Department of Modern Languages
2015

Morphological Disambiguation using Probabilistic Sequence Models

Miikka Pietari Silfverberg

Academic dissertation to be publicly discussed, by due permission of the Faculty of Arts at the University of Helsinki in auditorium FOO (in lecture room QUUX), on the XY^{th} of February at 12 o'clock.

University of Helsinki
Finland

Supervisor

Krister Lindén, Helsingin yliopisto, Finland, Anssi Yli-Jyrä, Helsingin yliopisto, Finland

Pre-examiners

XYZ, FOO, BAR
XYZ, FOO, BAR

Opponent

XYZ, FOO, BAR

Custos

XYZ, FOO, BAR

Contact information

Department of Modern Languages
P.O. Box 24 (Unioninkatu 40)
FI-00014 University of Helsinki
Finland

Email address: postmaster@helsinki.fi
URL: http://www.helsinki.fi/
Telephone: +358 9 1911, telefax: +358 9 191 51120

ISBN XXX-XXX-XX-XXXX-X (printed)
ISBN XXX-XXX-XX-XXXX-X (PDF)
Computing Reviews (1998) Classification: F.1.1, I.2.7, I.7.1
Helsinki 2015
FOO

Nobody is going to read your PhD thesis

Ancient Proverb

Contents

List of Figures

List of Tables

List of Algorithms

Notation

Acronyms

CRF	Conditional Random Field
HMM	Hidden Markov Model
MEMM	Maximum-Entropy Markov Model
NB	Naïve Bayes
PGM	Probabilistic Graphical Model
POS	Part-of-Speech
MAP	Maximum a posteriori

Mathematical Notation

$x[i]$	The element at index i (starting at 1) in vector or sequence x.
$\mathbb{R}_n^m$	The space of $m \times n$ real valued matrices.
$x[1:t]$	Prefix $(x_1,\ ...,\ x_t)$ of vector or sequence $x = (x_1,\ ...,\ x_T)$.
X^t	The cartesian product of t copies of the set X.
$M^\top$	Transpose of matrix M.
M^+	The More-Pennrose pseudoinverse of matrix M.
$f(x;\theta)$	Function f, paramerized by vector θ, applied to x.
$x \mapsto f(x),\ x \overset{f}{\mapsto} f(x)$	A mapping of values x to expressions $f(x)$.
$\lVert v \rVert$	Norm of vector v.
$\hat{p}$	Estimate of probability p.

Acknowledgements

Teemu Ruokolainen statistical methods, friendship, writing.

Måns Huldén finite-state, statistical methods, inspiration, courage, IPA.

Krister Lindén ideas, letting me do useless stuff once in a while, pushing me forward, encouragement, realism.

Anssi Yli-Jyrä inspiration, finite-state, positive attitude, friendship.

Kimmo Koskenniemi intellectual rigorosity, how to think of stuff, how to be legible, being critical.

Arto Voutilainen

Erik Axelson

Tommi Pirinen

Senka Drobac

Sam Hardwick

Graham Wilcock

Juliette Kennedy

Panu Kalliokoski

Coursera

Reetta Vuokko-Syrjänen and Kaj Syrjänen + Elsa

Family

Veera Nykänen

Jarmo Widemann

Antti, Sami, Ville, Jarmo, Rod, Marko and many more friends who have brought a lot of happiness through the time I’ve been working on this thesis project.

Chapter 1

Introduction

Chapter 2

Morphology

2.1 The Structure of Words

Many linguistc units are rather controversial. What is the definition of a sentence? Do sentences even exist in spoken language? Is there such a thing as a phoneme?

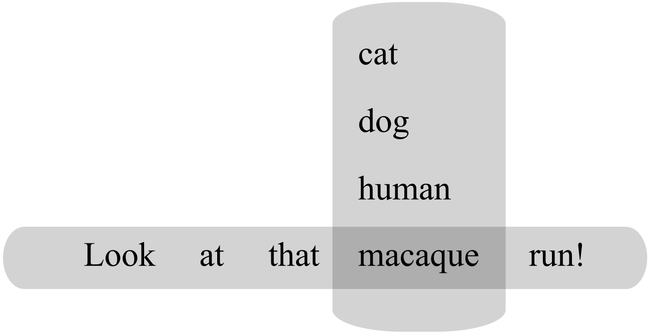
The concept of a word is, however, among the among the more universally accepted. The reason for this may be that there is in fact a fairly easy and readily applicable test for what is a word: X is a word if X can be used as an answer to a question. For example

> - You said you saw a what yesterday?
> - Dog. I saw a dog.

Although, singular nouns in English usually take a compulsory article, exceptional contexts where they are uttered without an article are acceptable for many language users. Therefore, it is plausible to stipulate that singular nouns are words in English. For most languages, this criterion can be used to find most words that language users intuitively group under the concept word. Even this criterion, however, fails for polysemous languages. Nevertheless, it very useful for many languages.

Of course, orthographic words are much easier to identify than spoken language words in many writing systems. Simply look for entities separated by punctuation and white-space. This, however, does not work for some languages such as Chinese where word spaces are not indicated in written text. Because this thesis exclusively deals with written language and because I have performed experiments on European languages

Figure 2.1: The syntagmatic and paradigmatic axes of language.



whose orthographies mark word boundaries, I will gloss over the difficulties of locating word boundaries although this is an interesting problem from the points of view of both linguistics and engineering.

Morphology is the sub-fields of linguistics that investigates the structure of words. For example the knowledge that appending a suffix “s” to a singular English nouns makes it plural is morphological knowledge. Of course this rule is not entirely correct because some nouns form plurals in some other way (e.g. “foot/feet”) and yet other nouns have no plural form (e.g. “music”).

Because words consist of phonemes or orthographic symbols, the structure of words cannot be investigated without considering the phonological system of a language. If we had to form the singular inessive of the the non-existent Finnish noun “looka”, we would say “lookassa”, not “lookassä” because vowel harmony, a phonological co-occurrence restriction, prohibits the latter form.

2.2 Languages with Rich Morphology

- Typological classification of languages.
- “Large label sets”.

2.3 Morphological Analyzers

- Finite-state morphology (Koskenniemi, 1984), (Kaplan and Kay, 1994).

Chapter 3

Machine Learning

This section outlines the basic methodology followed in machine learning approaches to Natural Language Processing. I will briefly discuss machine learning from a general point of view and then present supervised machine learning in more detail using linear regression as example. I will then elaborate on the different kinds of classifiers applied in NLP; both unstructured and structured.

3.0.1 Overview

Statistical and Rule-Based NLP Approaches to Natural Language Processing can be broadly divided into two groups: rule-based and machine learning approaches. Rule-based approaches utilize hand crafted rules for tasks such as text labeling or machine translation. Machine learning approaches solve the same set of problems using data driven (usually statistically motivated) models and samples from the problem domain, which are used to estimate model parameters.

This thesis focuses on extending machine learning techniques to the domain of morphologically complex languages, where they have been applied less frequent than for morphologically more straightforward languages like English. I want to emphasize that the thesis should not be seen as an attack against the rule-based paradigm for Natural Language Processing which has proven to be very successful in treatment of morphologically complex languages . This thesis is simply an investigation into extension of machine learning techniques. Truly successful language processing systems should probably combine rule-based and statistical techniques.

Machine learning and hand crafted rules have their respective merits and shortcomings and these are dependent on the task to some extent. For example, in the domain of machine translation, rule-based methods can give syntactically correct results . However, statistical machine translation systems are in general better at lexical choice .

Supervised and Unsupervised ML Machine learning is can be divided into two fields supervised and unsupervised. Supervised machine learning uses a training material of labeled training examples. An example would be a set of images with associated key words, where the task is to build a system which can associate previously unseen images with appropriate key words.

Supervised machine learning is typically employed for tasks such as labeling (that is classification) and translation. Typical examples of tasks include Part-of-Speech labeling and speech recognition where annotations consist of POS labels for the words in some text and written sentences corresponding to an acoustic speech signal respectively.

In contrast to supervised machine translation, unsupervised approaches exclusively utilize unannotated data. Unsupervised machine learning is most often used for various kinds of clustering tasks, that is grouping sets of examples into subsets of similar examples.

Finally, semi-supervised (and weakly supervised) systems use an annotated training set with in combination with a, typically, very large unannotated training set to improve the results beyond the maximum achievable by either approach in isolation.

In Natural Language processing, supervised learning approaches are most often used (for example ...). Unsupervised approaches, however, can also be useful for e.g. exploratory data analysis (...).

This thesis focuses on supervised machine learning.

3.1 Supervised Learning

In this section, I will illustrate the key concepts and techniques in supervised Machine Learning using the simple example of linear regression. I will explain the plain linear regression model and show how it can be fitted using training data. I will then briefly present ridge regression which is a regularized version of linear regression.

I chose linear regression as example because it is a simple model yet can be used to illustrate data driven techniques. Moreover, the model has several tractable properties such as smoothness and convexity. Additionally, it can be seen as the simplest example of a linear classifier which is a category of models encompassing

3.1.1 Basic Concepts

Linear Regression As a simple example, imagine a person called Jill who is a real estate agent. She is interested in constructing an application, for use by prospective clients, which would give rough estimates for the selling price of a property. Jill knows that a large number of factors affect housing prices but there are a few very robust quantifiable predictors of price that are easy to measure.

Jill decides to base the model on the following predictors:

1. The living area.
2. The number of rooms.
3. The number of bathrooms.
4. Size of the yard.
5. Distance of the house from the city center.
6. Age of the house.
7. Amount of time since the last major renovation.

Jill decides to use the simplest model which seems reasonable. This model is *linear regression* which models *the dependent variable* house price as a linear combination of the independent variables listed above and parameter values in $\mathbb{R}$. The linear regression model is probably not accurate. It fails in several regards. For example, increasing age of the house probably reduces the price up to a point but very old houses can in fact be more expensive then newly built houses especially if they have been renovated lately. Although, the linear model is unlikely to be entirely accurate, Jill is happy with it because the intention is just to give a ball park estimate of the price for the average prospective client.

To formalize the linear regression model, let us call the dependent variable price y and each of the independent variables living area, number of rooms and so on x_i. Given a vector $x = (x[1], ..., x[n])^\top \in \mathbb{R}^n$, which combines the independent variables[1], and a parameter vector $\theta \in \mathbb{R}^n$ the linear regression model is given in Equation 3.1.

$$y(x; \theta) = x^\top \theta \tag{3.1}$$

Two questions immediately arise: How to compute the price given parameters and predictors and how to compute the parameter vector θ. These questions are common for all supervised learning problems also when using other models than the linear regression model.

Inference The first question concerns *inference*, that is finding the most likely value for the dependent variable given the independent variables. In the case of linear regression, the answer to this question is straightforward. To compute the price, simply perform the inner product in Equation 3.1. The question is, however, not entirely settled because one might also ask for example how close to the actual price the estimate y is likely to be. A related question would be to provide an upper and lower bound for the price so that the actual price is very likely to be inside the provided bounds. To answer these questions, one would have to model the expected error.

Inference is very easy and also efficient in the case of linear regression. With more complex model such as structured graphical models used in this thesis, it can however be an algorithmically and computationally more challenging problem. The principle is still the same: Find the y which is most likely given the input parameters.

Estimation The second question concerns *estimation of model parameters* and it is more complex than the question of inference. First of all, Jill needs training data. She also needs to decide upon an *estimator*, that is, a method for estimating the parameters θ.

In the case of house price prediction, Jill can simply use data about houses she has brokered in the past. She decides to use a training data set $\mathcal{D} = \{(x_1, y_1), ..., (x_T, y_T)\}$, where each $x_t = (x_t[1]...x_t[n])$ is a vector of independent variable values (living area,

[1] In reality, each of the predictors would probably be transformed to give all of them the same average and variance. Although this ii not required in theory, it tends to give a better model **?**.

age of the house and so on) and y_t is the dependent variable value, that is the final selling price of the house. Now Jill needs to make a choice. How many training examples (x_t, y_t) does she need? The common wisdom is that more data is always better, however, this has bearing on how the parameters need to be estimated. In any case, the number of data points in the training data should ideally be higher than the number of parameters that need to be estimated. When one cannot accomplish this, one encounters the so called *data sparsity problem*.

Data Sparsity Whereas getting sufficient training data is fairly easy in the case of housing prices, it is vastly more difficult to accomplish with more complicated models in natural language processing. Therefore, one central question in this thesis is how to counteract *data sparsity*.

