^5$	$2.7 \cdot 10^5$	$1.8 \cdot 10^5$	$4 \cdot 10^4$	$2.1 \cdot 10^4$	$2.6 \cdot 10^2$
1,000	$1.3 \cdot 10^5$	$3.1 \cdot 10^5$	$2 \cdot 10^5$	$4.4 \cdot 10^4$	$2.3 \cdot 10^4$	$2.7 \cdot 10^2$

Table A.3: **SRL: number of fragments mined for different values of the threshold factor parameter.**

QC		RE	
Cl.	j	Cl.	j_{KSL}
ABBR	2	1	2
DESC	3	2	3
ENTY	3	3	3
HUM	4	4	4
LOC	3	5	3
NUM	4	6	4
		7	4

(a) KSL

QC			RE			SRL	
Cl.	L	j_{LSL}	Cl.	L	j_{LSL}	Cl.	L
ABBR	25	4.5	1	25	3	A0	25
DESC	50	2.5	2	25	3	A1	25
ENTY	25	2.5	3	100	2.5	A2	25
HUM	10	5.0	4	25	5	A3	100
LOC	10	3.0	5	75	2.5	A4	50
NUM	10	5.0	6	10	5	A5	10
			7	10	5		

(b) KSM and LSL

Table A.4: **Per-class best model parameters for the three tasks.**

L	$S = 1$	$S = 5$	$S = 10$	$S = 25$	$S = 50$
2	140	163	164	128	98
5	1,768	2,019	2,133	2,076	1,736
10	7,098	8,548	8,206	6,695	5,239
25	25,872	24,402	20,394	15,284	11,594
50	50,111	40,856	32,566	23,270	17,146
75	69,515	52,033	40,735	28,385	20,479
100	86,070	60,799	46,620	31,995	22,821
200	134,560	82,125	60,832	39,990	27,882
300	169,376	93,948	68,759	43,840	30,126
400	197,510	102,126	73,703	46,114	31,481
500	221,335	108,293	77,657	47,742	32,462
750	266,701	118,702	84,144	50,371	33,712
1000	306,883	126,603	88,505	51,944	34,472

Table A.5: **Number of fragments in the STKTree for A1 for different values of the threshold factor parameter L and number of splits S.**

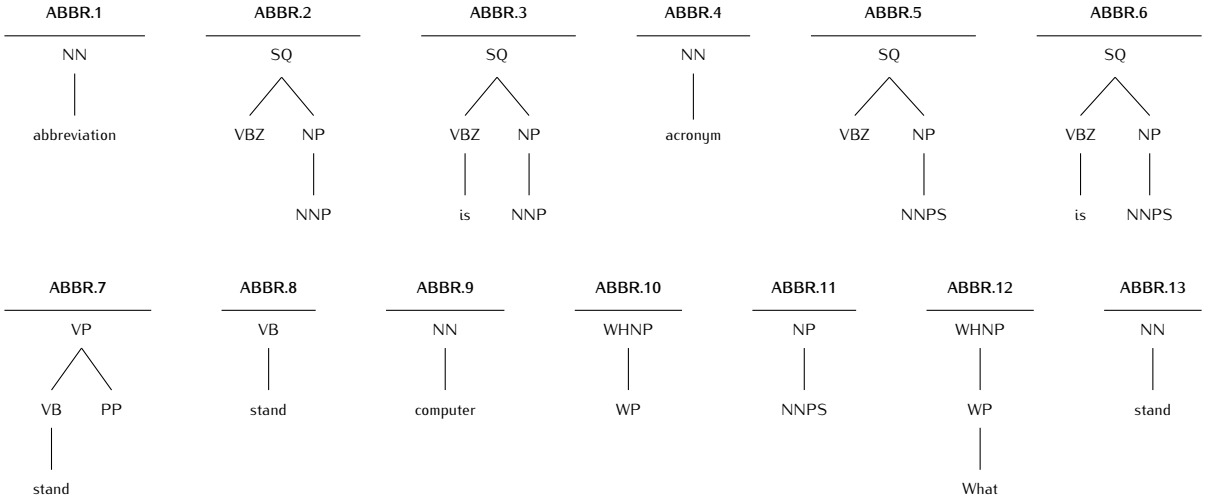
Appendix B

Relevant Fragments

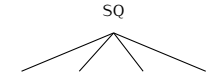
In this appendix we list the 100 more relevant fragments extracted from each class of the three tasks.

B.1 Question Classification

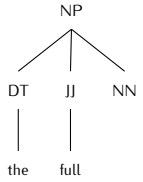
Fragments for Class ABBR



ABBR.14

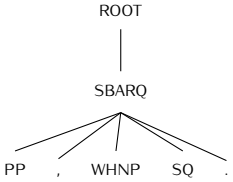


ABBR.15

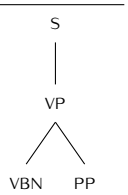


ABBR.16

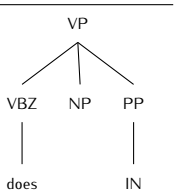
ABBR.17



ABBR.18

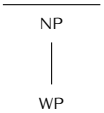


ABBR.19

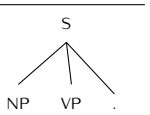


ABBR.20

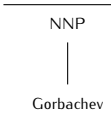
ABBR.21



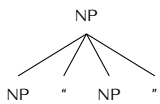
ABBR.22



ABBR.23



ABBR.24

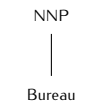


ABBR.25

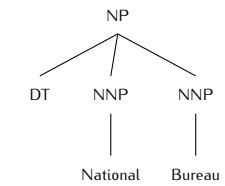
ABBR.26

ABBR.27

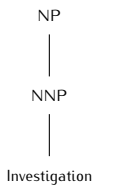
ABBR.28



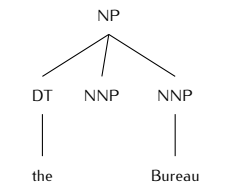
ABBR.29



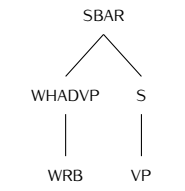
ABBR.30



ABBR.31



ABBR.32

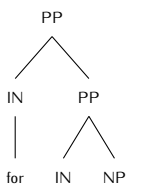


ABBR.33

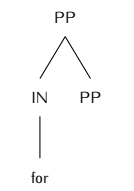
ABBR.34

ABBR.35

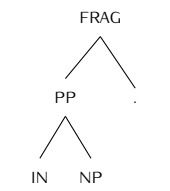
ABBR.36



ABBR.37



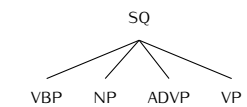
ABBR.38



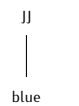
ABBR.39

ABBR.40

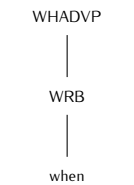
ABBR.41



ABBR.42

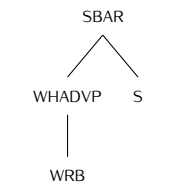


ABBR.43

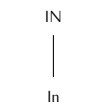


ABBR.44

ABBR.45

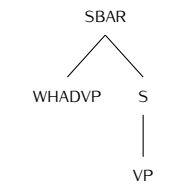


ABBR.46

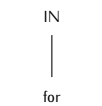


ABBR.47

ABBR.48

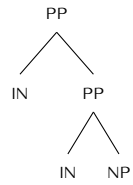


ABBR.49



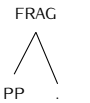
ABBR.50

ABBR.51

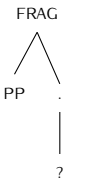


ABBR.52

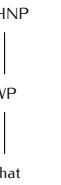
ABBR.53



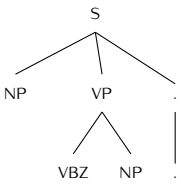
ABBR.54



ABBR.55

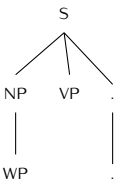


ABBR.56

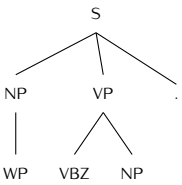


ABBR.57

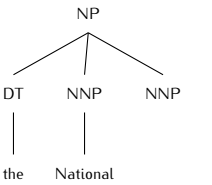
ABBR.58



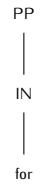
ABBR.59



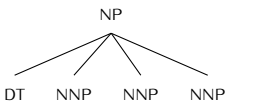
ABBR.60



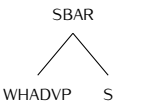
ABBR.61



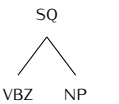
ABBR.62



ABBR.63



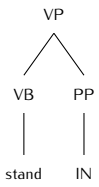
ABBR.64



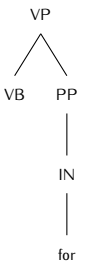
ABBR.65

ABBR.66

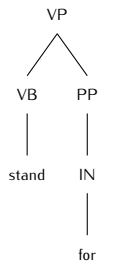
ABBR.67



ABBR.68



ABBR.69

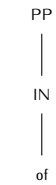


ABBR.70

ABBR.71

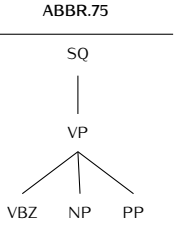
ABBR.72

ABBR.73



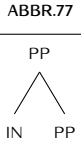
ABBR.74

ABBR.75

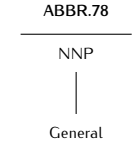


ABBR.76

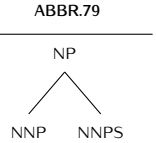
ABBR.77



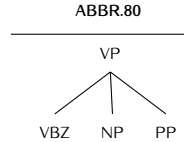
ABBR.78



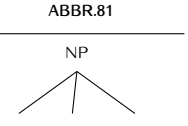
ABBR.79



ABBR.80



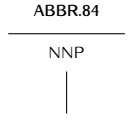
ABBR.81



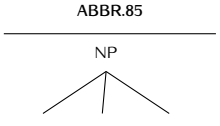
ABBR.82

ABBR.83

ABBR.84

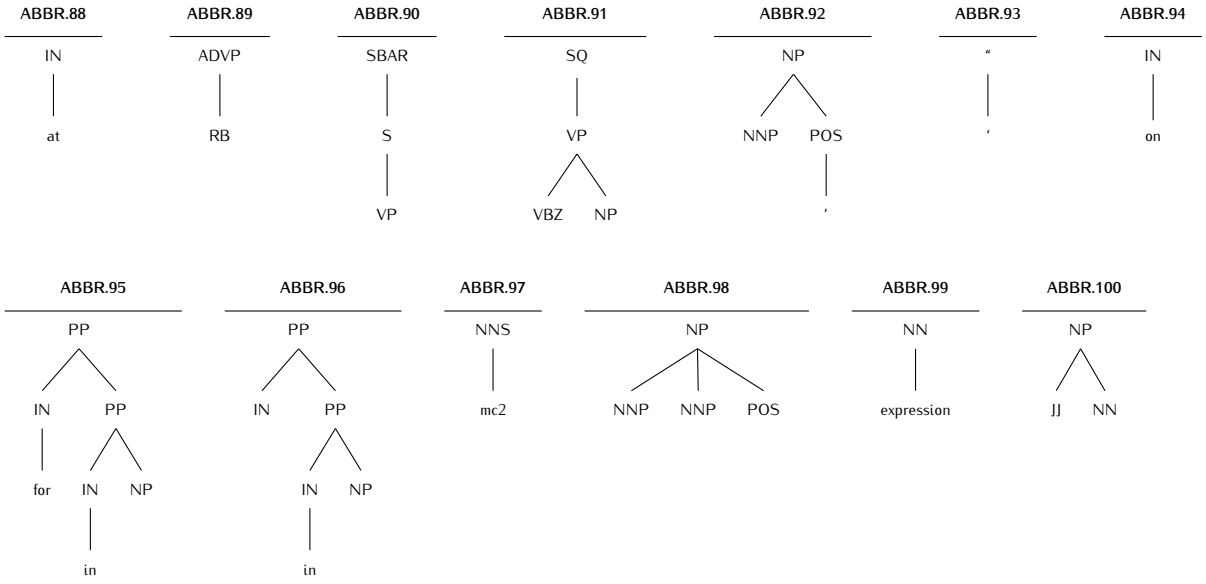


ABBR.85

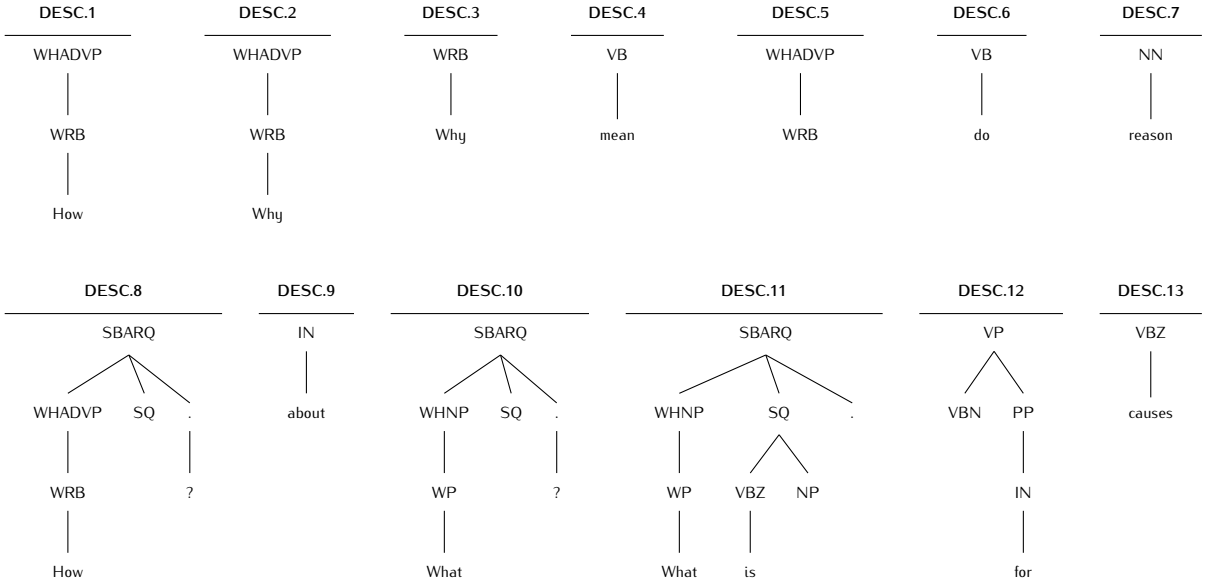


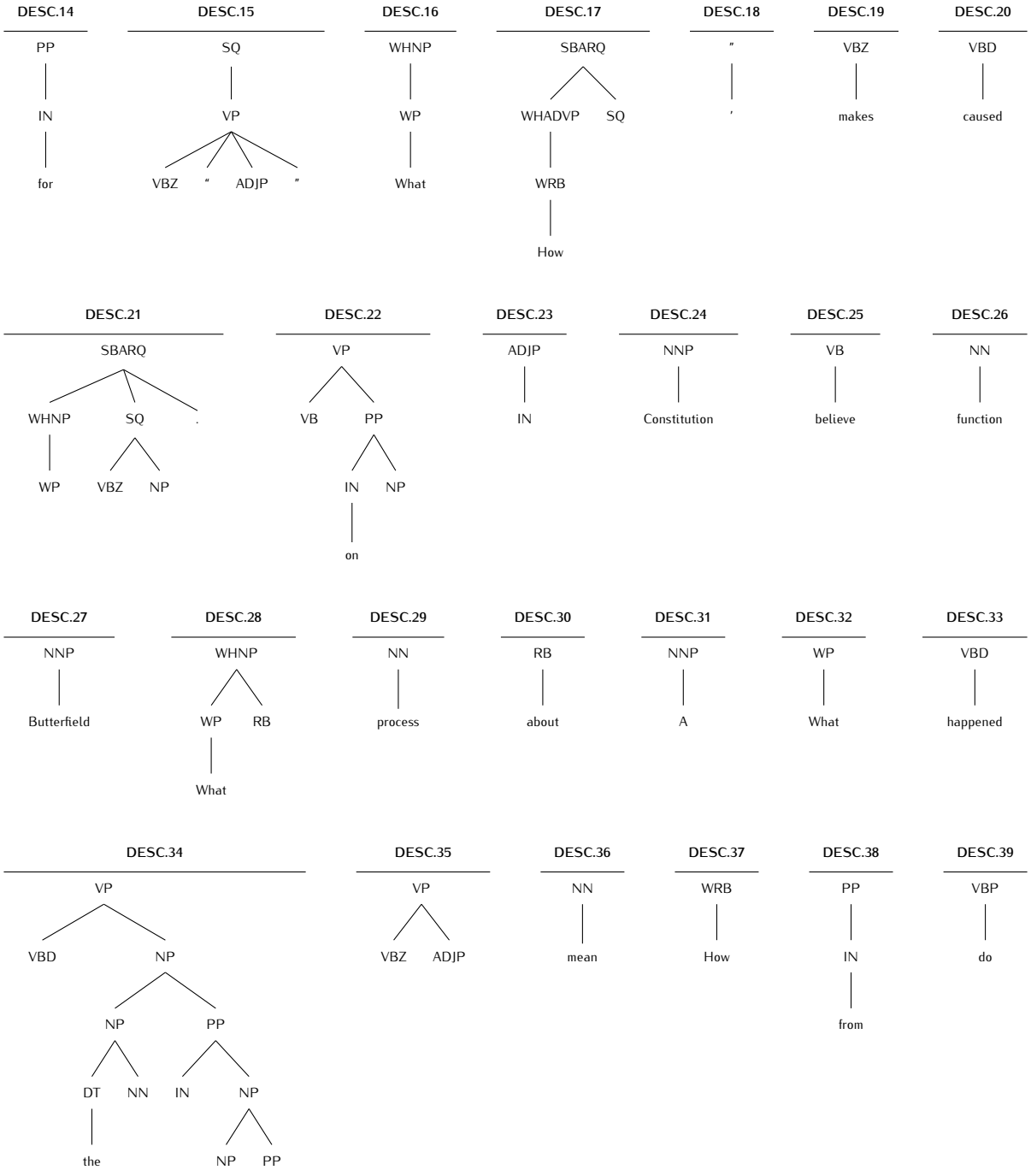
ABBR.86

ABBR.87

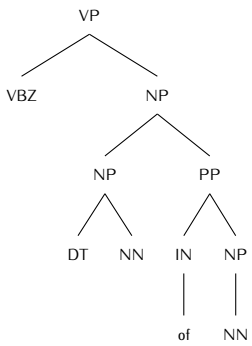


Fragments for Class DESC

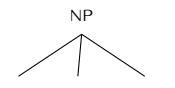



DESC.14
PP
IN
for
DESC.15
SQ
VP
VBZ
“
ADJP
”
DESC.16
WHNP
WP
What
DESC.17
SBARQ
WHADVP
SQ
WRB
How
DESC.18
”
’
DESC.19
VBZ
makes
DESC.20
VBD
caused
DESC.21
SBARQ
WHNP
SQ
.
WP
VBZ
NP
DESC.22
VP
VB
PP
IN
NP
on
DESC.23
ADJP
IN
DESC.24
NNP
Constitution
DESC.25
VB
believe
DESC.26
NN
function
DESC.27
NNP
Butterfield
DESC.28
WHNP
WP
RB
What
DESC.29
NN
process
DESC.30
RB
about
DESC.31
NNP
A
DESC.32
WP
What
DESC.33
VBD
happened
DESC.34
VP
VBD
NP
NP
PP
DT
NN
IN
NP
the
NP
PP
DESC.35
VP
VBZ
ADJP
DESC.36
NN
mean
DESC.37
WRB
How
DESC.38
PP
IN
from
DESC.39
VBP
do