Cost Functions The objective in estimation is to find a parameter vector θ which in some sense minimizes the error of the house price predictions $y(x_t; \theta)$ when compared to the actual realized house prices y_t in the training data. The usual minimization criterion used with linear regression is the least square sum criterion given in Equation 3.2. It is minimized by a parameter vector θ which gives as small square errors $|y_t - y(x_t; \theta)|^2$ as possible.

$$\theta = \arg\min_{\theta' \in \mathbb{R}^n} \sum_{x_t \in \mathcal{D}} |y_t - y(x_t; \theta')|^2 \tag{3.2}$$

The square sum is an example of a *cost function* (also called the objective function). A cost function assigns a non-negative real cost for each parameter vector. Using the concept of cost function, the objective of estimation can be reformulated: Find the parameter vector θ that minimizes the cost of the training data.

The Exact Solution In the case of linear regression, there is a well known exact solution for θ which utilizes linear algebra. The solution is given in Equation 3.3. The matrix $X \in \mathbb{R}_n^T$ is defined by $X_{t,i} = x_t[i]$ and its More-Pennrose pseudoinverse $X^+ \in \mathbb{R}_T^n$ is defined as $X^+ = (X^\top X)^{-1} X^\top$. The vector $Y \in R^T$ is simply the vector of realized house prices y_t. The solution θ exists only when none of the independent variables are linear combinations of each other in the training data.

$$\theta = X^{+}Y \tag{3.3}$$

Iterative Estimation Although the linear regression model is simple enough so that it can be estimated exactly, the same does not hold for most more complex models such as the Conditional Random Field investigated in this thesis. Moreover, the exact solution might often not be the one that is desired. Often the training data is quite sparse, that is there is too little of it compared to the amount of parameters that need to be estimated. Therefore, the model may *over-fit* the training data and fail to generalize well to examples not included in the training data.

Regularization Due to the problem of over-fitting, a family of heuristic techniques called *regularization* is often employed. They aim to transform the original problem in a way which will penalize both deviance from the gold standard and "complexity" of the solution θ. Regularization can be seen to convey the same idea as Occam's Maxim which states that a simpler explanation for a phenomenon should be preferred when compared to a more complex explanation yielding equivalent results. Of course, this does not explain what is meant by a "complex" parameter vector θ.

To illustrate simple and complex parameter vectors, examine a case of linear regression where the dependent variable y and the predictors x_i have mean 0 and variance 1 in the training data. This may seem restrictive but in fact any linear regression problem can easily be transformed into this form by applying an affine transformation $z \mapsto az - b$. When doing inference, this affine transformation can simply be reversed by applying $z \mapsto a^{-1}(z + b)$. The simplest parameter vector θ is clearly the zero vector $\theta = (0...0)^{\top}$. It corresponds to the hypothesis that the predictors x_i have no effect on the dependent variable y. According to this hypothesis, the prediction is always the average of the dependent variable values in the training data.

The zero solution to a linear regression problem is simple but also totally biased. Because we are assuming that the independent variables x_i explain the dependent variable y, a model that completely disregards them is unlikely to give a good fit to the training data. By introducing a regularization term into the cost function, we can however encourage simple solutions while at the same time also preferring solutions that give a good fit. There are several ways to accomplish this but the most commonly used are so called L_1 and L_2 regularization. These are general regularization methods that are employed in many models in machine learning.

The L_1 regularized cost function for linear regression is given in Equation 3.4. L_1 regularization, also called LASSO regularization **?**, enforces solutions where many of the parameter values are 0. These are also called sparse parameters. It is suitable in the situation where the model is over specified, that is, many of the predictors might not be necessary for good prediction. The expression is a sum

$$\theta = \arg\min_{\theta' \in \mathbb{R}^n} \sum_{x_t \in \mathcal{D}} |y_t - y(x_t; \theta')|^2 + \lambda \sum_i |\theta[i]| \tag{3.4}$$

The L_2 regularized cost function is given in 3.5. L_2 regularization is also called Tikhonov regularization. In contrast to L_1 regularization, it directly prefers solutions with small norm. A linear regression model with Tikhonov regularization is called a ridge regression model.

$$\theta = \arg\min_{\theta' \in \mathbb{R}^n} \sum_{x_t \in \mathcal{D}} |y_t - y(x_t; \theta')|^2 + \lambda \|\theta\|^2 = \arg\min_{\theta' \in \mathbb{R}^n} \sum_{x_t \in \mathcal{D}} |y_t - y(x_t; \theta')|^2 + \lambda \sum_i |\theta[i]|^2 \tag{3.5}$$

The coefficient $\lambda \in \mathbb{R}^+$ is called the *regularizer*. The regularizer determines the degree to which model fit and simplicity affect the cost. A higher λ will increase the cost for complex models more than a lower one. When λ increases, the optimal parameter vector θ approaches the zero vector and when it decreases θ approaches the parameters that fit the training data as closely as possible. This is called under-fitting.

The regularizer is a so called *hyper-parameter* of the regularized liner regression model. It is easy to see that increasing λ will automatically increase the cost. Therefore, there is no direct way to estimate its correct magnitude simply using the training data. Instead *held-out data* can be used. Held-out data is labeled data that is not used directly for estimating model parameters. If the model over-fits the training data, that is generalizes poorly to unseen examples, the held-out data will have a high cost. However, it will also have a high cost if the model under-fits, that is, performs poorly on all data. Held-out data can therefore be used to find an optimal values for the regularizer λ. Often one tries several potential values and chooses the one that minimizes the cost of the held-out data. Usually, one uses the unregularized cost function for the held-out data.

Iterative Estimation Regularization is an additional reason for introducing iterative estimation methods instead of exact estimation. There are several choices of regularization methods and some of them might not result in optimization problems that have closed form solutions[2]. When an exact solution cannot be computed, or it is undesirable, numerical methods can be used to estimate model parameters.

Because the cost of the training data is a function of the model parameters, one can apply analytical methods to try to find optimal parameter values. These methods include for example Newton's method which is an iterative procedure that can be used to find the zeros of a differentiable function or local extrema of a twice differentiable function. Approximations of Newton's method, so called Quasi-Newton methods **?**, have also been developed because Newton's method requires evaluation and inversion of the Hessian matrix of a function. This is a very costly operation for functions where the domain has high dimension. Quasi-Newton methods use approximations of the inverse Hessian.

A simpler method called gradient descent can be applied to functions that are only once differentiable.[3] In general, gradient descent converges toward the optimum more slowly than Newton's method, however, the computation of one step of the iterative process is much faster when using gradient descent. Therefore, it may be faster in practice.

All gradient based methods rely on differentiability of the cost function. For the models used in this thesis, differentiability holds. Gradient based methods work in the following general manner. Let $L_{\mathcal{D}} : \mathbb{R}^n \to \mathbb{R}$ be the cost of the training data $\mathcal{D}$.

1. Start at a random point θ_0 in the parameter space.

2. Determine the direction of steepest descent of the cost function. This is the negative gradient $-\nabla L_{\mathcal{D}}(\theta_t)$ at point θ_t.

3. Determine a suitable step size $\alpha_t \in \mathbb{R}_+$.

4. Take a step of length α_t in direction v_t to get to the next point in the parameter space θ_{t+1}, that is $\theta_{t+1} = \theta_t - \alpha_t \nabla L_{\mathcal{D}}(\theta_t)$.

[2] In the case of regression, non-linear *kernel functions* also result in optimization problems that don't have a closed form solution. I will elaborate this when discussing logistic regression.

[3] Essentially the same procedure can also be applied to functions that are not differentiable but are only guaranteed to have a sub-gradient **?**.

5. If the difference in cost $|L_{\mathcal{D}}(\theta_{t+1}) - L_{\mathcal{D}}(\theta_t)|$ is smaller than a threshold ρ, set $\theta = \theta_{t+1}$. Otherwise, set $\theta_t = \theta_{t+1}$ and return to line 2.

The main difference between first and second order methods is the computation of the step size α_t. Second order methods can take longer steps when the cost is plateauing. Thus they typically take fewer steps in total. In first order methods such as gradient descent, α_t can be constant, a decreasing function of t or can also be determined by a line search in the direction of $-\nabla L_{\mathcal{D}}(\theta_t)$ **?**. For example $\alpha_t = t^{-1}$ may work[4].

As the meta-algorithm above suggests, gradient based optimization algorithms are local in the sense that they always move in the direction of steepest descent of the cost function that is toward a local optimum. Therefore, they will in general not find the global optimum of the cost function. By choosing a *convex* cost function it is possible to avoid getting stuck at local optima. All local optima of convex functions are in fact global optima.

Convexity is, however, not enough to guarantee convergence to a global optimum. First of all, a global optimum might not exist[5]. Convergence could also be too slow thus leading to premature termination of the training procedure. This is specifically a problem for first order methods.

Online Estimation The optimization methods discussed up to this point have been so called *batch methods*. The derivatives of the cost function is computed over the entire training data and parameters are updated accordingly. Batch methods can be slow and subsequent training when new training examples become available is computationally intensive. *Online algorithms* are an alternative to batch methods, where the cost is instead computed for a randomly chosen training example and the parameters are the updated accordingly **?**. In practice, online methods can give fast convergence. Moreover, re-training is relatively efficient when new training examples become available.

Stochastic gradient descent is a well known online estimation algorithm. It has vastly superior convergence properties when compared to regular gradient descent **?** but is identical in all other respects except that it is an online estimation algorithm instead of a

[4] In general, stepsize $(\alpha_t)_{t\in\mathbb{N}}$ that resemble the harmonic sequence, that is $\sum \alpha_t^2 < \infty$ and $\sum \alpha_t = \infty$, guarantee convergence of gradient descent to an minimum of the cost function (if it exists) for a wide variety of functions **?**

[5] This can happen if the domain of the cost function is not compact. Unfortunately, it usually is not.

batch algorithm. The algorithm processes one random training example at a time. It uses the gradient $\nabla L_{\mathcal{D}[i]}(\theta)$ of the cost over the single training example $\mathcal{D}[i]$ to approximate the gradient $\nabla L_{\mathcal{D}}(\theta)$ for the entire training data $\mathcal{D}$.

3.2 Classifiers

The main topic of this thesis is morphological tagging. It can be seen as a structured *classification task* where each word in a sentence is assigned one morphological label[6]. Classification is a fundamental supervised machine learning task where the objective is to learn a mapping from objects like words, sentences, documents or images onto discrete classes such as morphological labels (Noun+Sg+Nom) or sentiments (Positive sentiment versus Negative Sentiment).

Classification resembles regression. The output of a classifier is, however, not a single real value y but a distribution over the set of nominal classes.

Probabilistic classifiers can broadly be divided into two groups, namely generative and discriminative. Generative classifiers learn a joint distribution $p(x, y)$ over labels y and input examples x. Given an example x, a distribution over classes y can be computed using the marginal probability of example x, which is $p(x) = \sum_{y' \in Y} p(y', x)$. The probability distribution over classes is defined by $p(y|x) = p(y, x)/p(x)$.

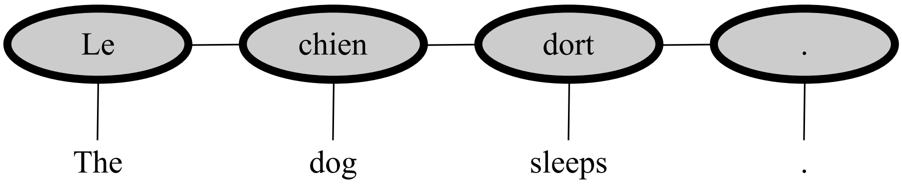
3.3 Structured Classifiers

There are tasks in natural language processing that cannot be adequately formulated as supervised classification tasks in the simple way that has been discussed earlier. Examples of tasks that require more sophisticated methods are syntactic parsing and translation between languages. One could think of parsing as a labeling task where the objective is to label a sentence with the appropriate syntactic tree. Likewise, the translation of a sentence could be seen as its label.

It is, however, easy to see that these approaches are deeply flawed. Firstly, there is an infinite number of syntactic trees and indeed an infinite number of sentences of finite length. Therefore, the label set would have to be infinite as well. Secondly, the data

[6] which may have internal structure as is seen in Section 8.8.

Figure 3.1: A translation from English to French



sparsity problem would be unmanageable. Just in order to see most relatively frequent syntax trees for moderately long English sentences, say twenty words long, we would need an unfathomable amount of training data.

Instead of a simplistic classification approach, *structured* classifiers can be employed. Structured classifiers relate parts of the complex label such as a translation, syntax tree or sequential labeling with the input. They also learn how to piece together complex labels from simple constituents. A very simple (and probably very bad) translation software could learn how to translate isolated English words into French words and then learn how to combine French words into sentence as shown in Figure 3.1. Note that although both "Le" and "La" are perfectly good translations for "The", only "Le" is usually possible before "chien", which is a masculine French word.

In general, structured classifiers rely on two models, the unstructured and structured model. The unstructured model learns to relate simple labels such as words in a translation or nodes in a syntax tree with the input. In contrast, the structured model learns how to combine simple labels into complex labels such as entire translations or syntax trees. In practice, some models make a stricter division then other models.

Structured models are the main subject matter of this thesis and I will present two structured models, the Hidden Markov model and the Conditional random field in detail in Chapters **??** and **??**.

3.4 The Aim of Research – Improving the State of the Art

Like most language technological research, the work documented in this thesis aims at improving existing solutions for language processing. Specifically, my work is targeted at improving morphological taggers for morphologically rich languages. It is not entirely easy to define what constitutes an improvement to the field of morphological tagging or indeed any sub-field of language technology. Nevertheless, most researchers would probably agree that the following kinds of changes are improvements compared to existing approaches:

1. Improving labeling accuracy.
2. Speeding up estimation.
3. Speeding up tagging.
4. Reducing model size.
5. Clarifying the underlying theoretical foundations.
6. Simplifying implementation of taggers.
7. Uncovering best practices for building taggers.

Items 1 though 4 in the list above are *quantifiable* improvements. It is possible to perform experiments to measure the labeling accuracy, tagging speed, estimation speed and model size given by different morphological taggers and derive conclusions about the respective merits of the taggers. In contrast, the rest of the items in the list cannot be measured as easily.

Although, most probably would agree that clarifying theoretical foundations of a field is a substantial contribution, it may not be as easy to agree upon what constitutes a clarification. This could for example be dependent on the background of individual researchers. Similarly, one model might be more straight-forward to implement than another model using some programming language, however, this can very well depend on the specific programming language and available libraries to some degree.

Because quantifiable improvements are easier to ascertain, this thesis focuses on demonstrating such improvements compared to other state of the art approaches. Nevertheless, I will also aim to show that the model demonstrated in the thesis is conceptually simpler than other state of the art approaches.

Although quantifiable improvements are easier to demonstrate than other improvements, there are still caveats. First of all, it is impossible to compare machine learning models directly. One can only compare implementations of the models. Because of differences between platforms even different implementations of the exact same model can have radically different run time on the same data. Moreover, bugs in the implementation of different models can reduce the accuracy or have sporadic effects on run time.

Besides practical concerns like dependence on implementation, there are also theoretical issues that interfere with measuring performance of different models. It is easy to say that the accuracy of one system is better than another system on *a specific data set*. This does, however, not imply that the accuracy is better on *all data sets*. In formal terms, we can only conduct experiments on samples of the distribution of all texts in a given language. Therefore, our experiments will yield results only in a probabilistic sense: it is likely that the labeling accuracy of one system is better than the accuracy of another, if the accuracy was better on the sample used for testing.

Using large test sets and test sets compared from a variety of different genres will probably give more reliable results. This is, however, also to some extent a debatable matter. Some methods are very accurate on the same genre that they were trained on but perform worse on other genres. Other methods instead perform well on average but, as a trade-off, cannot reach as high accuracy on a specific text domain. It is not easy to say which kind of system is preferable. This trade-off is called the bias variance trade-off **?** and it has bearing on measuring the performance difference of systems.

When using a high variance system, accuracy on different texts varies a lot. When instead using a system with high bias, the accuracy tends to vary much less.

When comparing two systems, it is not sufficient to simply look at the performance of the systems on a test set. If we measure the performance of the systems on a particular test, one of them will almost certainly perform better than the other regardless of whether there is an actual difference in the performances of the systems. The probability of exactly equal performance is simply very small.

The larger the test sets are, the more accurately the results of experiments will on average reflect the true performance of the systems. This is easy to see, because ultimately the test set will represent the entire domain. Unfortunately, the amount of test material is usually restricted and producing more test data might not be feasible[7].

An approach that is often used is splitting the test data into segments and performing several experiments – one for each segment. If one system outperforms the other system on most segments with a large margin, then we are more confident in saying that there is in fact a difference in the performance of the systems. If on the hand each system outperforms the other on roughly half of the segments, we might be inclined to doubt whether there is any real difference in performance between the systems. The higher the variance between the results, the larger the margins between the systems need to be in order for us to be able to conclude that there is a difference in performance. This argument can be made rigorous using statistical tests which measure the significance of the difference in performance of two systems.

The usual set up of statistical significance testing is to make a null hypothesis H_0 that the average performance of two systems is the same. After this a test statistic is computed. The test statistic is simply a real number whose value indicates the significance of the difference in performances between the systems. If the statistic is high enough, the null hypothesis can be discarded in favor of the alternative hypothesis that there is a genuine difference in performance between the systems. The test statistic incorporates information about the difference in performance of the systems on test data all segments as well as the variance of the performances.

In the work conducted presented in this thesis, I have used the Wilcoxon signed-rank test **?** to ascertain the significance of results. I use it instead of the T-test because it is not dependent on the fact that the distribution of differences in performance are normally distributed[8]. In practice it is less sensitive than the T-test.

[7] Especially when using standard data sets, one is restricted to the given amount of test data

[8] In practice it might be a fairly accurate assumption that the differences are normally distributed. This often holds for measurements **?**.

Chapter 4

Part-of-Speech Tagging

4.1 Background

4.2 Statistical Part-of-Speech Taggers

Morphological disambiguation and part-of-speech tagging are interesting tasks from the perspective of machine learning because they represent labeling tasks where both the set of inputs and outputs are unfathomably large. Since each word in a sentence $x = (x_1,\ ...,\ x_T)$ of length T receives one label, the complete sentence has n^T possible labels $y = (y_1,\ ...,\ y_T)$ when the POS label set has size n.

The exact number of potential English sentences of any given length, say ten, is difficult to estimate because all strings of words are not valid sentences[1]. However, it is safe to say that it is very large – indeed much larger than the combined number of sentences in POS annotated English language corpora humankind is likely to ever produce. Direct ML-estimation of the conditional distributions $p(y \mid x)$, for POS label sequences y and sentences x, by counting is therefore impossible.

Because the POS labels of words in a sentence depend on each other, predicting the label y_t for each position t separately is not an optimal solution. Consider the sentence "The police dog me constantly although I haven't done anything wrong!".

The labels of the adjacent words "police", "dog", "me" and "constantly" help to disambiguate each other. A priori, we think that "dog" is a noun since the verb "dog" is

[1] Moreover, it is not easy to say how many word types the English language includes.

quite rare. This hypothesis is supported by the preceding word “police” because “police dog” is an established noun–noun collocation. However, the next word “me” can only be a pronoun, which brings this interpretation into question. The fourth word “constantly” is an adverb, which provides additional evidence against a noun interpretation of “dog”. In total, the evidence points toward a verb interpretation for “dog”.

The disambiguation of the POS label for “dog” utilizes both so called *unstructured* and *structured* information. The information that “police dog” is a frequent nominal compound is unstructured information, because it refers only to the POS label (the prediction) of the word “dog”. The information that verbs are much more likely to be followed by pronouns than nouns is a piece of structured information because it refers to the combination of several POS labels. Structured information refers to the combination of predictions. Both kinds of information are very useful, but a model which predicts the label y_t for each position in isolation cannot utilize structured information.

Even though structured information is useful, structure is probably mostly useful in a limited way. For example the labels of “dog” and “anything” in the example are not especially helpful for disambiguating each other. It is probably a sensible assumption that the further apart two words are situated in the sentence, the less likely it is that they can significantly aid in disambiguating each other. However, this does not mean that the interpretations of words that are far apart cannot depend on each other – in fact they frequently do. For example embedded clauses introduce long range dependencies inside sentences.

It is said that machine learning is sophisticated counting of co-occurrences. This statement applies extremely well to POS tagging. Counting is an adequate approach to capturing correlations between the labels of words inside a small window (in the league of five words), because most adjacent words indeed do depend on each other is some way. However, sophisticated counting fails for larger windows, because the number of meaningful dependencies in large windows is negligible in comparison to the space of possibilities.

4.3 Morphological Disambiguation

Chapter 5

Hidden Markov Models

AN INTRO

5.1 Example

I will illustrate Hidden Markov Models using an example. Imagine a person called Jill who is hospitalized and occupies a windowless room. The only way for her to know what is happening in the outside world is to observe a nurse who passes her room daily[1].

Suppose, Jill is interested in weather phenomena and she decides to pass time by guessing if it is raining outside. She bases her guesses on whether or not the nurse is carrying an umbrella. In other words, she predicts an unobserved variable, the weather, based on an observed variable, the nurse's umbrella.

There are several probabilistic models Jill might use. The simplest useful model assigns probability 1 to the event of rain, if the nurse carries an umbrella, and assign it the probability 0 otherwise. This simplistic model would certainly give the correct prediction most of the time, but Jill believes that she can do better.

Jill knows that people often carry an umbrella when it is raining. She also knows that they rarely carry one when the weather is clear. However, people sometimes do forget their umbrella on rainy days, perhaps because they are in a hurry. Moreover, people sometimes carry an umbrella even when it is not raining. For example the weather might be murky and they might anticipate rain. Therefore, Jill decides to reserve some

[1] To make things simple, imagine the nurse never gets a day off.

probability, say 0.2, for the event that the nurse is carrying an umbrella when the weather is clear. She reserves an equal probability for the event that the nurse arrives at work without an umbrella although it is in fact raining.

Without additional information, this more complicated model will give exactly the same MAP predictions as the simplistic one. Knowledge of meteorology, however, also factors in. Let us suppose Jill is a weather enthusiast and she knows that the probability of rain is 0.25 a priori, making the probability of clear weather 0.75. She also knows that the probability of rain increases markedly on days following rainy days at which time it is 0.7. Similarly, the probability of clear weather increases to 0.9 if the weather was clear on the previous day. Figure 5.1 summarizes these probabilities.[2]

ι	
CLEAR	0.75
RAIN	0.25

T	CLEAR	RAIN
CLEAR	0.9	0.1
RAIN	0.3	0.7

E	☂̸	☂
CLEAR	0.8	0.2
RAIN	0.2	0.8

Figure 5.1: Foo FIXME

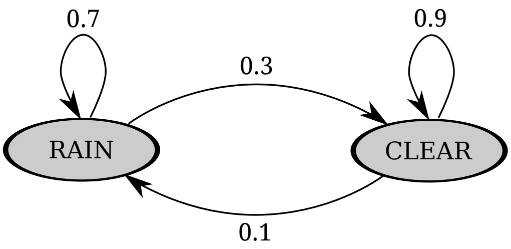
Let us assume that Jill observes the nurse for one week. She sees the nurse carry an umbrella on all days except Tuesday. The MAP prediction given by the simplistic model is that Tuesday is clear and all other days are rainy. The more complex model will, however, give a different MAP prediction: the probability is maximized by assuming that all days are rainy. Under the more complex model, it is simply more likely that the nurse forgot to bring an umbrella on Tuesday.

The model Jill is using for weather prediction is called a Hidden Markov Model. It can be used to make predictions about a series of events based on indirect observations.

The HMM is commonly visualized as a directed graph. Each hidden state, for example Rain and Clear, represents a vertex in the graph. Transitions from one hidden state to another are represented by arrows labeled with probabilities. Figure 5.2 shows a graph representing the transition structure of the HMM outlined in Figure 5.1.

[2] Since the author of this thesis has very little knowledge about meteorology, these probabilities are likely to be nonsense. The overall probability of rain and clear weather is, however, chosen to be the steady state of the Markov chain determined by the probabilities of transitioning between states. Consistency is therefore maintained.

Figure 5.2: Foo



5.2 Formal Definition

Abstracting from the example above, an HMM is a probabilistic model that generates sequences of state-observation pairs. At each step t in the generation process, the model generates an observation by sampling the *emission distribution* ε_{y_t} of the current state y_t. It will then generate a successor state y_{t+1} by sampling the *transition distribution* τ_{y_t} of state y_t. The first hidden state y_1 is sampled from the *initial distribution* ι of the HMM.

Since the succession of days is infinite for all practical purposes, there was no need to consider termination in the example presented in Figure 5.2. Nevertheless, many processes, such as sentences, do have finite duration. Therefore, a special *final state* f is required. When the process arrives at the final state, it stops: no observations or successor states are generated.

Following Rabiner (1989), I formally define a *discrete* HMM as a structure (Y, X, i, T, E, F) where:

1. Y is the set of hidden states ($Y = \{\mathrm{CLEAR}, \mathrm{RAIN}\}$ in the example in Figure 5.1).

2. X is the set of emissions, also called observations ($X = \{\cancel{☂}, ☂\}$ in the example in Figure 5.1).

3. $\iota : Y \rightarrow \mathbb{R}$ is the initial state distribution, that is the probability distribution determining the initial state of an HMM process (array ι in Figure 5.1).

4. T is the collection of transition distributions, $\tau_y : Y \rightarrow \mathbb{R}$, that determine the probability of transitioning from a state y to each state $y' \in Y$ (array T in Figure

5.1).