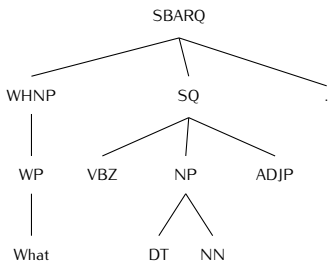
DESC.40



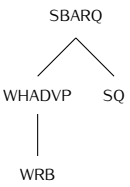
DESC.41



DESC.42

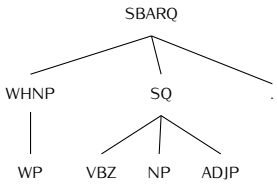


DESC.43

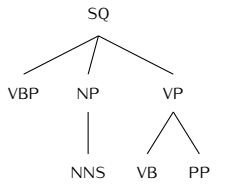


DESC.44

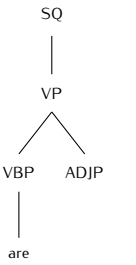
DESC.45



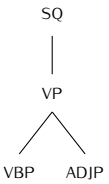
DESC.46



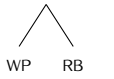
DESC.47



DESC.48

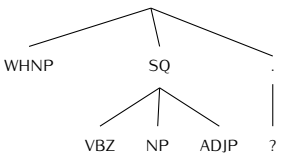


DESC.49



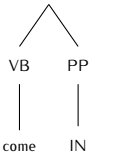
DESC.50

DESC.51

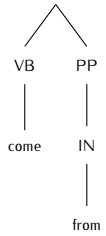


DESC.52

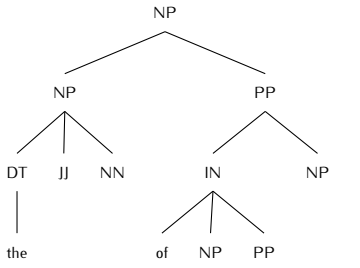
DESC.53



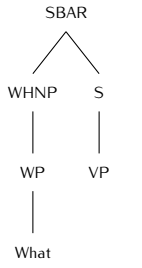
DESC.54



DESC.55

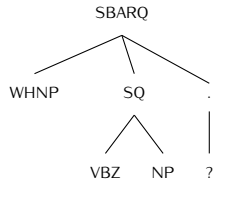


DESC.56



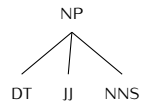
DESC.57

DESC.58



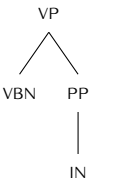
DESC.59

DESC.60



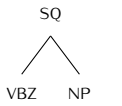
DESC.61

DESC.62



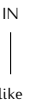
DESC.63

DESC.64

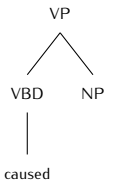


DESC.65

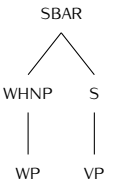
DESC.66



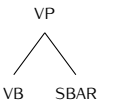
DESC.67



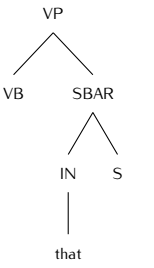
DESC.68



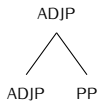
DESC.69



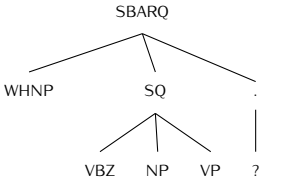
DESC.70



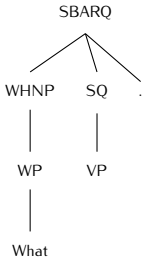
DESC.71



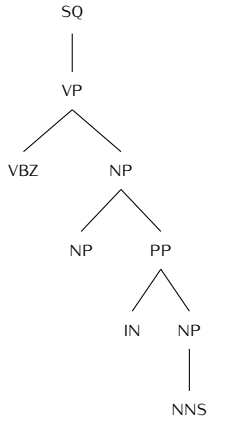
DESC.72



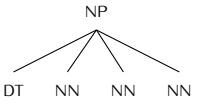
DESC.73



DESC.74

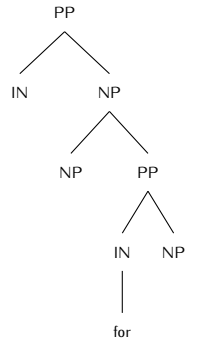


DESC.75



DESC.76

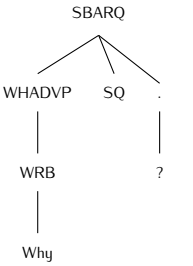
DESC.77



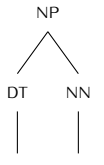
DESC.78

DESC.79

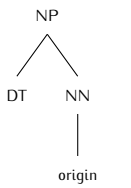
DESC.80



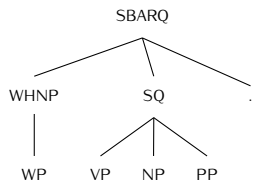
DESC.81



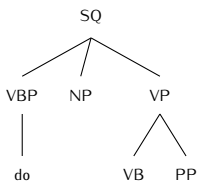
DESC.82



DESC.83

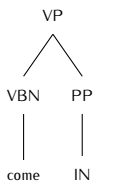


DESC.84

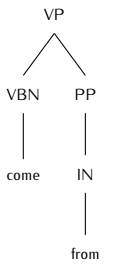


DESC.85

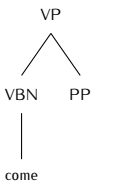
DESC.86



DESC.87

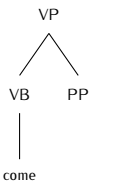


DESC.88

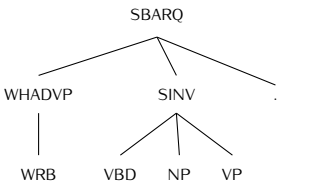


DESC.89

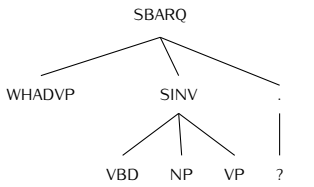
DESC.90



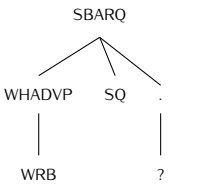
DESC.91



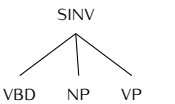
DESC.92



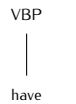
DESC.93



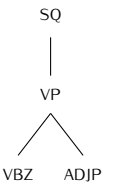
DESC.94



DESC.95

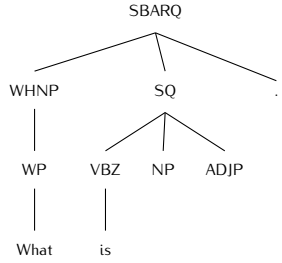


DESC.96

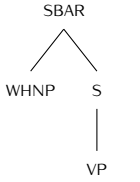


DESC.97

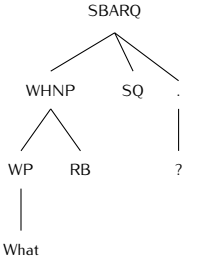
DESC.98



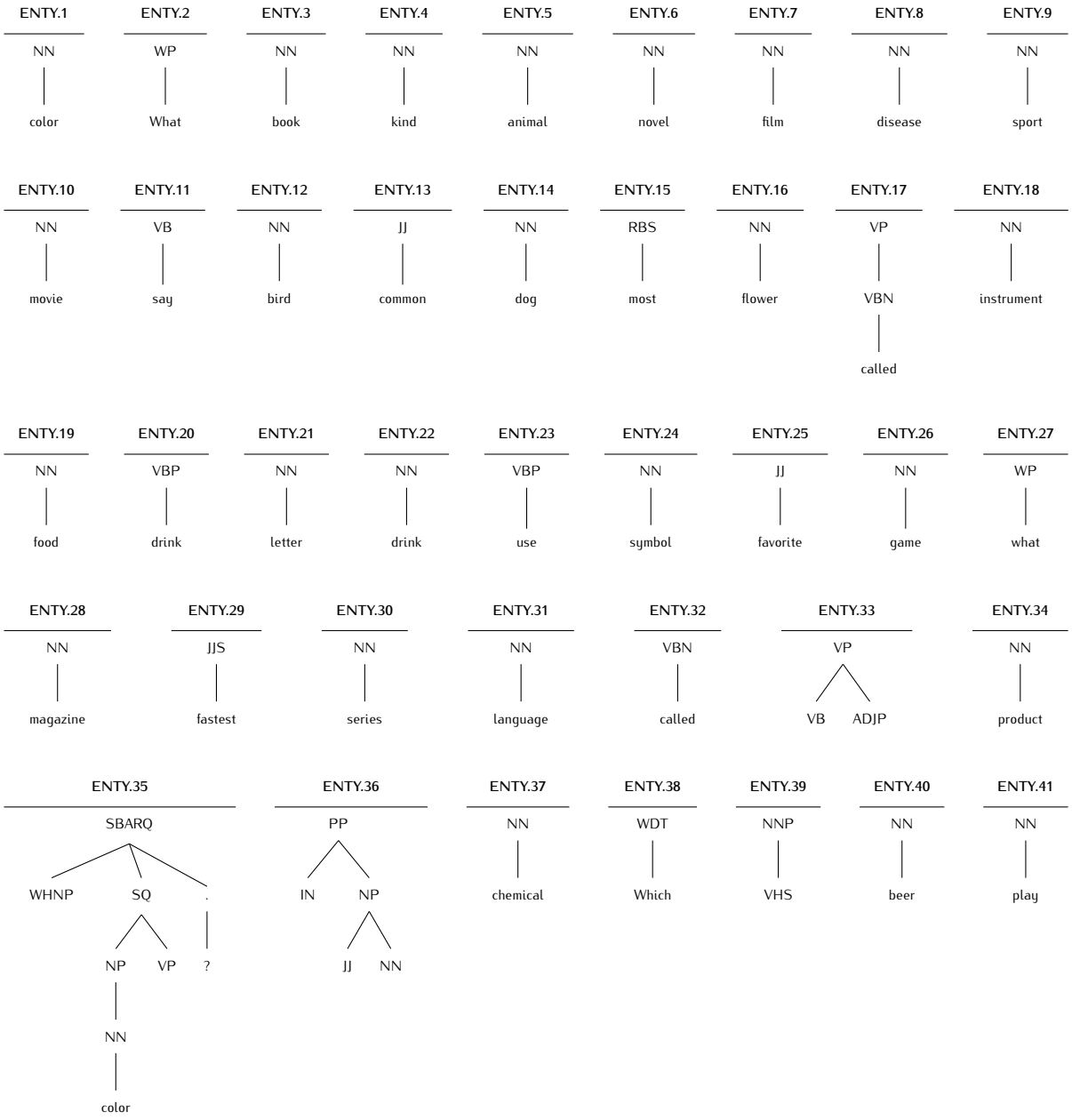
DESC.99

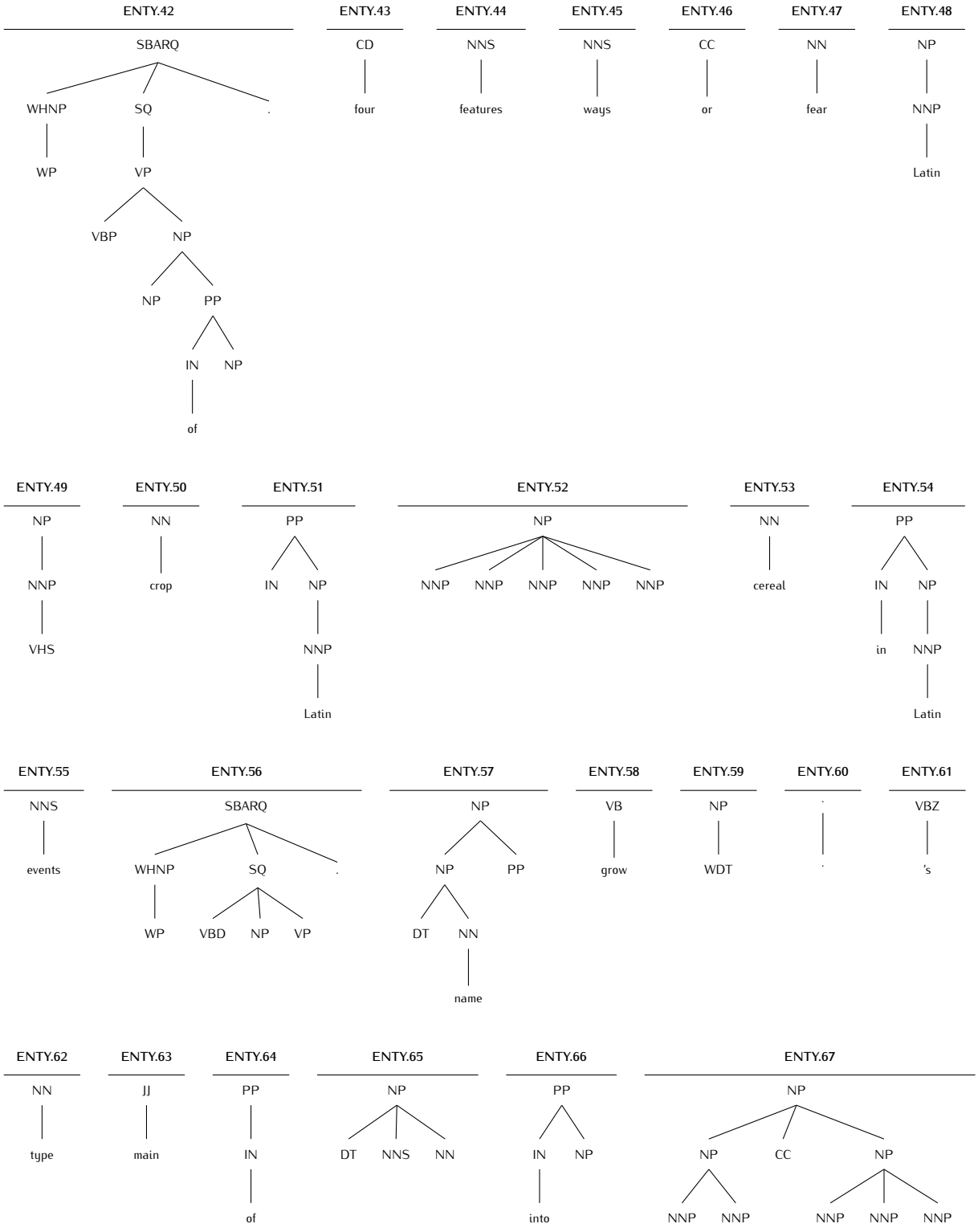


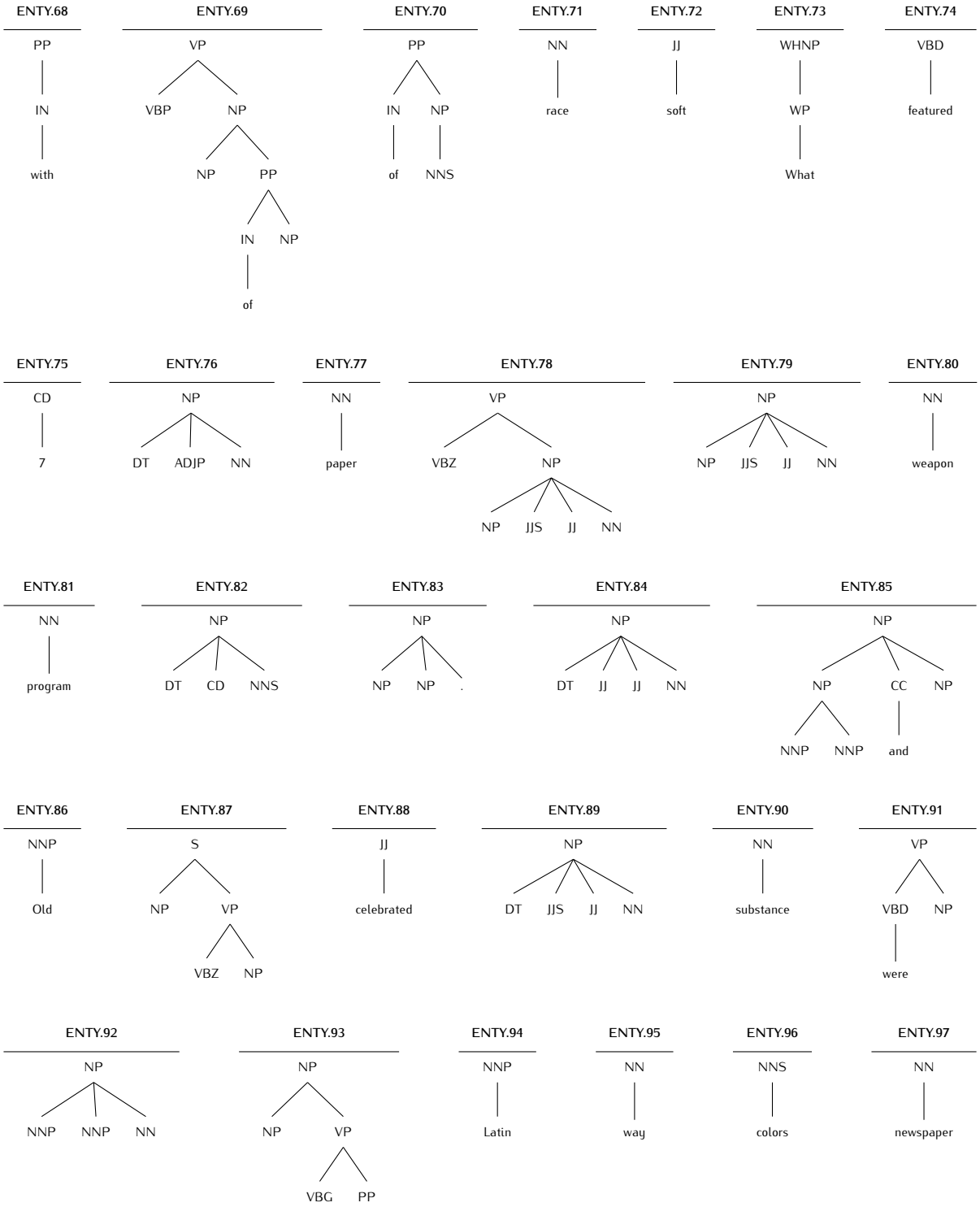
DESC.100

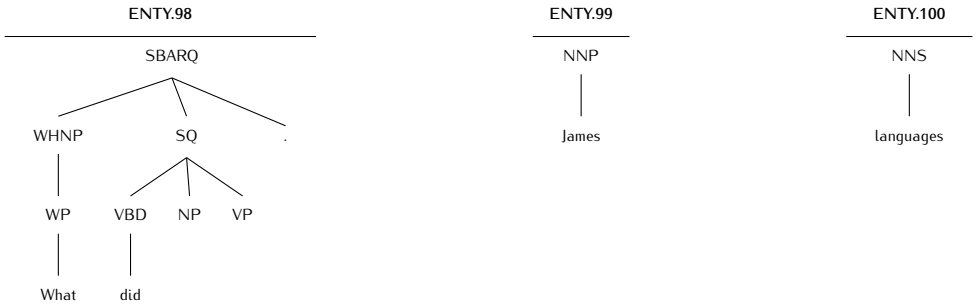


Fragments for Class ENTY

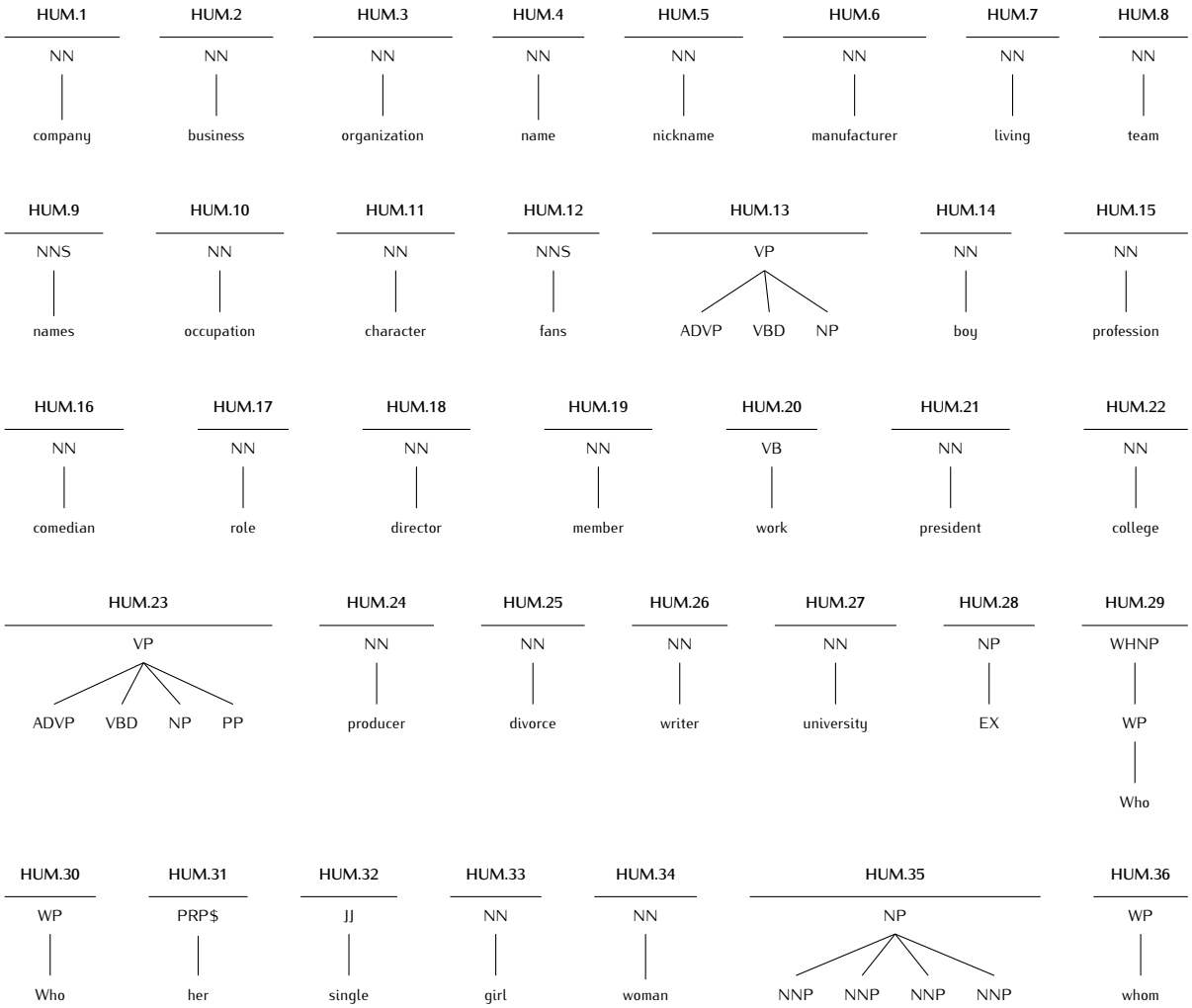


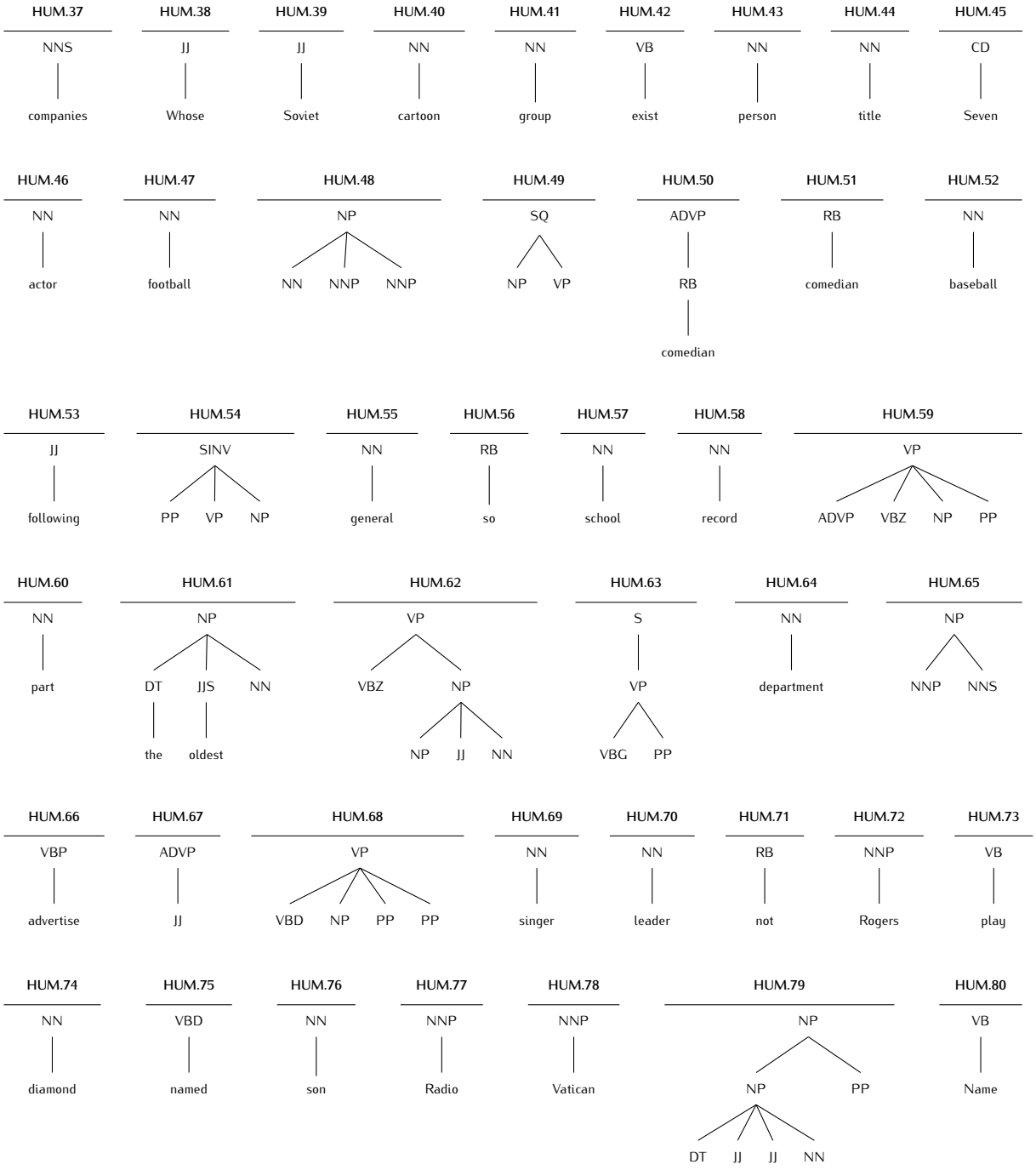

ENTY.42
SBARQ
WHNP
SQ
.
WP
VP
VBP
NP
NP
PP
IN
NP
of
ENTY.43
CD
four
ENTY.44
NNS
features
ENTY.45
NNS
ways
ENTY.46
CC
or
ENTY.47
NN
fear
ENTY.48
NP
NNP
Latin
ENTY.49
NP
NNP
VHS
ENTY.50
NN
crop
ENTY.51
PP
IN
NP
NNP
Latin
ENTY.52
NP
NNP
NNP
NNP
NNP
NNP
ENTY.53
NN
cereal
ENTY.54
PP
IN
NP
in
NNP
Latin
ENTY.55
NNS
events
ENTY.56
SBARQ
WHNP
SQ
.
WP
VBD
NP
VP
ENTY.57
NP
NP
PP
DT
NN
name
ENTY.58
VB
grow
ENTY.59
NP
WDT
ENTY.60
.
.
ENTY.61
VBZ
's
ENTY.62
NN
type
ENTY.63
JJ
main
ENTY.64
PP
IN
of
ENTY.65
NP
DT
NNS
NN
ENTY.66
PP
IN
NP
into
ENTY.67
NP
NP
CC
NP
NNP
NNP
NNP
NNP
NNP

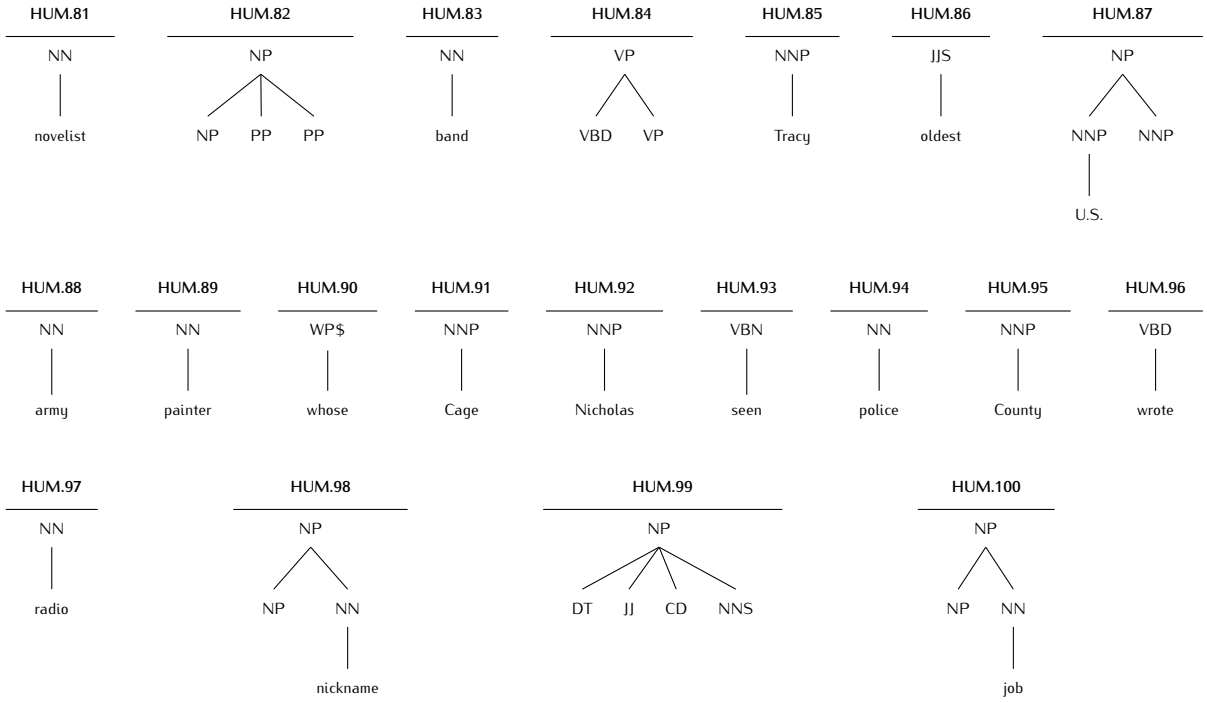

ENTY.68
PP
IN
with
ENTY.69
VP
VBP
NP
NP
PP
IN
NP
of
ENTY.70
PP
IN
NP
of
NNS
ENTY.71
NN
race
ENTY.72
JJ
soft
ENTY.73
WHNP
WP
What
ENTY.74
VBD
featured
ENTY.75
CD
7
ENTY.76
NP
DT
ADJP
NN
ENTY.77
NN
paper
ENTY.78
VP
VBZ
NP
NP
JJS
JJ
NN
ENTY.79
NP
NP
JJS
JJ
NN
ENTY.80
NN
weapon
ENTY.81
NN
program
ENTY.82
NP
DT
CD
NNS
ENTY.83
NP
NP
NP
.
ENTY.84
NP
DT
JJ
JJ
NN
ENTY.85
NP
NP
CC
NP
NNP
NNP
and
ENTY.86
NNP
Old
ENTY.87
S
NP
VP
VBZ
NP
ENTY.88
JJ
celebrated
ENTY.89
NP
DT
JJS
JJ
NN
ENTY.90
NN
substance
ENTY.91
VP
VBD
NP
were
ENTY.92
NP
NNP
NNP
NN
ENTY.93
NP
NP
VP
VBG
PP
ENTY.94
NNP
Latin
ENTY.95
NN
way
ENTY.96
NNS
colors
ENTY.97
NN
newspaper



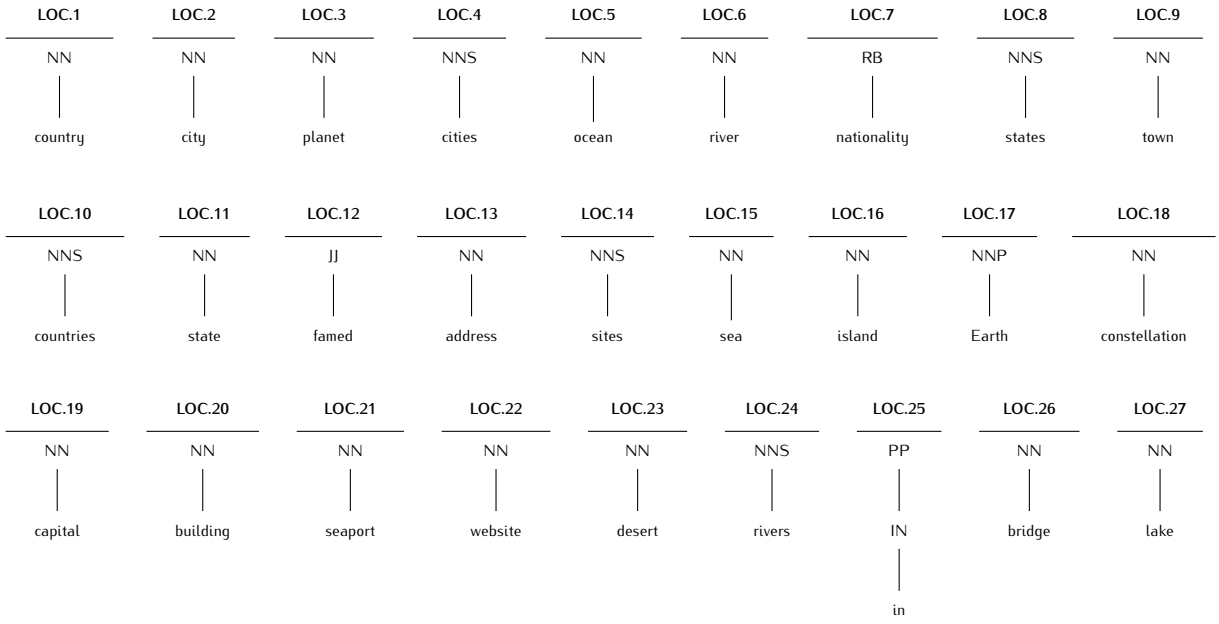
Fragments for Class HUM

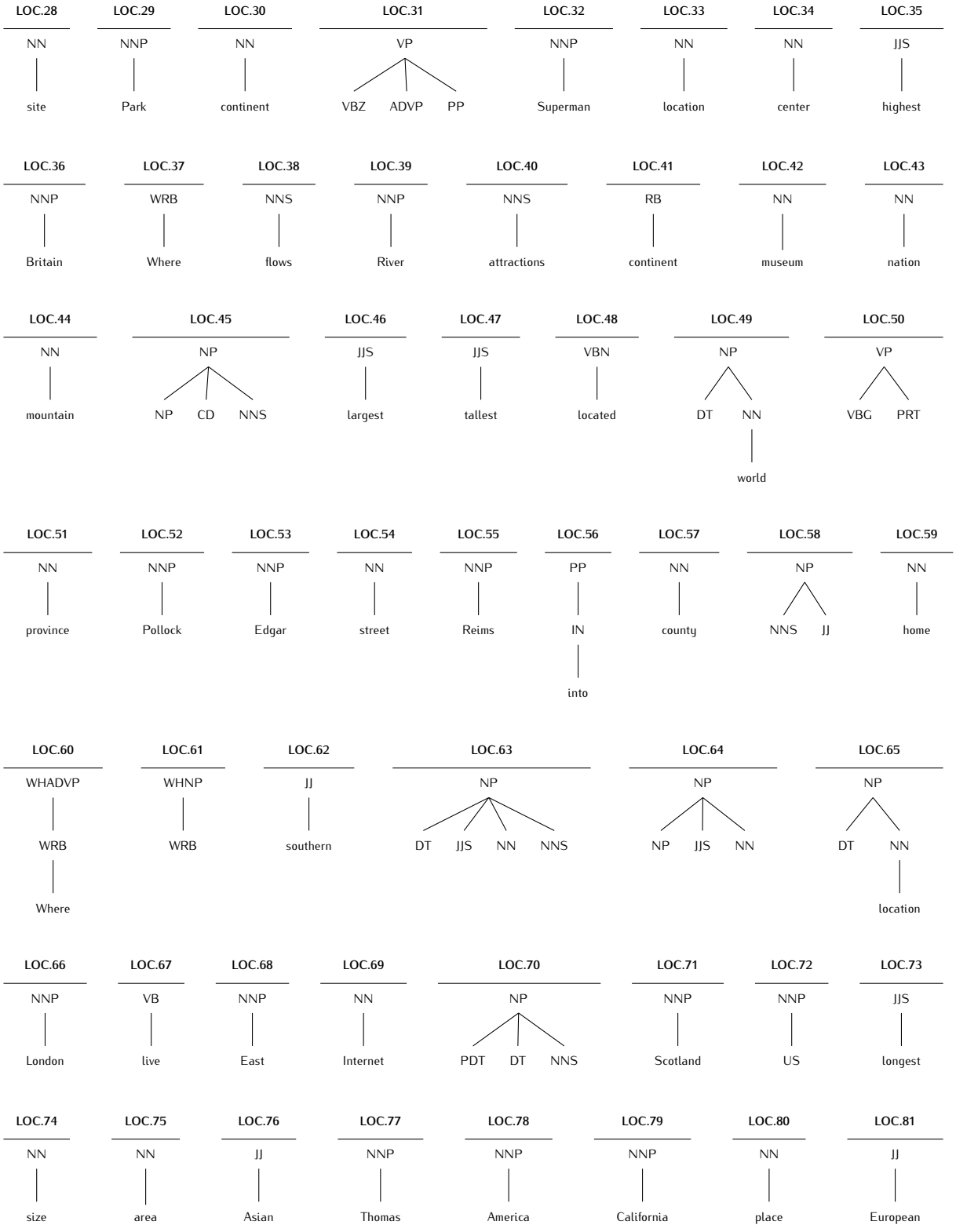


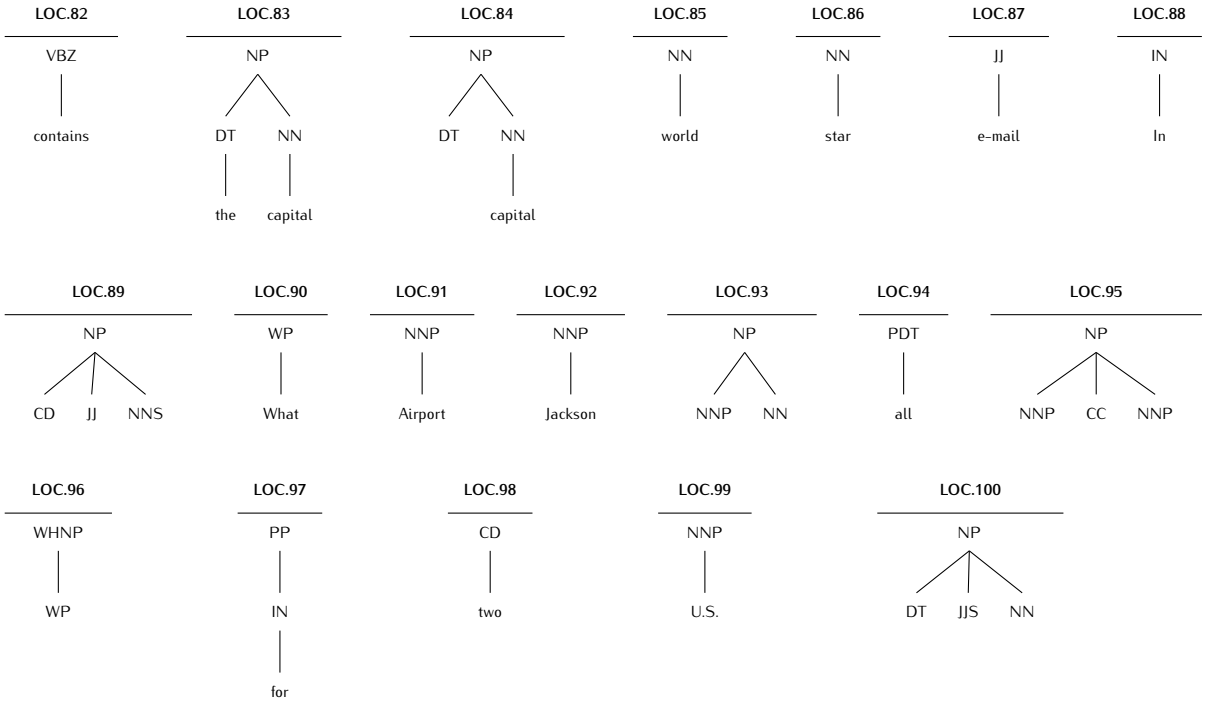

HUM.37
NNS
companies
HUM.38
JJ
Whose
HUM.39
JJ
Soviet
HUM.40
NN
cartoon
HUM.41
NN
group
HUM.42
VB
exist
HUM.43
NN
person
HUM.44
NN
title
HUM.45
CD
Seven
HUM.46
NN
actor
HUM.47
NN
football
HUM.48
NP
NN NNP NNP
HUM.49
SQ
NP VP
HUM.50
ADVP
RB
comedian
HUM.51
RB
comedian
HUM.52
NN
baseball
HUM.53
JJ
following
HUM.54
SINV
PP VP NP
HUM.55
NN
general
HUM.56
RB
so
HUM.57
NN
school
HUM.58
NN
record
HUM.59
VP
ADVP VBZ NP PP
HUM.60
NN
part
HUM.61
NP
DT JJS NN
the oldest
HUM.62
VP
VBZ NP
NP JJ NN
HUM.63
S
VP
VBG PP
HUM.64
NN
department
HUM.65
NP
NNP NNS
HUM.66
VBP
advertise
HUM.67
ADVP
JJ
HUM.68
VP
VBD NP PP PP
HUM.69
NN
singer
HUM.70
NN
leader
HUM.71
RB
not
HUM.72
NNP
Rogers
HUM.73
VB
play
HUM.74
NN
diamond
HUM.75
VBD
named
HUM.76
NN
son
HUM.77
NNP
Radio
HUM.78
NNP
Vatican
HUM.79
NP
NP PP
DT JJ JJ NN
HUM.80
VB
Name



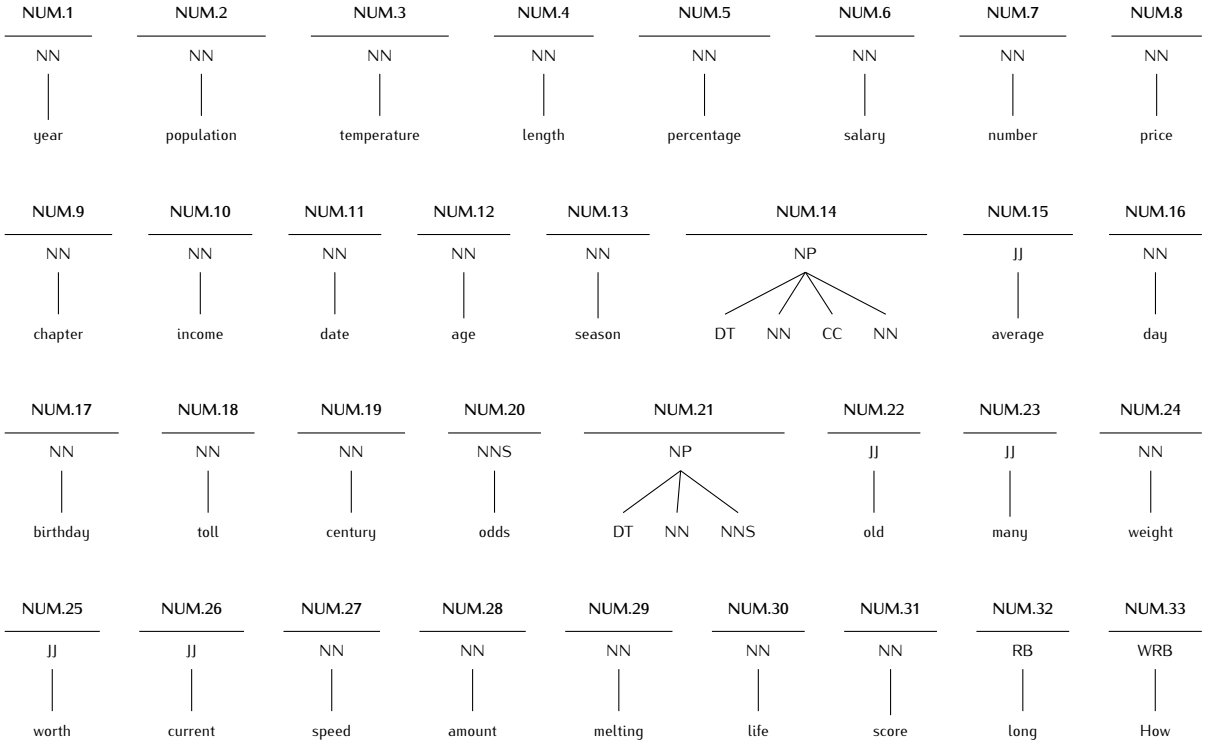
Fragments for Class LOC

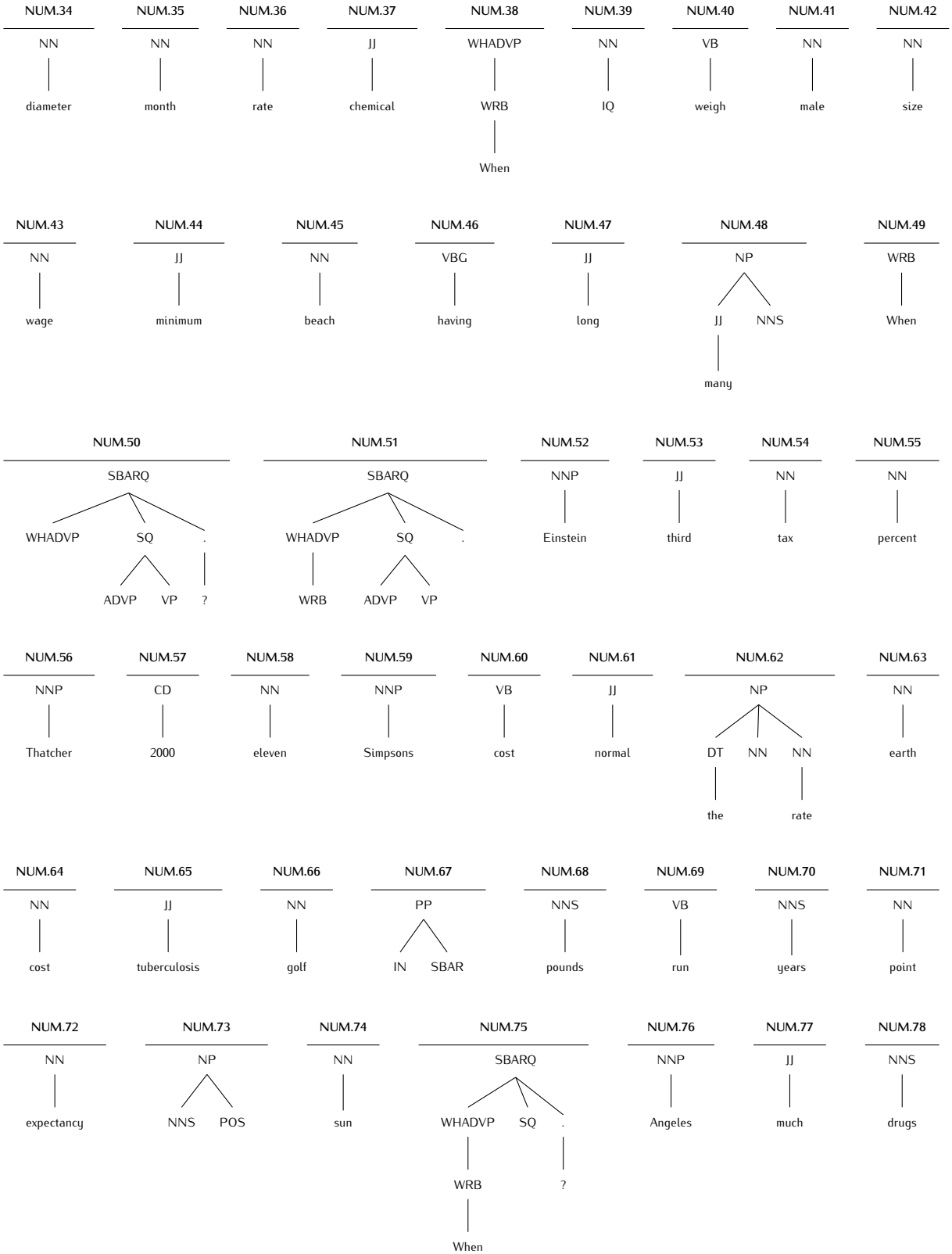


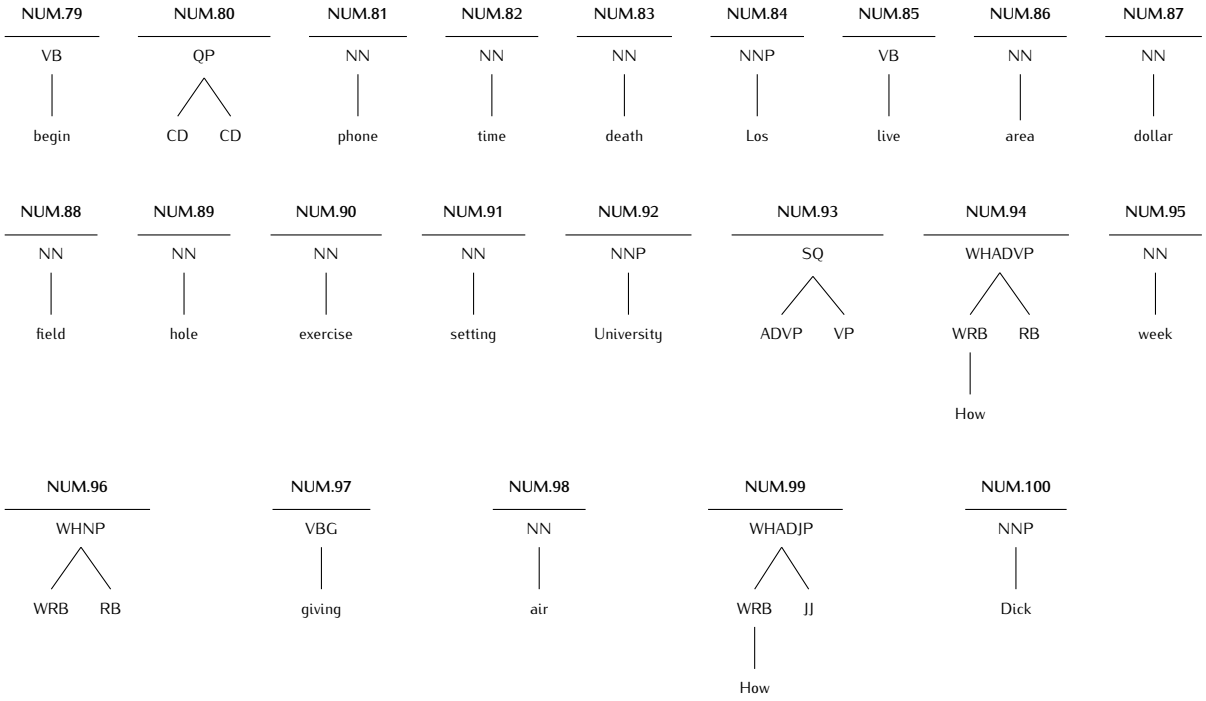

LOC.28
NN
site
LOC.29
NNP
Park
LOC.30
NN
continent
LOC.31
VP
VBZ ADVP PP
LOC.32
NNP
Superman
LOC.33
NN
location
LOC.34
NN
center
LOC.35
JJS
highest
LOC.36
NNP
Britain
LOC.37
WRB
Where
LOC.38
NNS
flows
LOC.39
NNP
River
LOC.40
NNS
attractions
LOC.41
RB
continent
LOC.42
NN
museum
LOC.43
NN
nation
LOC.44
NN
mountain
LOC.45
NP
NP CD NNS
LOC.46
JJS
largest
LOC.47
JJS
tallest
LOC.48
VBN
located
LOC.49
NP
DT NN
world
LOC.50
VP
VBG PRT
LOC.51
NN
province
LOC.52
NNP
Pollock
LOC.53
NNP
Edgar
LOC.54
NN
street
LOC.55
NNP
Reims
LOC.56
PP
IN
into
LOC.57
NN
county
LOC.58
NP
NNS JJ
LOC.59
NN
home
LOC.60
WHADVP
WRB
Where
LOC.61
WHNP
WRB
LOC.62
JJ
southern
LOC.63
NP
DT JJS NN NNS
LOC.64
NP
NP JJS NN
LOC.65
NP
DT NN
location
LOC.66
NNP
London
LOC.67
VB
live
LOC.68
NNP
East
LOC.69
NN
Internet
LOC.70
NP
PDT DT NNS
LOC.71
NNP
Scotland
LOC.72
NNP
US
LOC.73
JJS
longest
LOC.74
NN
size
LOC.75
NN
area
LOC.76
JJ
Asian
LOC.77
NNP
Thomas
LOC.78
NNP
America
LOC.79
NNP
California
LOC.80
NN
place
LOC.81
JJ
European