5. E is the collection of emission distributions $\varepsilon_y : X \to \mathbb{R}$, which determine the probability of observing each emission $o \in X$ in state $y \in Y$ (array E in Figure 5.1).

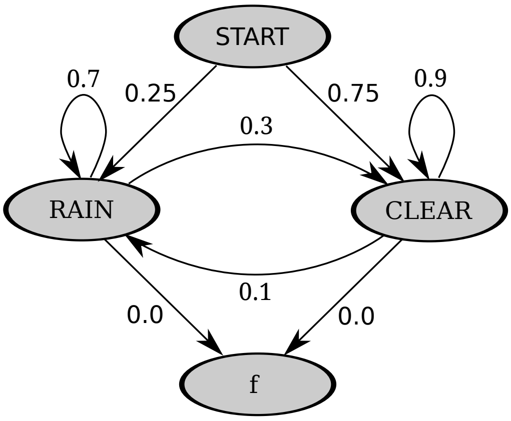
6. $f \in Y$ is the final state. The state f emits no observations and there are no transitions from f.

The definition of HMMs in this thesis differs slightly from Rabiner (1989) since I utilize final states. Final states were used in for example **?**. Absorbing states **?** could also be used. FIXME.

Figure 5.3 is a more accurate description of the HMM in Figure 5.1 than Figure 5.2. The image has been augmented with initial distribution and a final state.

Because the progression of days is infinite for all practical purposes, the probability of transitioning to the final state f in example 5.2 is 0 regardless of the current state. Hence, the probability of any single sequence of states and emissions is 0. Still, the probability of an initial segment of a state sequence may still be non-zero[3].

Figure 5.3: Foo



An HMM models a number of useful quantities:

[3] The probability of an initial segment up to position t can be computed using the forward algorithm, which is presented in Section 5.3.1.

1. The *joint probability* $p(x, y; \theta)$ of a observation sequence x and state sequence y. This is the probability that an HMM with parameters θ will generate the state sequence y and generate the observation x_t in every state y_t.

2. The *marginal probability* $p(x; \theta)$ of an observation sequence x. This is the overall probability that the observation sequence generated by an HMM is x.

3. The *conditional probability* $p(y \mid x; \theta)$ of a state sequence y given an observation sequence x. That is, how likely it is that the model passes through the states in y when emitting the observations in x in order.

4. The *marginal probability* $p(z, t,\ x; \theta)$ *of state* z *at position* t when emitting the observation sequence x. That is, the probability of emitting observation sequence x under the single constraint that the state at position t has to be z.

To formally define these probabilities, let $\theta = \{\iota,\ T,\ E\}$ be the parameters of some HMM with observation set X and hidden state set Y, $x = (x_1,\ ...,\ x_T) \in X^T$ be a sequence of observations and $y = (y_1,\ ...,\ y_T, y_{t+1} = f) \in Y^{T+1}$ a sequence of hidden states. Note, that the last state in y has to be the final state f. Then the joint probability $p(x, y; \theta)$ of x and y given θ is defined by Equation (5.1).

$$p(x, y; \theta) = p(y; \theta) \cdot p(x \mid y; \theta) = \left(\iota(y_1) \cdot \prod_{t=1}^{T} \tau_{y_t}(y_{t+1}) \right) \cdot \prod_{t=1}^{T} \varepsilon_{y_t}(x_t) \tag{5.1}$$

Equation (5.1) is a product of two factors: the probability of the hidden state sequence y, determined by the initial and transition probabilities, and the probability of the emissions x_t given hidden states y_t determined by the emission probabilities.

FIXME: Talk about language models and stuff.

There is no limit on the number of hidden states in Y that can emit a given observation. Therefore, it is quite possible that several state sequence $y \in Y^{T+1}$ can be generate the same sequence of observations $x \in X^T$. The marginal probability $p(x; \theta)$ of an observation sequence x can be found by summing over all state sequences that could have generated x. It is defined by Equation (5.2).

$$p(x; \theta) = \sum_{y \in Y^{T+1},\ y_{T+1} = f} p(x, y; \theta) \tag{5.2}$$

Possibly the most important probability associated to the HMM is the conditional probability $p(y \,|\, x;\theta)$ of state sequence y given observations x. This is an important quantity because maximizing $p(y \,|\, x;\theta)$ with regard to y will give the MAP assignment of observation sequence x. It is defined by Equation (5.3).

$$p(y \,|\, x;\theta) = \frac{p(x,y;\theta)}{p(x;\theta)} \tag{5.3}$$

It is noteworthy, that $p(y \,|\, x;\theta) \propto p(x,y;\theta)$ because the marginal probability $p(x;\theta)$ is independent of y. Therefore, y maximizes $p(y \,|\, x;\theta)$ if and only if, it maximizes $p(x,y;\theta)$. This facilitates inference.

Finally, the posterior marginal probability of state z at position t given the observation sequence x is computed by summing, or marginalizing, over all state sequence y, where $y_t = z$. It is defined by Equation (5.4)

$$p(z,t,\ x;\theta) = \sum_{y' \in Y^{T+1},\ y'_t = z,\ y'_{T+1} = f} p(x,y';\theta) \tag{5.4}$$

5.3 Inference

Informally, inference in HMMs means finding a maximally probable sequence of hidden states y that might have emitted the observation x. As Rabiner (1989) points out, this statement is not strong enough to suggest an algorithm.

Maximally probable is an ambiguous term when dealing with structured models. It could be taken to mean at least two distinct things. The MAP assignment y_{MAP} of the hidden state sequence is the most probable joint assignment of states defined by Equation (5.5) and depicted in Figure 5.4a.

$$y_{MAP} = \arg\max_{y \in Y^T} p(y \,|\, x;\theta) \tag{5.5}$$

Another possible definition would be the *maximum marginal* (MM) assignment. It chooses the most probable hidden state for each word considering all possible assignments of states for the remaining words. The MM assignment y_{MM} is defined by Equation (5.6). Figure 5.4c shows the marginal for one position and state.

$$y_{MM} = \arg\max_{y \in Y^T} \prod_{t=1}^{T} p(y_t, t \mid x; \theta) \tag{5.6}$$

As Merialdo (1994) and many others have noted, the MAP and MM assignments maximize different objectives. The MM assignment maximizes the accuracy of correct states per observations whereas the MAP assignment maximizes the number of completely correct state sequences. Both objectives are important from the point of view of POS tagging. However, they are often quite correlated and, at least in POS tagging, it does not matter in practice which of the criteria is used (Merialdo, 1994). Most systems, for example Church (1988), Brants (2000), Halácsy et al. (2007), have chosen to use MAP inference.

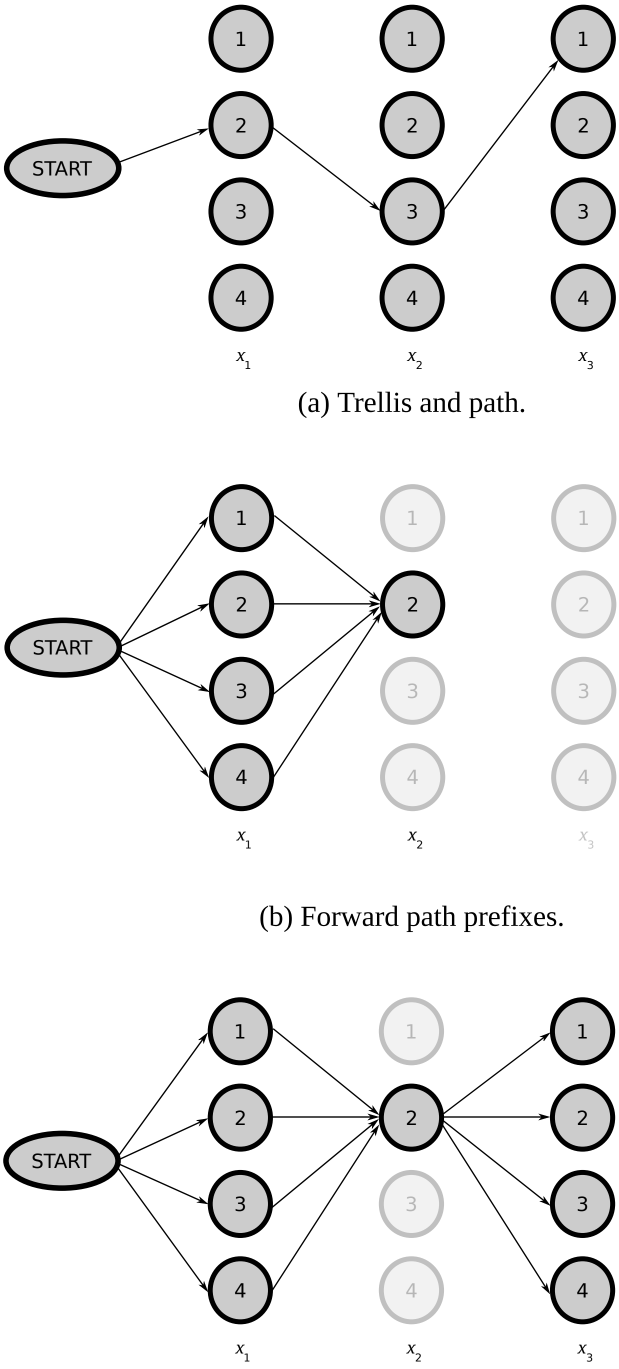
Although, MM inference is more rarely used with HMMs, computing the marginals is important both in unsupervised estimation of HMMs and discriminative estimation of sequence models. Therefore, an efficient algorithm for MM inference, the *forward algorithm*, is presented below.

There are a number of strongly related algorithms for both exact MAP and MM inference. The work presented in this thesis, uses the Viterbi algorithm for MAP inference and the forward-backward algorithm for MM inference (Rabiner, 1989). *Belief propagation*, introduced by Pearl (1982), computes the MM assignment and can be modified to compute the MAP assignment as well. For sequence models, such as the HMM where hidden states form a directed sequence, belief propagation is very similar to the forward-backward algorithm. It can, however, be extended to cyclic graphs (Weiss, 2000) unlike the Viterbi algorithm.

Since cyclic models fall beyond the scope of this thesis and both the Viterbi and forward-backward algorithms are amenable to well known optimizations, which are of great practical importance, I will not discuss belief propagation belief propagation further. Koller and Friedman (2009) gives a nice treatment of belief propagation and graphical models at large.

Before introducing the Viterbi and forward-backward algorithm, it is necessary to investigate the forward algorithm, which is used to compute the marginal probability of an observation and also as part of the forward-backward algorithm. The forward algorithm and Viterbi algorithm are closely related.

Figure 5.4: foo.



(a) Trellis and path.

(b) Forward path prefixes.

(c) Marginal paths.

5.3.1 The Forward Algorithm

Equations (5.5) and (5.6) reveal, that both MAP and MM inference require knowledge of the entire observation x. In the weather prediction example, observations are, however, always infinite. What kind of inference is possible in this case?

Even when we only know a prefix $x[1:t]$ (of length t) of the entire observation x, we can still compute the *belief state* (Boyen and Koller, 1998) of the HMM given the prefix. The belief state is in fact not a single state, but rather a distribution over the set of hidden states Y. It tells us how likely we are to be in state z at time t, when we have emitted the prefix $x[1:t]$.

To compute the belief state at position t, we first need to compute the *forward probabilities* for each state $z \in Y$. The forward probability $\mathrm{fw}_{t,z}(x)$ of state z at position t is the probability of emitting prefix $x[1:t]$ and ending up in state $z \in Y$. For example, given an infinite observation {☂, ☂, ☂̸, ...}, the forward probability $\mathrm{fw}_{3,\text{RAIN}}$ is the probability that the third day is rainy, when the nurse carried an umbrella on the first and second days, but did not carry one on the third day.

I am going to make a technical but useful definition. The *prefix probability* of observation sequence $x = (x_1,\ ...,\ x_T)$ and state sequence $y = (y_1,\ ...,\ y_t)$ at position, where $t < T + 1$ is given by Equation (5.7).

$$p(x,\ y;\theta) = \left(\iota(y_1) \cdot \left(\prod_{u=1}^{t-1} \tau_{y_u}(y_{u+1})\right) \cdot \prod_{u=1}^{t} \varepsilon_{y_u}(x_u)\right),\ t \leq T \tag{5.7}$$

When $t = T$, this is very similar to the joint probability of x and y, but the final transition is missing.

Conceptually, the forward probability is computed by summing over the probabilities of all path prefixes up to position t, where the state at position t is z, see Figure 5.4b. Formally, the forward probability is defined by Equation (5.8).

$$\mathrm{fw}_{t,z} = \sum_{y \in Y^t,\ y_t = z} p(x,\ y;\theta) \tag{5.8}$$

Comparing Equations (5.8) and (5.1) shows that the forward probability in a sense represents the probability of a prefix of observation x.