Fragments for Class NUM

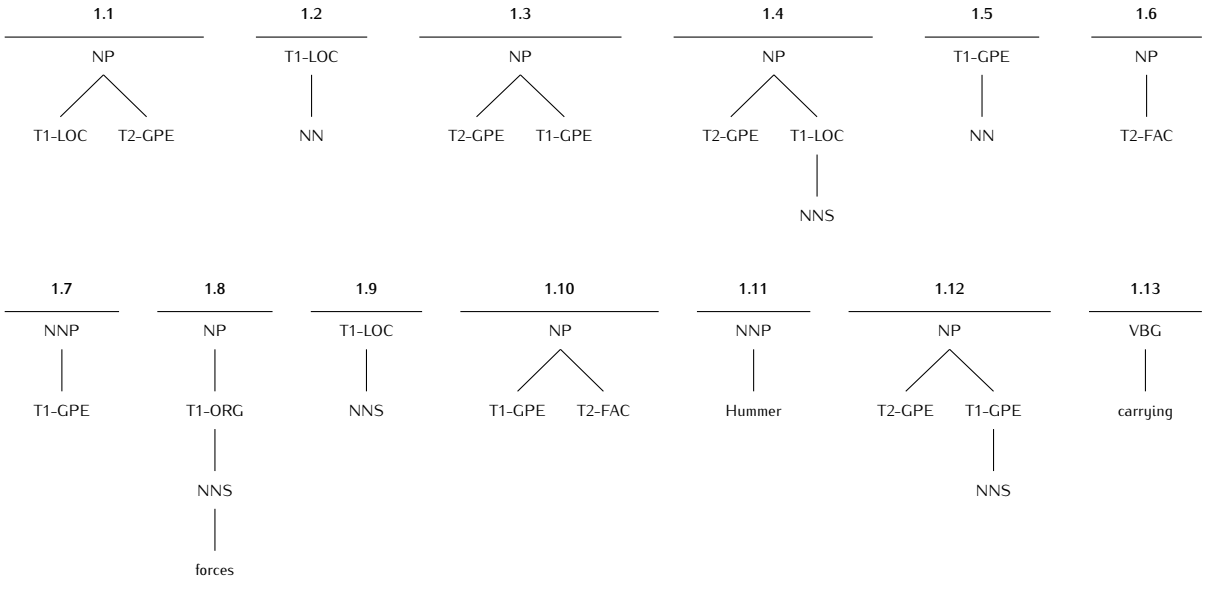


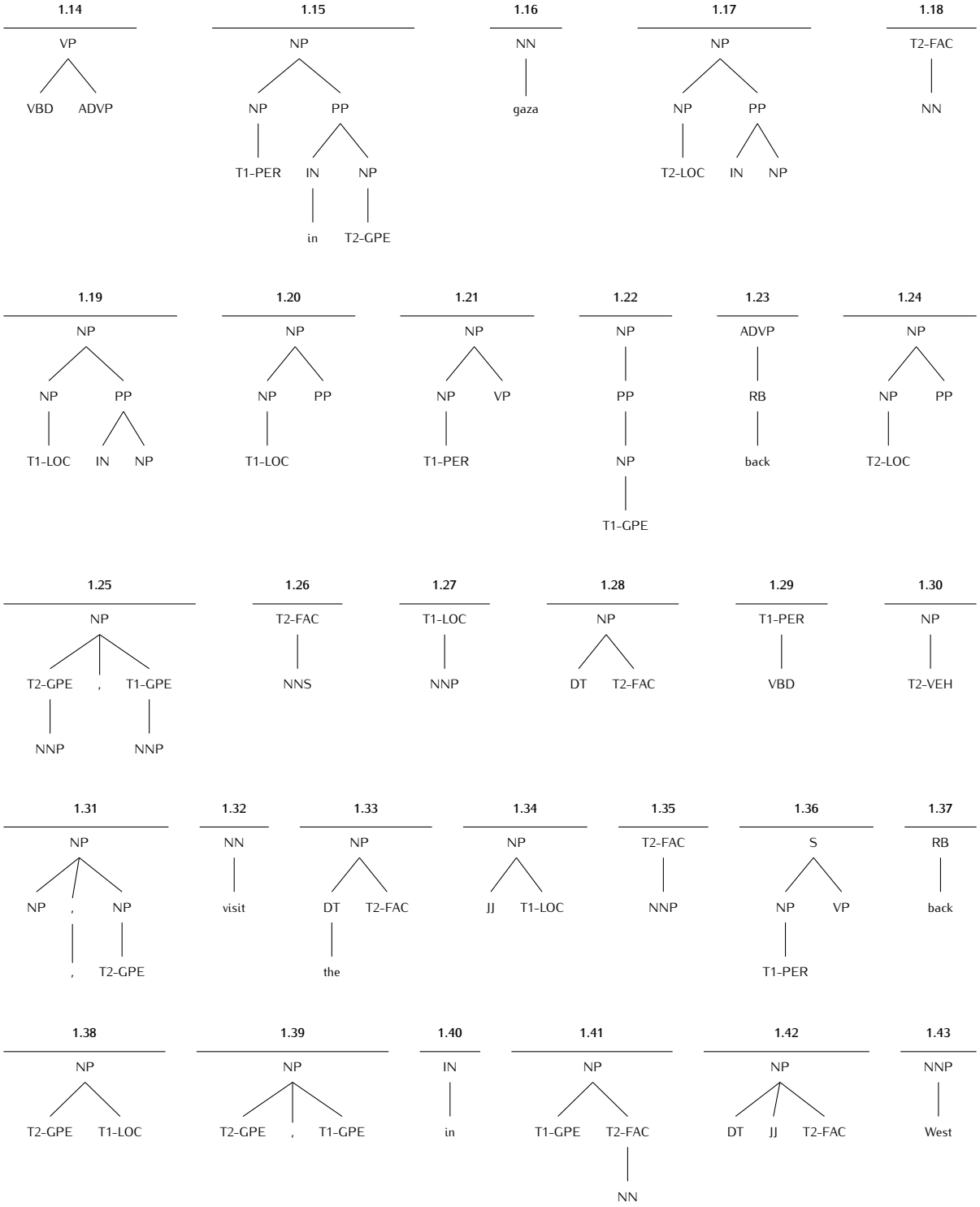

NUM.34
NN
diameter
NUM.35
NN
month
NUM.36
NN
rate
NUM.37
JJ
chemical
NUM.38
WHADVP
WRB
When
NUM.39
NN
IQ
NUM.40
VB
weigh
NUM.41
NN
male
NUM.42
NN
size
NUM.43
NN
wage
NUM.44
JJ
minimum
NUM.45
NN
beach
NUM.46
VBG
having
NUM.47
JJ
long
NUM.48
NP
JJ
NNS
many
NUM.49
WRB
When
NUM.50
SBARQ
WHADVP
SQ
.
ADVP
VP
?
NUM.51
SBARQ
WHADVP
SQ
.
WRB
ADVP
VP
NUM.52
NNP
Einstein
NUM.53
JJ
third
NUM.54
NN
tax
NUM.55
NN
percent
NUM.56
NNP
Thatcher
NUM.57
CD
2000
NUM.58
NN
eleven
NUM.59
NNP
Simpsons
NUM.60
VB
cost
NUM.61
JJ
normal
NUM.62
NP
DT
NN
NN
the
rate
NUM.63
NN
earth
NUM.64
NN
cost
NUM.65
JJ
tuberculosis
NUM.66
NN
golf
NUM.67
PP
IN
SBAR
NUM.68
NNS
pounds
NUM.69
VB
run
NUM.70
NNS
years
NUM.71
NN
point
NUM.72
NN
expectancy
NUM.73
NP
NNS
POS
NUM.74
NN
sun
NUM.75
SBARQ
WHADVP
SQ
.
WRB
?
When
NUM.76
NNP
Angeles
NUM.77
JJ
much
NUM.78
NNS
drugs

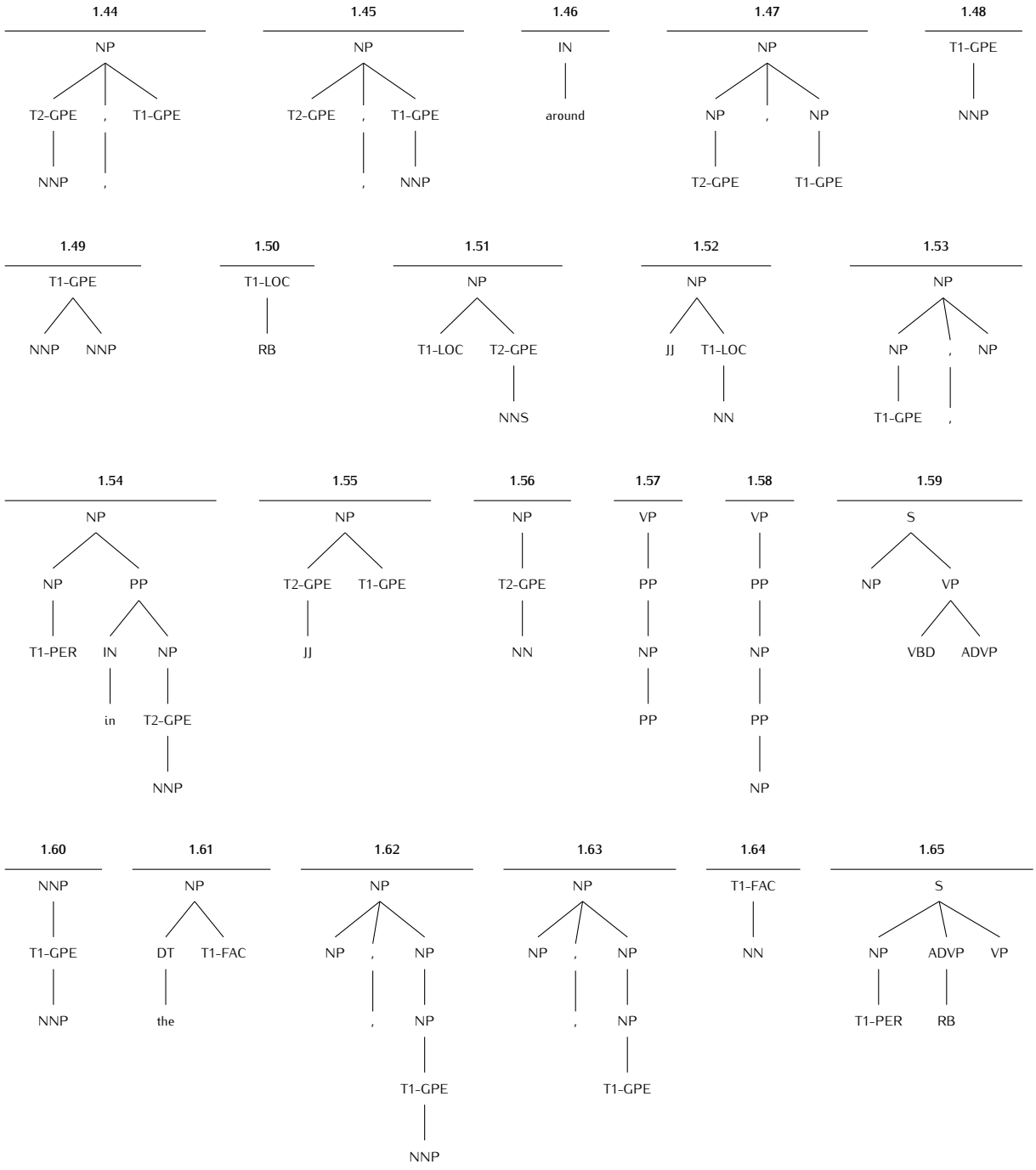


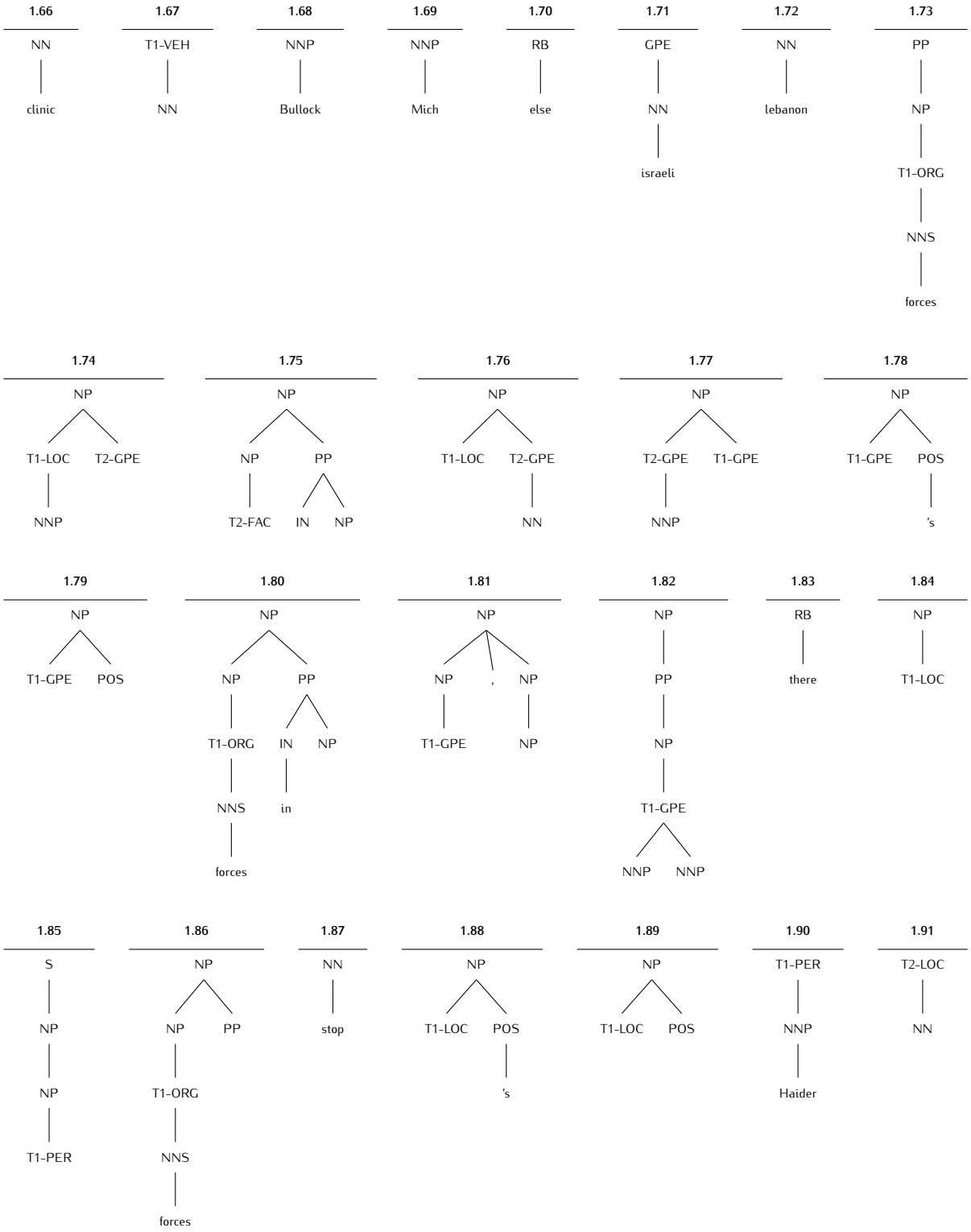
B.2 Relation Extraction

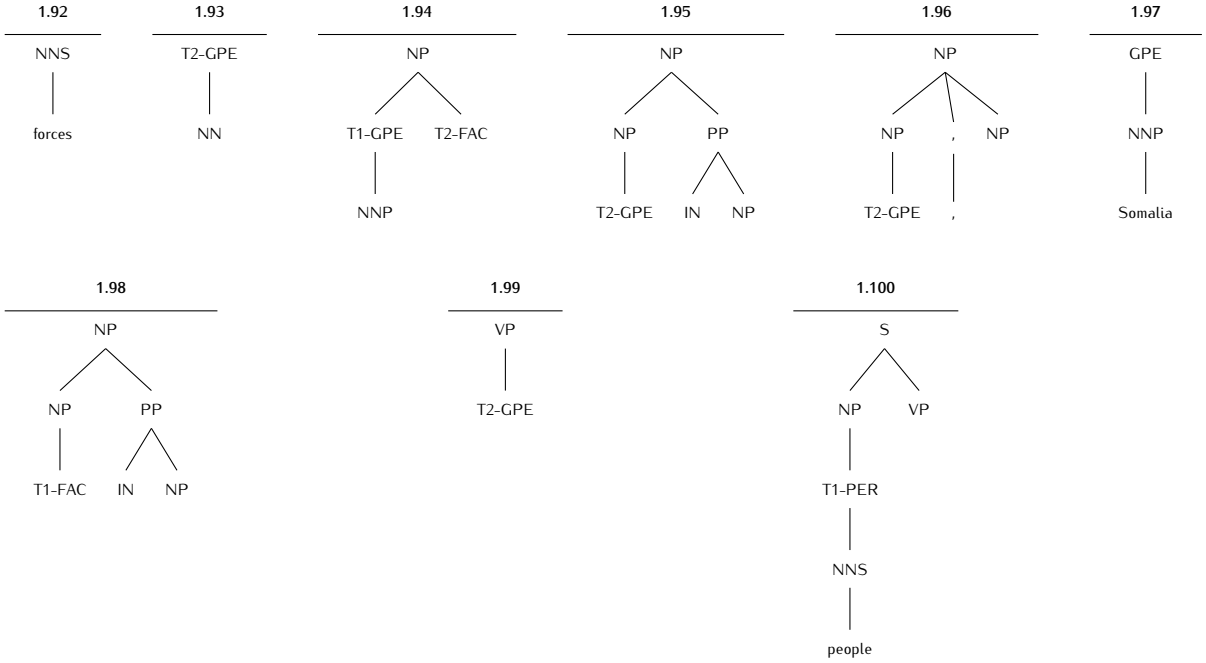
Fragments for Class 1



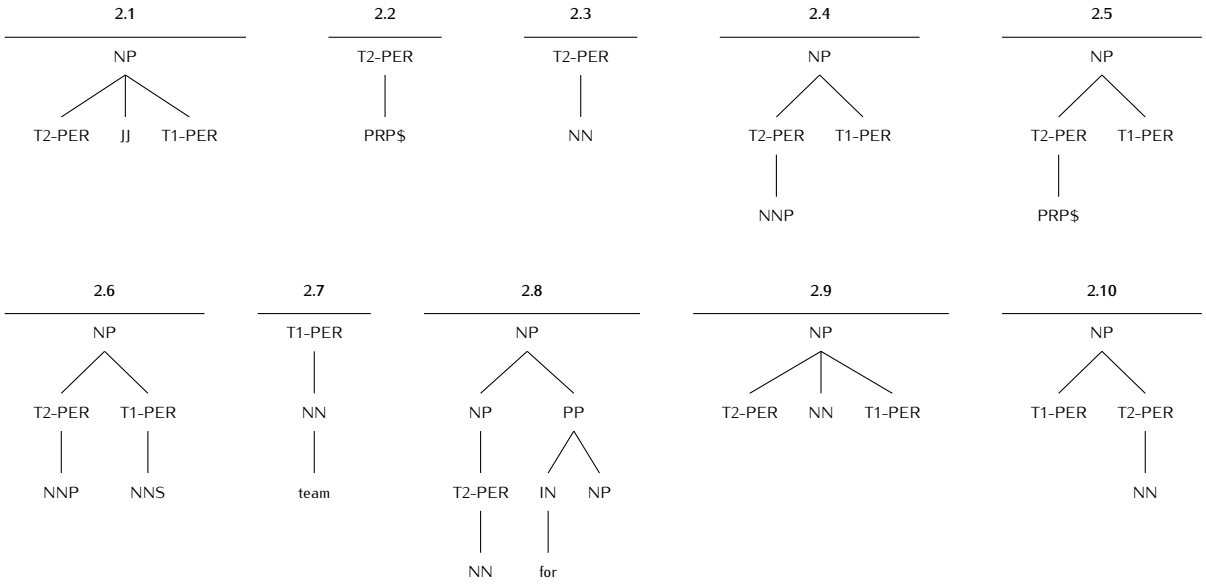

1.14
VP
VBD ADVP
1.15
NP
NP PP
T1-PER IN NP
in T2-GPE
1.16
NN
gaza
1.17
NP
NP PP
T2-LOC IN NP
1.18
T2-FAC
NN
1.19
NP
NP PP
T1-LOC IN NP
1.20
NP
NP PP
T1-LOC
1.21
NP
NP VP
T1-PER
1.22
NP
PP
NP
T1-GPE
1.23
ADVP
RB
back
1.24
NP
NP PP
T2-LOC
1.25
NP
T2-GPE , T1-GPE
NNP NNP
1.26
T2-FAC
NNS
1.27
T1-LOC
NNP
1.28
NP
DT T2-FAC
1.29
T1-PER
VBD
1.30
NP
T2-VEH
1.31
NP
NP , NP
, T2-GPE
1.32
NN
visit
1.33
NP
DT T2-FAC
the
1.34
NP
JJ T1-LOC
1.35
T2-FAC
NNP
1.36
S
NP VP
T1-PER
1.37
RB
back
1.38
NP
T2-GPE T1-LOC
1.39
NP
T2-GPE , T1-GPE
1.40
IN
in
1.41
NP
T1-GPE T2-FAC
NN
1.42
NP
DT JJ T2-FAC
1.43
NNP
West


1.44
NP
T2-GPE , T1-GPE
NNP ,
1.45
NP
T2-GPE , T1-GPE
, NNP
1.46
IN
around
1.47
NP
NP , NP
T2-GPE T1-GPE
1.48
T1-GPE
NNP
1.49
T1-GPE
NNP NNP
1.50
T1-LOC
RB
1.51
NP
T1-LOC T2-GPE
NNS
1.52
NP
JJ T1-LOC
NN
1.53
NP
NP , NP
T1-GPE ,
1.54
NP
NP PP
T1-PER IN NP
in T2-GPE
NNP
1.55
NP
T2-GPE T1-GPE
JJ
1.56
NP
T2-GPE
NN
1.57
VP
PP
NP
PP
1.58
VP
PP
NP
PP
NP
1.59
S
NP VP
VBD ADVP
1.60
NNP
T1-GPE
NNP
1.61
NP
DT T1-FAC
the
1.62
NP
NP , NP
, NP
T1-GPE
NNP
1.63
NP
NP , NP
, NP
T1-GPE
1.64
T1-FAC
NN
1.65
S
NP ADVP VP
T1-PER RB

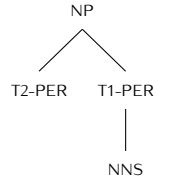

1.66
NN
clinic
1.67
T1-VEH
NN
1.68
NNP
Bullock
1.69
NNP
Mich
1.70
RB
else
1.71
GPE
NN
israeli
1.72
NN
lebanon
1.73
PP
NP
T1-ORG
NNS
forces
1.74
NP
T1-LOC
T2-GPE
NNP
1.75
NP
NP
PP
T2-FAC
IN
NP
1.76
NP
T1-LOC
T2-GPE
NN
1.77
NP
T2-GPE
T1-GPE
NNP
1.78
NP
T1-GPE
POS
's
1.79
NP
T1-GPE
POS
1.80
NP
NP
PP
T1-ORG
IN
NP
NNS
in
forces
1.81
NP
NP
,
NP
T1-GPE
NP
1.82
NP
PP
NP
T1-GPE
NNP
NNP
1.83
RB
there
1.84
NP
T1-LOC
1.85
S
NP
NP
T1-PER
1.86
NP
NP
PP
T1-ORG
NNS
forces
1.87
NN
stop
1.88
NP
T1-LOC
POS
's
1.89
NP
T1-LOC
POS
1.90
T1-PER
NNP
Haider
1.91
T2-LOC
NN



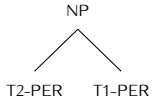
Fragments for Class 2



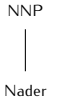
2.11



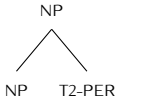
2.12



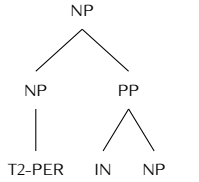
2.13



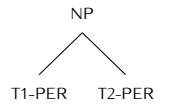
2.14



2.15



2.16

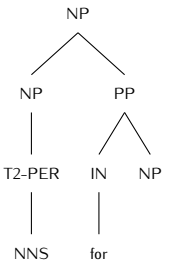


2.17

2.18

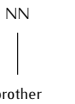
2.19

2.20

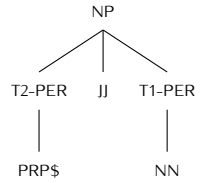


2.21

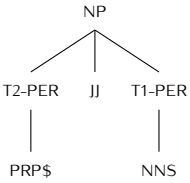
2.22



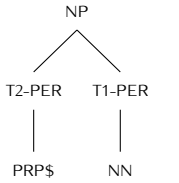
2.23



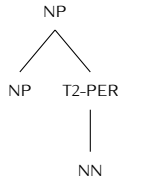
2.24



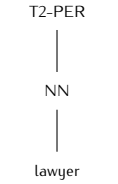
2.25



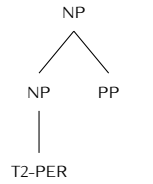
2.26



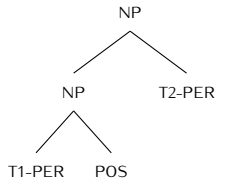
2.27



2.28

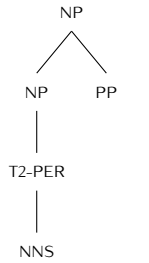


2.29

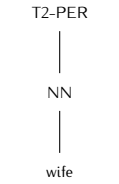


2.30

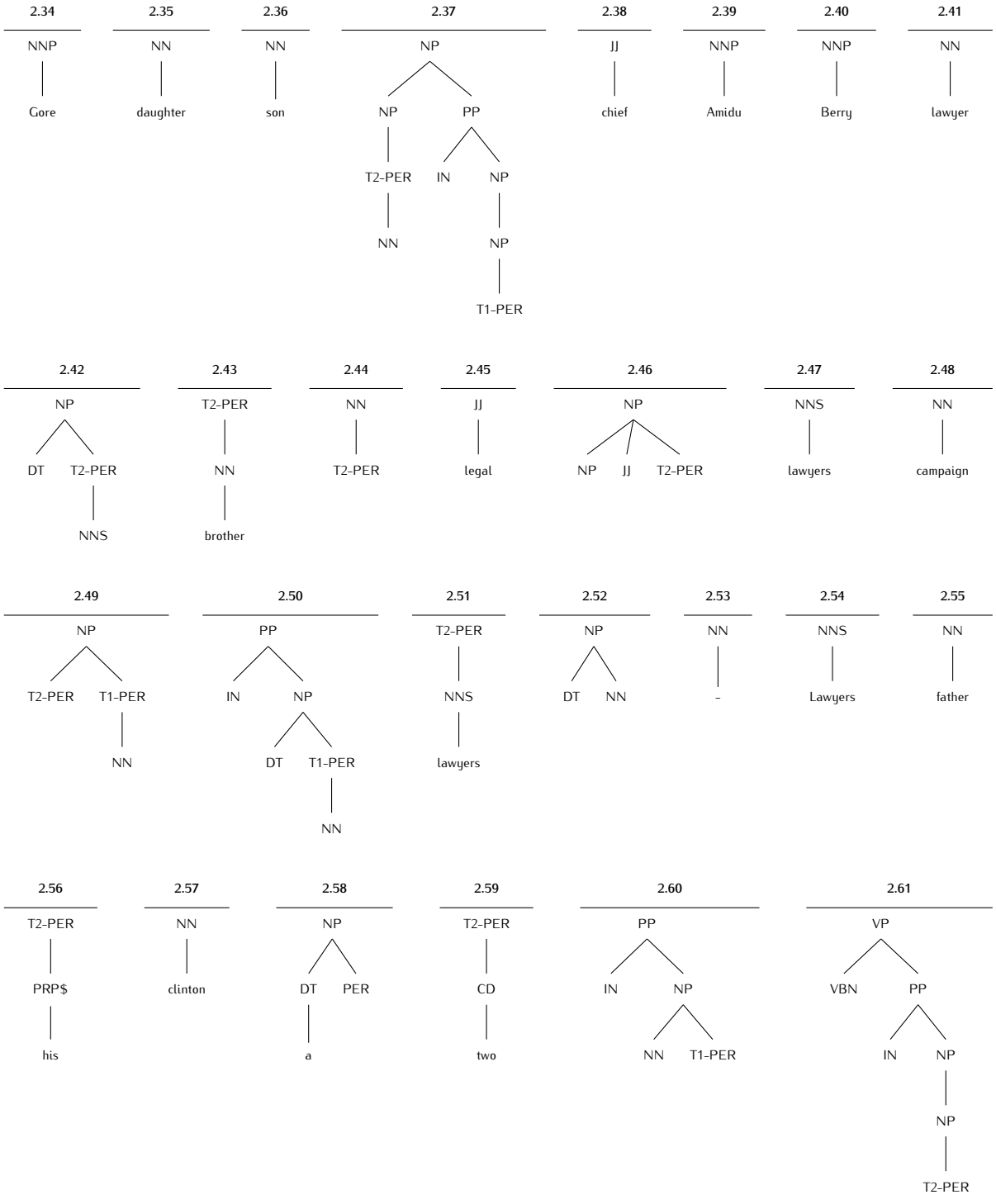
2.31



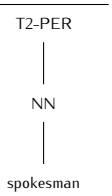
2.32



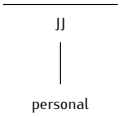
2.33


2.34
NNP
Gore
2.35
NN
daughter
2.36
NN
son
2.37
NP
NP
PP
T2-PER
IN
NP
NN
NP
T1-PER
2.38
JJ
chief
2.39
NNP
Amidu
2.40
NNP
Berry
2.41
NN
lawyer
2.42
NP
DT
T2-PER
NNS
2.43
T2-PER
NN
brother
2.44
NN
T2-PER
2.45
JJ
legal
2.46
NP
NP
JJ
T2-PER
2.47
NNS
lawyers
2.48
NN
campaign
2.49
NP
T2-PER
T1-PER
NN
2.50
PP
IN
NP
DT
T1-PER
NN
2.51
T2-PER
NNS
lawyers
2.52
NP
DT
NN
2.53
NN
-
2.54
NNS
Lawyers
2.55
NN
father
2.56
T2-PER
PRP$
his
2.57
NN
clinton
2.58
NP
DT
PER
a
2.59
T2-PER
CD
two
2.60
PP
IN
NP
NN
T1-PER
2.61
VP
VBN
PP
IN
NP
NP
T2-PER

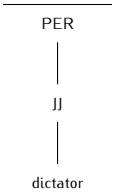
2.62



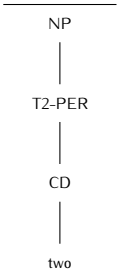
2.63



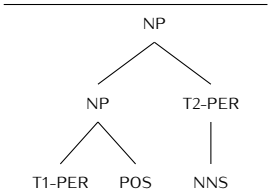
2.64



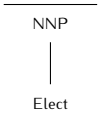
2.65



2.66

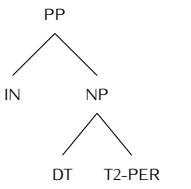


2.67

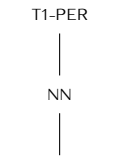


2.68

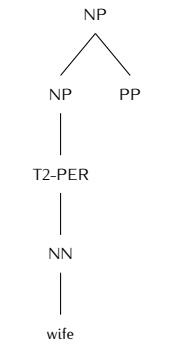
2.69



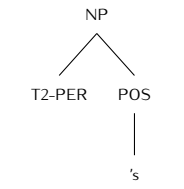
2.70



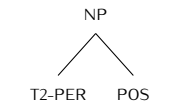
2.71



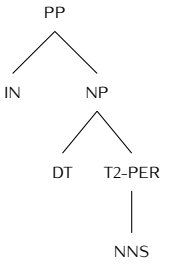
2.72



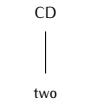
2.73



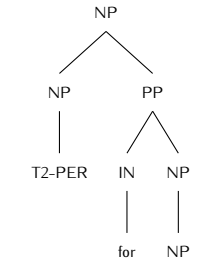
2.74



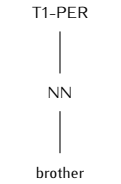
2.75



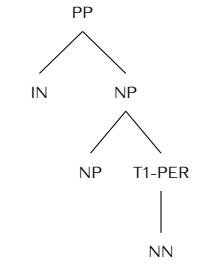
2.76



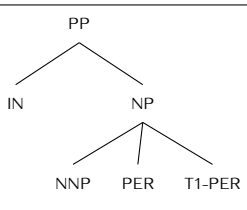
2.77



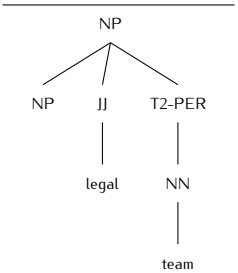
2.78



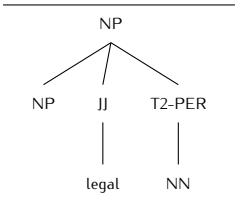
2.79



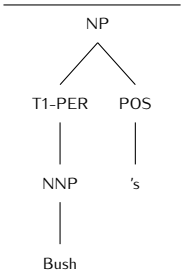
2.80

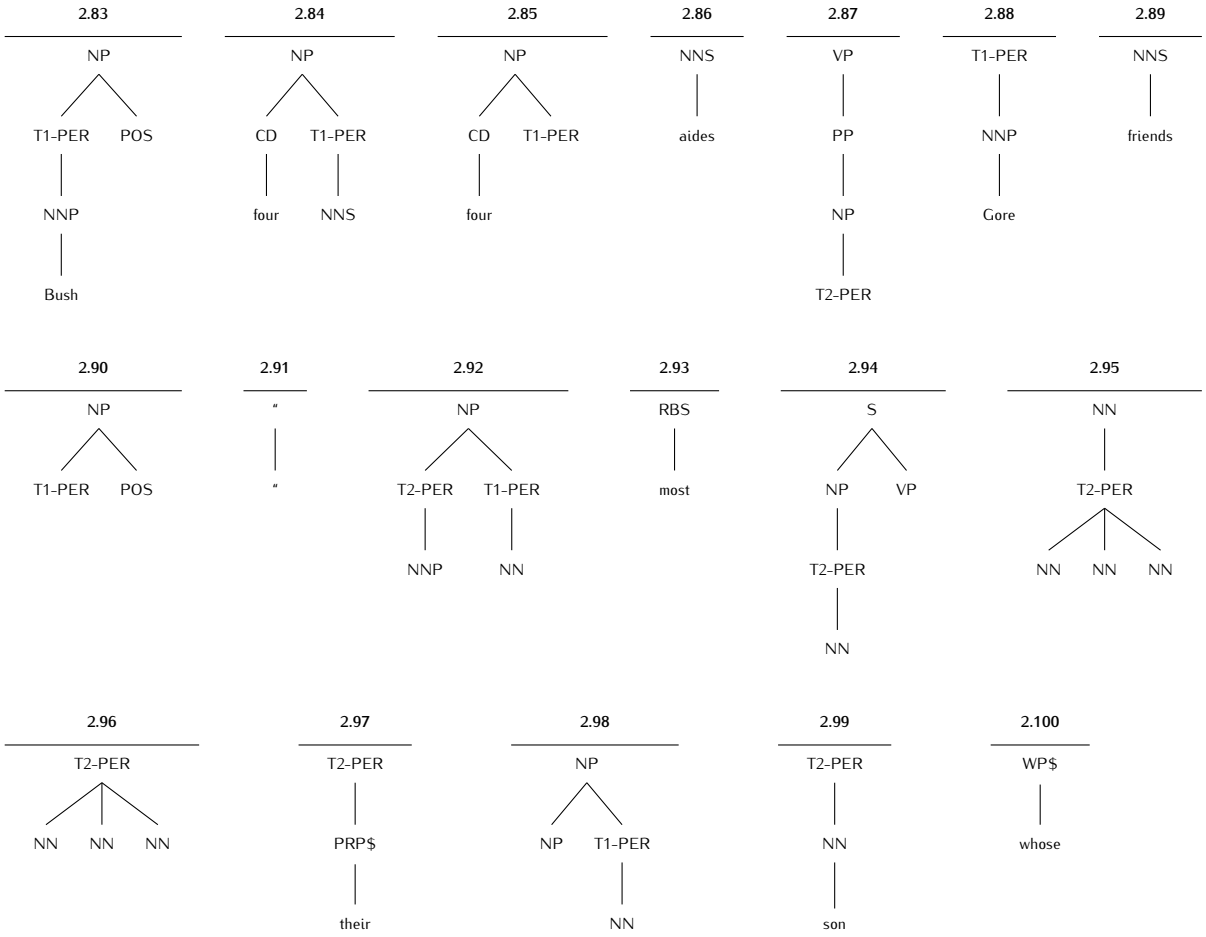


2.81

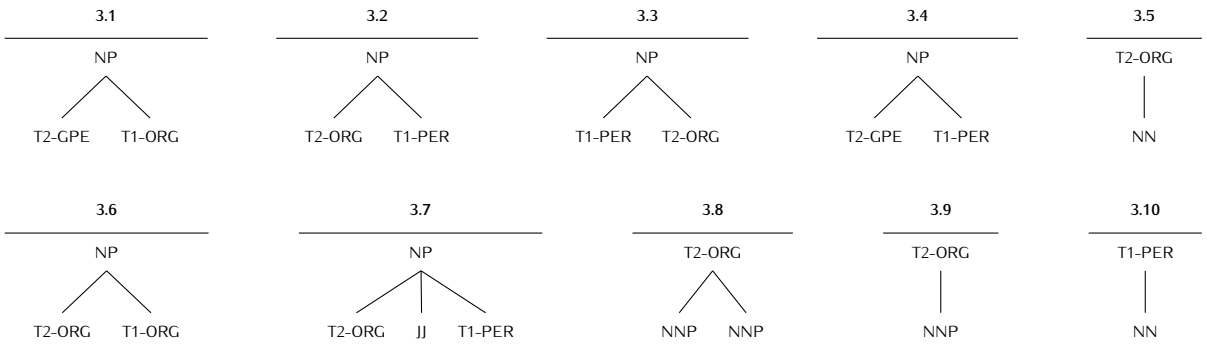


2.82

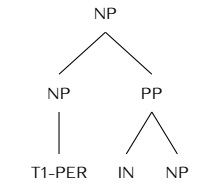




Fragments for Class 3

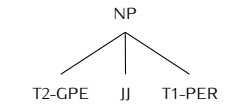


3.11



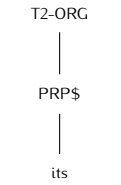
3.12

3.13

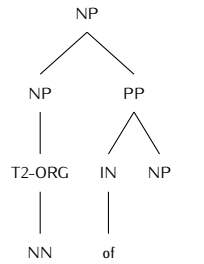


3.14

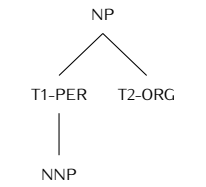
3.15



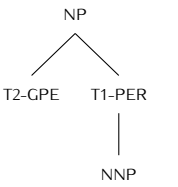
3.16



3.17



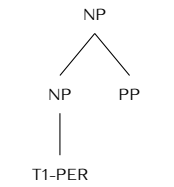
3.18



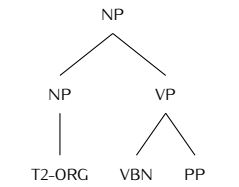
3.19

3.20

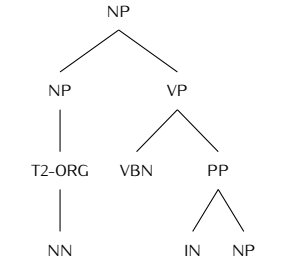
3.21



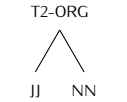
3.22



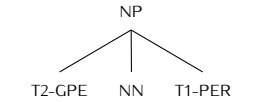
3.23



3.24

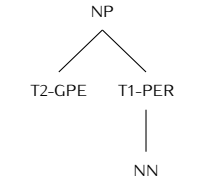


3.25

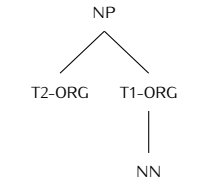


3.26

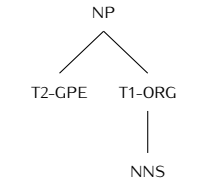
3.27



3.28

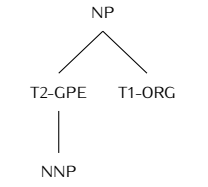


3.29

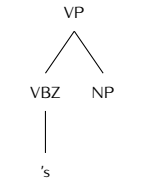


3.30

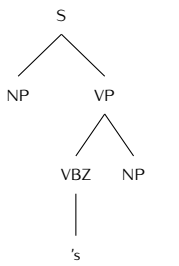
3.31



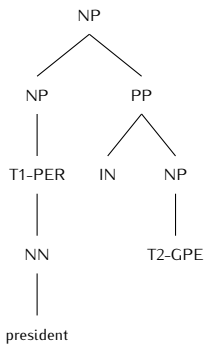
3.32



3.33

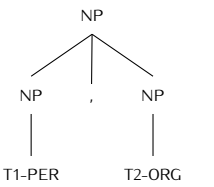


3.34



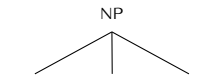
3.35

3.36



3.37

3.38

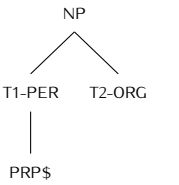


3.39

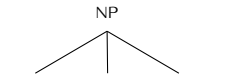
3.40

3.41

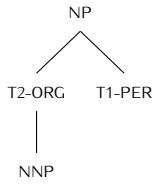
3.42



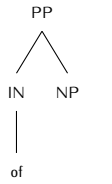
3.43



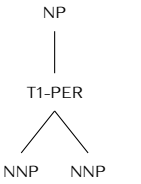
3.44



3.45

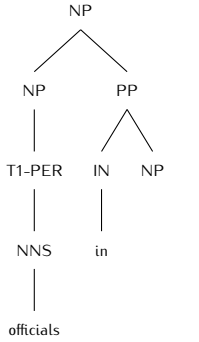


3.46

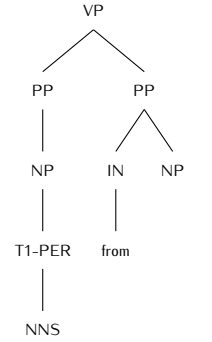


3.47

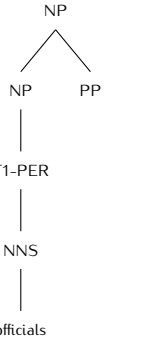
3.48



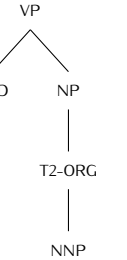
3.49



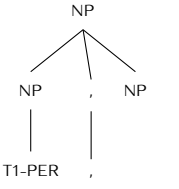
3.50



3.51



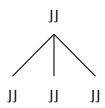
3.52



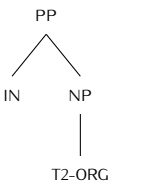
3.53

3.54

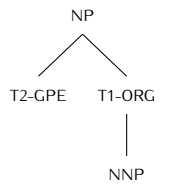
3.55



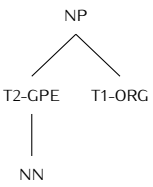
3.56



3.57

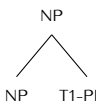


3.58



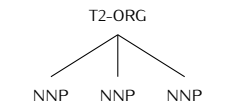
3.59

3.60

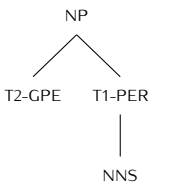


3.61

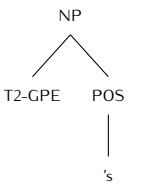
3.62



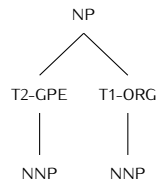
3.63



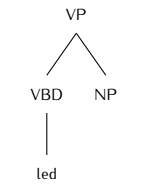
3.64



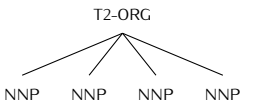
3.65



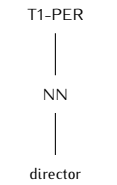
3.66



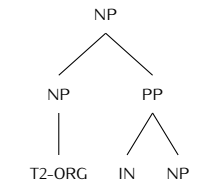
3.67



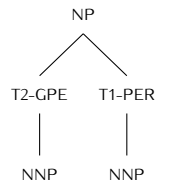
3.68



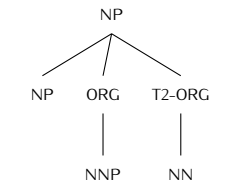
3.69



3.70



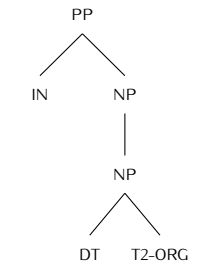
3.71



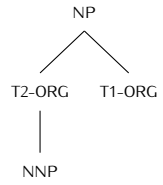
3.72

3.73

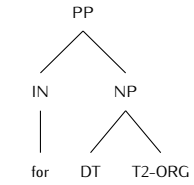
3.74



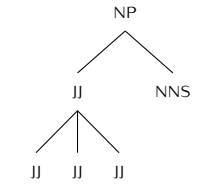
3.75



3.76

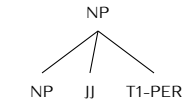


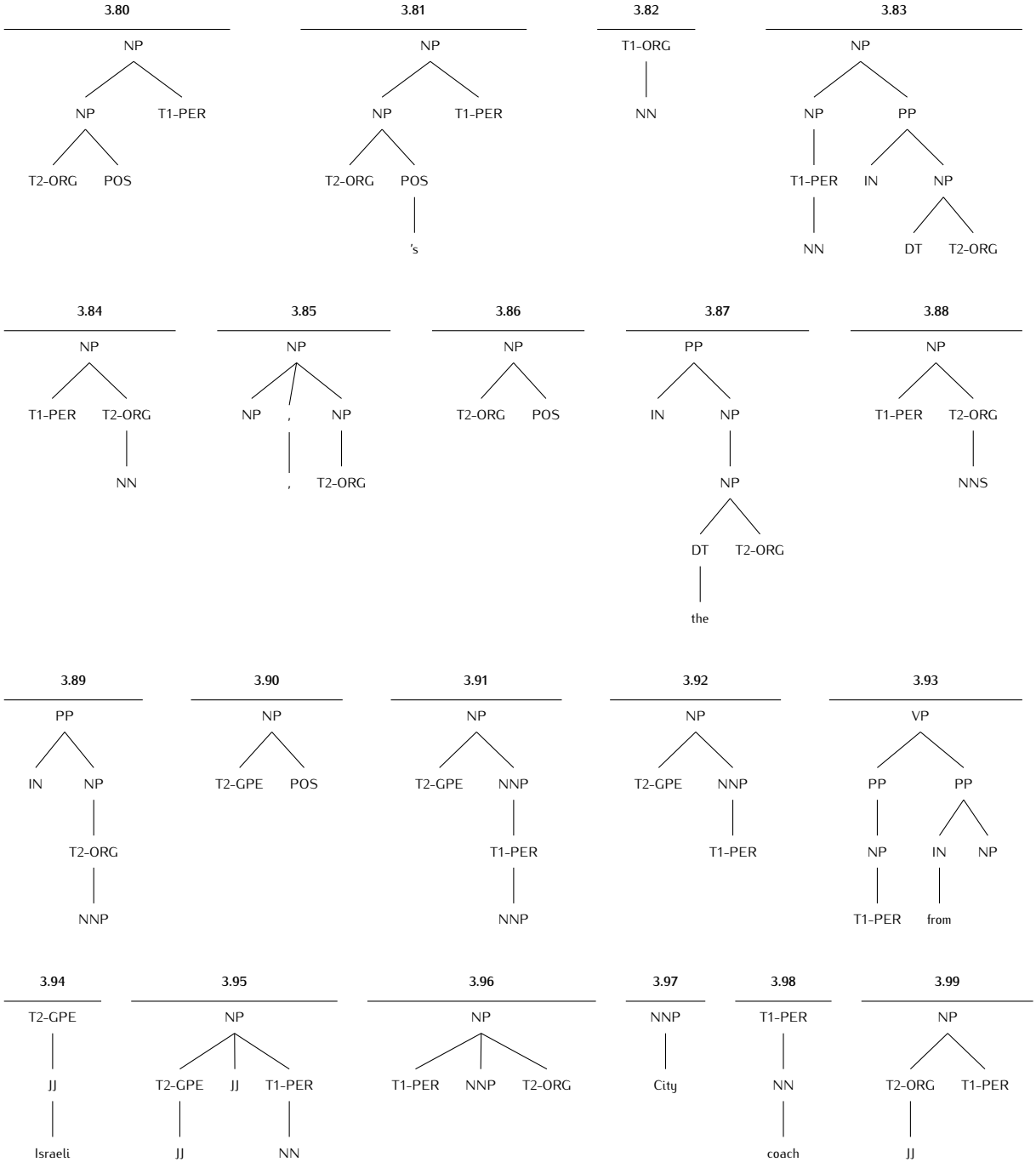
3.77

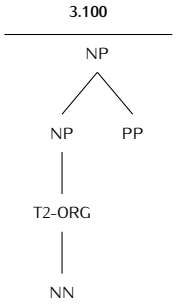


3.78

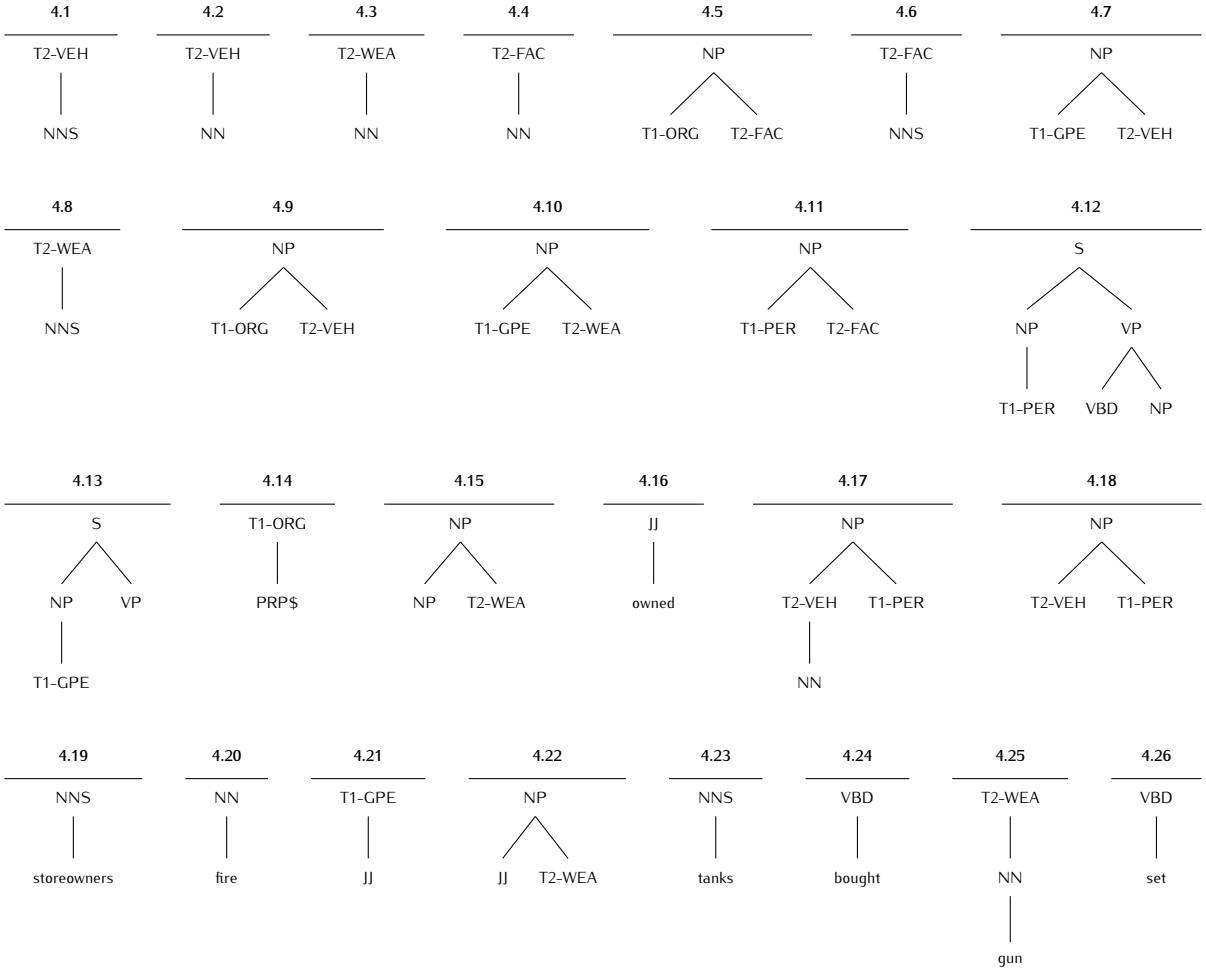
3.79

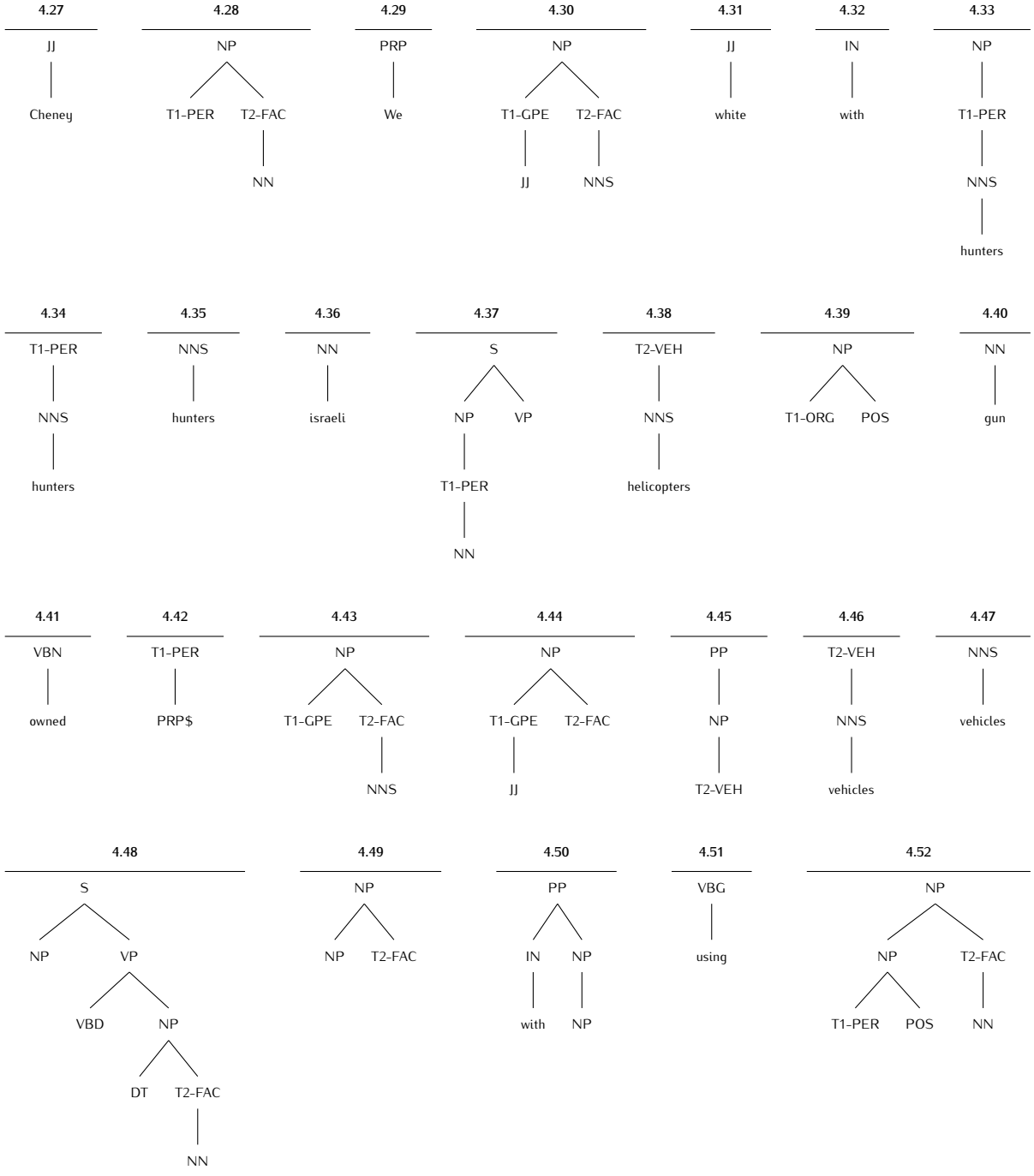


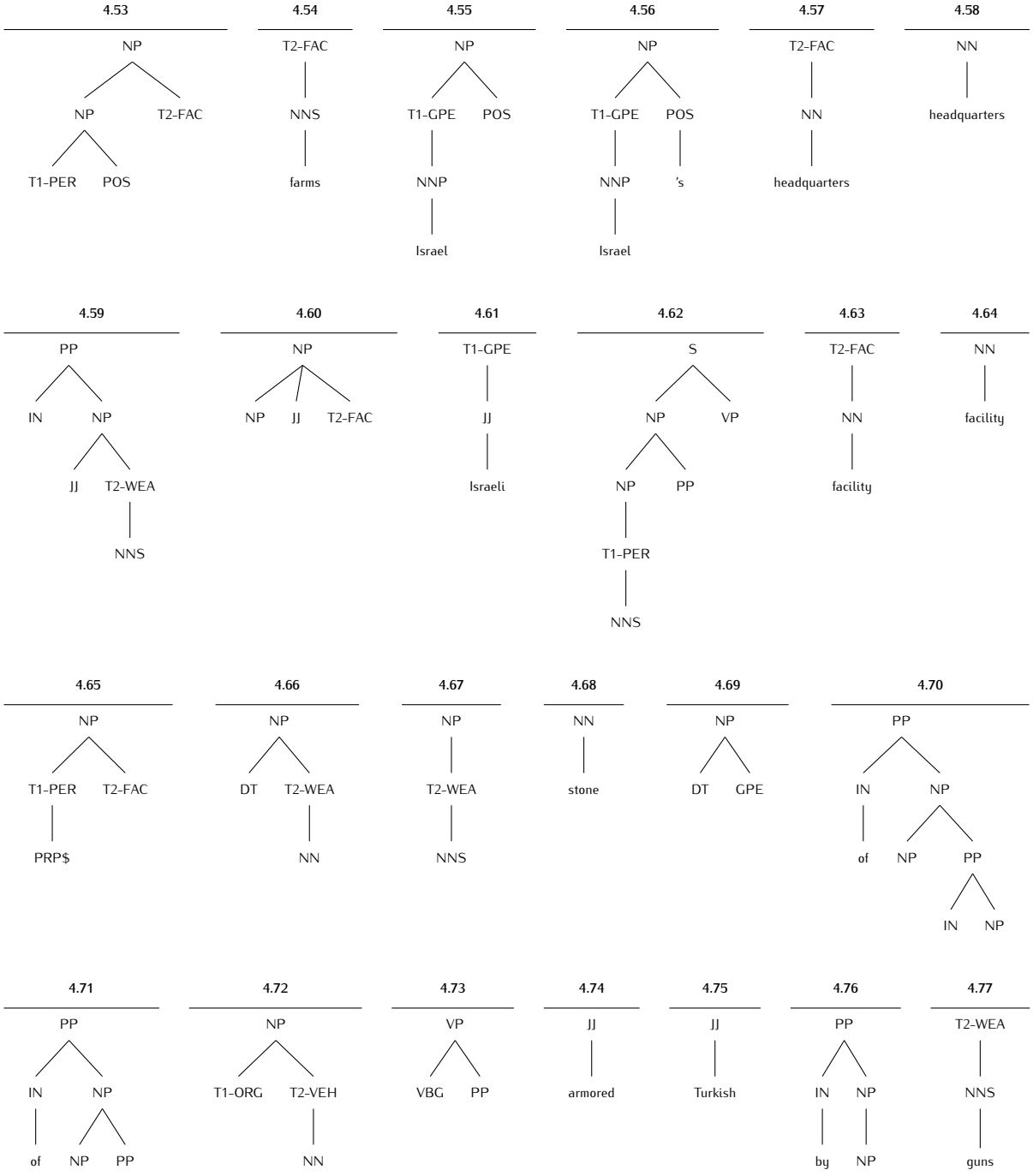

3.80
NP
NP
T1-PER
T2-ORG
POS
3.81
NP
NP
T1-PER
T2-ORG
POS
's
3.82
T1-ORG
NN
3.83
NP
NP
PP
T1-PER
IN
NP
NN
DT
T2-ORG
3.84
NP
T1-PER
T2-ORG
NN
3.85
NP
NP
,
NP
,
T2-ORG
3.86
NP
T2-ORG
POS
3.87
PP
IN
NP
NP
DT
T2-ORG
the
3.88
NP
T1-PER
T2-ORG
NNS
3.89
PP
IN
NP
T2-ORG
NNP
3.90
NP
T2-GPE
POS
3.91
NP
T2-GPE
NNP
T1-PER
NNP
3.92
NP
T2-GPE
NNP
T1-PER
3.93
VP
PP
PP
NP
IN
NP
T1-PER
from
3.94
T2-GPE
JJ
Israeli
3.95
NP
T2-GPE
JJ
T1-PER
JJ
NN
3.96
NP
T1-PER
NNP
T2-ORG
3.97
NNP
City
3.98
T1-PER
NN
coach
3.99
NP
T2-ORG
T1-PER
JJ

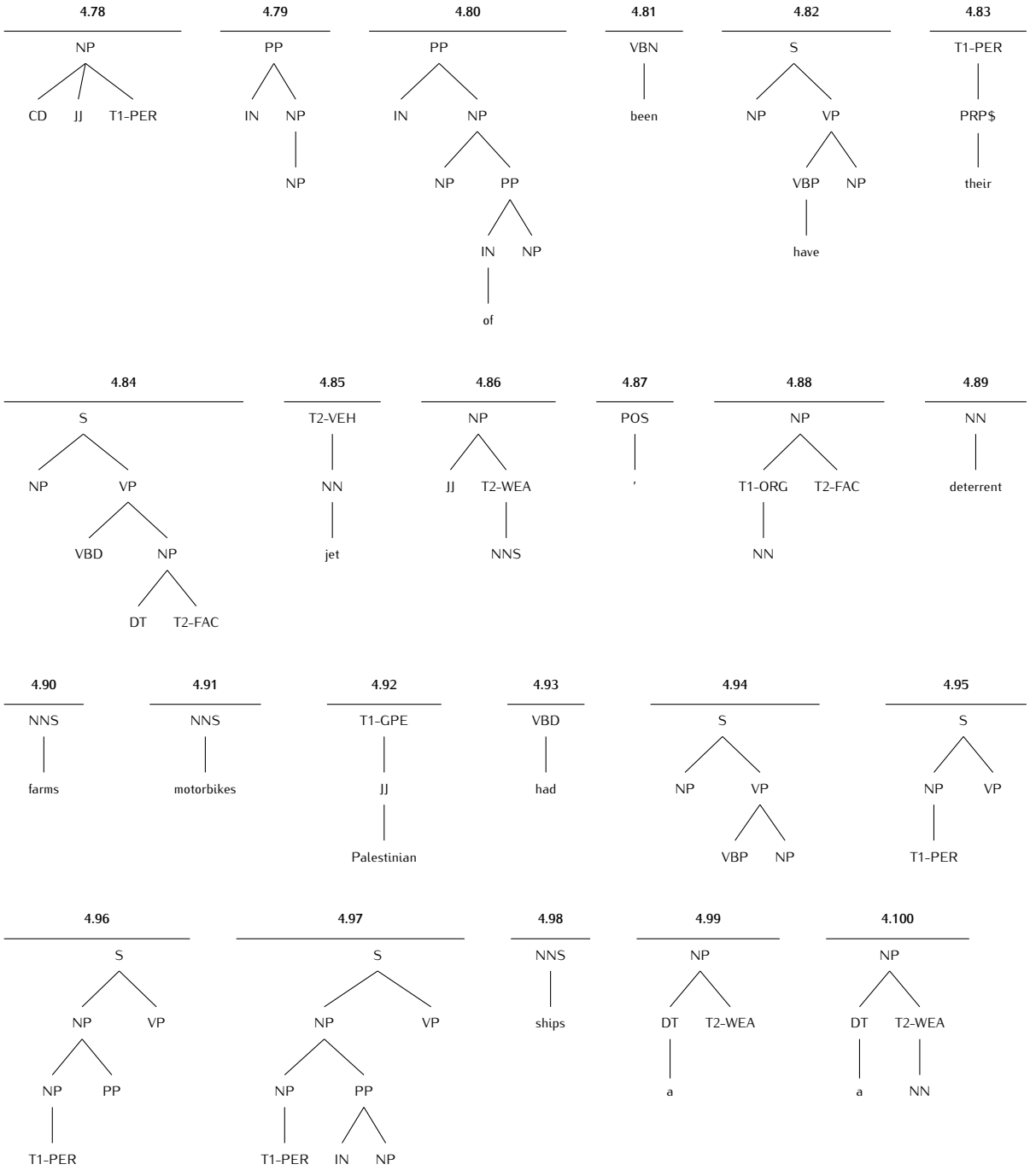


Fragments for Class 4

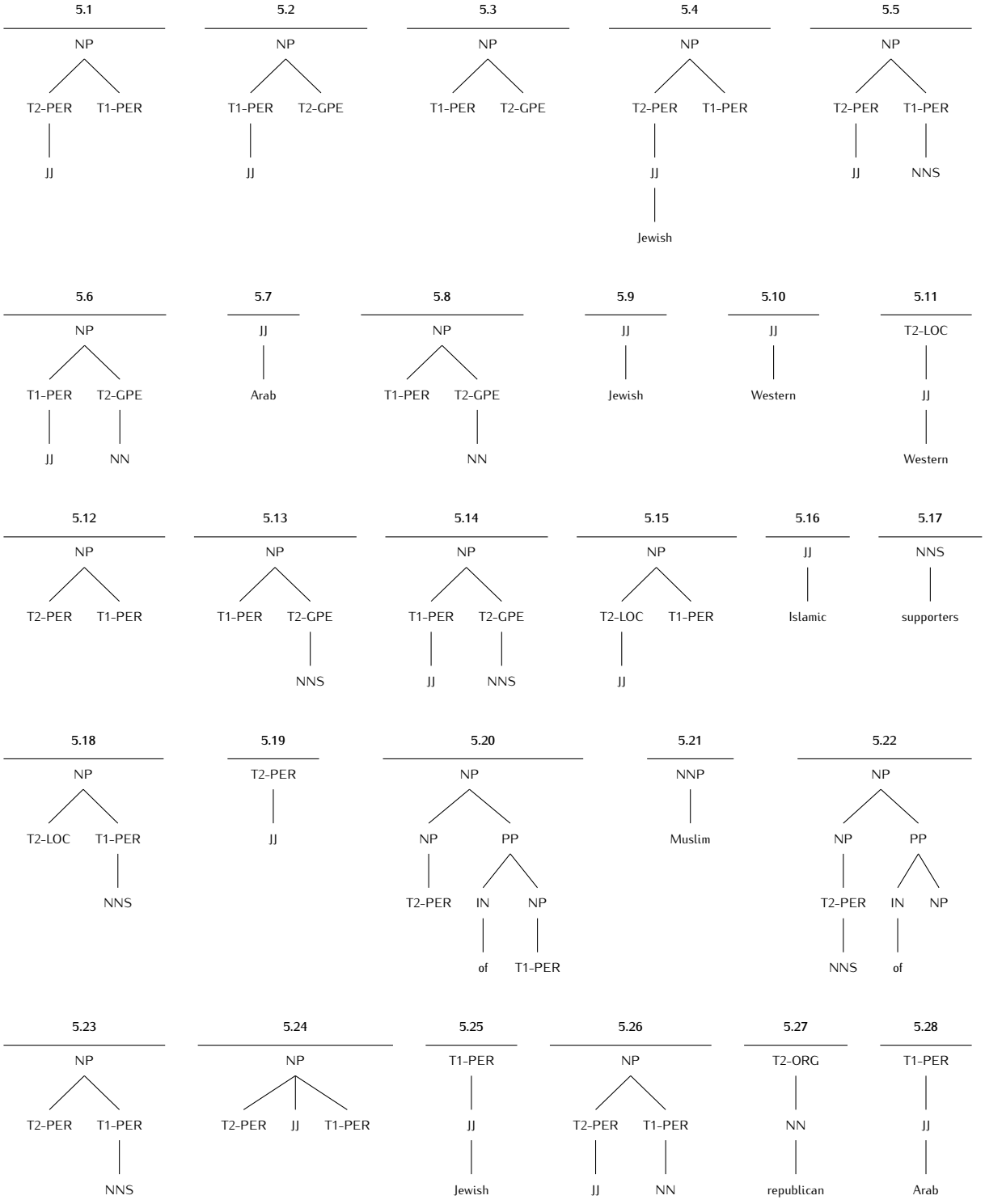



4.27
JJ
Cheney
4.28
NP
T1-PER
T2-FAC
NN
4.29
PRP
We
4.30
NP
T1-GPE
T2-FAC
JJ
NNS
4.31
JJ
white
4.32
IN
with
4.33
NP
T1-PER
NNS
hunters
4.34
T1-PER
NNS
hunters
4.35
NNS
hunters
4.36
NN
israeli
4.37
S
NP
VP
T1-PER
NN
4.38
T2-VEH
NNS
helicopters
4.39
NP
T1-ORG
POS
4.40
NN
gun
4.41
VBN
owned
4.42
T1-PER
PRP$
4.43
NP
T1-GPE
T2-FAC
NNS
4.44
NP
T1-GPE
T2-FAC
JJ
4.45
PP
NP
T2-VEH
4.46
T2-VEH
NNS
vehicles
4.47
NNS
vehicles
4.48
S
NP
VP
VBD
NP
DT
T2-FAC
NN
4.49
NP
NP
T2-FAC
4.50
PP
IN
NP
with
NP
4.51
VBG
using
4.52
NP
NP
T2-FAC
T1-PER
POS
NN

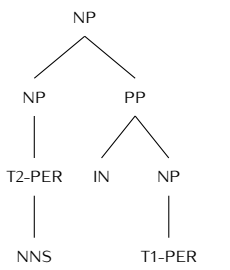

4.53
NP
NP
T2-FAC
T1-PER
POS
4.54
T2-FAC
NNS
farms
4.55
NP
T1-GPE
POS
NNP
Israel
4.56
NP
T1-GPE
POS
NNP
's
Israel
4.57
T2-FAC
NN
headquarters
headquarters
4.58
NN
headquarters
4.59
PP
IN
NP
JJ
T2-WEA
NNS
4.60
NP
NP
JJ
T2-FAC
4.61
T1-GPE
JJ
Israeli
4.62
S
NP
VP
NP
PP
T1-PER
NNS
4.63
T2-FAC
NN
facility
facility
4.64
NN
facility
4.65
NP
T1-PER
T2-FAC
PRP$
4.66
NP
DT
T2-WEA
NN
4.67
NP
T2-WEA
NNS
4.68
NN
stone
4.69
NP
DT
GPE
4.70
PP
IN
NP
of
NP
PP
IN
NP
4.71
PP
IN
NP
of
NP
PP
4.72
NP
T1-ORG
T2-VEH
NN
4.73
VP
VBG
PP
4.74
JJ
armored
4.75
JJ
Turkish
4.76
PP
IN
NP
by
NP
4.77
T2-WEA
NNS
guns


4.78
NP
CD JJ T1-PER
4.79
PP
IN NP
NP
4.80
PP
IN NP
NP PP
IN NP
of
4.81
VBN
been
4.82
S
NP VP
VBP NP
have
4.83
T1-PER
PRP$
their
4.84
S
NP VP
VBD NP
DT T2-FAC
4.85
T2-VEH
NN
jet
4.86
NP
JJ T2-WEA
NNS
4.87
POS
'
4.88
NP
T1-ORG T2-FAC
NN
4.89
NN
deterrent
4.90
NNS
farms
4.91
NNS
motorbikes
4.92
T1-GPE
JJ
Palestinian
4.93
VBD
had
4.94
S
NP VP
VBP NP
4.95
S
NP VP
T1-PER
4.96
S
NP VP
NP PP
T1-PER
4.97
S
NP VP
NP PP
T1-PER IN NP
4.98
NNS
ships
4.99
NP
DT T2-WEA
a
4.100
NP
DT T2-WEA
a NN