The belief state and posterior marginal distribution may seem similar. They are,

Figure 5.5: foo

y_1	y_2	y_3	p
CLEAR	CLEAR	CLEAR	$(0.75 \cdot 0.8 \cdot 0.9 \cdot 0.2) \cdot 0.9 \cdot 0.8 \approx 0.078$
RAIN	CLEAR	CLEAR	$(0.25 \cdot 0.2 \cdot 0.3 \cdot 0.2) \cdot 0.9 \cdot 0.8 \approx 0.002$
CLEAR	RAIN	CLEAR	$(0.75 \cdot 0.8 \cdot 0.1 \cdot 0.8) \cdot 0.3 \cdot 0.8 \approx 0.012$
RAIN	RAIN	CLEAR	$(0.25 \cdot 0.2 \cdot 0.7 \cdot 0.8) \cdot 0.3 \cdot 0.8 \approx 0.007$
☂	☂	~~☂~~	
			≈ 0.098

however, distinct distributions because the belief state disregards all information about observation x after position t. In contrast, the marginal distribution encompasses information about the entire observation. For example the marginal probability of RAIN at position 3 is likely to depend strongly on whether or not Jill observes the nurse carry an umbrella on the fourth day. However, this will have no impact on the belief state.

Figure 5.5 demonstrates a naive approach to computing the forward probabilities. Simply list all relevant state sequences, compute the probability of each sequence and sum the probabilities. Unfortunately, the naive approach fails for large t because the number of distinct state sequences depends on the sequence length in an exponential manner.

The complexity of the naive algorithm is $|Y|^t$, which is infeasible. For example, $f_{20,\text{RAIN}}(x)$ requires us to sum approximately 20 million probabilities and $f_{30,\text{RAIN}}(x)$ entails summation of approximately 540 million probabilities. Since observation sequences in domains such as natural language processing frequently reach lengths of 100, a more efficient approach is required.

The belief state can be computed in linear time with regard to t and quadratic time with regard to $|Y|$ using the *forward algorithm* (Rabiner, 1989), which is in fact simply a recursive application of the right distributive rule of algebra

$$a_1 \cdot b + ... + a_n \cdot b = (a_1 + ... + a_n) \cdot b$$

for real numbers a_1 up to a_n and b.

Instead of computing the probability separately for each path, the forward probabilities for longer paths are computed incrementally using the forward probabilities of shorter paths. Examine Figure 5.5. By grouping rows one and two, as well as three and four into pairs, it is easy to see that

$$\text{fw}_{3,\text{CLEAR}} = (\text{fw}_{2,\text{RAIN}} \cdot \tau_{\text{RAIN}}(\text{CLEAR}) + \text{fw}_{2,\text{CLEAR}} \cdot \tau_{\text{CLEAR}}(\text{CLEAR})) \cdot \varepsilon_{\text{CLEAR}}(\cancel{\text{☂}})$$

Generalizing, we get the recursion in Equation (5.9).

$$\text{fw}_{t,z} = \begin{cases} \iota(z) \cdot \varepsilon_z(x_1) & , t = 1 \\ \left(\sum_{z' \in Y} \text{fw}_{t-1,z'} \cdot \tau_{z'}(z) \right) \cdot \varepsilon_z(x_t) & , 1 < t \leq T \\ \sum_{z' \in Y} \text{fw}_{T,z'} \cdot \tau_{z'}(f) & , t = T+1, \ z = f. \end{cases} \tag{5.9}$$

The remaining forward probabilities $\text{fw}_{T+1,z}$, where $z \neq f$ are defined to be 0.

The forward probability $f_{T+1,f} = p(x;\theta)$. In fact one of the principal applications for the forward algorithm is computing the marginal probability of an observation. The other central application is in the forward-backward algorithm, which computes the state marginals.

The forward algorithm is outlined in Algorithm 8.1. Assuming that accessing the data structures `x`, `i_prob`, `e_prob`, `tr_prob` and `trellis` is constant time, the complexity of the algorithm is dominated by the three nested loops on lines 27–37. This shows that the complexity of the forward algorithm is linear with regard to the length of the sequence and quadratic with regard to the size of the hidden state set.

Although, the forward algorithm depends linearly on the observation length, its quadratic dependence on the size of the hidden state set is problematic from the perspective of morphological disambiguation of morphologically complex languages, where the size of the hidden state set is measured in the hundreds or thousands for regular HMMs. When using second order HMMs presented below, the state set can grow to tens of thousands or millions, which can slow down systems to a degree that makes them infeasible in practice. I will present partial solutions to these problems below.

Algorithm 5.1: The forward algorithm in Python 3.

```python
def forward(x, i_prob, e_prob, tr_prob):
    """
        x       - The observation as a list.
        i_prob  - Initial state distribution.
        e_prob  - Emission distributions.
        tr_prob - Transition distributions.

        Return the trellis of forward probabilities.
    """

    assert(not x.empty())

    trellis = {}

    # Indexing in python starts at 0.
    x_1 = x[0]
    T   = len(x) + 1

    # Set final state F. States are consequtive integers
    # in the range [0, F].
    F = len(i_prob) - 1

    # Initialize first trellis column.
    for z in range(F):
        trellis[(1,z)] = i_prob[z] * e_prob[z][x_1]

    # Set all except the final column.
    for t in range(2, T):
        trellis[(t, z)] = 0

        x_t = x[t - 1]

        for z in range(F):
            for s in range(F):
                trellis[(t, z)] = trellis[(t - 1, s)] * tr_prob[s][z]

            trellis[(t, z)] *= em_prob[z][x_t]

    # Set the last column.
    for z in range(s_count):
        trellis[(T + 1, z)] = trellis[(T, z)] * tr_prob[z][F]

    return trellis
```

5.3.2 The Viterbi Algorithm

Whereas the forward algorithm incrementally computes the marginal probability of an observation x, the Viterbi algorithm incrementally computes the MAP assignment for observation x.

A naive approach to finding the MAP assignment is to list all the hidden state paths, compute their probabilities and pick the one with the highest probability. Similarly as for the forward algorithm, the complexity of this approach is exponential with regard to the length of observation x.

Just as in the case of forward probabilities, the MAP assignment of hidden states for a prefix of the observation x can be computed incrementally. Formally, the MAP assignment for a prefix $x[1 : t]$ is defined by equation (5.10) utilizing the joint prefix probability of x and a state sequence y of length t. Intuitively, it is the sequence of hidden states $y_{t,z}$ which maximizes the joint probability and ends at state z.

$$y_{t,z} = \operatorname*{arg\,max}_{y \in Y^t,\ y_t = z} p(x,\ y; \theta) \tag{5.10}$$

Comparing this equation with the definition of the forward probability $f_{t,z}$ in Equation 5.8, we can see that the only difference is that the sum has been changed to $\arg\max$.

I will now show that the MAP prefix $y_{t,z}$ can be computed incrementally in a similar fashion as the forward probability $f_{t,z}$. Suppose that $y_{t+1,z'} = (y_1,\ \ldots,\ y_t = z,\ y_{t+1} = z')$. I will show that $y_{t+1,z'}[1 : t] = y_{t,z}$. Let y' be the concatenation of $y_{t,z}$ and z'. If $y_{t+1,z'}[1 : t] \neq y_{t,z}$, then

$$\begin{aligned} p(x,\ y_{t+1,z'}; \theta) &= p(x,\ y_{t+1,z'}[1 : t]; \theta) \cdot \tau_z(z') \cdot \varepsilon_{z'}(x_{t+1}) \\ &< p(x,\ y_{t,z}; \theta) \cdot \tau_z(z') \cdot \varepsilon_{z'}(x_{t+1}) \\ &= p(x, y'; \theta) \end{aligned}$$

This contradicts the definition in Equation (5.10)[4].

[4] As long as we suppose that there is exactly one MAP prefix.

We now get Equation (5.11), which gives us a recursion.

$$y_{t+1,z} = \arg\max_{z \in Y} \begin{cases} \iota(z) \cdot \varepsilon_z(x_1) & , t = 1 \\ y_{t-1,z'} \cdot \tau_{z'}(z) \cdot \varepsilon_z(x_t) & , 1 < t \leq T \\ y_{T,z'} \cdot \tau_{z'}(f) & , t = T+1,\ z = f. \end{cases} \tag{5.11}$$

5.3.3 Beam Search

As seen in the previous section, the complexity of the Viterbi algorithm depends on the square of the size of the hidden state set. This can be problematic when the set of hidden states is large, for example when the states represent POS tags in a very large label set or when they represent combinations of POS tags. When tagging, a morphologically complex language, the state set may easily encompass hundreds or even thousands of states.

Beam search is is a heuristic which prunes the search space explored by the Viterbi algorithm based on the following observation: in many practical applications, the number of hidden states which emit a given observation with appreciable probability is small. This is true even when the total number of hidden states is very large. For example, when the states represent POS labels, a given word such as “dog” can usually only be emitted by a couple of states (maybe Noun and Verb in this case).

When the Viterbi algorithm maximizes (5.11) for $y_{t+1,z}$, a large number of histories $y_{t,z}$ can, therefore, be ignored.

Often a constant number, the *beam width*, of potential histories are considered in the maximization. The complexity of the Viterbi algorithm with beam search is $o(|Y|\cdot \log|Y|)$. The log factor stems from the fact that the histories need to be sorted before maximization.[5]

In addition to histories, the possible hidden states for output can also be filtered. The simplest method in to use a so called tag dictionary. These techniques are described in Section 5.6.

[5] Sequential decoding, an approximate inference algorithm, which was used for decoding before the Viterbi algorithm was in common use (Forney, 2005) is very similar to beam search. Indeed, it could be said that Viterbi invented an exact inference algorithm, which is once more broken by beam search.

5.3.4 The Forward-Backward Algorithm

The Viterbi algorithm computes the MAP assignment for the hidden states efficiently. For efficiently computing the marginal probability for a every state and position (see Figure 5.4c), the forward-backward algorithm is required.

Intuitively, the probability that a state sequence y has state $z \in Y$ at position t, that is the probability that $y_t = z$ is the product of the probabilities that the prefix $y[1 : t]$ ends up at state z and the probability that the suffix $y[t : T]$ originates at z.

The name forward-backward algorithm stems from the fact, that the algorithm essentially consists of one pass of the forward algorithm, which computes prefix probabilities, and another pass of the forward algorithm starting at the end of the sentence and moving towards the beginning which computes suffix probabilities. Finally, the forward and suffix probabilities are combined to give the marginal probability of all paths where the state at position t is z. These passes are called the forward and backward pass.

Formally, the suffix probabilities computed by the backward pass are defined by equation (5.12).

Since a backward pass of the forward algorithm carries the same complexity as the forward pass, we can see that the complexity of the forward-backward algorithm is the same as the complexity of the forward algorithm, however, there is a constant factor of two compared to the forward algorithm.

$$b_{t,z} = \begin{cases} \left(\sum_{z' \in Y} t_z(z') \cdot b_{t+1,z'} \right) \cdot e_z(x_{t+1}) & , 1 < t < T \\ t_z(f) & , t = T + 1,\ z = f. \end{cases} \tag{5.12}$$

5.3.5 Sparse Forward-Backward Algorithms

FIXME (Pal et al., 2006).

5.4 Estimation

HMMs can be trained in different ways depending on the quality of the available data, but also on the task at hand. The classical setting presented by Rabiner (1989) is nearly completely unsupervised: the HMM is trained exclusively from observations. Some supervision is nevertheless usually required to determine the number of hidden states[6]. Additionally priors on the emission and transitions distributions may be required to avoid undesirably even distributions (Cutting et al., 1992, Johnson, 2007).

The unsupervised training setting has two important and interrelated applications:

1. Modeling a complex stochastic process from limited data. Here the HMM can be contrasted to a Markov chain (Manning and Schütze, 1999, 318–320), where each emission can occur in a unique state leading to a higher degree of data sparsity and inability to model under-lying structure.

2. Uncovering structure in data, for example part-of-speech induction (Johnson, 2007).

The classical method for unsupervised Maximum likelihood estimation of HMMs is the *Baum-Welch algorithm* (Rabiner, 1989), which is an instance of the *expectation maximization algorithm* (EM) (Dempster et al., 1977) for HMMs.

In POS tagging and morphological disambiguation, the supervised training scenario is normally used. Supervised training consists of annotating a text corpus with POS labels and estimating the emission and transition probabilities from the annotated data.

Straight-forward counting is sufficient to get the ML estimates for the transition and emission distributions. For example, one can simply count how often a determiner is followed by a noun, an adjective or some other class. Similarly, one can count how many often a verb label emits “dog” and how often the noun label emits “dog”.