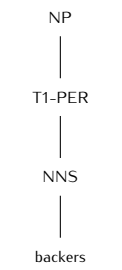
Fragments for Class 5



5.29



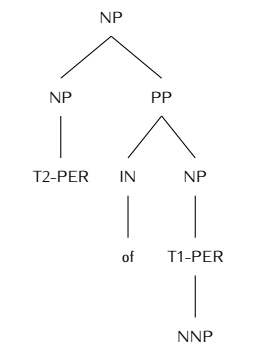
5.30



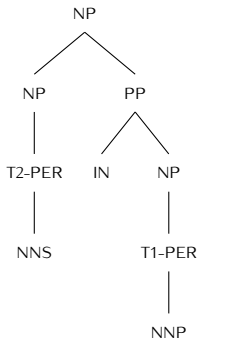
5.31

5.32

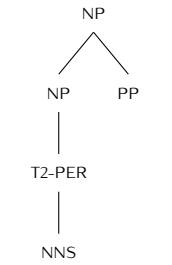
5.33



5.34

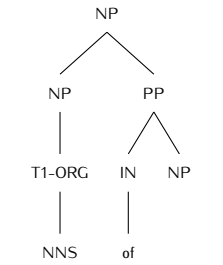


5.35

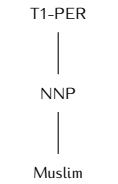


5.36

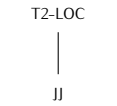
5.37



5.38

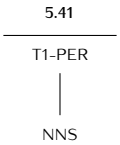


5.39

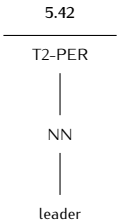


5.40

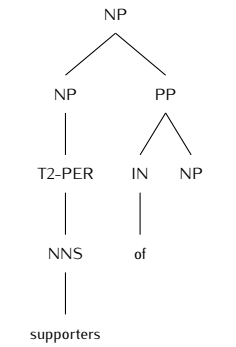
5.41



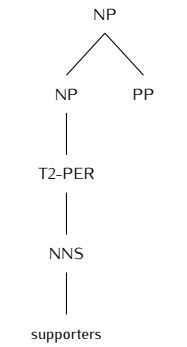
5.42



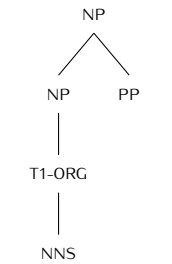
5.43



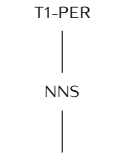
5.44



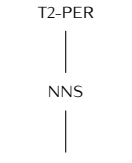
5.45



5.46

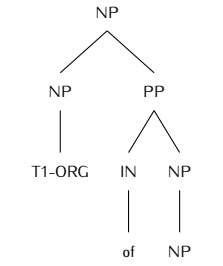


5.47

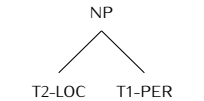


5.48

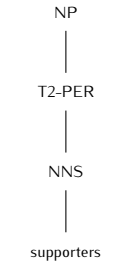
5.49



5.50

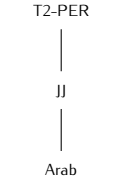


5.51

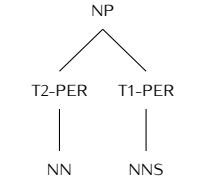


5.52

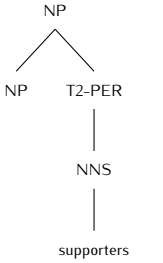
5.53



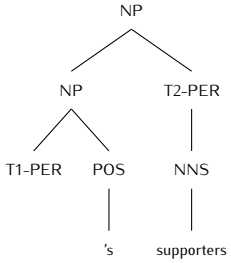
5.54



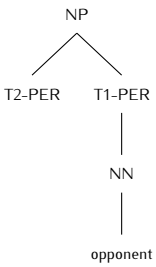
5.55



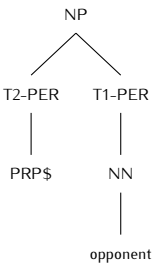
5.56



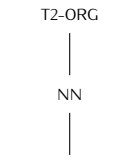
5.57



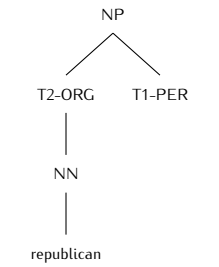
5.58



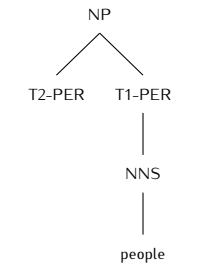
5.59



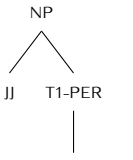
5.60



5.61



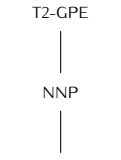
5.62



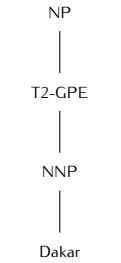
5.63

5.64

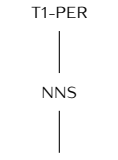
5.65



5.66

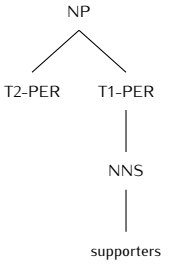


5.67

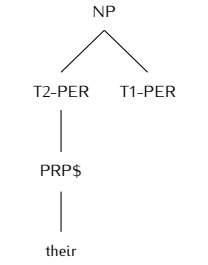


5.68

5.69



5.70

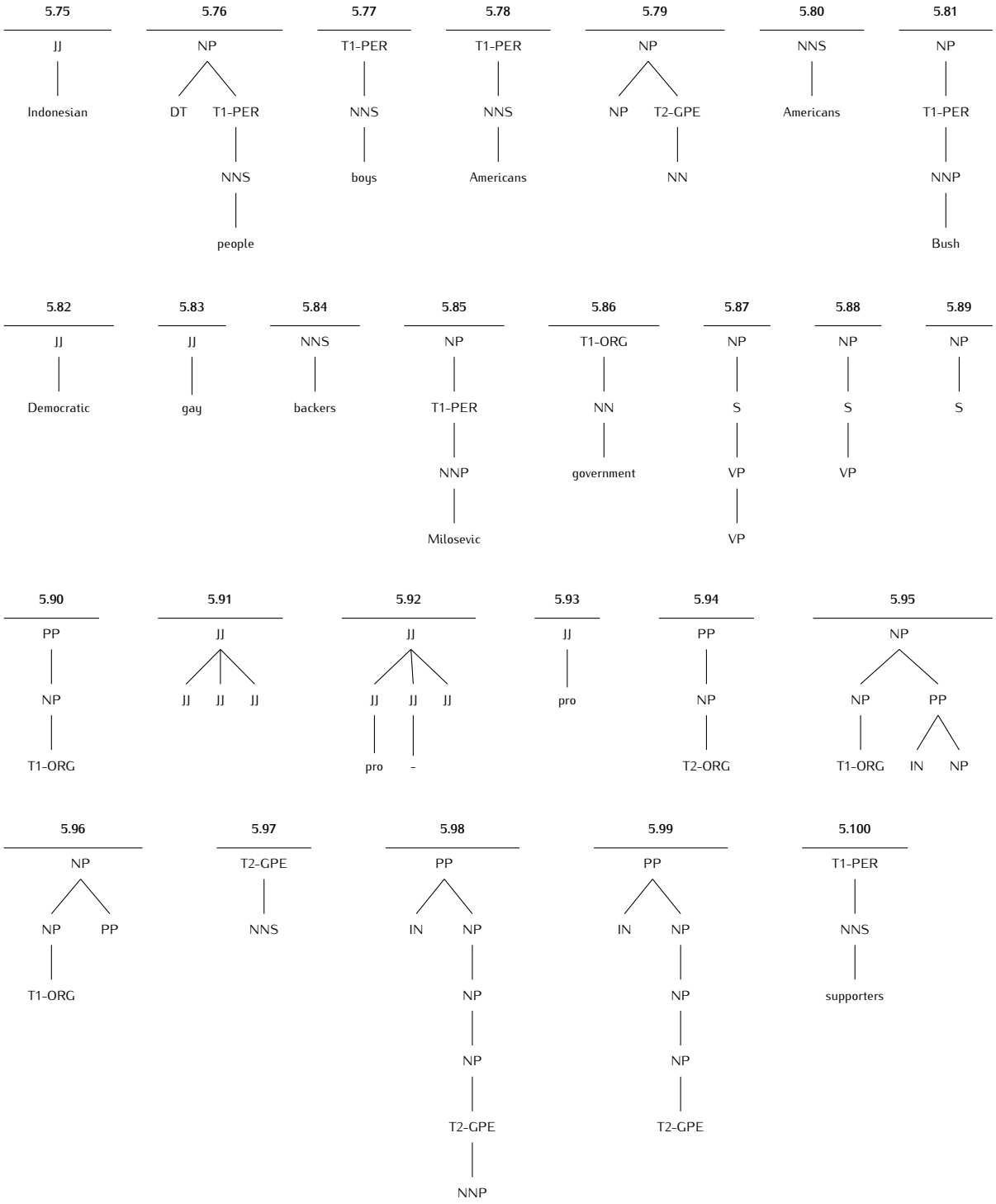


5.71

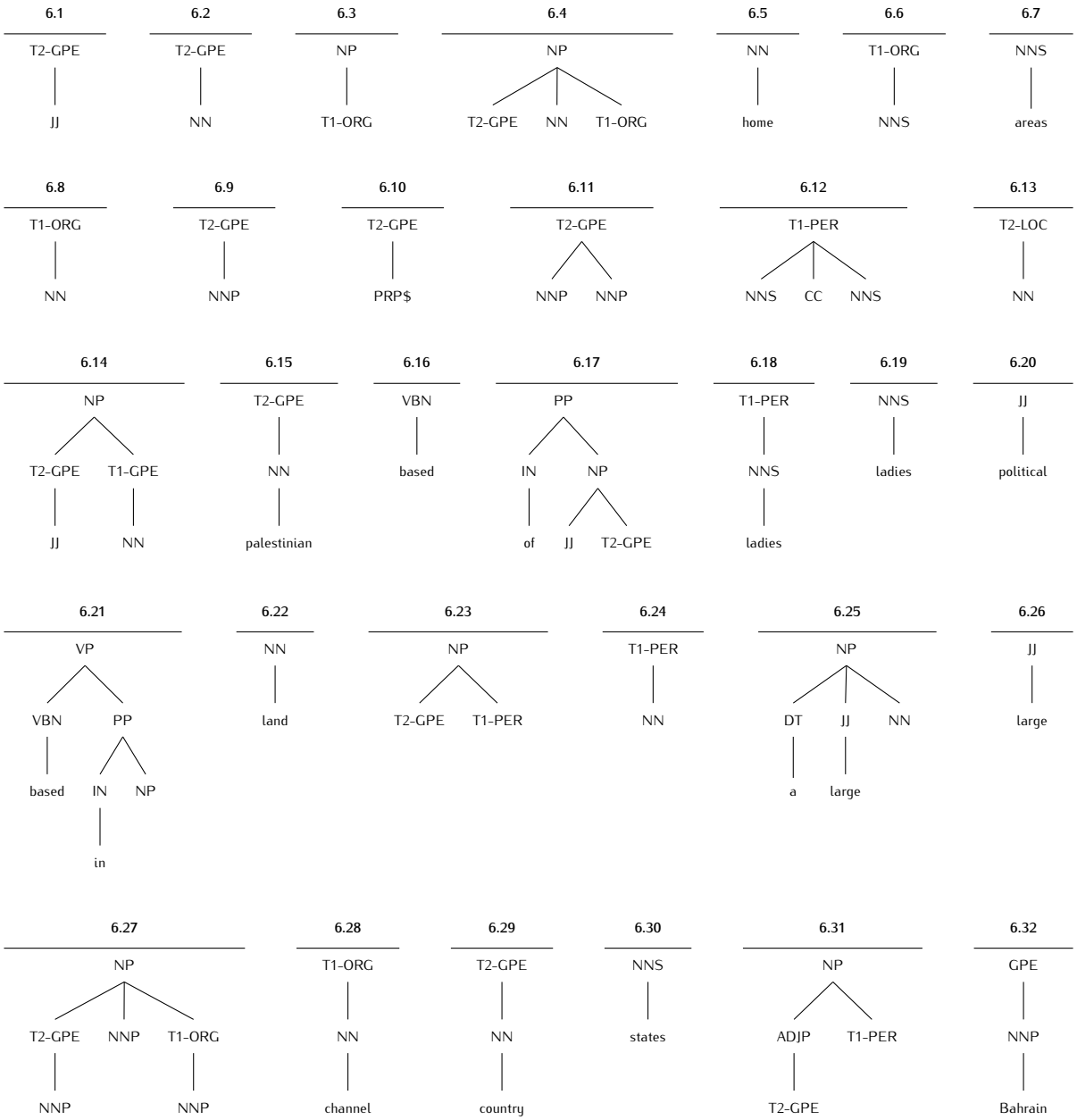
5.72

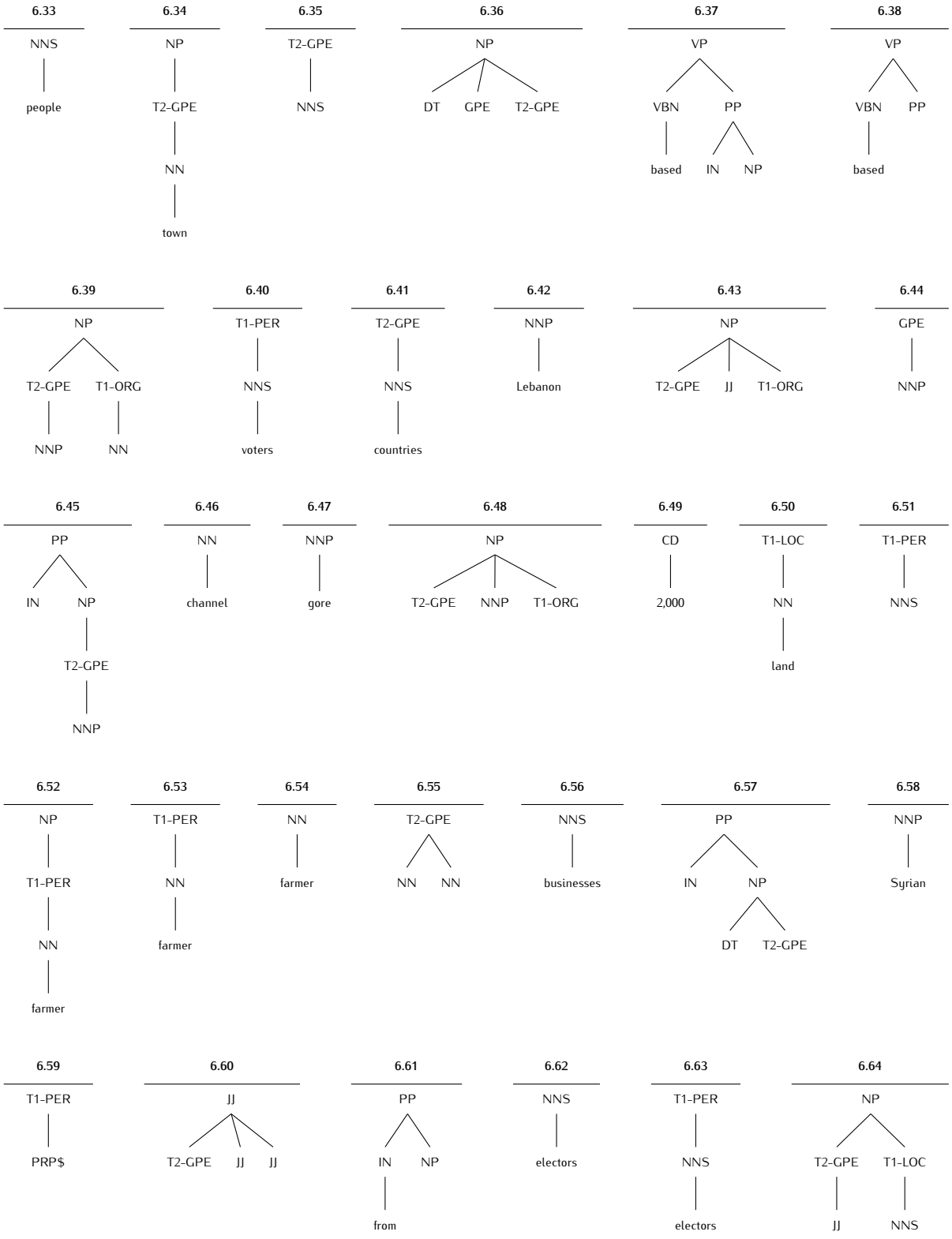
5.73

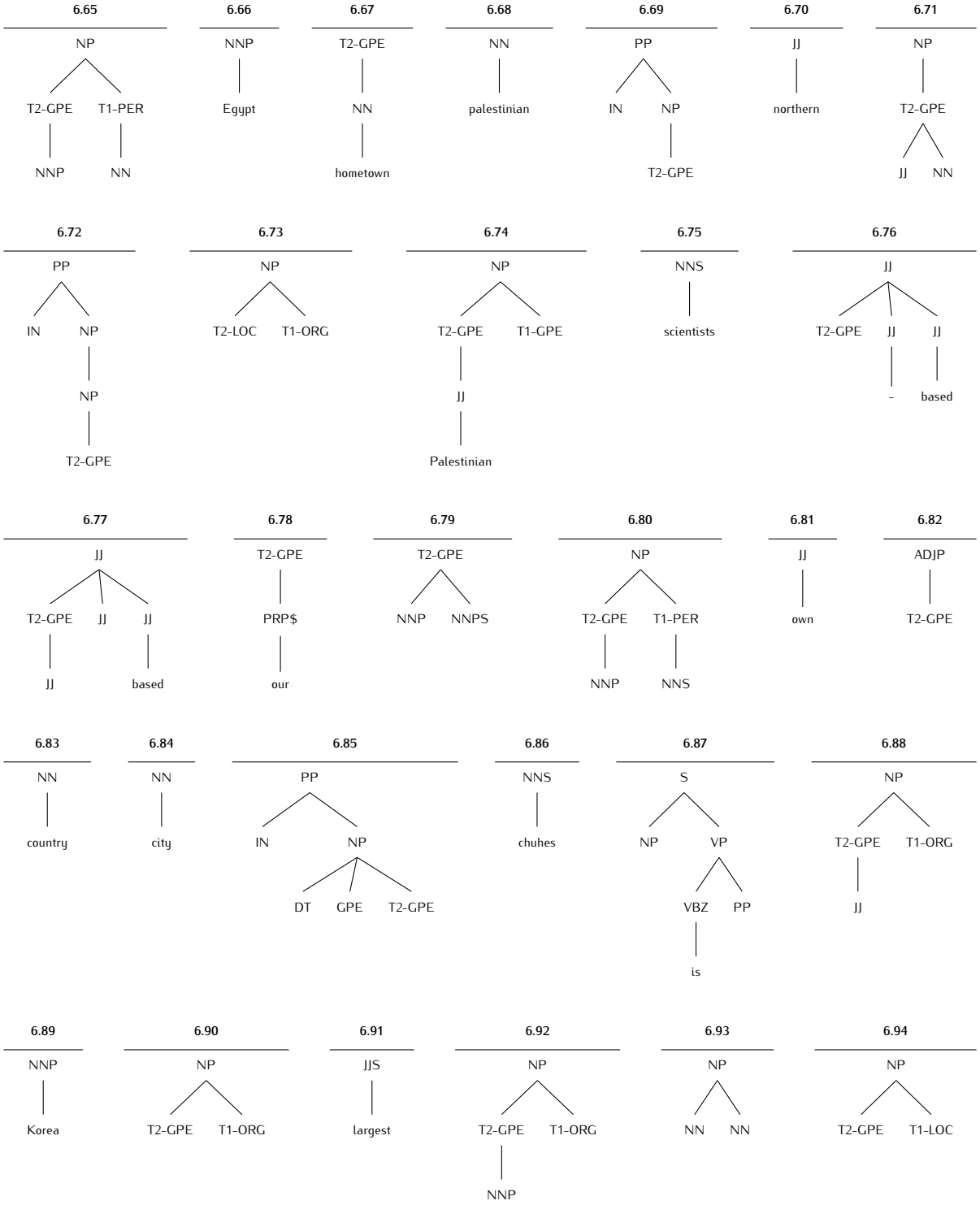
5.74

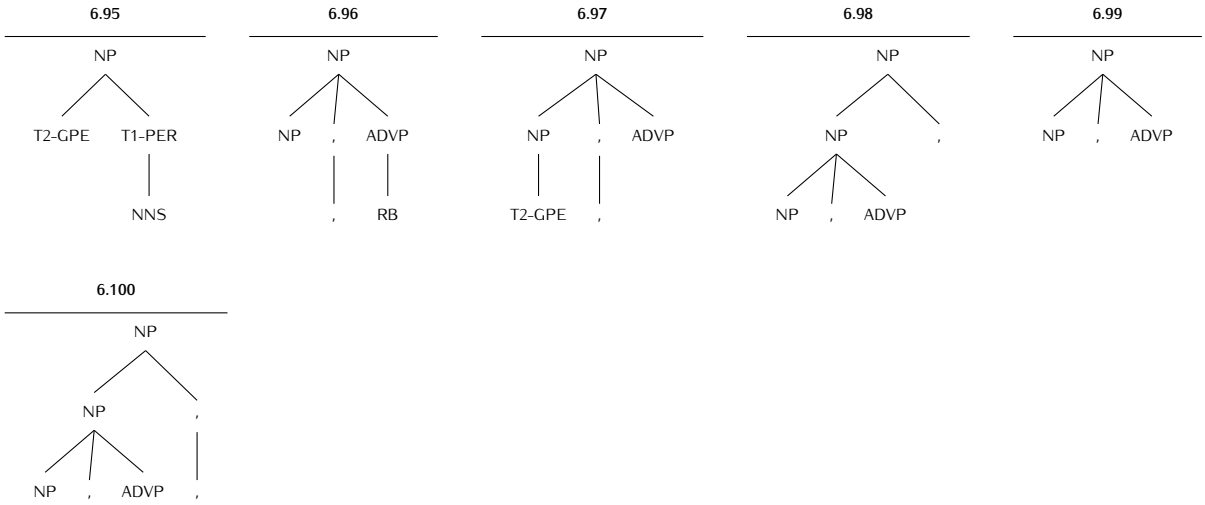

5.75
JJ
Indonesian
5.76
NP
DT
T1-PER
NNS
people
5.77
T1-PER
NNS
boys
5.78
T1-PER
NNS
Americans
5.79
NP
NP
T2-GPE
NN
5.80
NNS
Americans
5.81
NP
T1-PER
NNP
Bush
5.82
JJ
Democratic
5.83
JJ
gay
5.84
NNS
backers
5.85
NP
T1-PER
NNP
Milosevic
5.86
T1-ORG
NN
government
5.87
NP
S
VP
VP
5.88
NP
S
VP
5.89
NP
S
5.90
PP
NP
T1-ORG
5.91
JJ
JJ
JJ
JJ
5.92
JJ
JJ
JJ
JJ
pro
-
5.93
JJ
pro
5.94
PP
NP
T2-ORG
5.95
NP
NP
PP
T1-ORG
IN
NP
5.96
NP
NP
PP
T1-ORG
5.97
T2-GPE
NNS
5.98
PP
IN
NP
NP
NP
T2-GPE
NNP
5.99
PP
IN
NP
NP
NP
T2-GPE
5.100
T1-PER
NNS
supporters

Fragments for Class 6

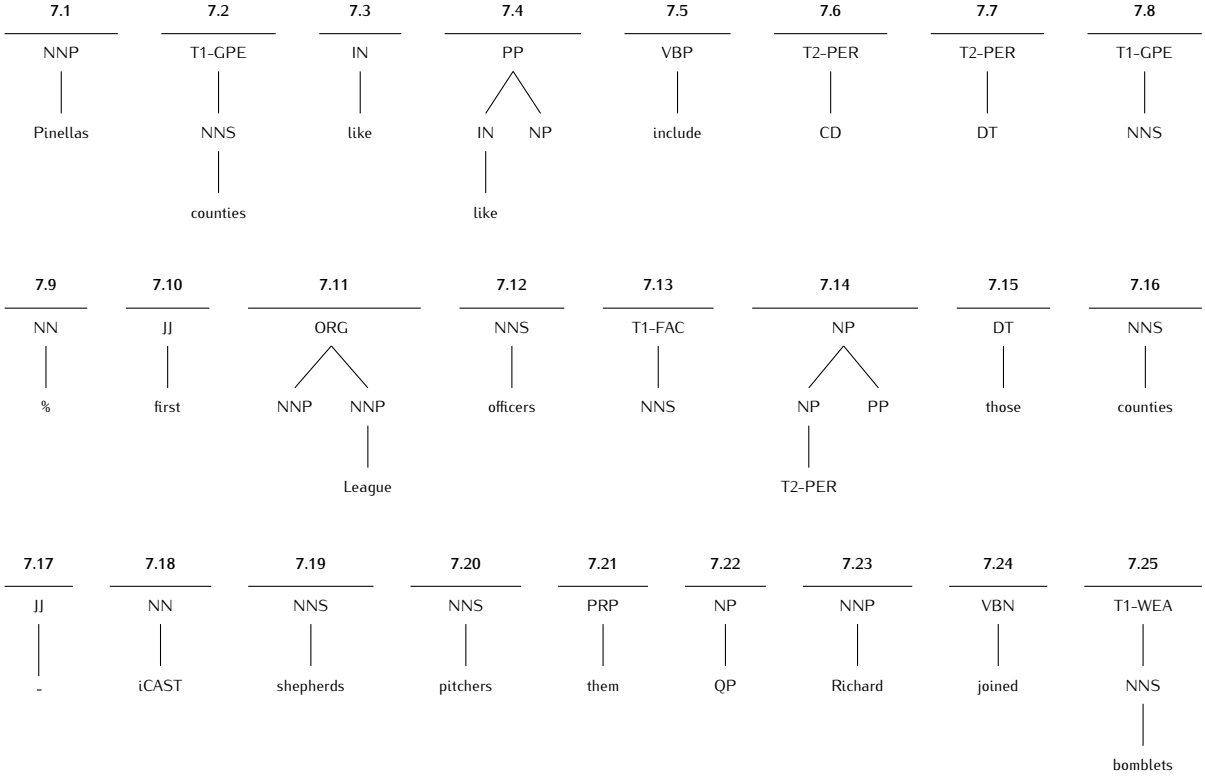



6.33
NNS
people
6.34
NP
T2-GPE
NN
town
6.35
T2-GPE
NNS
6.36
NP
DT GPE T2-GPE
6.37
VP
VBN PP
based IN NP
6.38
VP
VBN PP
based
6.39
NP
T2-GPE T1-ORG
NNP NN
6.40
T1-PER
NNS
voters
6.41
T2-GPE
NNS
countries
6.42
NNP
Lebanon
6.43
NP
T2-GPE JJ T1-ORG
6.44
GPE
NNP
6.45
PP
IN NP
T2-GPE
NNP
6.46
NN
channel
6.47
NNP
gore
6.48
NP
T2-GPE NNP T1-ORG
6.49
CD
2,000
6.50
T1-LOC
NN
land
6.51
T1-PER
NNS
6.52
NP
T1-PER
NN
farmer
6.53
T1-PER
NN
farmer
6.54
NN
farmer
6.55
T2-GPE
NN NN
6.56
NNS
businesses
6.57
PP
IN NP
DT T2-GPE
6.58
NNP
Syrian
6.59
T1-PER
PRP$
6.60
JJ
T2-GPE JJ JJ
6.61
PP
IN NP
from
6.62
NNS
electors
6.63
T1-PER
NNS
electors
6.64
NP
T2-GPE T1-LOC
JJ NNS


6.65
NP
T2-GPE T1-PER
NNP NN
6.66
NNP
Egypt
6.67
T2-GPE
NN
hometown
6.68
NN
palestinian
6.69
PP
IN NP
T2-GPE
6.70
JJ
northern
6.71
NP
T2-GPE
JJ NN
6.72
PP
IN NP
NP
T2-GPE
6.73
NP
T2-LOC T1-ORG
6.74
NP
T2-GPE T1-GPE
JJ
Palestinian
6.75
NNS
scientists
6.76
JJ
T2-GPE JJ JJ
- based
6.77
JJ
T2-GPE JJ JJ
JJ based
6.78
T2-GPE
PRP$
our
6.79
T2-GPE
NNP NNPS
6.80
NP
T2-GPE T1-PER
NNP NNS
6.81
JJ
own
6.82
ADJP
T2-GPE
6.83
NN
country
6.84
NN
city
6.85
PP
IN NP
DT GPE T2-GPE
6.86
NNS
chuhes
6.87
S
NP VP
VBZ PP
is
6.88
NP
T2-GPE T1-ORG
JJ
6.89
NNP
Korea
6.90
NP
T2-GPE T1-ORG
6.91
JJS
largest
6.92
NP
T2-GPE T1-ORG
NNP
6.93
NP
NN NN
6.94
NP
T2-GPE T1-LOC