Even in large training corpora, “dog” might very well never receive a verb label[7]. Nevertheless, “dog” can be a verb, for example in the sentence “Fans may dog Muschamp, but one thing’s for certain: he did things the right way off the field.”. To avoid this kind of problems caused by data sparsity, both emission and transition counts need to be smoothed.

5.4.1 Counting for Supervised ML Estimation

When HMMs are used in linguistic labeling tasks, such as part-of-speech tagging, they are usually estimated in a supervised manner. Each label is thought to represent a hidden

[6] Although methods for determining the number of states from the data exist (**?**).

[7] There are ten occurrences of “dog” in the Penn Treebank and all of them are analyzed as nouns.

variable, and the HMM models the transitions from one label type to another and the emission of words from each label type.

`Mr.`	`NNP`
`Vinken`	`NNP`
`is`	`VBZ`
`chairman`	`NN`
`of`	`IN`
`Elsevier`	`NNP`
`N.V.`	`NNP`
`,`	`,`
`the`	`DT`
`Dutch`	`NNP`
`publishing`	`VBG`
`group`	`NN`
`.`	`.`

Figure 5.6: foo

Figure 5.6 shows one sentence from the Penn Treebank (Marcus et al., 1993). The sentence is labeled with POS tags which are taken to be the hidden states of an HMM. When estimating an HMM tagger for the corpus, transitions probabilities, for example $t_{\texttt{NNP},\texttt{VBZ}}$, and emission probabilities, for example $e_{\texttt{NNP}}(\texttt{Dutch})$ can in principle be computed directly from the corpus. For example the transition probability $t_{\texttt{NNP},\texttt{VBZ}}$ and the emission probability $e_{\texttt{NNP}}(\texttt{Dutch})$ in the Penn Treebank are simply:

$$t_{\texttt{NNP},\texttt{VBZ}} = \frac{\text{Count of POS tag pair } \texttt{NNP VBZ} \text{ in the corpus}}{\text{Count of POS tag } \texttt{NNP} \text{ in the corpus}} = \frac{4294}{114053} \approx 0.04$$

$$e_{\texttt{NNP}}(\texttt{Dutch}) = \frac{\text{Number of times } \texttt{Dutch} \text{ was tagged } \texttt{NNP} \text{ in the corpus}}{\text{Count of POS tag } \texttt{NNP} \text{ in the corpus}} = \frac{14}{114053} \approx 1.2{\cdot}10^{-4}$$

Simple computation of co-occurrences is insufficient because of data-sparsity. Words do not occur with all POS tags in the training corpus and all compbinations of POS tags are never observed. Sometimes this is not a problem. For example, “Dutch” could never be a preposition. We know that the probability that a preposition state emits “Dutch” is 0. However, there are at least three analyses that are perfectly plausible: noun (the

Dutch language), adjective (property of being from The Nederlands) and proper noun (for example in the restaurant name “The Dutch”).

Since “Dutch” occurs only 14 times in the Penn Treebank, it is not surprising that all of these analyses do not occur. Specifically, the noun analysis is missing. An HMM based on direct counts will therefore never analyze “Dutch” as a noun.

It is tempting to think that missing analyses are a minor problem because they only occur for relatively rare words such as “Dutch”. Unfortunately, a large portion of text is made up from rare words. The problem therefore has very real consequences.

The usual approach is to use a family of techniques called *smoothing*. In smoohing, zero counts and all other counts are modified slightly to counter-act aprsity.

Smoothing of emission proabilities and transition probabilities differ slightly. For transition probabilities it is common practice to use counts of both tag pairs and single tags to estimate tag probabilities either in a back-off scheme (**?**) or using interpolation (Brants, 2000). Many sophisticated interpolation schemes can be used for example Kneser-Ney (**?**).

Many systems such as the HMM tagger by Brants (2000) do not smooth emission probabilities for words seen in the training corpus. However, words *not* seen in the training corpus, or out-of-vocabulary (OOV) words still require special processing. The simplest method is to estimate combined statistics for all words occurring one time in the training corpus and use these statistics for OOV words (**?**). Lidstone smoothing is another similar approach (**?**). Another approach would be to build models to guess the analysis of OOV words using the longest suffix of the word shared with a word in the training data.

Brants (2000) employs a specialized emission model for OOV words, which combines both approaches. It assigns a probability $p(y|x)$ for any label $y \in \mathcal{Y}$ and an arbitrary word x based on suffixes s_i of the word different lengths. The subscript i indicates suffix length.

The model uses relative frequencies $\hat{p}(y|s_i)$ of label y given each suffix s_i of x that occurs in the training data. The frequencies for different suffix lengths are recursively combined into probability estimates $p(y|s_i)$ using successive interpolations

$$p(y|s_{i+1}) = \frac{\hat{p}(y|s_{i+1}) + \theta \cdot p(y|s_i)}{1 + \theta}.$$

The base case $p(y|s_0)$, for the empty suffix s_0, is given by the overall frequency of label type y in the training data, i.e. $p(y|s_0) = \hat{p}(y)$, and the interpolation coefficient θ is the variance of the frequencies of label types in the training data

$$\theta = \frac{1}{|\mathcal{Y}|-1}\sum_{y\in\mathcal{Y}}(\hat{p} - \hat{p}(y))^2.$$

Here $\hat{p}$ is the average frequency of a label type. Finally, $p(y|x) = p(y|s_I)$, where s_I is the longest suffix of x that occurs in the training data. However, a maximal suffix length is imposed to avoid over-fitting. Brants (2000) uses 10 for English. Moreover, the training data for the emission model is restricted to include only "rare" words, that is words whose frequency does not exceed a given threshold. This is necessary, because the distribution labels for OOV words usually differs significantly from the overall label distribution in the training data.

Brants (2000) does not discuss the choice of θ in great length. It is, however, instructive to consider the effect of the magnitude of θ on the emission model. When the variance of label type frequencies, that is θ, is great, shorter suffixes and the prior distribution of label types will weigh more than long suffixes. This is sensible as (1) a high θ implies that the distribution of words into label types is eschewed a priori and (2) long suffix statistics are sparse and thus prone to overfitting. When θ is low, the prior distribution of word classes is closer to the even distribution. Therefore, there is no choice but to trust longer suffixes more.

For morphologically complex languages, the smoothing scheme employed by Brants (2000) may be inferior to a longest suffix approach utilized by Silfverberg and Lindén (2011) and Lindén (2009). This may happen because productive compounding. For languages with writing systems that radically differ from English, such as Mandarin Chinese, suffix based methods work poorly. Other methods, such as basing the guess on all symbols in the word, may work better. Huang et al. (2007) smooth symbol emission probabilities using geometric mean.

5.4.2 The EM algorithm for Unsupervised ML Estimation

The Baum-Welch, or Expectation Maximization, algorithm for HMMs is an iterative hill-climbing algorithm, that can be used to find locally optimal parameters for an HMM given a number of unlabeled independent training examples which are drawn from the distribution that is being modeled by the HMM. Here is a short outline of the algorithm:

1. Random initialize the emission and transition parameters.

2. Use the forward-backward algorithm to compute posterior marginals over input positions.

3. Use the posterior marginals as *soft counts* to estimate new parameters.

4. Repeat steps 2 and 3 until the improvement of likelihood of the training data is below a threshold value.

In step 2, the algorithm computes the maximally likely state distribution for each position given the current parameters. In step 3, the state distributions for each position in the input data are used to infer the MAP parameters for the HMM. Therefore, the marginal probability of the training data has to increase on every iteration of steps 2 and 3, or possible remain the same, if the current parameters are optimal.

There are no guarantees that the optimum found by the EM algorithm is global. Therefore, several random restarts are used and parameters giving the best marginal probability for the training data are used.

A more formal treatment of the EM algorithm can be found in Bilmes (1997).

5.5 Model Order

The standard HMM presented above is called a *first order model* because the next hidden state is determined solely based on the current hidden state. This model is easy to estimate and resistant to over-fitting caused by data-sparsity, but it fails to capture some key properties of language. For example, in the Penn Treebank, the probability of seeing a second adverb `RB` following and adverb is approximately, 0.08. If the first order assumption were valid, the probability of seeing a third adverb following two adverbs should also be 8%, however it is lower, around 5%.

The example with adverbs is poor at best, but it illustrates the kind of effect *second order* information can have. Second order HMMs are models where transitions are conditioned on two preceding hidden states. Equivalently, in POS tagging, the hidden states

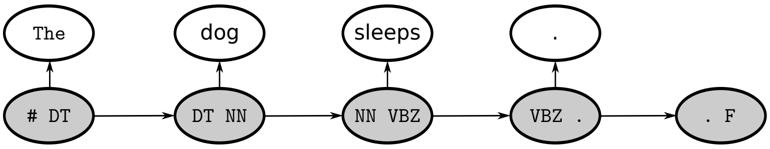


Figure 5.7: foo

can be taken to be pairs of POS tags, e.g. `DT, NN`. In such a model transitions can only occur to a subset of the hidden state set. For example a transition from `DT, NN` to `NN, VBZ` is possible, but a transition to `JJ, NN` is impossible. Figure 5.7 illustrates a path with legal transitions.

Figure 5.7 implies that emissions in a second order model are conditioned on two labels like the transitions. However, many existing HMM based POS tagging systems such as Brants (2000) condition emissions only on one label, that is use $e_{t_i,t_{i-1}}(w_i) = p(w_i \,|\, t_i)$ instead of $e_{t_i,t_{i-1}}(w_i) = p(w_i \,|\, t_{i-1},\ t_i)$. The reason is probably data-sparsity. Therefore, these systems cannot be called HMMs in the strictest sense of the word. They are instead called trigram taggers.

Halácsy et al. (2007), show that it is possible to maintain the correct HMM formulation over-come the data sparseness problem and achieve gains over the more commonly used trigram tagger. However, they fail to describe the smoothing scheme used, which is crucial. This defect is partly remedied by the fact that the system is open-source. One of the chief contributions of Silfverberg and Lindén (2011) was to investigate the effect of different ways of estimating the emission parameters in a generative trigram tagger paying attention to smoothing.

Increasing model order unfortunately leads to increased data sparsity, because the number of hidden states increases. Therefore, smoothing transition probabilities is even more important than in the first order case. Even using smoothing, third and higher order models tend to generalize to unseen data more poorly than lower order models because of over-fitting (**?**).

An alternative to increasing model order, is to use so called latent annotations (Huang et al., 2009) in an otherwise regular first order HMM. Conceptually, each label for example `NN` is split into a number of sub-states `NN1`, `NN2` and so on. Expectation maximization is used to train the model in a partly supervised fashion. Splitting labels, and indeed any increase in order, is probably works better for label sets with quite few labels. Otherwise, it will simply contribute data sparsity.

5.6 HMM taggers and Morphological Analyzers

The inventory of POS labels that are possible for a given word form tends to be small. For example the English “dog” can get two of the Penn Treebank POS tags singular noun `NN` and `VB` infinitive verb form. The remaining 43 POS tags can never label “dog”. Consequently, in an HMM POS tagger, only the states corresponding to `VB` and `NN` should ever emit the word “dog”.

A `tag dictionary` (Brants, 2000) can be used in combination with the Viterbi algorithm to limit the set of hidden states that could emit a word. The tag dictionary can be constructed from the training corpus. Additionally, an external lexical resource, such as a morphological analyzer, can be used. Such a lexical resource can help to compensate for missing statistics for OOV words. In the frequent setting, where most rare words have quite few analyses, this can have a substantial effect on tagging accuracy.

Chapter 6

Finite-State Machines

This thesis deals primarily with machine learning but Silfverberg and Lindén (2011) presents a weighted finite-state implementation of Hidden Markov Models. Therefore, a quick overview of finite-state calculus is required.

6.1 Weighted Finite-State Automata

Automata can be seen as a formalization of the concept of algorithm. They represent a restricted type of algorithm in the sense that they can only operate on symbol strings and they only operation they perform is to accept or reject a string. Although, this might sound quite restrictive, it turns out that most computational problems can be formalized as such *decision problems* involving only acceptance or rejection of symbol strings.

Every automaton is associated with a *formal language* which is a (possible infinite) set of finite strings from the set of all strings of some finite alphabet. The automaton solves the decision problem of this language, i.e. accepts exactly those strings that belong to the language. *Weighted formal languages* are an extension to the concept of formal language, where each string should be assigned a weight. If the weights can be interpreted as probabilities, a weighted formal language over an alphabet Σ can be seen as a probability distribution over the set of strings of the alphabet. However, this is not always possible because all weights cannot be interpreted as probabilities.

Several well know classes of algorithms exist. The most famous ones are Turing machines which, in a sense, subsume all existing automata **?**. In this thesis I have, however, utilized another class of automata, Finite-state machines. They are more restricted than Turing machines or another well know class, stack automata, because they utilize only a finite amount of memory. All real world implementations of automata of course utilize only a finite amount of memory.