Fragments for Class 7



7.26

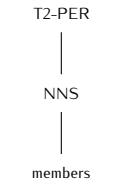
7.27

7.28

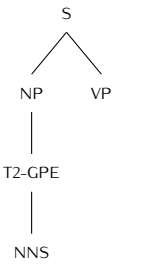
7.29

7.30

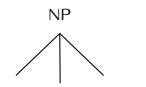
7.31



7.32

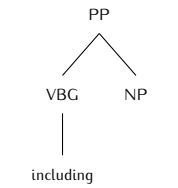


7.33



7.34

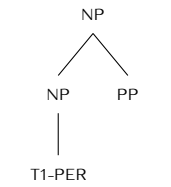
7.35



7.36

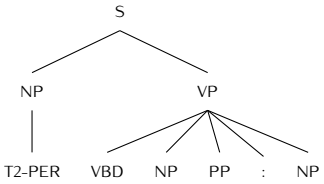
7.37

7.38



7.39

7.40

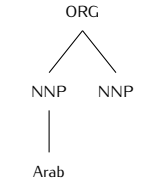


7.41

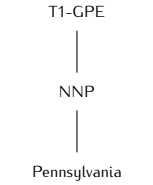
7.42

7.43

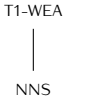
7.44



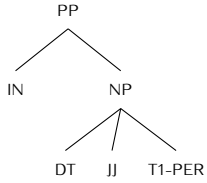
7.45



7.46



7.47

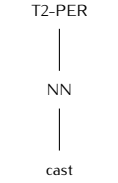


7.48

7.49

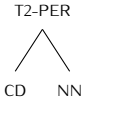
7.50

7.51



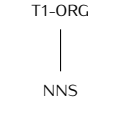
7.52

7.53

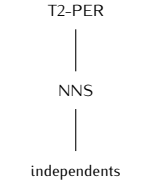


7.54

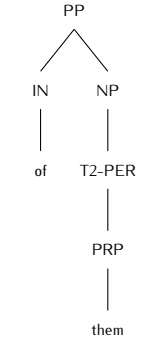
7.55



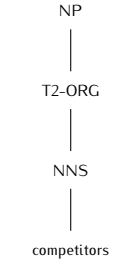
7.56



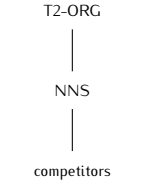
7.57

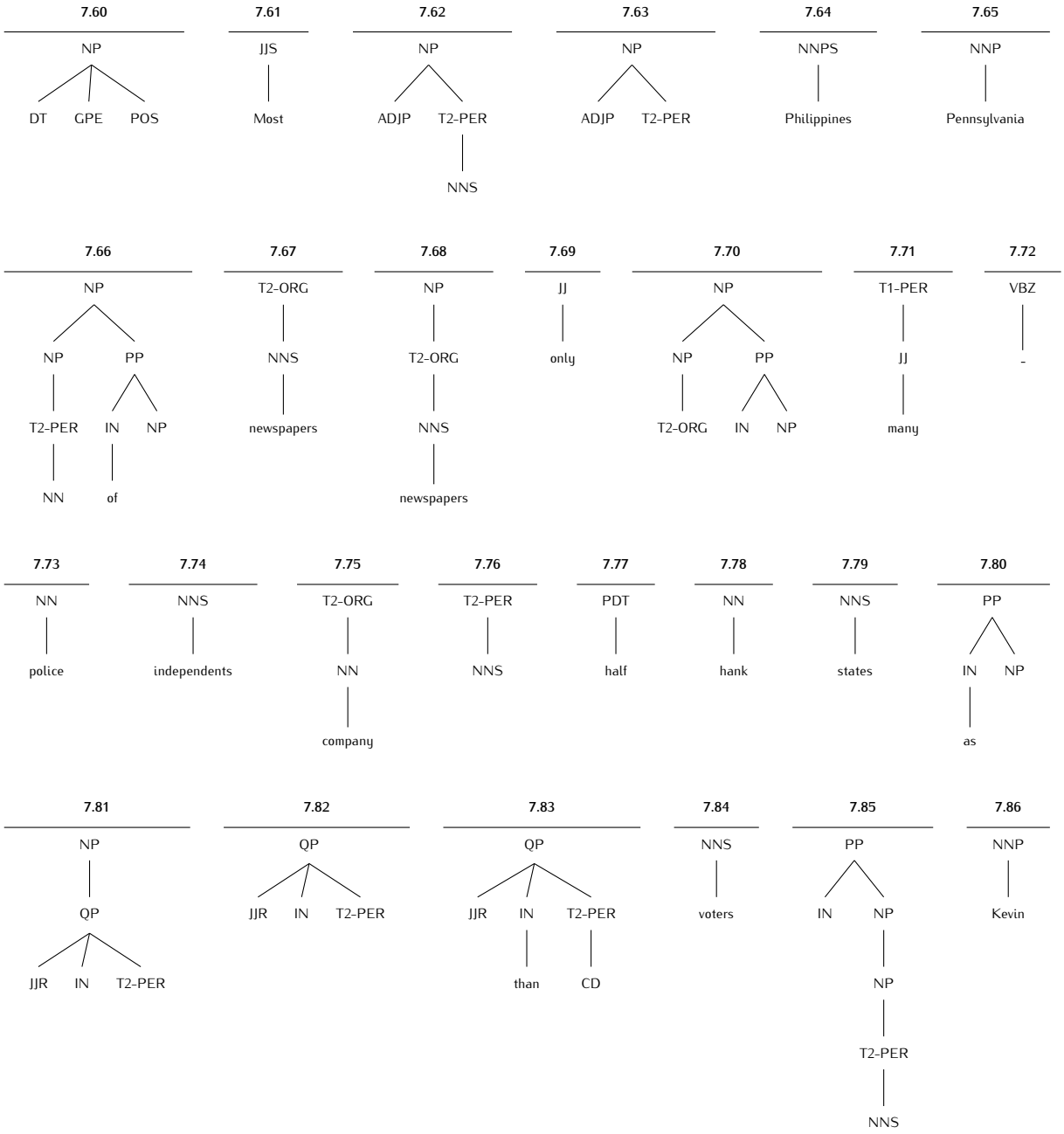


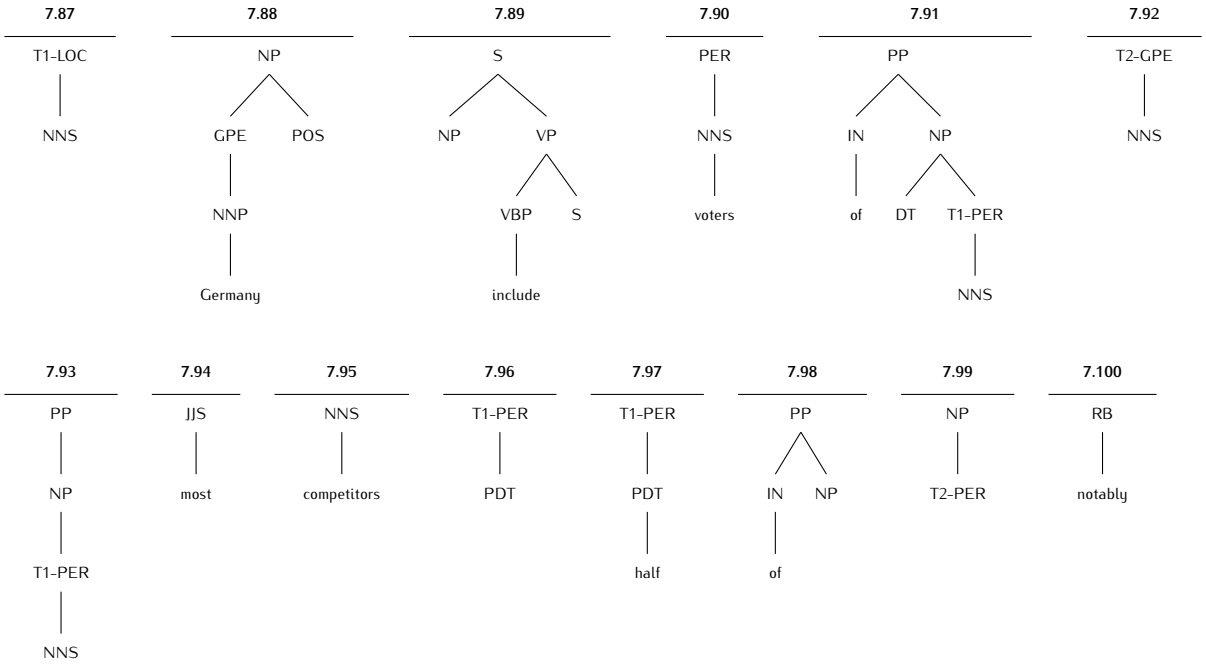
7.58



7.59

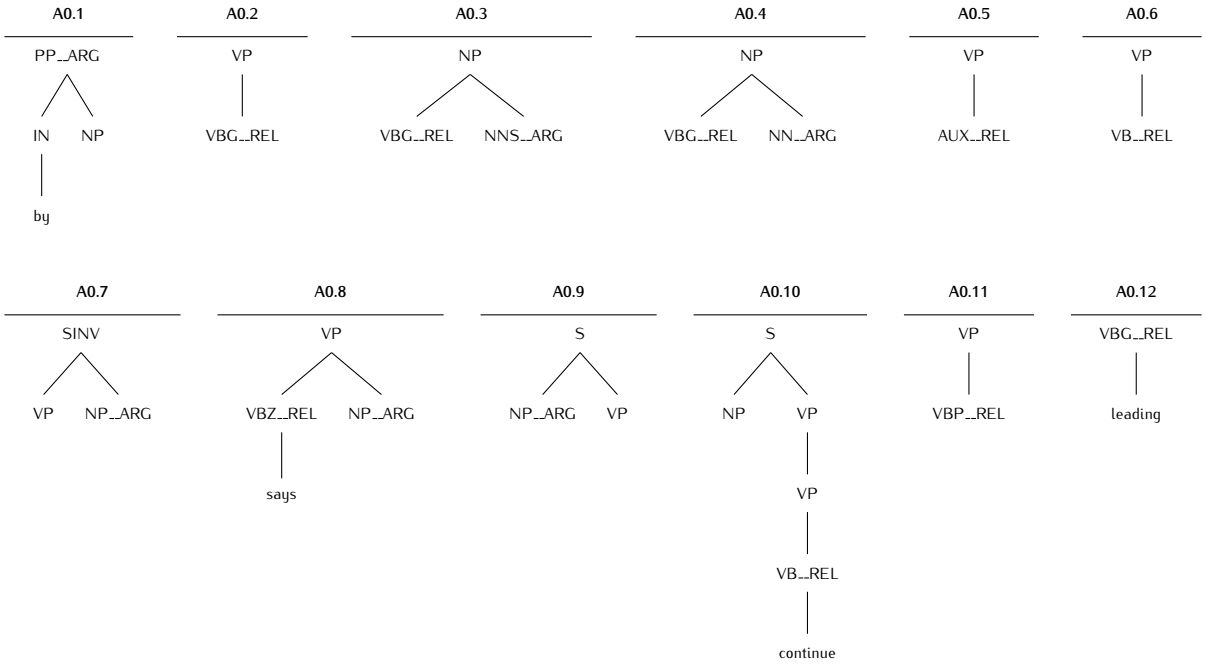


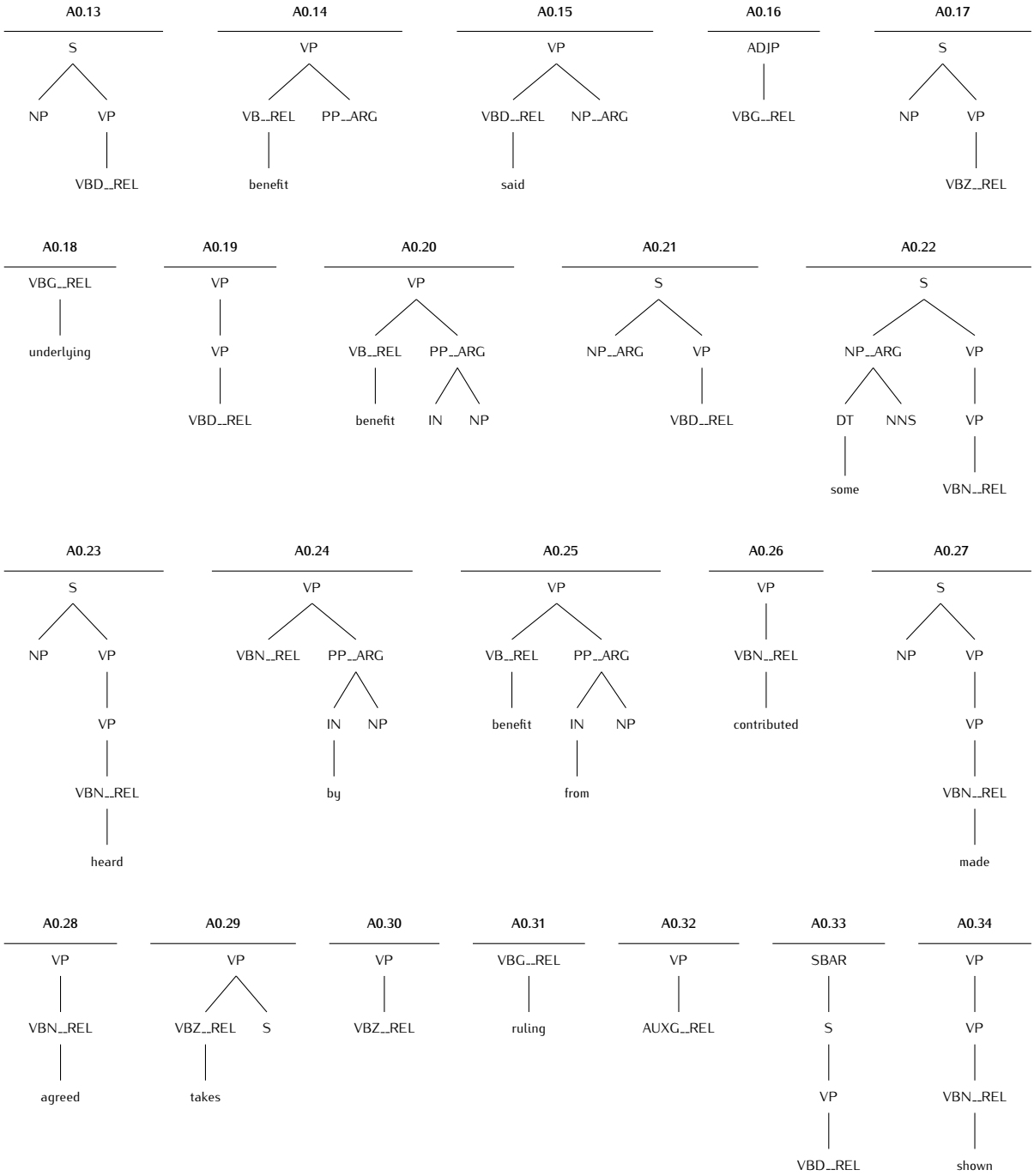

7.60
NP
DT GPE POS
7.61
JJS
Most
7.62
NP
ADJP T2-PER
NNS
7.63
NP
ADJP T2-PER
7.64
NNPS
Philippines
7.65
NNP
Pennsylvania
7.66
NP
NP PP
T2-PER IN NP
NN of
7.67
T2-ORG
NNS
newspapers
7.68
NP
T2-ORG
NNS
newspapers
7.69
JJ
only
7.70
NP
NP PP
T2-ORG IN NP
7.71
T1-PER
JJ
many
7.72
VBZ
-
7.73
NN
police
7.74
NNS
independents
7.75
T2-ORG
NN
company
7.76
T2-PER
NNS
7.77
PDT
half
7.78
NN
hank
7.79
NNS
states
7.80
PP
IN NP
as
7.81
NP
QP
JJR IN T2-PER
7.82
QP
JJR IN T2-PER
7.83
QP
JJR IN T2-PER
than CD
7.84
NNS
voters
7.85
PP
IN NP
NP
T2-PER
NNS
7.86
NNP
Kevin

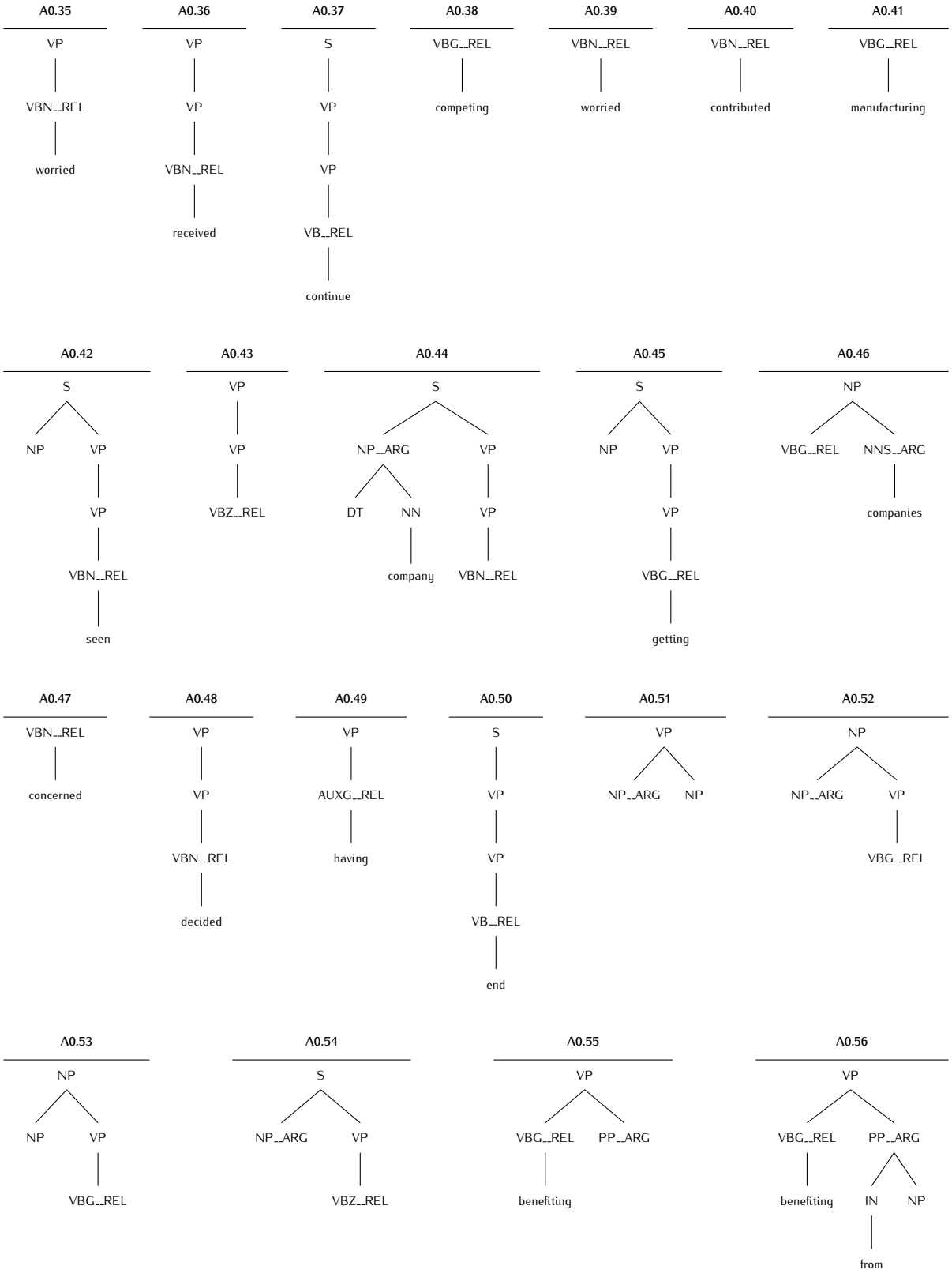


B.3 Semantic Role Labeling

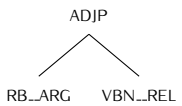
Fragments for Class A0



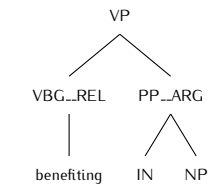

A0.13
S
NP
VP
VBD__REL
A0.14
VP
VB__REL
PP__ARG
benefit
A0.15
VP
VBD__REL
NP__ARG
said
A0.16
ADJP
VBG__REL
A0.17
S
NP
VP
VBZ__REL
A0.18
VBG__REL
underlying
A0.19
VP
VP
VBD__REL
A0.20
VP
VB__REL
PP__ARG
benefit
IN
NP
A0.21
S
NP__ARG
VP
VBD__REL
A0.22
S
NP__ARG
VP
DT
NNS
VP
some
VBN__REL
A0.23
S
NP
VP
VP
VBN__REL
heard
A0.24
VP
VBN__REL
PP__ARG
IN
NP
by
A0.25
VP
VB__REL
PP__ARG
benefit
IN
NP
from
A0.26
VP
VBN__REL
contributed
A0.27
S
NP
VP
VP
VBN__REL
made
A0.28
VP
VBN__REL
agreed
A0.29
VP
VBZ__REL
S
takes
A0.30
VP
VBZ__REL
A0.31
VBG__REL
ruling
A0.32
VP
AUXG__REL
A0.33
SBAR
S
VP
VBD__REL
A0.34
VP
VP
VBN__REL
shown


A0.35
VP
VBN_-REL
worried
A0.36
VP
VP
VBN_-REL
received
A0.37
S
VP
VP
VB_-REL
continue
A0.38
VBG_-REL
competing
A0.39
VBN_-REL
worried
A0.40
VBN_-REL
contributed
A0.41
VBG_-REL
manufacturing
A0.42
S
NP
VP
VP
VBN_-REL
seen
A0.43
VP
VP
VBZ_-REL
A0.44
S
NP_-ARG
VP
DT
NN
VP
company
VBN_-REL
A0.45
S
NP
VP
VP
VBG_-REL
getting
A0.46
NP
VBG_-REL
NNS_-ARG
companies
A0.47
VBN_-REL
concerned
A0.48
VP
VP
VBN_-REL
decided
A0.49
VP
AUXG_-REL
having
A0.50
S
VP
VP
VB_-REL
end
A0.51
VP
NP_-ARG
NP
A0.52
NP
NP_-ARG
VP
VBG_-REL
A0.53
NP
NP
VP
VBG_-REL
A0.54
S
NP_-ARG
VP
VBZ_-REL
A0.55
VP
VBG_-REL
PP_-ARG
benefiting
A0.56
VP
VBG_-REL
PP_-ARG
benefiting
IN
NP
from

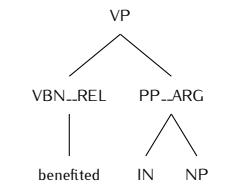
A0.57



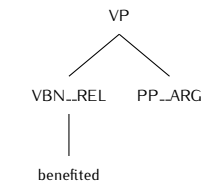
A0.58



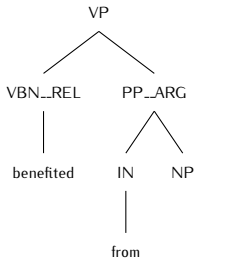
A0.59



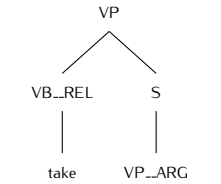
A0.60



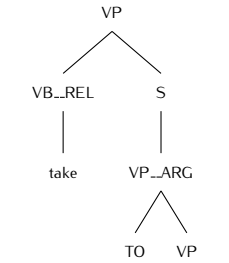
A0.61



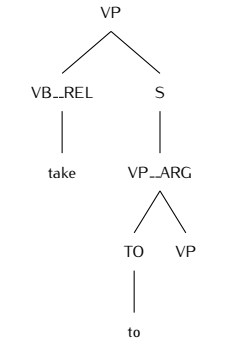
A0.62



A0.63

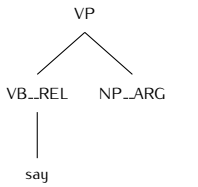


A0.64

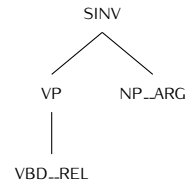


A0.65

A0.66



A0.67



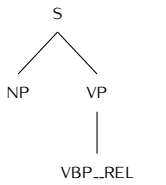
A0.68

A0.69

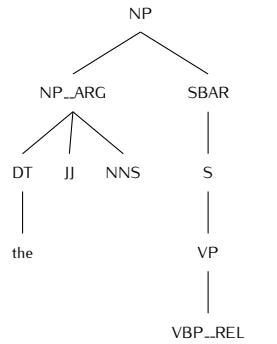
A0.70

A0.71

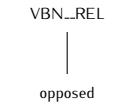
A0.72



A0.73

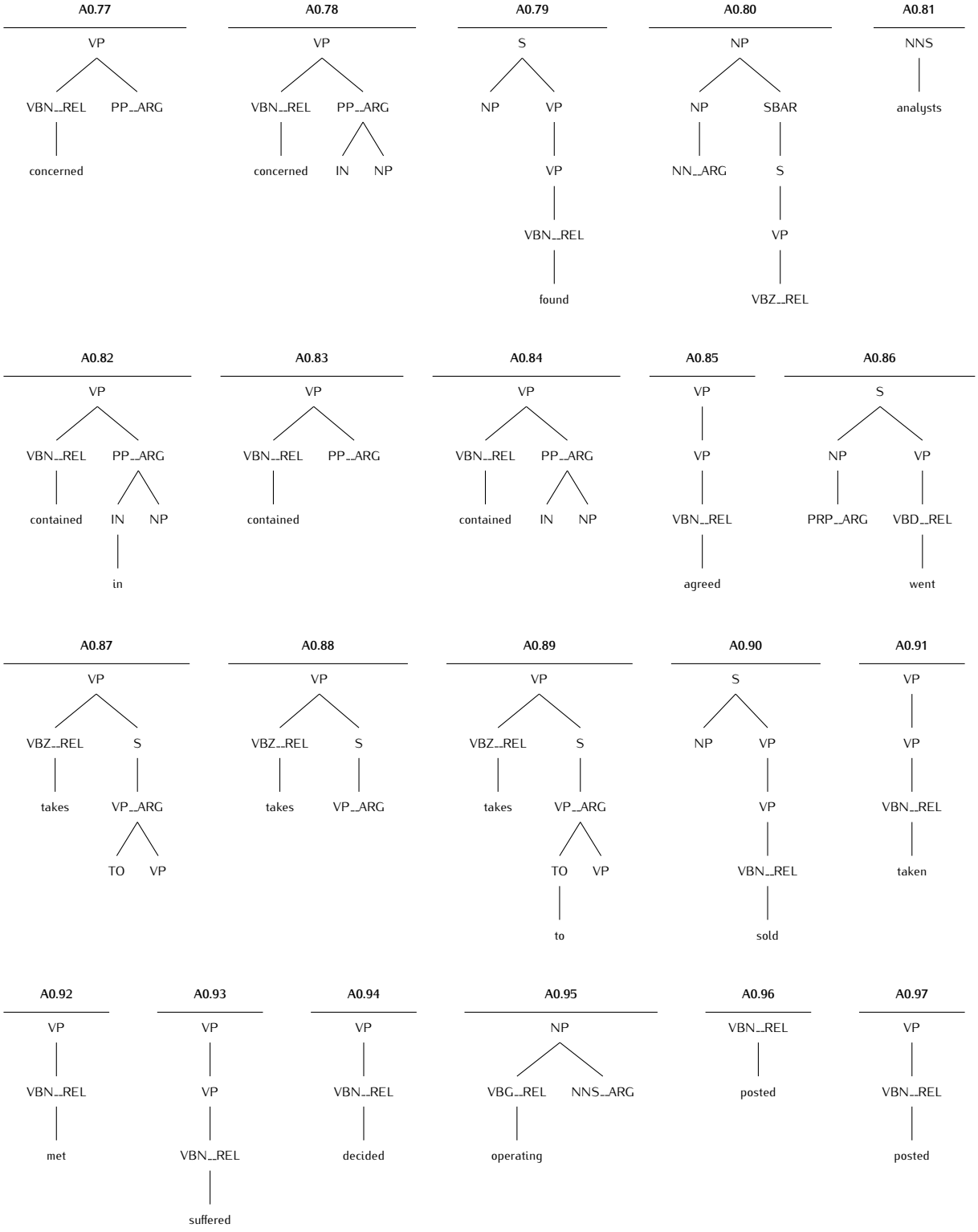


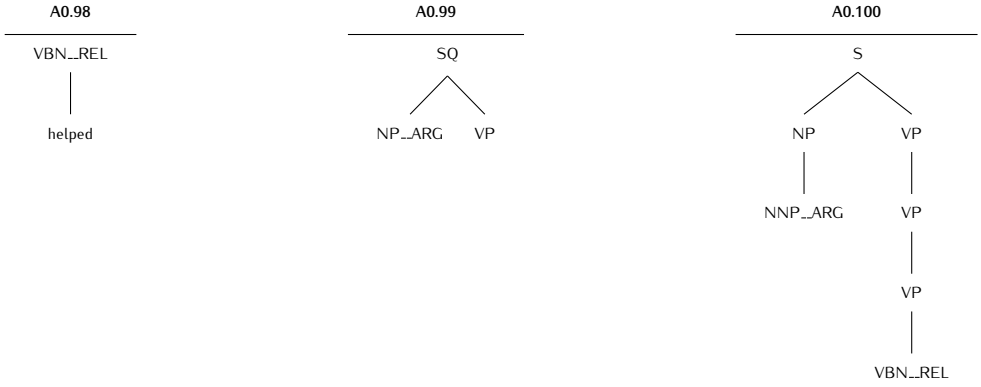
A0.74



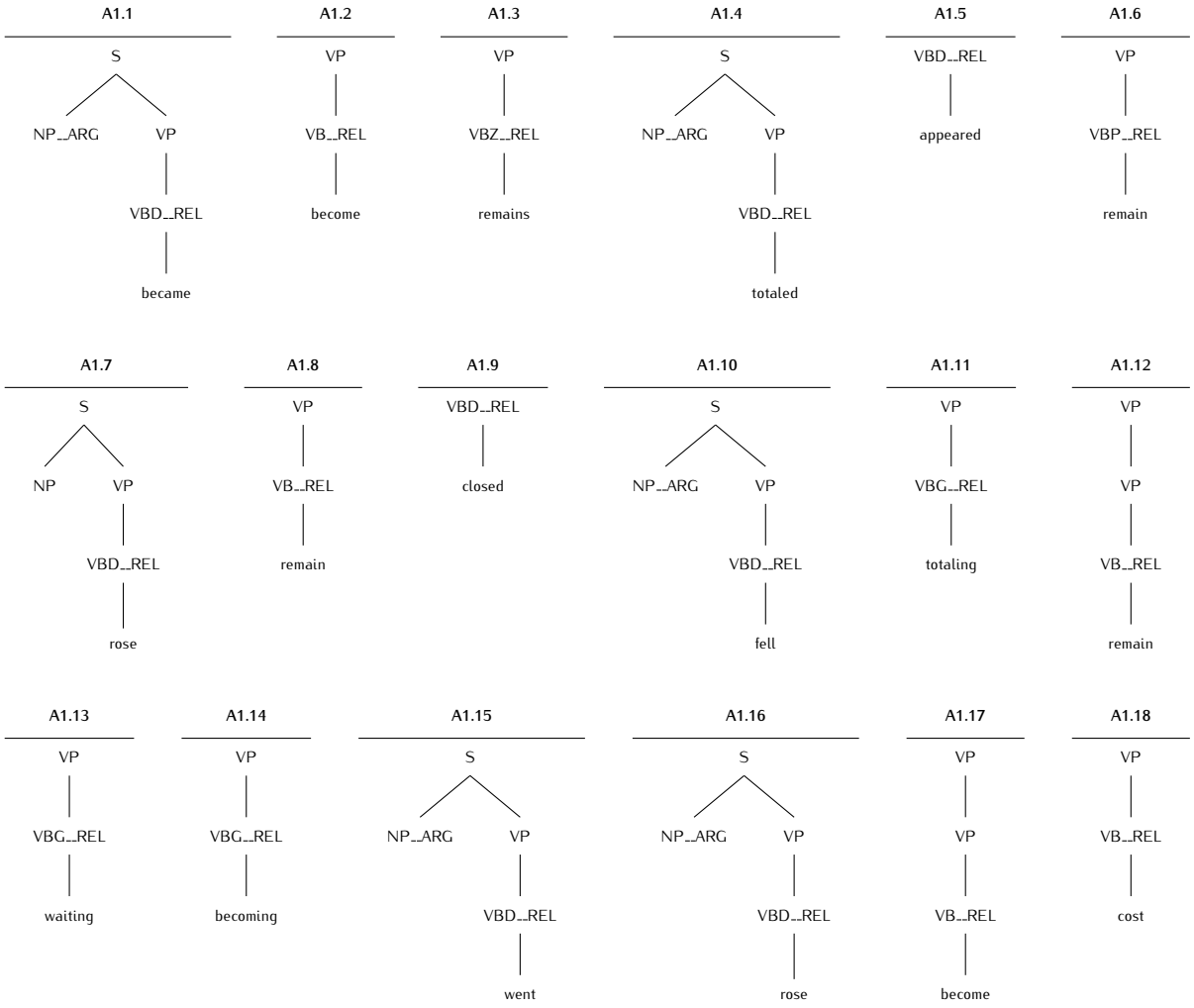
A0.75

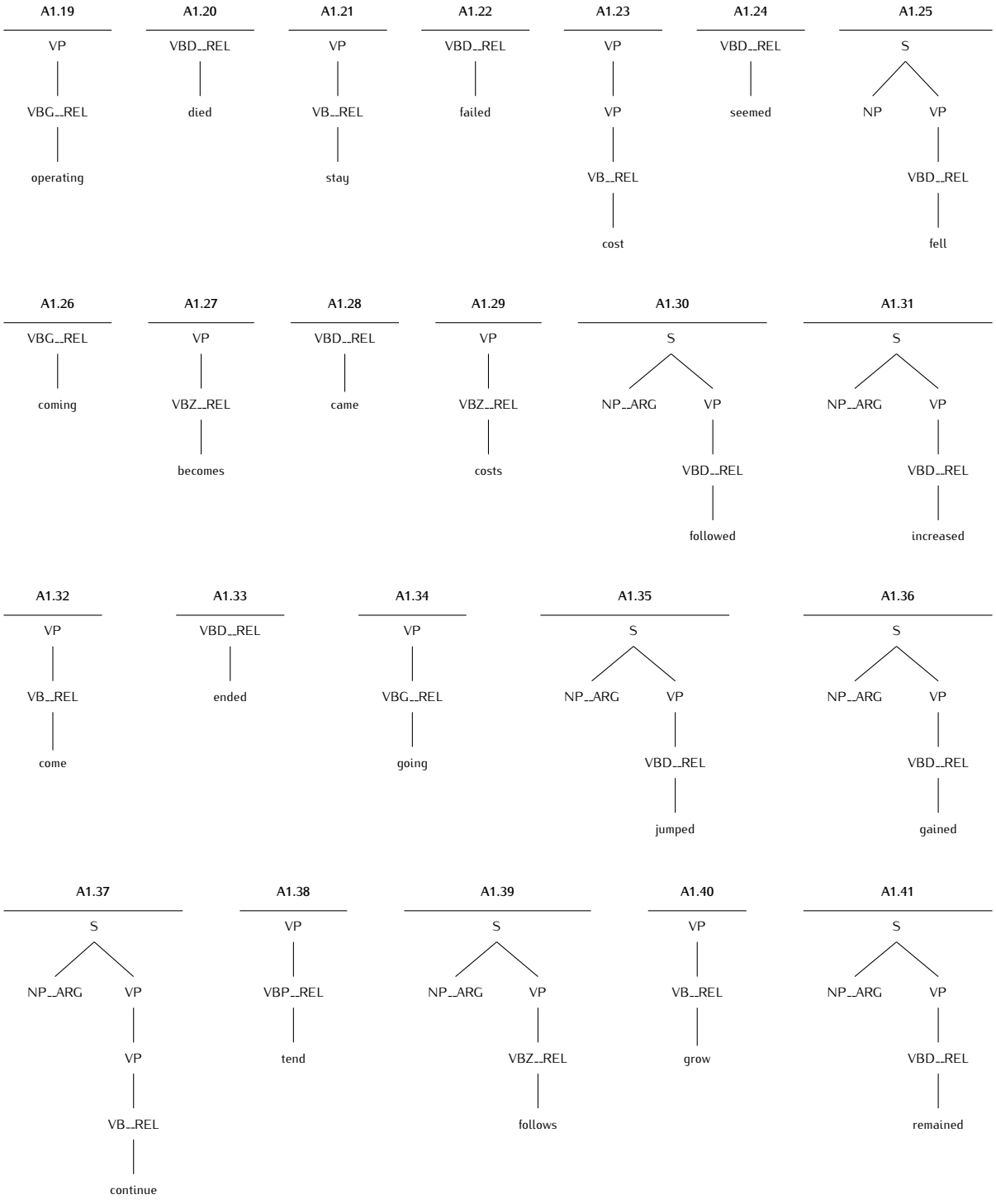
A0.76

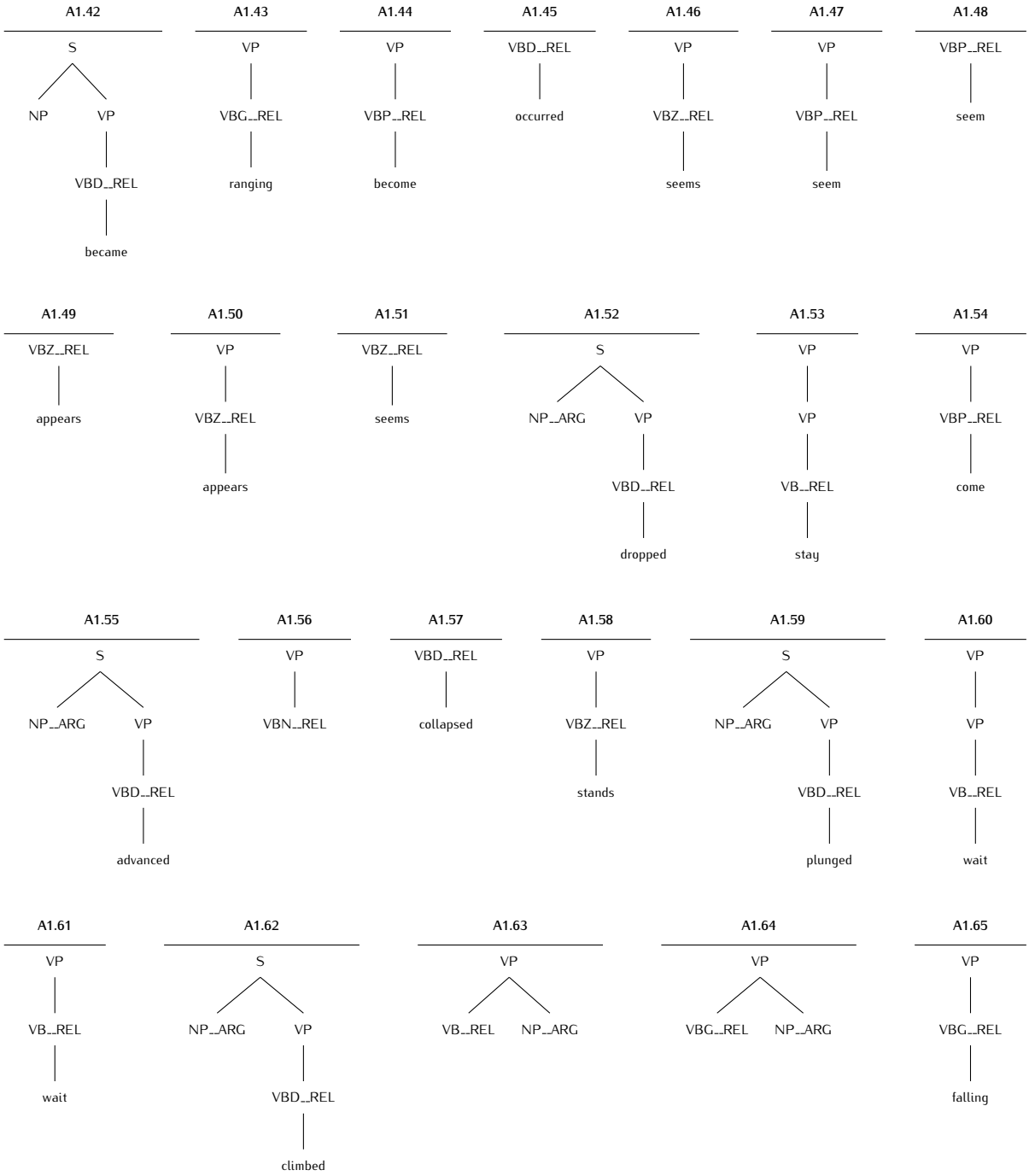

A0.77
VP
VBN_-REL
PP_-ARG
concerned
A0.78
VP
VBN_-REL
PP_-ARG
concerned
IN
NP
A0.79
S
NP
VP
VP
VBN_-REL
found
A0.80
NP
NP
SBAR
NN_-ARG
S
VP
VBZ_-REL
A0.81
NNS
analysts
A0.82
VP
VBN_-REL
PP_-ARG
contained
IN
NP
in
A0.83
VP
VBN_-REL
PP_-ARG
contained
A0.84
VP
VBN_-REL
PP_-ARG
contained
IN
NP
A0.85
VP
VP
VBN_-REL
agreed
A0.86
S
NP
VP
PRP_-ARG
VBD_-REL
went
A0.87
VP
VBZ_-REL
S
takes
VP_-ARG
TO
VP
A0.88
VP
VBZ_-REL
S
takes
VP_-ARG
A0.89
VP
VBZ_-REL
S
takes
VP_-ARG
TO
VP
to
A0.90
S
NP
VP
VP
VBN_-REL
sold
A0.91
VP
VP
VBN_-REL
taken
A0.92
VP
VBN_-REL
met
A0.93
VP
VP
VBN_-REL
suffered
A0.94
VP
VBN_-REL
decided
A0.95
NP
VBG_-REL
NNS_-ARG
operating
A0.96
VBN_-REL
posted
A0.97
VP
VBN_-REL
posted

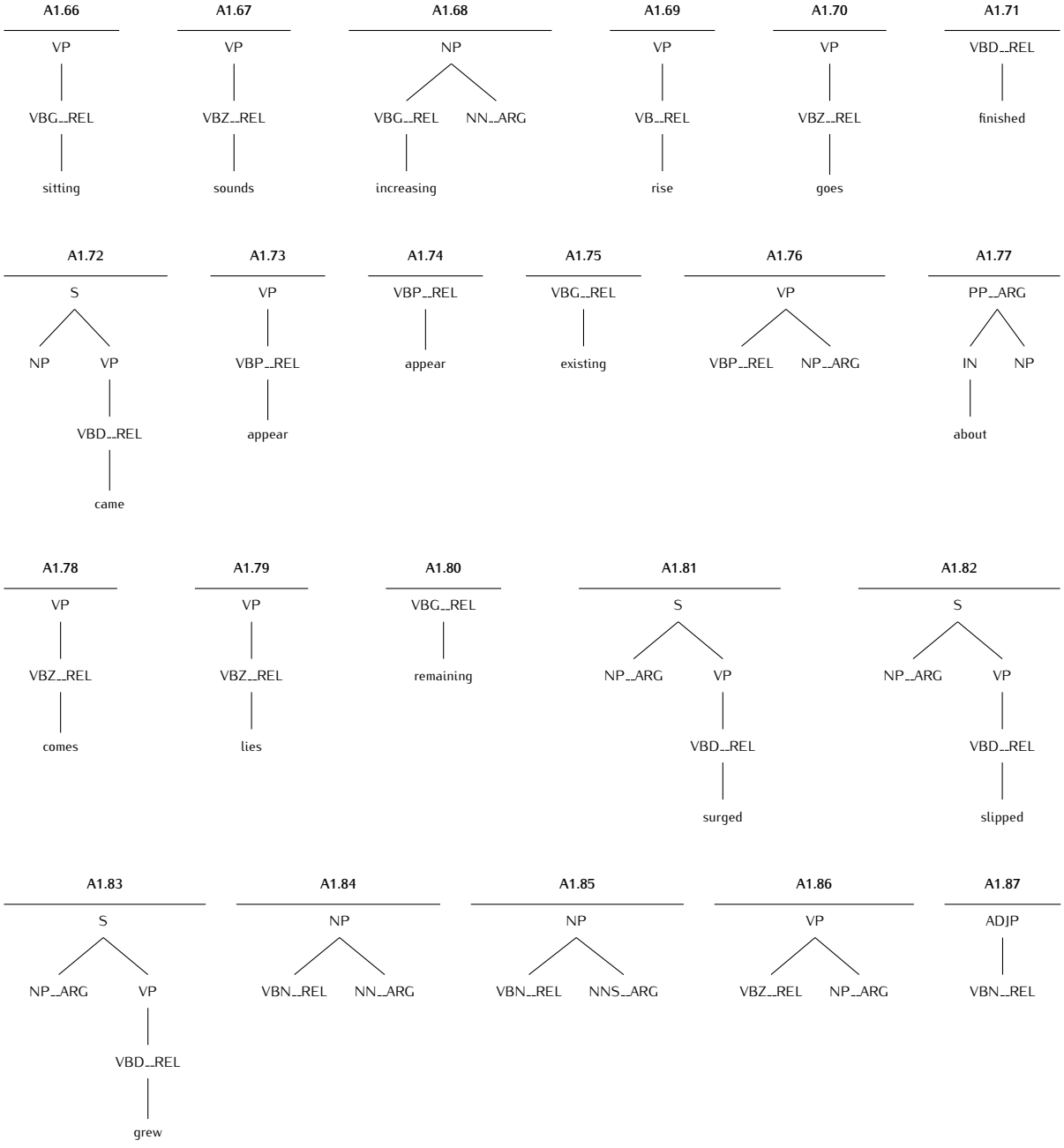


Fragments for Class A1



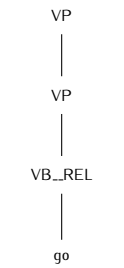

A1.19
VP
VBG_-REL
operating
A1.20
VBD_-REL
died
A1.21
VP
VB_-REL
stay
A1.22
VBD_-REL
failed
A1.23
VP
VP
VB_-REL
cost
A1.24
VBD_-REL
seemed
A1.25
S
NP
VP
VBD_-REL
fell
A1.26
VBG_-REL
coming
A1.27
VP
VBZ_-REL
becomes
A1.28
VBD_-REL
came
A1.29
VP
VBZ_-REL
costs
A1.30
S
NP_-ARG
VP
VBD_-REL
followed
A1.31
S
NP_-ARG
VP
VBD_-REL
increased
A1.32
VP
VB_-REL
come
A1.33
VBD_-REL
ended
A1.34
VP
VBG_-REL
going
A1.35
S
NP_-ARG
VP
VBD_-REL
jumped
A1.36
S
NP_-ARG
VP
VBD_-REL
gained
A1.37
S
NP_-ARG
VP
VP
VB_-REL
continue
A1.38
VP
VBP_-REL
tend
A1.39
S
NP_-ARG
VP
VBZ_-REL
follows
A1.40
VP
VB_-REL
grow
A1.41
S
NP_-ARG
VP
VBD_-REL
remained


A1.42
S
NP
VP
VBD_REL
became
A1.43
VP
VBG_REL
ranging
A1.44
VP
VBP_REL
become
A1.45
VBD_REL
occurred
A1.46
VP
VBZ_REL
seems
A1.47
VP
VBP_REL
seem
A1.48
VBP_REL
seem
A1.49
VBZ_REL
appears
A1.50
VP
VBZ_REL
appears
A1.51
VBZ_REL
seems
A1.52
S
NP_ARG
VP
VBD_REL
dropped
A1.53
VP
VP
VB_REL
stay
A1.54
VP
VBP_REL
come
A1.55
S
NP_ARG
VP
VBD_REL
advanced
A1.56
VP
VBN_REL
A1.57
VBD_REL
collapsed
A1.58
VP
VBZ_REL
stands
A1.59
S
NP_ARG
VP
VBD_REL
plunged
A1.60
VP
VP
VB_REL
wait
A1.61
VP
VB_REL
wait
A1.62
S
NP_ARG
VP
VBD_REL
climbed
A1.63
VP
VB_REL
NP_ARG
A1.64
VP
VBG_REL
NP_ARG
A1.65
VP
VBG_REL
falling

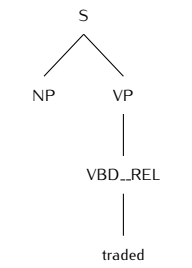

A1.66
VP
VBG--REL
sitting
A1.67
VP
VBZ--REL
sounds
A1.68
NP
VBG--REL
NN--ARG
increasing
A1.69
VP
VB--REL
rise
A1.70
VP
VBZ--REL
goes
A1.71
VBD--REL
finished
A1.72
S
NP
VP
VBD--REL
came
A1.73
VP
VBP--REL
appear
A1.74
VBP--REL
appear
A1.75
VBG--REL
existing
A1.76
VP
VBP--REL
NP--ARG
A1.77
PP--ARG
IN
NP
about
A1.78
VP
VBZ--REL
comes
A1.79
VP
VBZ--REL
lies
A1.80
VBG--REL
remaining
A1.81
S
NP--ARG
VP
VBD--REL
surged
A1.82
S
NP--ARG
VP
VBD--REL
slipped
A1.83
S
NP--ARG
VP
VBD--REL
grew
A1.84
NP
VBN--REL
NN--ARG
A1.85
NP
VBN--REL
NNS--ARG
A1.86
VP
VBZ--REL
NP--ARG
A1.87
ADJP
VBN--REL

A1.88

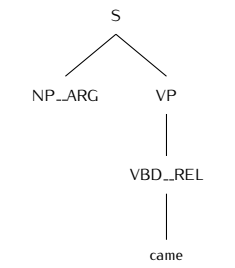
A1.89



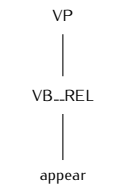
A1.90



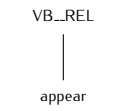
A1.91



A1.92

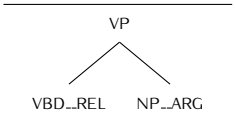


A1.93

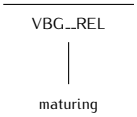


A1.94

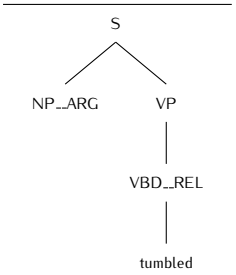
A1.95



A1.96

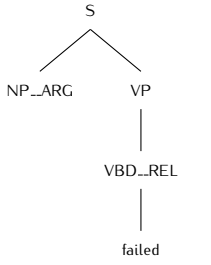


A1.97

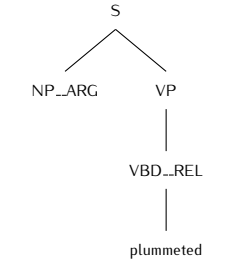


A1.98

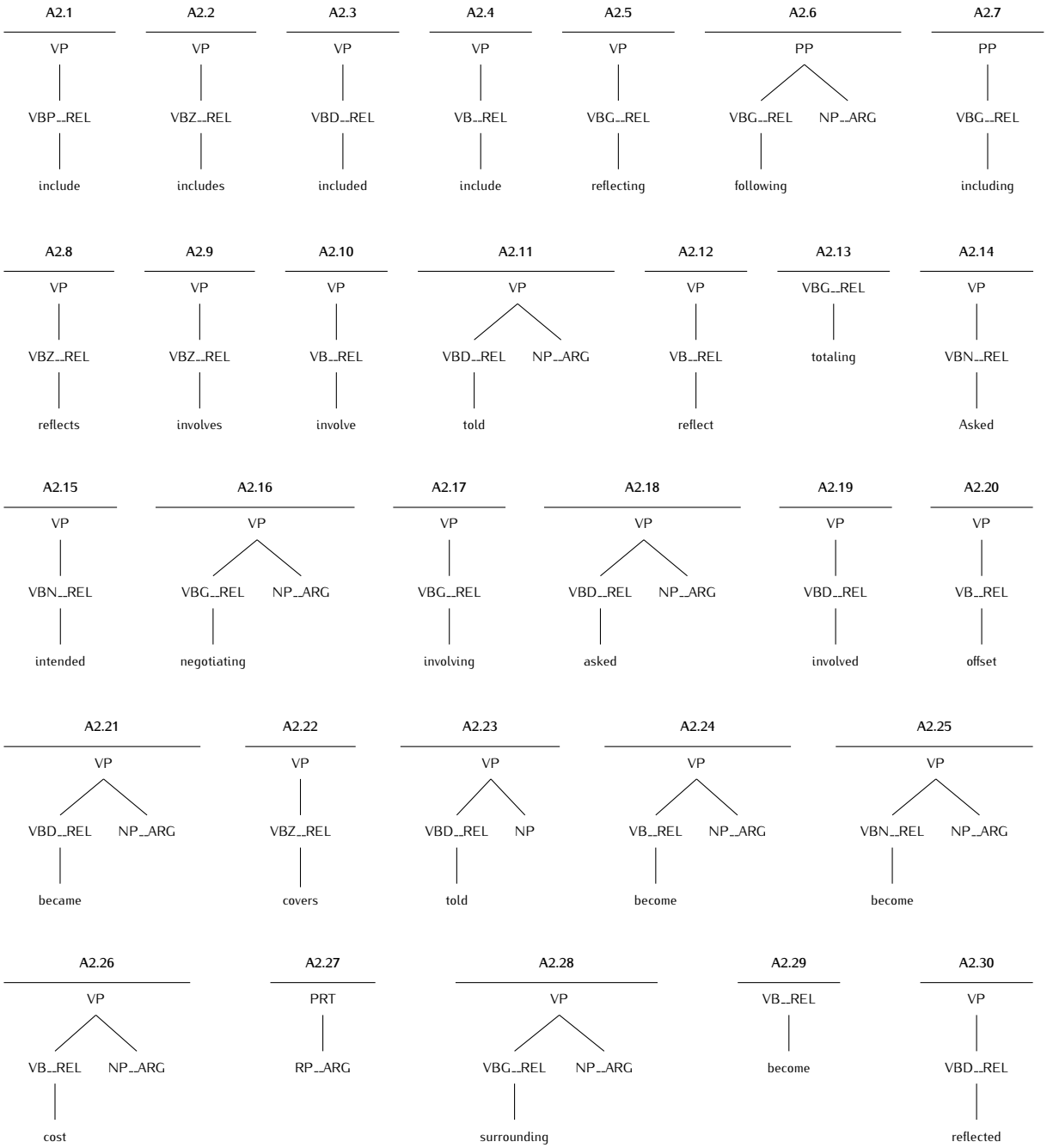
A1.99

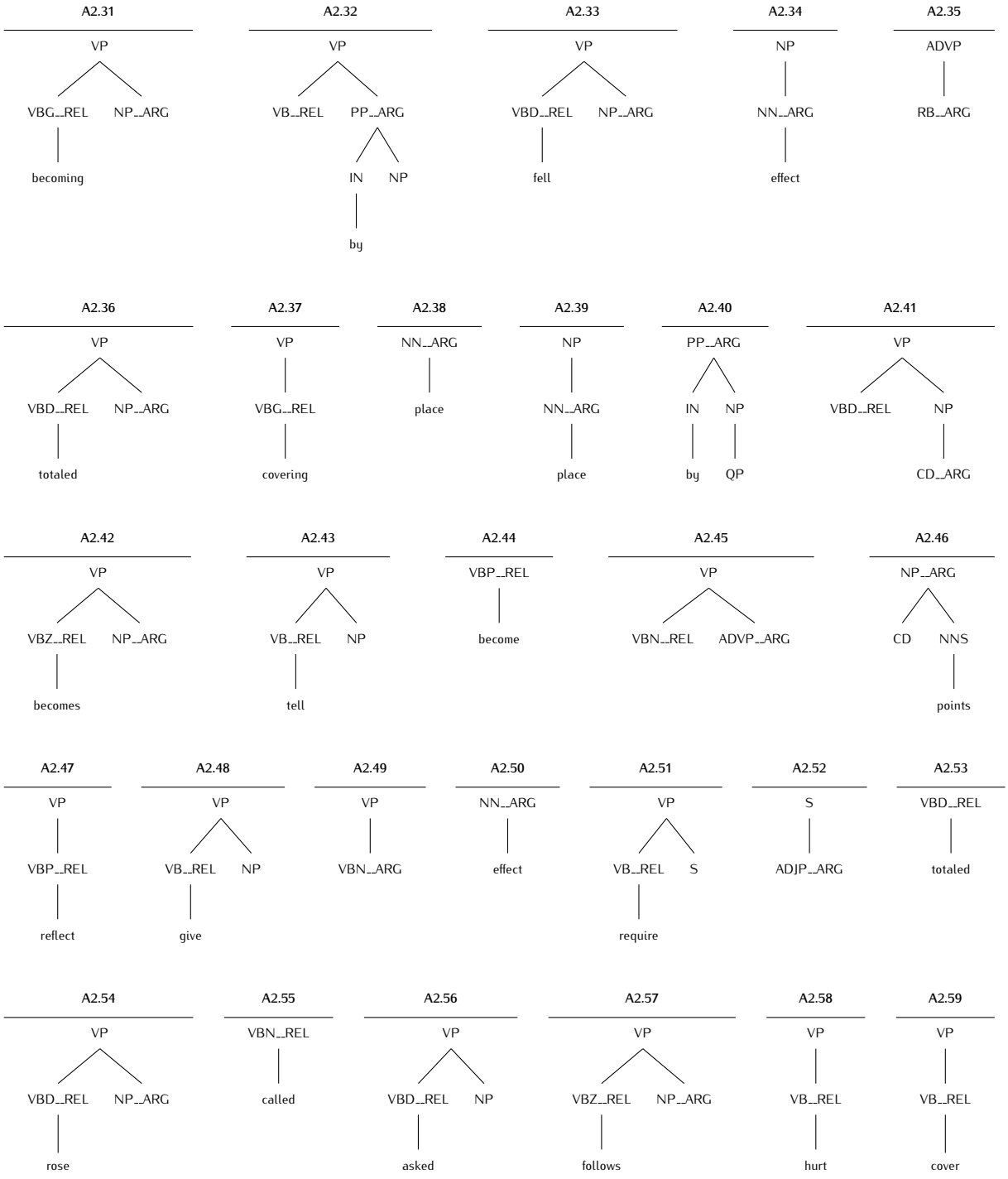


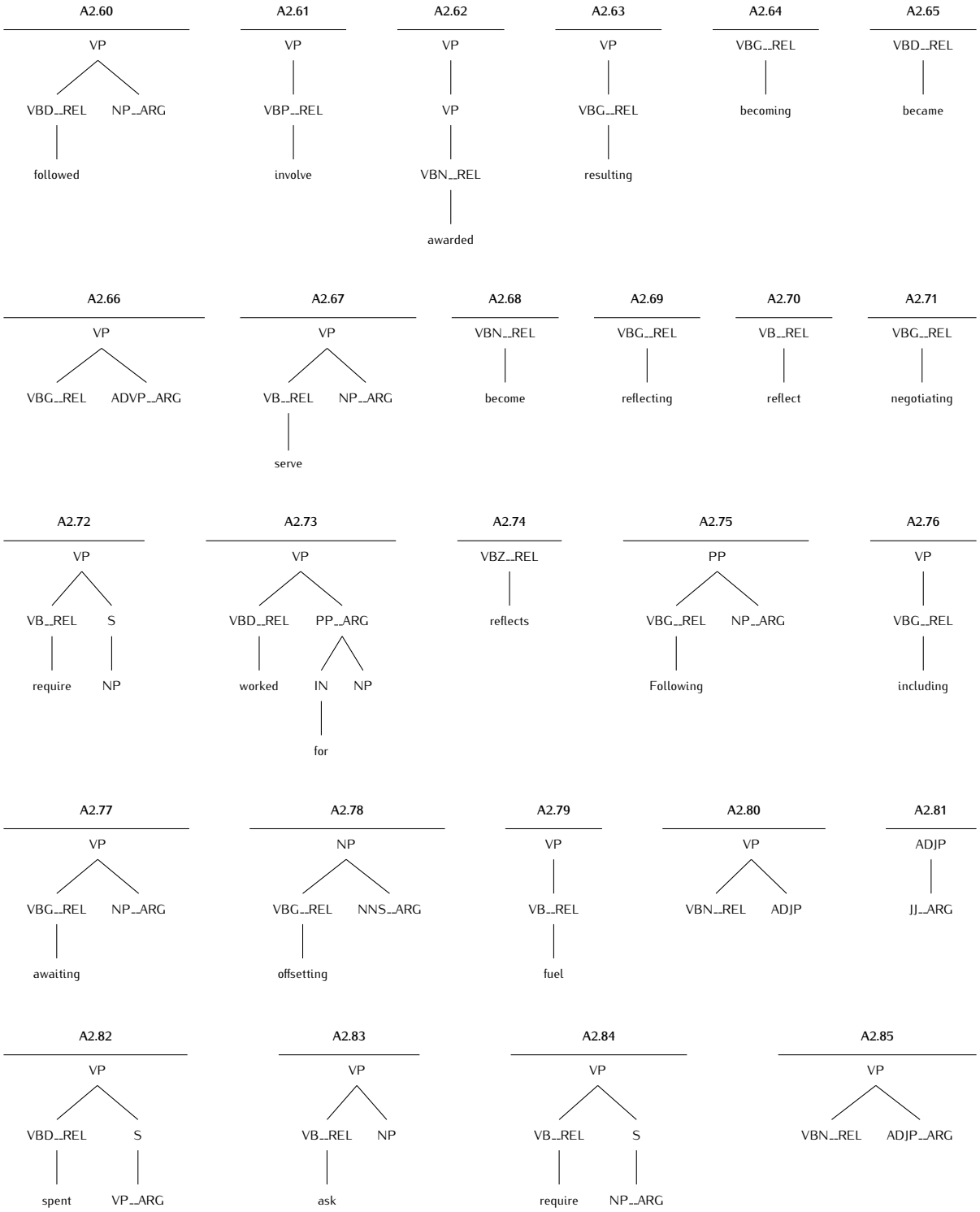
A1.100

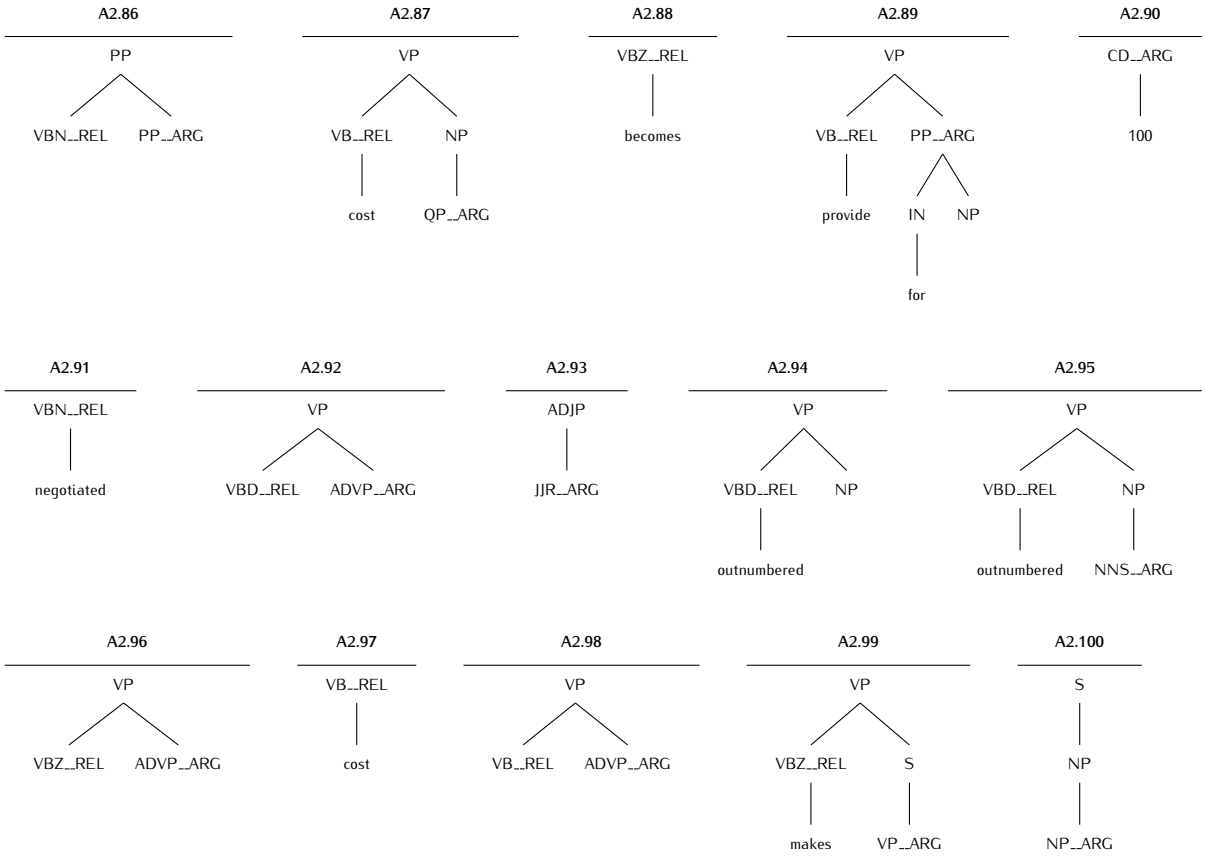


Fragments for Class A2

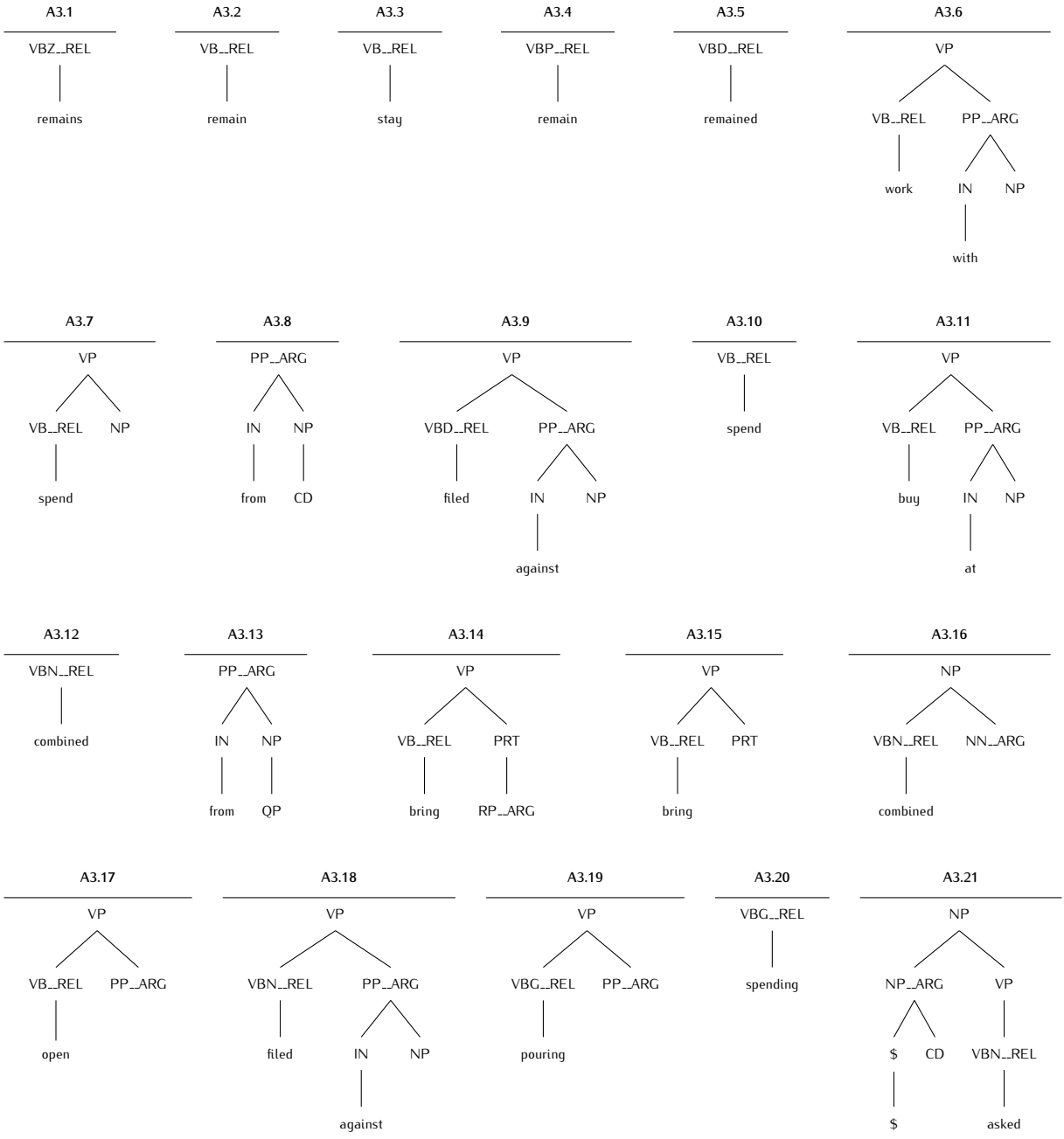



A2.31
VP
VBG_REL
NP_ARG
becoming
A2.32
VP
VB_REL
PP_ARG
IN
NP
by
A2.33
VP
VBD_REL
NP_ARG
fell
A2.34
NP
NN_ARG
effect
A2.35
ADVP
RB_ARG
A2.36
VP
VBD_REL
NP_ARG
totaled
A2.37
VP
VBG_REL
covering
A2.38
NN_ARG
place
A2.39
NP
NN_ARG
place
A2.40
PP_ARG
IN
NP
by
QP
A2.41
VP
VBD_REL
NP
CD_ARG
A2.42
VP
VBZ_REL
NP_ARG
becomes
A2.43
VP
VB_REL
NP
tell
A2.44
VBP_REL
become
A2.45
VP
VBN_REL
ADVP_ARG
A2.46
NP_ARG
CD
NNS
points
A2.47
VP
VBP_REL
reflect
A2.48
VP
VB_REL
NP
give
A2.49
VP
VBN_ARG
A2.50
NN_ARG
effect
A2.51
VP
VB_REL
S
require
A2.52
S
ADJP_ARG
A2.53
VBD_REL
totaled
A2.54
VP
VBD_REL
NP_ARG
rose
A2.55
VBN_REL
called
A2.56
VP
VBD_REL
NP
asked
A2.57
VP
VBZ_REL
NP_ARG
follows
A2.58
VP
VB_REL
hurt
A2.59
VP
VB_REL
cover

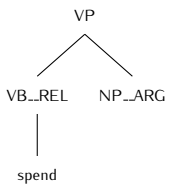

A2.60
VP
VBD--REL
NP--ARG
followed
A2.61
VP
VBP--REL
involve
A2.62
VP
VP
VBN--REL
awarded
A2.63
VP
VBG--REL
resulting
A2.64
VBG--REL
becoming
A2.65
VBD--REL
became
A2.66
VP
VBG--REL
ADVP--ARG
A2.67
VP
VB--REL
NP--ARG
serve
A2.68
VBN--REL
become
A2.69
VBG--REL
reflecting
A2.70
VB--REL
reflect
A2.71
VBG--REL
negotiating
A2.72
VP
VB--REL
S
require
NP
A2.73
VP
VBD--REL
PP--ARG
worked
IN
NP
for
A2.74
VBZ--REL
reflects
A2.75
PP
VBG--REL
NP--ARG
Following
A2.76
VP
VBG--REL
including
A2.77
VP
VBG--REL
NP--ARG
awaiting
A2.78
NP
VBG--REL
NNS--ARG
offsetting
A2.79
VP
VB--REL
fuel
A2.80
VP
VBN--REL
ADJP
A2.81
ADJP
JJ--ARG
A2.82
VP
VBD--REL
S
spent
VP--ARG
A2.83
VP
VB--REL
NP
ask
A2.84
VP
VB--REL
S
require
NP--ARG
A2.85
VP
VBN--REL
ADJP--ARG


A2.86
PP
VBN_REL
PP_ARG
A2.87
VP
VB_REL
NP
cost
QP_ARG
A2.88
VBZ_REL
becomes
A2.89
VP
VB_REL
PP_ARG
provide
IN
NP
for
A2.90
CD_ARG
100
A2.91
VBN_REL
negotiated
A2.92
VP
VBD_REL
ADVP_ARG
A2.93
ADJP
JJR_ARG
A2.94
VP
VBD_REL
NP
outnumbered
A2.95
VP
VBD_REL
NP
outnumbered
NNS_ARG
A2.96
VP
VBZ_REL
ADVP_ARG
A2.97
VB_REL
cost
A2.98
VP
VB_REL
ADVP_ARG
A2.99
VP
VBZ_REL
S
makes
VP_ARG
A2.100
S
NP
NP_ARG