The restriction on memory limits the set of problems that finite-state machines can solve. However, at the same time it allows for a rich algebra which allows one to combine finite-state machines to produce new finite-state machines.

The machine in Figure 6.1 accepts sequences of Penn Treebank labels which correspond to singular noun phrases that can have an adjective attribute (JJ) with an optional adverb (RB) expressing degree.

Typically finite-state machines are represented as state transition graphs. An example is given in Figure 6.1. The machine has a set of numbered *states* (0, 1, 2, 3 and 4) and *transitions* from one state to another. The transitions are labeled using symbols from a finite alphabet (e.g. NN) and weights from a *weight semiring*, when the machine is weighted. The weight semiring in this example is the *probability semiring* Allauzen et al. (2007) where weights are probabilities that can be added and multiplied in the usual fashion.

Operation starts in the designated *start state* (0 in Figure 6.1) and it has to end in a *final state* (2 in Figure 6.1 marked with a double ring). A string is accepted, if and only if there is a sequence of transitions leading from the start state to a final state where the symbols in the transition labels spell out the input string and the product of the weights along the path is non-zero. For example the machine in Figure **??** accepts the string “DT NN” with weight 0.7 but rejects “DT JJ” because the string inds when the machine is in state 4 but that is not a finite state.

There may be several or no *accepting paths* for a given string. If there are several paths for some string, the machine is called *ambiguous*. More generally, a machine is called *non-deterministic* if some state has several out-going transitions with the same label. Trivially, ambiguous machines are non-deterministic. In the case of ambiguous machines, the weight of a string is the sum of the weights of all of its accepting paths or zero if there are no accepting paths.

Weighted formal languages Formally, weighted languages are mappings $M : \Sigma^* \to \mathbb{K}$ from the set of arbitrary strings Σ^* of some finite alphabet Σ into some *weight semiring*

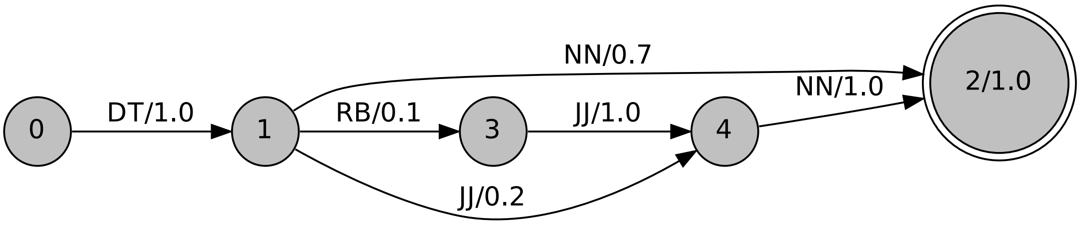


Figure 6.1: A finite-state machine accepting a subset of the singular noun phrases in Penn Treebank.

$(\mathbb{K}, \oplus, \otimes, \mathbb{0}, \mathbb{1})$ where $\mathbb{K}$ is a set of weights, $\oplus$ and $\otimes$ are addition and product operators for weights and $\mathbb{0} \in \mathbb{K}$ and $\mathbb{1} \in \mathbb{K}$ are the additive and multiplicative identity.

Weight semirings As mentioned above, the weight semiring in Figure 6.1 is the probability semiring, where weights belong to the set of non-negative reals $[0, \infty)$ and addition and product are given by the regular real addition and product operations. Even though probabilities reside in $[0, 1]$, the addition of probabilities can result in quantities that are larger than 1. Because finite-state algebra does include operations that add probabilities, the weight set needs to include all non-negative reals.

The general weight semiring $(\mathbb{K}, \oplus, \otimes, \mathbb{0}, \mathbb{1})$ is an abstraction of the probability semiring. The set $\mathbb{K}$ is an arbitrary set but the operations $\oplus \to \mathbb{K} \times \mathbb{K} \to \mathbb{K}$ and $\oplus \to \mathbb{K} \times \mathbb{K} \to \mathbb{K}$ need to satify the following requirements Allauzen et al. (2007)

1. $(\mathbb{K}, \oplus, \mathbb{0})$ is a commutative and associative monoid.
2. $(\mathbb{K}, \otimes, \mathbb{1})$ is an associative monoid.
3. $\otimes$ distributes with respect to $\oplus$.
4. $\mathbb{0}$ is the annihilator of $\otimes$.

If $\oplus$ is a group operation, the semiring is in fact a ring. Many useful weight semirings are, however, not rings. Often the semiring $(\mathbb{K}, \oplus, \otimes, \mathbb{0}, \mathbb{1})$ is denoted simply as $\mathbb{K}$. Paralleling the regular sum notation $\Sigma_{i \in I} x_i$ and product notation $\prod_{i \in I} x_i$, I use the notations $\bigoplus_{i \in I} x_i$ and $\bigotimes_{i \in I} x_i$, where I is a denumerable set.[1] Specifically $\oplus_{i \in \emptyset} x_i = \mathbb{0}$

[1] Infinite sums and products are not defined in all semi rings, for example the probability semiring. In this case, it is often possible to augment the semirings with an infinity element ∞. This problem does not often cause problems in practice and will not be examined further.

and $\otimes_{i\in\emptyset} x_i = \mathbb{1}$.

Weights in the probability semiring and the addition and product operations have an intuitive interpretation but from a practical point of view the probability semiring is sub optimal. The biggest drawback is that repeated multiplication of probabilities gives rise to very small quantities and under-flow becomes a problem. Therefore, the so called *logarithmic semiring* is more commonly used. It can be derived by applying a logarithmic transformation $x \mapsto -\log(x)$ to the weights and operations in the probability semiring.

In the logarithmic semiring, numerical under-flow is not a problem but the addition operation given by $x \oplus y = -\log(\exp(-x) + \exp(-y))$ is resource intensive. Therefore, another semiring, the *tropical semiring*, is often used. The addition operation in the tropical semiring is given by $x \oplus y = \min(x, y)$. Computationally, this operation is light. A theoretical motivation is given by the fact that $\min$ preserves the magnitude of the sum although not its exact value. In practice, this is often sufficient **?**. All machines in this thesis use tropical weights.

Weighted finite-state automata Formally, a weighted finite-state automaton of weight semiring $\mathbb{K}$ is a structure $M = (\Sigma, Q, q_0, \rho, E)$ where

1. Σ is a finite symbol set (also called an alphabet).

2. Q is a finite set of states.

3. $q_0 \in Q$ is the unique start state.

4. $\rho : Q \to \mathbb{K}$ is the final weight map.

5. $T \subset Q \times (\Sigma \cup \{\varepsilon\}) \times Q \times \mathbb{K}$ is a finite set of transitions.

The final weight map ρ associates each state $q \in Q$ with a final weight. The final states of M are $\{q \in Q \mid \rho(q) \neq \mathbb{0}\}$.

Each transition $x \in T$ consists of a source state $\mathrm{s}(x)$, an input symbol $\mathrm{i}(x)$, a target state $\mathrm{t}(x)$ and a weight $\mathrm{w}(x)$. The input symbol may be ε which denotes the empty symbol. For the single outgoing transition x in state 4 in Figure 6.1, the source state $\mathrm{s}(x) = 4$, the input symbol $\mathrm{i}(x) = $ "NN", the target state $\mathrm{t}(x) = 2$ and the weight $\mathrm{w}(x) = 1.0$.

A sequence of transitions $x_1, ..., x_n \in T$ is a *path*, if $\mathrm{s}(x_1) = q_0$ and $\mathrm{t}(x_{k-1}) = \mathrm{s}(x_k)$ for all k. As a special case, the empty sequence is also a path. The weight of path $p = (x_1, ..., x_n)$ is $\mathrm{w}_M(p) = (\prod_{i=1}^{n} \mathrm{w}(x_i)) \otimes \rho(x_n)$ and the weight of the empty sequence is $\rho(q_0)$. A path p is called successful when $\mathrm{w}_M(p) \neq \mathbb{0}$.

Each path $p = (x_1, ..., x_n)$ is associated with a unique, possibly empty, string $s \in \Sigma^*$ such that there is a subsequence $j_i, ..., j_m$ of $1, ..., n$ for which $i(x_{j_1}), ..., i(x_{j_m}) = s$ and $i(x_l) = \varepsilon$ whenever $l \notin \{j_1, ..., j_m\}$. If the path p is associated with the string s, it is called a *path of* s.

The set of paths of a string s is denoted P_s and the weight assigned to s by the automaton M is $\mathrm{w}_M(s) = \bigoplus_{p \in P_s} \mathrm{w}_M(p)$. When $P_s = \emptyset$, $\mathrm{w}_M(s) = \mathbb{0}$. The string s belongs to the weighted language $L(M)$ accepted by M if $L(M)(s) = \mathrm{w}_M(s) \neq \mathbb{0}$.

6.2 Finite-State Algebra

Finite-state machines use only a finite fixed amount of memory. Therefore, they cannot solve all computational problems. There is, for example, no finite-state machine which accepts only strings of prime number length. This can be proved using the pumping lemma of regular language (Sipser, 1996) which are the languages whose decision problems finite-state automata can solve.

Although the computational power of finite-state machines is limited, they are closed under a number of very useful operations which correspond to set theoretical operations of the languages recognized by the machines. The collection of these operations is called the *finite-state algebra*.

Table 6.1 gives an overview of a number of well known finite-state operations.

6.3 Finite-State Transducers

Language processing frequently requires translation of one signal such as speech or text into another signal, for example text in another language in the case of machine translation or annotated text in the case of part of speech tagging. These do not seem like the kinds of processing tasks that finite-state machines can solve because, as we saw above, finite-state machines solve decision problems.

Operation	Symbol	Definition
Union	$M_1 \oplus M_2$	$\mathrm{w}_{M_1 \oplus M_2}(s) = \mathrm{w}_{M_1}(s) \oplus \mathrm{w}_{M_2}(s)$
Concatenation	$M_1 \otimes M_2$	$\mathrm{w}_{M_1 \otimes M_2}(s) = \bigoplus_{s_1 s_2 = s} \mathrm{w}_{M_1}(s_1) \otimes \mathrm{w}_{M_2}(s_2)$
Power	M^n	$\mathrm{w}_{M^n}(s) = \bigoplus_{s_1 \ldots s_n = s} \mathrm{w}_M(s_1) \otimes \ldots \otimes \mathrm{w}_M(s_n)$
Closure	$\bigoplus_{n=0}^{\infty} M^n$	
Intersection	$M_1 \cap M_2$	$\mathrm{w}_{M_1 \cap M_2}(s) = \mathrm{w}_{M_1}(s) \otimes \mathrm{w}_{M_2}(s)$

Table 6.1: A selection of operators from the finite-state algebra.

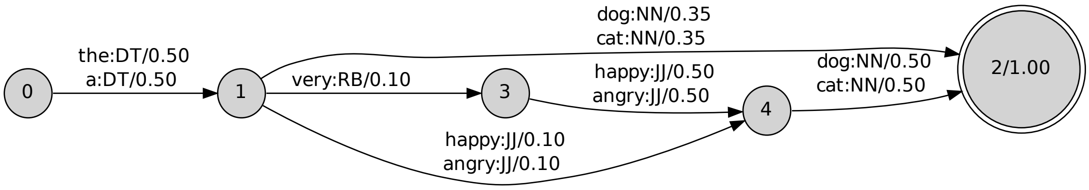


Figure 6.2: A finite-state transducer that analyzes a subset of the singular noun phrases in Penn Treebank.

Weighted Finite-state transducers are an extension of weighted finite-state automata. Whereas finite-state automata simply accept or reject a symbol sequence, finite-state transducers simultaneously output another symbol sequence.

Formally, a weighted finite-state transducer of weight semiring $\mathbb{K}$ is a structure $M = (\Sigma, \Omega, Q, q_0, \rho, E)$ where

1. Σ is a finite input symbol set (also called an input alphabet).

2. Ω is a finite output symbol set (also called an output alphabet).

3. Q is a finite set of states.

4. $q_0 \in Q$ is the unique start state.

5. $\rho : Q \to \mathbb{K}$ is the final weight map.

6. $T \subset Q \times (\Sigma \cup \{\varepsilon\}) \times (\Omega \cup \{\varepsilon\}) \times Q \times \mathbb{K}$ is a finite set of transitions.

Transitions $e \in T$ are similar to transitions in automata but in addition to the input symbol $\mathrm{i}(e)$, they also contain an output symbol (e). Paths of transitions are defined in the same way as paths for automata, however, paths of transducer transitions are associated with two strings – an input and an output string. The input string of path $p = (x_1, ..., x_n)$ is the string $s_i = \mathrm{i}(x_1)...\mathrm{i}(x_n)$ disregarding epsilons and the output string is $s_o = (x_1)...(x_n)$ also disregarding epsilons. Path p itself is associated with the string pair $s = s_i{:}s_o$. The set of paths associated with the input string of s is $\mathrm{P}_i(s)$ and the set paths associated with of output string is $\mathrm{P}_o(s)$.