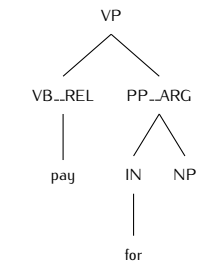
Fragments for Class A3



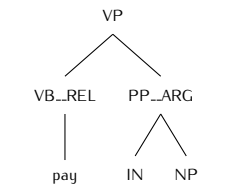
A3.22



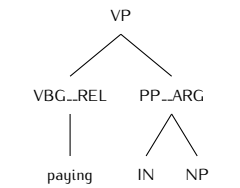
A3.23



A3.24

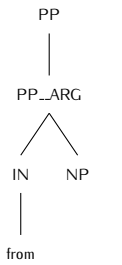


A3.25

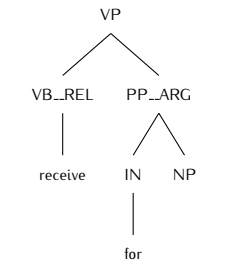


A3.26

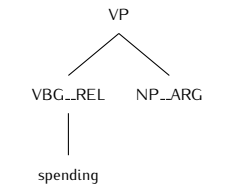
A3.27



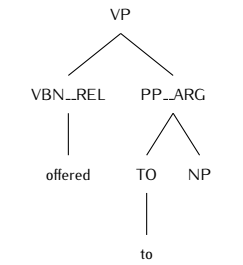
A3.28



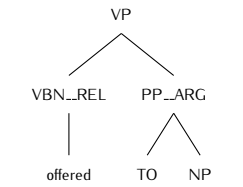
A3.29



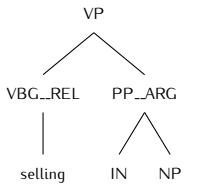
A3.30



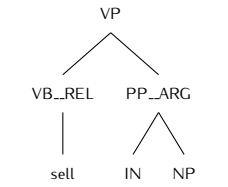
A3.31



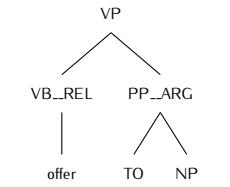
A3.32



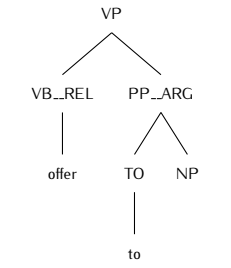
A3.33



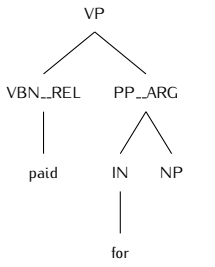
A3.34



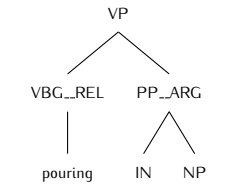
A3.35



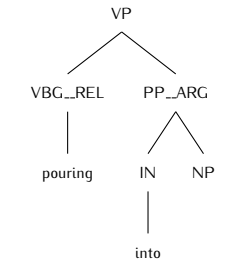
A3.36



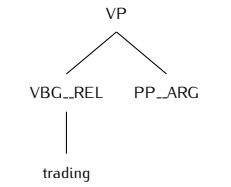
A3.37



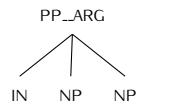
A3.38



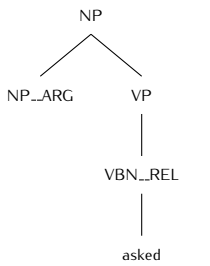
A3.39



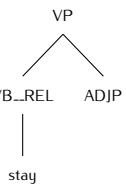
A3.40



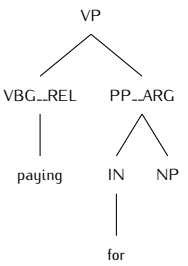
A3.41



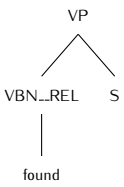
A3.42



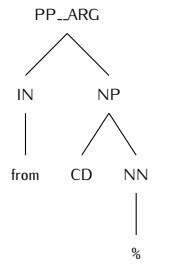
A3.43



A3.44

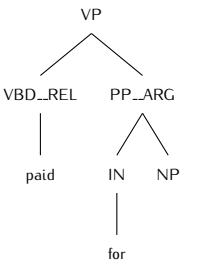


A3.45

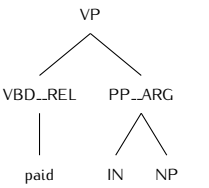


A3.46

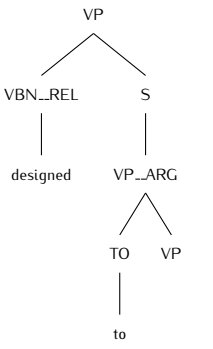
A3.47



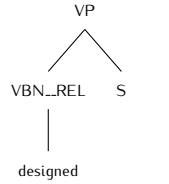
A3.48



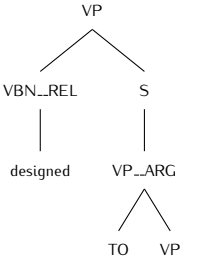
A3.49



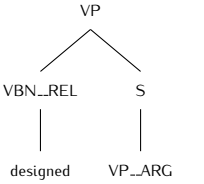
A3.50



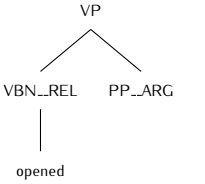
A3.51



A3.52

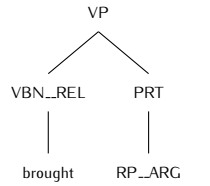


A3.53



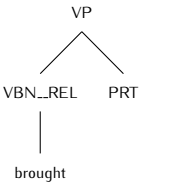
A3.54

A3.55

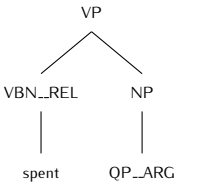


A3.56

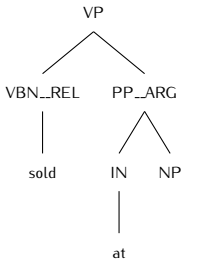
A3.57



A3.58

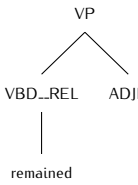


A3.59

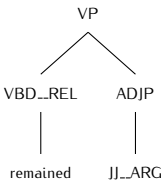


A3.60

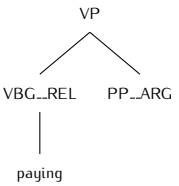
A3.61



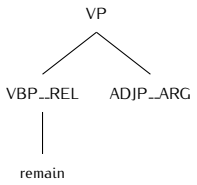
A3.62



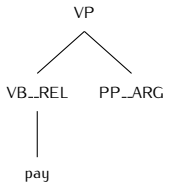
A3.63



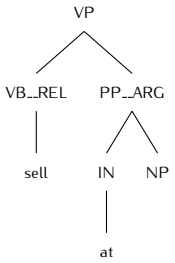
A3.64



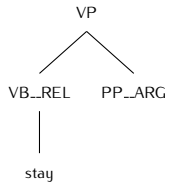
A3.65



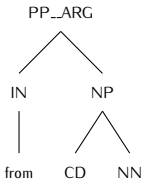
A3.66



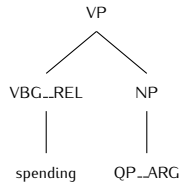
A3.67



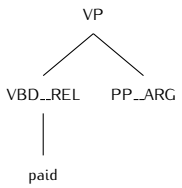
A3.68



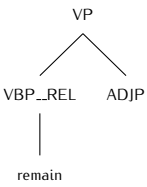
A3.69



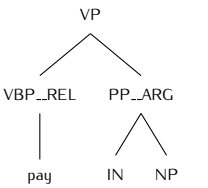
A3.70



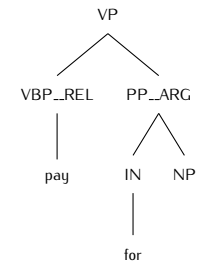
A3.71



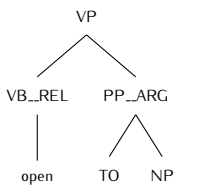
A3.72



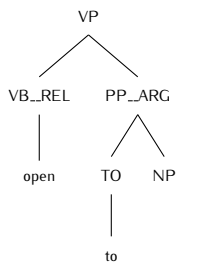
A3.73



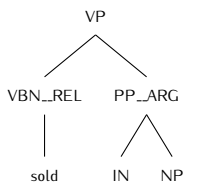
A3.74



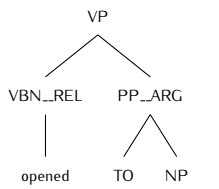
A3.75



A3.76

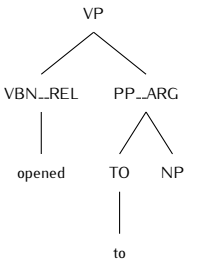


A3.77

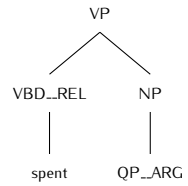


A3.78

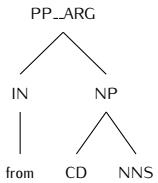
A3.79



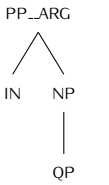
A3.80



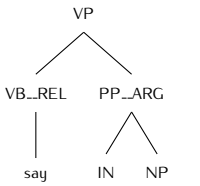
A3.81



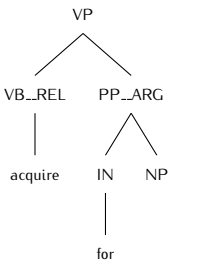
A3.82



A3.83



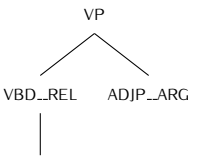
A3.84



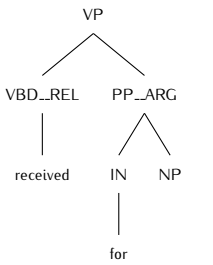
A3.85

A3.86

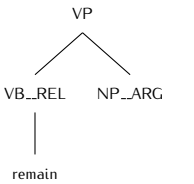
A3.87



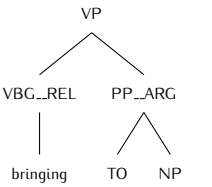
A3.88



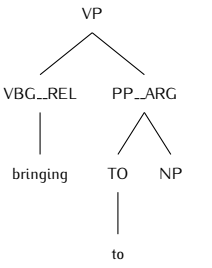
A3.89



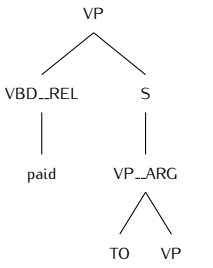
A3.90



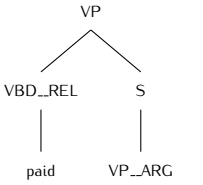
A3.91



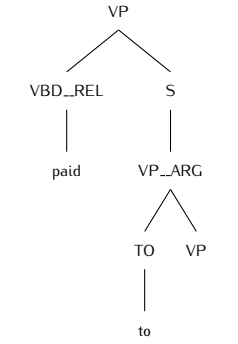
A3.92



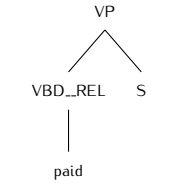
A3.93



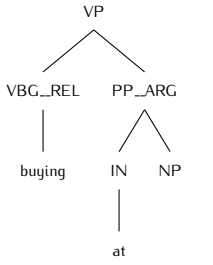
A3.94

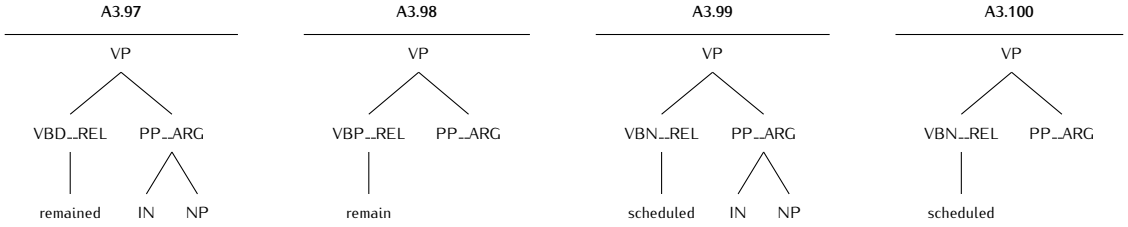


A3.95

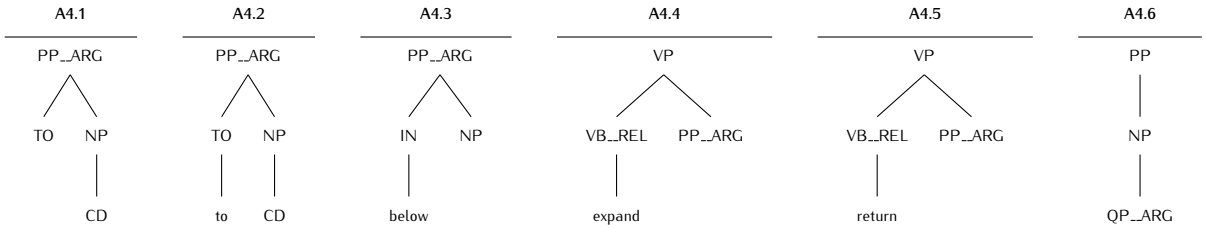


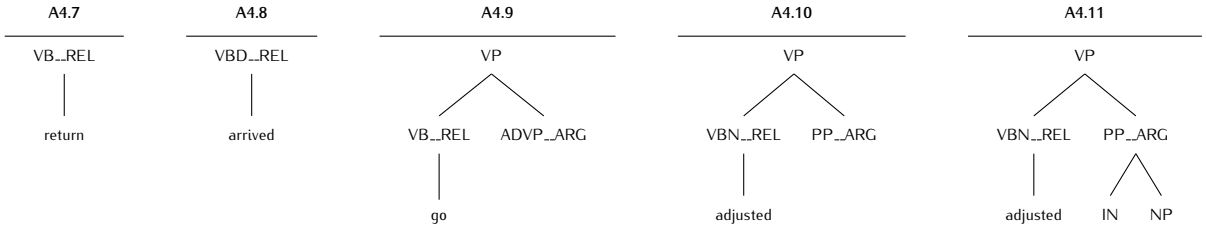
A3.96

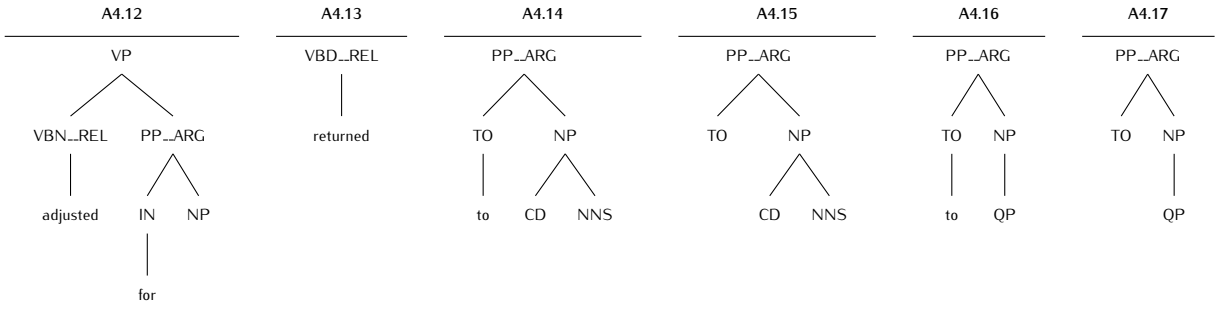


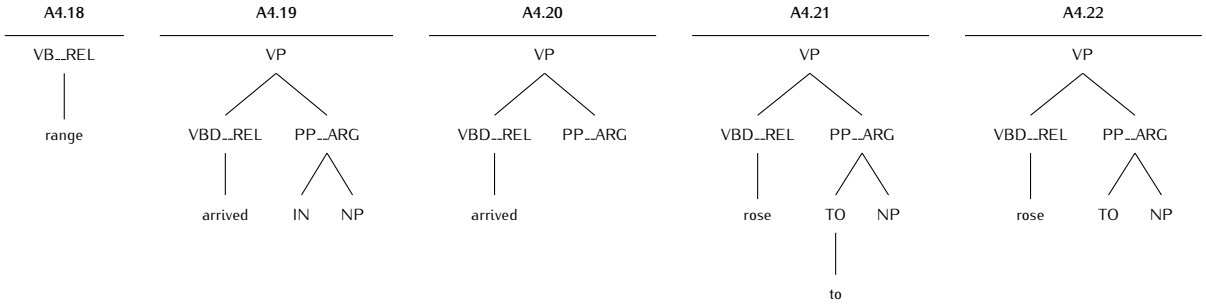


Fragments for Class A4

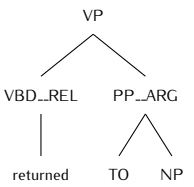




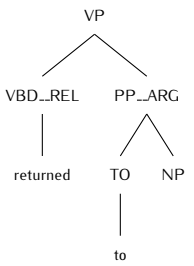




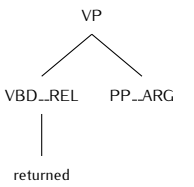
A4.23



A4.24

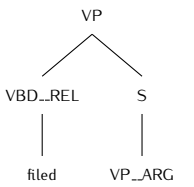


A4.25

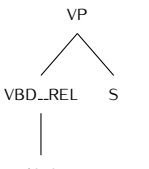


A4.26

A4.27

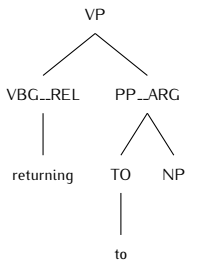


A4.28

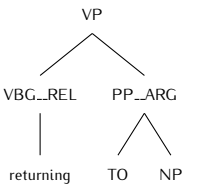


A4.29

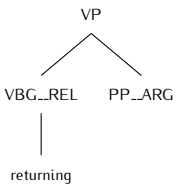
A4.30



A4.31



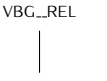
A4.32



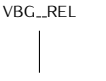
A4.33

A4.34

A4.35

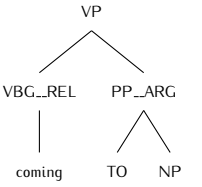


A4.36

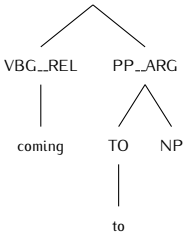


A4.37

A4.38



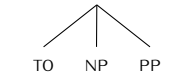
A4.39



A4.40

A4.41

A4.42



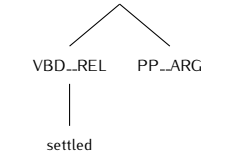
A4.43

A4.44

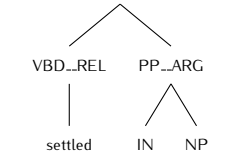
A4.45

A4.46

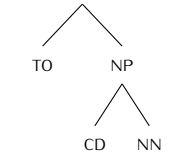
A4.47



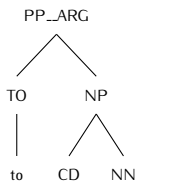
A4.48



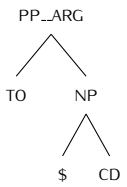
A4.49



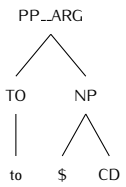
A4.50



A4.51

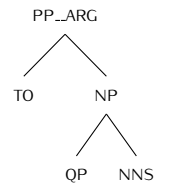


A4.52

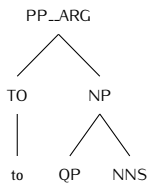


A4.53

A4.54

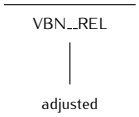


A4.55

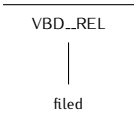


A4.56

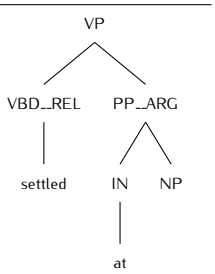
A4.57



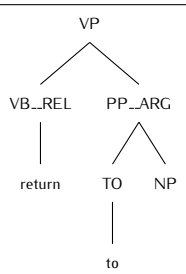
A4.58



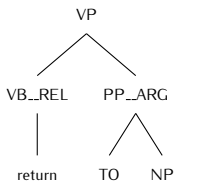
A4.59



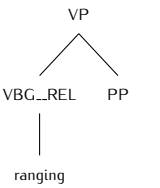
A4.60



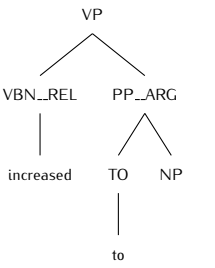
A4.61



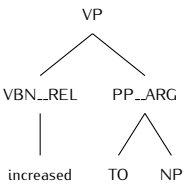
A4.62



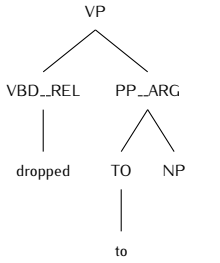
A4.63



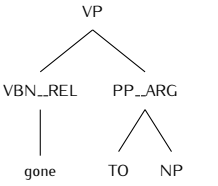
A4.64



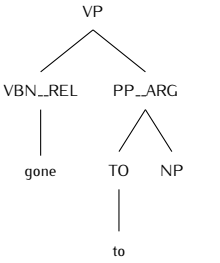
A4.65



A4.66

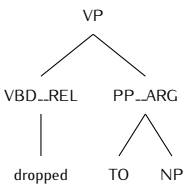


A4.67

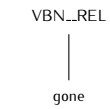


A4.68

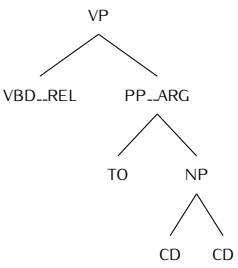
A4.69



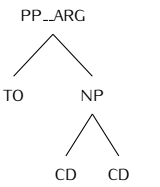
A4.70



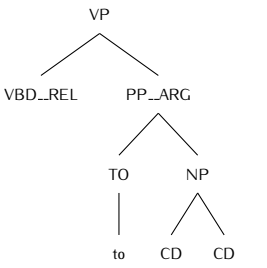
A4.71



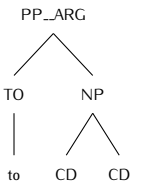
A4.72



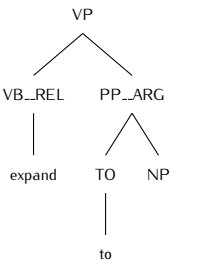
A4.73



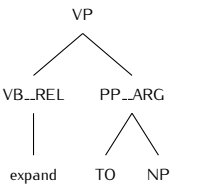
A4.74



A4.75

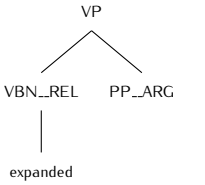


A4.76



A4.77

A4.78

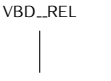


A4.79

A4.80

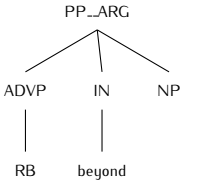
A4.81

A4.82

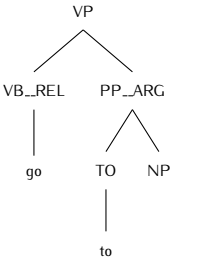


A4.83

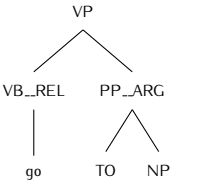
A4.84



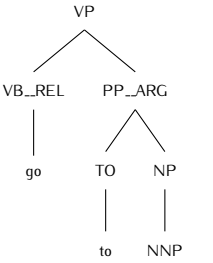
A4.85



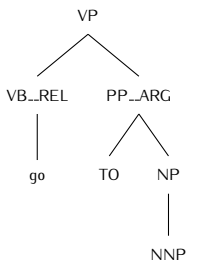
A4.86



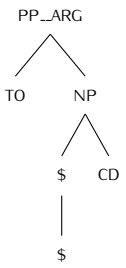
A4.87



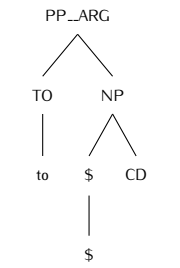
A4.88



A4.89

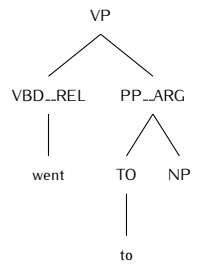


A4.90

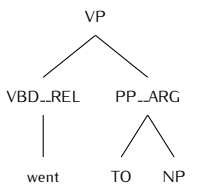


A4.91

A4.92

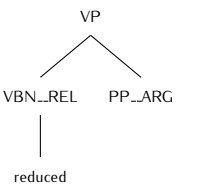


A4.93



A4.94

A4.95

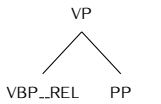


A4.96

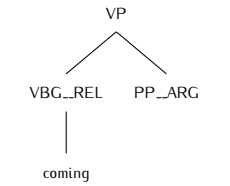
A4.97

A4.98

A4.99

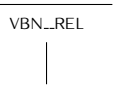


A4.100

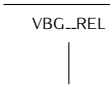


Fragments for Class A5

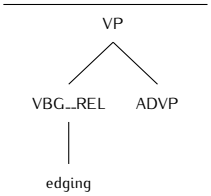
A5.1



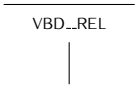
A5.2



A5.3

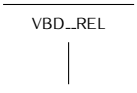


A5.4

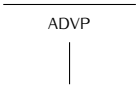


A5.5

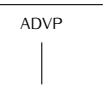
A5.6



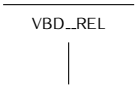
A5.7



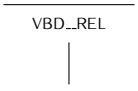
A5.8



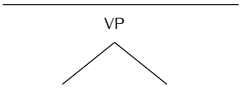
A5.9



A5.10



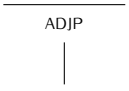
A5.11

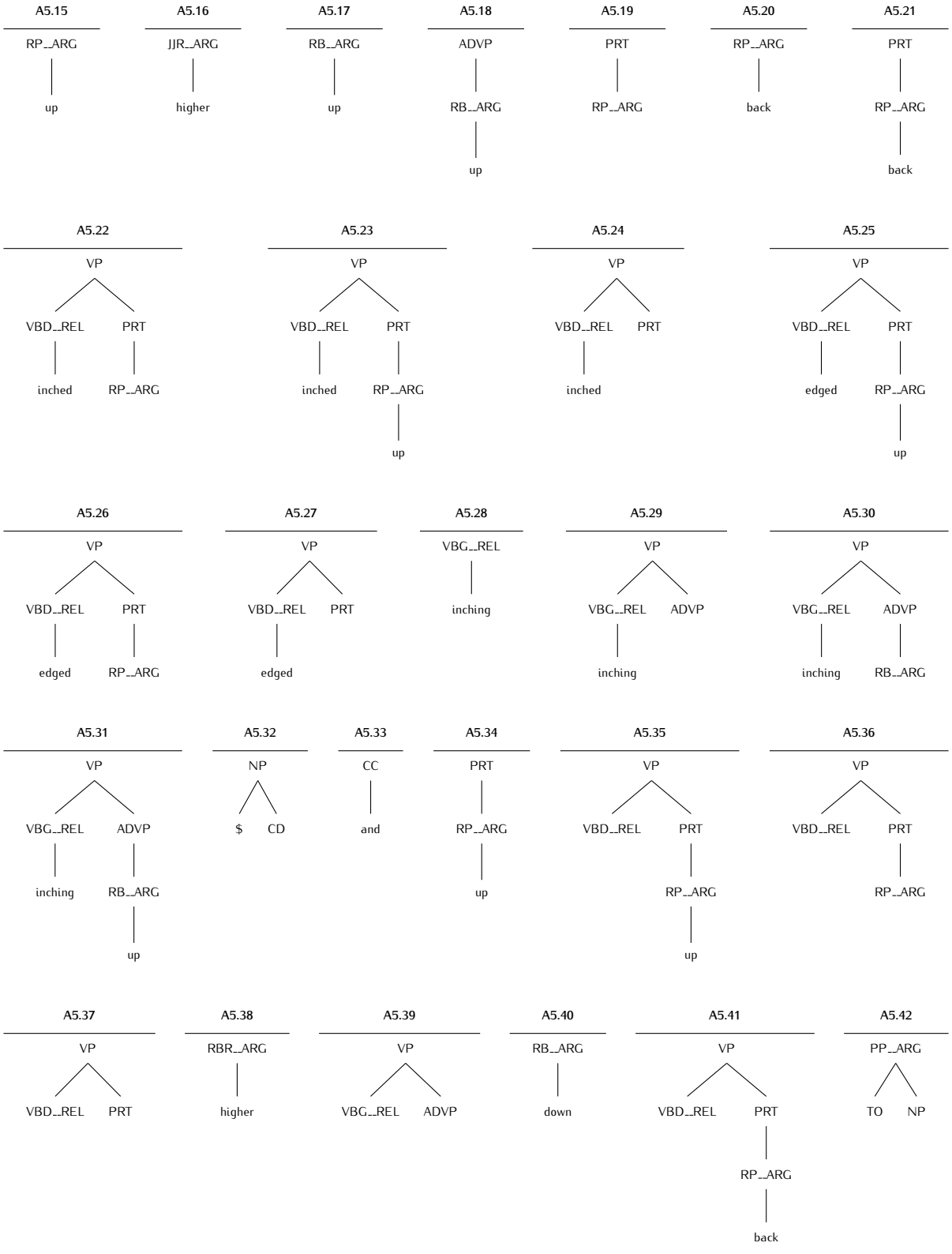


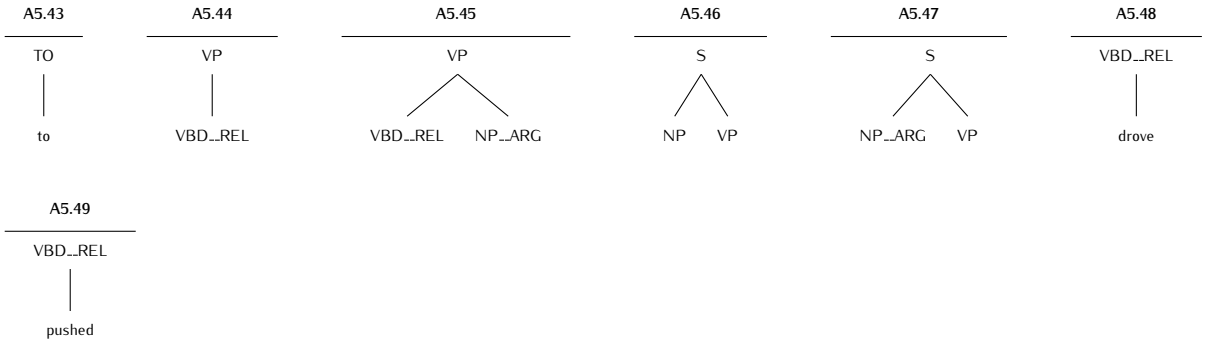
A5.12

A5.13

A5.14




A5.15
RP--ARG
up
A5.16
JJR--ARG
higher
A5.17
RB--ARG
up
A5.18
ADVP
RB--ARG
up
A5.19
PRT
RP--ARG
A5.20
RP--ARG
back
A5.21
PRT
RP--ARG
back
A5.22
VP
VBD--REL
PRT
inched
RP--ARG
A5.23
VP
VBD--REL
PRT
inched
RP--ARG
up
A5.24
VP
VBD--REL
PRT
inched
A5.25
VP
VBD--REL
PRT
edged
RP--ARG
up
A5.26
VP
VBD--REL
PRT
edged
RP--ARG
A5.27
VP
VBD--REL
PRT
edged
A5.28
VBG--REL
inching
A5.29
VP
VBG--REL
ADVP
inching
A5.30
VP
VBG--REL
ADVP
inching
RB--ARG
A5.31
VP
VBG--REL
ADVP
inching
RB--ARG
up
A5.32
NP
$
CD
A5.33
CC
and
A5.34
PRT
RP--ARG
up
A5.35
VP
VBD--REL
PRT
RP--ARG
up
A5.36
VP
VBD--REL
PRT
RP--ARG
A5.37
VP
VBD--REL
PRT
A5.38
RBR--ARG
higher
A5.39
VP
VBG--REL
ADVP
A5.40
RB--ARG
down
A5.41
VP
VBD--REL
PRT
RP--ARG
back
A5.42
PP--ARG
TO
NP


A5.43
TO
to
A5.44
VP
VBD--REL
A5.45
VP
VBD--REL
NP--ARG
A5.46
S
NP
VP
A5.47
S
NP--ARG
VP
A5.48
VBD--REL
drove
A5.49
VBD--REL
pushed

Bibliography

[Aiolli et al., 2006] Aiolli, F., Martino, G. D. S., Sperduti, A., and Moschitti, A. (2006). Fast On-line Kernel Learning for Trees. In *Proceedings of ICDM'06*. (Cited on page 72).

[Aizerman et al., 1964] Aizerman, M., Braverman, E., and Rozonoer, L. (1964). Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837. (Cited on page 20).

[Aksu et al., 2008] Aksu, Y., Miller, D., and Kesidis, G. (2008). Margin-based feature selection techniques for support vector machine classification. In *First IAPR Workshop on Cognitive Information Processing*, pages 176–181, Santorini, Greece. (Cited on pages 3 and 38).

[Bartlett and Shawe-Taylor, 1998] Bartlett, P. and Shawe-Taylor, J. (1998). *Advances in Kernel Methods –– Support Vector Learning*, chapter Generalization Performance of Support Vector Machines and other Pattern Classifiers. MIT Press. (Cited on page 16).

[Bellman, 1961] Bellman, R. (1961). Adaptive control processes - A guided tour. *Naval Research Logistics Quarterly*, 8(3):315–316. (Cited on page 29).

[Boser et al., 1992] Boser, B. E., Guyon, I., and Vapnik, V. (1992). A Training Algorithm for Optimal Margin Classifiers. In *Proceedings of the 5th Annual Workshop on Computational learning theory*. (Cited on page 16).

[Cancedda et al., 2003] Cancedda, N., Gaussier, E., Goutte, C., and Renders, J. M. (2003). Word sequence kernels. *Journal of Machine Learning Research*, 3:1059–1082. (Cited on pages 35 and 40).

[Cao et al., 2007] Cao, B., Shen, D., Sun, J.-T., Yang, Q., and Chen, Z. (2007). Feature selection in a kernel space. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 121–128, New York, NY, USA. ACM. (Cited on pages 3 and 39).

[Carreras and Màrquez, 2005] Carreras, X. and Màrquez, L. (2005). Introduction to the CoNLL-2005 Shared Task: Semantic Role Labeling. In *Proceedings of CoNLL'05*. (Cited on page 91).

[Charniak, 2000] Charniak, E. (2000). A maximum-entropy-inspired parser. In *Proceedings of NAACL'00*. (Cited on page 91).

[Collins and Duffy, 2001] Collins, M. and Duffy, N. (2001). Convolution kernels for natural language. In *Advances in Neural Information Processing Systems 14*, pages 625–632. MIT Press. (Cited on pages 25, 34, 40, and 64).

[Collins and Duffy, 2002] Collins, M. and Duffy, N. (2002). New Ranking Algorithms for Parsing and Tagging: Kernels over Discrete Structures, and the Voted Perceptron. In *Proceedings of ACL'02*. (Cited on pages 2, 25, 34, 36, 51, and 64).

[Cortes and Vapnik, 1995] Cortes, C. and Vapnik, V. (1995). Support-Vector Networks. *Machine Learning*, 20(3):273–297. (Cited on page 19).

[Culotta and Sorensen, 2004] Culotta, A. and Sorensen, J. (2004). Dependency Tree Kernels for Relation Extraction. In *Proceedings of ACL'04*. (Cited on pages 2 and 36).

[Cumby and Roth, 2003] Cumby, C. and Roth, D. (2003). Kernel Methods for Relational Learning. In *Proceedings of ICML 2003*. (Cited on page 2).

[Diab et al., 2008] Diab, M., Moschitti, A., and Pighin, D. (2008). Semantic Role Labeling Systems for Arabic Using Kernel Methods. In *Proceedings of ACL-08: HLT*, pages 798–806. (Cited on pages 2 and 36).

[Doddington et al., 2004] Doddington, G., Mitchell, A., Przybocki, M., Ramshaw, L., Strassel, S., and Weischedel, R. (2004). The Automatic Content Extraction (ACE) Program–Tasks, Data, and Evaluation. *Proceedings of LREC 2004*, pages 837–840. (Cited on page 90).

[Freund and Schapire, 1999] Freund, Y. and Schapire, R. E. (1999). Large Margin Classification Using the Perceptron Algorithm. *Machine Learning*, 37(3):277–296. (Cited on pages 23 and 34).

[Gildea and Jurafsky, 2002] Gildea, D. and Jurafsky, D. (2002). Automatic labeling of semantic roles. *Computational Linguistics*, 28:245–288. (Cited on pages 36 and 112).

[Graf et al., 2004] Graf, H. P., Cosatto, E., Bottou, L., Durdanovic, I., and Vapnik, V. (2004). Parallel Support Vector Machines: The Cascade SVM. In *Neural Information Processing Systems*. (Cited on page 48).

[Guyon and Elisseeff, 2003] Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182. (Cited on page 30).

[Guyon et al., 2002] Guyon, I., Weston, J., Barnhill, S., and Vapnik, V. (2002). Gene Selection for Cancer Classification using Support Vector Machines. *Machine Learning*, 46(1-3):389–422. (Cited on pages 3, 4, and 37).

[Haussler, 1999] Haussler, D. (1999). Convolution kernels on discrete structures. Technical report, Dept. of Computer Science, University of California at Santa Cruz. (Cited on pages 2 and 24).

[Joachims, 2000] Joachims, T. (2000). Estimating the Generalization Performance of a SVM Efficiently. In *Proceedings of ICML'00*. (Cited on page 88).

[Kashima and Koyanagi, 2002] Kashima, H. and Koyanagi, T. (2002). Kernels for Semi-Structured Data. In *Proceedings of ICML'02*. (Cited on page 25).

[Kazama and Torisawa, 2005] Kazama, J. and Torisawa, K. (2005). Speeding up training with tree kernels for node relation labeling. In *Proceedings of HLT-EMNLP'05*. (Cited on page 36).

[Kira and Rendell, 1992] Kira, K. and Rendell, L. A. (1992). The Feature Selection Problem: Traditional Methods and a New Algorithm. In *AAAI*, pages 129–134. (Cited on page 39).

[Klein and Manning, 2003] Klein, D. and Manning, C. D. (2003). Accurate unlexicalized parsing. In *Proceedings of ACL'03*, pages 423–430. (Cited on pages 89 and 90).

[Kudo and Matsumoto, 2003] Kudo, T. and Matsumoto, Y. (2003). Fast methods for kernel-based text analysis. In *Proceedings of ACL'03*. (Cited on pages 2, 4, and 40).

[Kudo et al., 2005] Kudo, T., Suzuki, J., and Isozaki, H. (2005). Boosting-based Parse Reranking with Subtree Features. In *Proceedings of ACL'05*. (Cited on page 2).

[Li and Roth, 2006] Li, X. and Roth, D. (2006). Learning question classifiers: the role of semantic information. *Natural Language Engineering*, 12(3):229–249. (Cited on page 88).

[Lodhi et al., 2002] Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., Watkins, C., and Scholkopf, B. (2002). Text classification using string kernels. *Journal of Machine Learning Research*, 2:563–569. (Cited on pages 27 and 35).

[Mercer, 1909] Mercer, J. (1909). Functions of Positive and Negative Type, and their Connection with the Theory of Integral Equations. *Royal Society of London Philosophical Transactions Series A*, 209:415–446. (Cited on page 22).

[Morik et al., 1999] Morik, K., Brockhausen, P., and Joachims, T. (1999). Combining statistical learning with a knowledge-based approach - a case study in intensive care monitoring. In *ICML '99: Proceedings of the Sixteenth International Conference on Machine Learning*, pages 268–277, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. (Cited on page 98).

[Moschitti, 2006a] Moschitti, A. (2006a). Efficient Convolution Kernels for Dependency and Constituent Syntactic Trees. In *Proceedings of ECML'06*, pages 318–329. (Cited on pages 27, 34, and 88).

[Moschitti, 2006b] Moschitti, A. (2006b). Making Tree Kernels Practical for Natural Language Learning. In *Proccedings of EACL'06*. (Cited on pages 3 and 25).

[Moschitti et al., 2008] Moschitti, A., Pighin, D., and Basili, R. (2008). Tree Kernels for Semantic Role Labeling. *Computational Linguistics*, 34(2):193–224. (Cited on pages 2, 35, and 91).

[Moschitti et al., 2007] Moschitti, A., Quarteroni, S., Basili, R., and Manandhar, S. (2007). Exploiting Syntactic and Shallow Semantic Kernels for Question/Answer Classification. In *Proceedings of ACL'07*. (Cited on page 35).

[Neumann et al., 2005] Neumann, J., Schnorr, C., and Steidl, G. (2005). Combined SVM-Based Feature Selection and Classification. *Machine Learning*, 61(1-3):129–150. (Cited on page 38).

[Nguyen et al., 2009] Nguyen, T.-V. T., Moschitti, A., and Riccardi, G. (2009). Convolution kernels on constituent, dependency and sequential structures for relation extraction. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 1378–1387, Singapore. Association for Computational Linguistics. (Cited on pages 37 and 90).

[Nocedal and Wright, 2000] Nocedal, J. and Wright, S. J. (2000). *Numerical Optimization*. Springer. (Cited on page 20).

[Palmer et al., 2005] Palmer, M., Gildea, D., and Kingsbury, P. (2005). The Proposition Bank: An Annotated Corpus of Semantic Roles. *Comput. Linguist.*, 31(1):71–106. (Cited on page 91).

[Pei et al., 2001] Pei, J., Han, J., Asl, M. B., Pinto, H., Chen, Q., Dayal, U., and Hsu, M. C. (2001). PrefixSpan Mining Sequential Patterns Efficiently by Prefix Projected Pattern Growth. In *Proceedings of ICDE'01*. (Cited on pages 6, 40, and 41).

[Pighin and Moschitti, 2009a] Pighin, D. and Moschitti, A. (2009a). Efficient Linearization of Tree Kernel Functions. In *Proceedings of CoNLL'09*. (Cited on pages 6 and 112).

[Pighin and Moschitti, 2009b] Pighin, D. and Moschitti, A. (2009b). Reverse Engineering of Tree Kernel Feature Spaces. In *Proceedings of EMNLP*, pages 111–120, Singapore. Association for Computational Linguistics. (Cited on pages 6 and 112).

[Pradhan et al., 2005] Pradhan, S., Hacioglu, K., Krugler, V., Ward, W., Martin, J. H., and Jurafsky, D. (2005). Support vector learning for semantic argument classification. *Mach. Learn.*, 60(1-3):11–39. (Cited on pages 36 and 112).

[Rakotomamonjy, 2003] Rakotomamonjy, A. (2003). Variable selection using SVM based criteria. *Journal of Machine Learning Research*, 3:1357–1370. (Cited on pages 4 and 38).

[Reichartz et al., 2009] Reichartz, F., Korte, H., and Paass, G. (2009). Dependency Tree Kernels for Relation Extraction from Natural Language Text. In Buntine, W. L., Grobelnik, M., Mladenic, D., and Shawe-Taylor, J., editors, *Proceedings of ECML/PKDD 2009*, volume 5782 of *Lecture Notes in Computer Science*, pages 270–285. Springer. (Cited on page 37).

[Rieck et al., 2010] Rieck, K., Krueger, T., Brefeld, U., and Müller, K.-R. (2010). Approximate tree kernels. *J. Mach. Learn. Res.*, 11:555–580. (Cited on page 41).

[Rosenblatt, 1958] Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*, 65(6):386 – 408. (Cited on page 15).

[Schölkopf and Smola, 2001] Schölkopf, B. and Smola, A. J. (2001). *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond (Adaptive Computation and Machine Learning)*. The MIT Press. (Cited on pages 14, 22, and 37).

[Shen et al., 2003] Shen, L., Sarkar, A., and Joshi, A. K. (2003). Using ltag based features in parse reranking. In *Proceedings of the 2003 conference on Empirical methods in natural language processing*, pages 89–96, Morristown, NJ, USA. Association for Computational Linguistics. (Cited on pages 2 and 36).

[Suzuki and Isozaki, 2005] Suzuki, J. and Isozaki, H. (2005). Sequence and Tree Kernels with Statistical Feature Mining. In *Proceedings of NIPS'05*. (Cited on pages 4 and 40).

[Toutanova et al., 2004] Toutanova, K., Markova, P., and Manning, C. (2004). The Leaf Path Projection View of Parse Trees: Exploring String Kernels for HPSG Parse Selection . In *Proceedings of EMNLP 2004*. (Cited on page 2).

[Vapnik, 1998] Vapnik, V. N. (1998). *Statistical Learning Theory*. Wiley-Interscience. (Cited on pages 13, 14, 37, and 50).

[Viswanathan and Smola, 2003] Viswanathan, S. V. N. and Smola, A. J. (2003). Fast Kernels for String and Tree Matching. In S. Becker, S. T. and Obermayer, K., editors, *Advances in Neural Information Processing Systems 15*. "MIT Press"address. (Cited on page 25).

[Voorhees, 2001] Voorhees, E. M. (2001). Overview of the TREC 2001 Question Answering Track. In *In Proceedings of the Tenth Text REtrieval Conference (TREC)*, pages 42–51. (Cited on page 88).

[Weston et al., 2003] Weston, J., Elisseeff, A., Schölkopf, B., and Tipping, M. (2003). Use of the zero norm with linear models and kernel methods. *J. Mach. Learn. Res.*, 3:1439–1461. (Cited on pages 4 and 39).

[Xue and Palmer, 2004] Xue, N. and Palmer, M. (2004). Calibrating Features for Semantic Role Labeling. In Lin, D. and Wu, D., editors, *Proceedings of EMNLP 2004*. (Cited on page 36).

[Zaki, 2002] Zaki, M. J. (2002). Efficiently mining frequent trees in a forest. In *Proceedings of KDD'02*. (Cited on page 6).

[Zelenko et al., 2003] Zelenko, D., Aone, C., Richardella, A., K, J., Hofmann, T., Poggio, T., and Shawe-taylor, J. (2003). Kernel Methods for Relation Extraction. *Journal of Machine Learning Research*, 3:2003. (Cited on page 36).

[Zhang and Lee, 2003] Zhang, D. and Lee, W. S. (2003). Question classification using support vector machines. In *Proceedings of SIGIR'03*, pages 26–32. (Cited on pages 35 and 88).

[Zhang et al., 2006] Zhang, M., Zhang, J., and Su, J. (2006). Exploring Syntactic Features for Relation Extraction using a Convolution Tree Kernel. In *Proceedings of NAACL*. (Cited on pages 3, 36, and 51).