The weight of path p is defined in exaclty the same way as for automata

$$\mathrm{w}_M(p) = \Big(\prod_{i=1}^{n} \mathrm{w}(x_i)\Big) \otimes \rho(x_n).$$

A transducer M can be viewed as a machine that maps input strings to sets of output strings with weights. Equivalently, it can be seen as a machine accepting string pairs $s = s_i{:}s_o$ with some weight $\mathrm{w}_M(s)$. The weight $\mathrm{w}_M(s)$ given to string pair s by transducer M is defined

$$\mathrm{w}_M(s) = \bigoplus_{p \in \mathrm{P}_i(s) \cap \mathrm{P}_o(s)} \mathrm{w}_M(p)$$

The defition takes into account the fact that there may be several alignments of input and output string giving the exact same string pairs. For example, $(\varepsilon{:}b, a{:}b)$ and $(a{:}b, \varepsilon{:}b)$ represent the same string pair $a{:}bb$.

Chapter 7

Generative Taggers using Finite-State Transducers

This section presents a finite-state implementation of granarative taggers such as HMMs using finite-state algebra. This exapnds on Silfverberg and Lindén (2010) and Silfverberg and Lindén (2011).

7.1 Enriching the emission and transition models

- Halácsy et al. (2007) and Silfverberg and Lindén (2011).

7.2 Problems

In the standard first order HMM an observation depends only on the current hidden state. If the hidden state is given, the observation cannot be influenced by the hidden states at other positions in the input or the other observations. This facilitates estimation and makes the system resistant to over-fitting, but at the same time it severly limits the set phenomena tht the model can capture.

An example from POS tagging the Penn Treebank illustrates this problem. The Penn Treebank has two labels for common nouns: `NN` for singular nouns and `NNS` for plural nouns.

- Local normalization.

- Inability to use word context.
- Label bias.
- The inability to use rich features, causes problems in domains other than POS tagging and in POS tagging for MR languages.
- Smoothing is difficult.
- Differences in performance between generative HMMs and discriminative approaches are even greater for morphologically complex languages and small training sets.
- OOV words may require different mechanisms depending on language again causing problems for MR lanuages.

7.3 Finite-State Implementation of Hidden Markov Models

As seen in Chapter **?**, an HMM can be decomposed into a emission model, which models the conditional distribution $p(y|x)$ of all labels y given a state x and a transition model, which models the conditional distribution $p(y_{n+1}|y_1, ..., y_n)$ of states y_{n+1} given a state history $y_1, ..., y_n$.

Both the emission and transition models can be compiled into finite-state machines. The emission model is compiled into one finite-state transducer but the transition model is made up from a number of component models. All the models are combined using a run-time variant of composition and intersection.

7.3.1 Interpreting HMMs as Finite-State Machines

Given a sequence of observations $x = (x_1, ..., x_n)$ and states $y = (y_1, ..., y_n)$, the joined probability for x and y given by an HMM with parameters θ is

$$p(x, y; \theta) = p(y; \theta) \cdot p(x \mid y; \theta) = \left(\iota(y_1) \cdot \prod_{t=1}^{T} \tau_{y_t}(y_{t+1}) \right) \cdot \prod_{t=1}^{T} \varepsilon_{y_t}(x_t)$$

where ι, τ and ε are initial, transition and emission distributions respectively.

In an HMM the states y_i correspond to actual states of the model. When interpreting an HMM as a finite-state transducer, they will instead correspond to output symbols. The transducer will then map the input sequence x to the states sequence y with weight $p(x, y; \theta)$.

Mohri et al. (2002) describe an implementation of HMMs for speech recognition where each individual phoneme can be recognized by a small HMM. Their construction, however, requires $O(|Y|)^n$ states where Y is the set of states of the HMM and n its order. It is easy to see that this is infeasible. For example when using a second order model with a morphological label set of 1000 labels the resulting transducer has around one billion states.

7.3.2 The Emission Model

7.3.3 The Transition Model

7.3.4 Weighted Intersecting Composition

Chapter 8

Conditional Random Fields

8.1 Discriminative modeling

As seen in Chapter 5, the HMM POS tagger can be viewed as a state machine which alternates between sampling words from state specific observation distributions and sampling morphological labels from state specific transition distributions. Each emission word and each morphological label is conditioned *solely* on the current morphological label. These independence assumptions are harsh. For example collocations cannot be adequately modeled, because the model does not include direct information about neighboring words in a sentence.

Although information about word sequences and orthography is quite useful in morphological labeling, it is often difficult to incorporate such information in a generative model. As Sutton and McCallum (2012) note, two principal approaches could, however, be attempted:

1. Extending the emission model presented in Chapter 5 to incorporate dependencies between words and orthographic features.

2. Replacing the usual emission model with a Naive Bayes' model which in theory can handle arbitrary features (although overlapping features cause).

Approach 1 is difficult in a fully generative setting because the emission model needs to account for the complex dependencies that exist between sentence contexts and orthography. There simply does not seem to exist a straightforward way of modeling the dependencies.

In a domain closely related to morphological labeling, namely biomedical entity extraction,Ruokolainen and Silfverberg (2013) show that approach 2 also fails. In fact their experiments indicate that adding richer context modeling such as adjacent words may worsen the performance of a tagger with a Naive Bayes emission model. One reason for this may be that overlapping information sources required in a rich emission model tend to cause the Naive Bayes model to give overly confident probability estimates (Sutton and McCallum, 2012). Combining the probabilities given by the emission model with the transition model can therefore be problematic.

In contrast to generative sequence models, discriminative sequence models such as Maximum Entropy Markov Models (Ratnaparkhi, 1998) and Conditional Random Fields (Lafferty et al., 2001) can incorporate overlapping sources of information. They model the conditional distribution of label sequences $p(y \mid x)$ directly instead of modeling the joint distribution $p(x,\ y)$. Therefore, they do not need to model the distribution of sentences at all.

Discriminative models assign probabilities $p(y \mid x)$ for label sequences $y = (\mathrm{DT},\ \mathrm{NN},\ \mathrm{VBZ},\ .)$ and word sequences $x = (\mathrm{The},\ \mathrm{dog},\ \mathrm{eats},\ .)$ by extracting local features from the input sentence and label sequence. Examples of local features include *(x_1 is “dog” and y_1 is “NN”)* and *(y_0 is “DT” and y_1 is “NN”)*. Each feature is associated with a parameter value and the parameter values are combined to give the conditional likelihood of the entire label sequence. Naturally, the label sequence which maximizes the conditional likelihood given sentence x is the label sequence returned by the discriminative POS tagger.

In generative models, emissions and transitions are independent. Both are determined exclusively based on the current label. Contrastingly, in discriminative models, there are no emissions or transitions. Instead, it is customary to speak about unstructured features relating exclusively to the input sentence, and structured features, which incorporate information about the label sequence. Simplifying a bit, discriminative models make no independence assumptions among features relating to a single position in the sentence. This allows for improved fit to training data but parameter estimation becomes more complex as we shall soon see. Moreover, discriminative models are more prone to over-fitting. This is of course an example of the famous bias-variance trade-off (Geman et al., 1992).

8.2 Maximum Entropy Modeling

8.2.1 Example

8.3 Basics

I will now describe a CRF POS tagger from a practical point-of-view. The tagging procedure encompasses two stages: feature extraction and inference using an exact or approximate inference algorithm. Whereas inference in CRFs is very similar to inference in HMMs, we did not discuss feature extraction in association to HMMs. This is because HMM taggers use a restricted set of features (the current word and preceding labels).

y	DT	NN	VBZ	.
x	The	dog	eats	.

Features are true logical propositions, which connect aspects of the input sentence with labels or parts of the label sequence. Given sentence x and label sequence y of length n, we extract features at every position t in the sentence. For example at position 2 in sentence x, we could extract *The current word x_t is "dog" and the label is "NN"* and *The previous label is "DT" and the current label is "NN"*. We could, however, not extract the same feature *The current word is "dog" and the label is "NN"* at position 1, because this proposition is false when $t = 1$ (the word at position 1 is "The" and the label is "VBZ").

Features are conjunctions of two parts: a feature template, for example *The current word is "dog"* and a label *the label is "NN"*. The set of features recognized by a CRF POS tagger contains all conjunctions $f\&y$ of feature templates f present in the training data and labels y. For example, the tagger would know *The current word is "dog" and the label is "DT"* although it is unlikely that this feature would ever be observed in actual training data.

Ratnaparkhi (1996) introduced a rather rudimentary feature set and variations of this feature set are commonly used in the literature (for example Collins (2002) and Lafferty et al. (2001)). Let W be the set of word forms in the training data. Additionally let P

and S be the sets of prefixes and suffixes of maximal length 4 of all words $w \in W$. Then, the Ratnaparkhi feature set contains the unstructured feature templates in Table 8.1 and the structured feature templates in Table 8.2.

Feature template	Example
The current word is w	The current word is “dog”
The current word has prefix p	The current word has prefix “d-”
The current word has suffix s	The current word has suffix “-og”
The current word contains a digit	
The current word contains a hyphen	
The current word contains a upper case letter	
The previous word is w	The previous word is “The”.
The next word is w	The next word is “eats”.
The word before the previous word is w	
The word after the next word is w	

Table 8.1: foo

Feature template	Example
The label of the previous word is y	The label of the previous word is “NN”.
The label of the previous two words are y' and y	The labels of the two previous words are “DT” and “NN”.

Table 8.2: foo

These feature templates are then combined with all labels occurring in the training set. It is instructive to try to estimate the number of features when using a realistic training set of size around a million words. The number of features is be $|Y|^3 + |WY|$. For small label sets and large training data, the bulk of the feature set tends to consist of unstructured features. However, for large label sets in the order of $1,000$ labels, there will be a significant number of structured features (one billion in this case). This necessitates either dropping second order structured features or using sparse feature representations. All structured features simply cannot be represented in memory. We will see techniques to circumvent these problems. Especially the averaged perceptron is essential.

It is common to represent the CRF using linear algebra. Each position t in the sentence is represented as a vector ϕ_t whose dimensions correspond to the entire space of possible features. The selection of features is finite because it is limited by the training data. There are only finitely many word forms, suffixes, morphological labels and so on in the training data. The elements of each vector ϕ_t represent activations of features. In the present work all elements are either 0 or 1 mirroring false and true truth values, but other activations in $\mathbb{R}$ can also be used if the truth values of the feature propositions exist on a non-binary scale.

In order to represent sentence positions as vectors, we need an injective index function I which maps features onto consecutive integers starting at 1. For each feature f, $I(f)$ will correspond to one dimension in ϕ_t. In concrete implementations, the index function I can be implemented as a hash function.

Given a sentence x and label sequence y, we can extract the set of features $F_t(x)$ for each position t in x. Let $\phi_t \in \mathbb{R}^N$ be a vector defined by

$$\phi_t(i) = 1, \text{ if } i \leq N \text{ and } I(f) = i \text{ for some } f \in F_t$$

all other entries in ϕ_t are 0.

Given a parameter vector $\omega \in \mathbb{R}^N$, the probability $p(y|x)$ is

$$p(y|x) \propto \prod_{t=1}^{T} \exp(\omega^\top \phi_t)$$

Specifically, the same parameter vector ω is shared by all sentence positions and the probability $p(y|x)$ is a log linear combination of parameter values in ω.

Lafferty et al. (2001)

8.4 Logistic Regression

The simplest Conditional Random Field is the *Logistic Regression Model* (LRM). It is an unstructured probabilistic discriminative model. In this section, I will present a formal treatment of the LRM because it aids in understanding more general CRFs.

Regular linear regression models a *real valued quantity* y based on independent variables x_1, ..., x_n. In contrast the LRM is a regression model which models *the probability* that an observation x belongs to a class y in a finite class set Y. For example, the logistic classifier can be used to model the probability of a tumor belonging to the class MALIGNANT or BENIGN. The probability is based on quantifiable information about the tumor such as its size, shape and the degrees of expression of different genes in the tumor cells. These quantifiable information sources are the *feature templates* of the logistic classifier and combined with class labels they make up the features of the model.

The material at hand deals with linguistic data where most information sources are binary, for example whether a word ends in suffix "-s" and whether a word is preceded by the word "an". In other domains such as medical diagnostics, more general features can be used. These can be real valued numerical measurements such as the diameter of a tumor. This treatment of logistic classifiers will focus on the binary valued case. When using binary features, we can equate the example x with the set of feature templates $F_x \subset F$ that it *activates,* that is *Tumor diameter* $\geq$ *5 cm, Preceded by "an"* and so on. Examples that activate the exactly same feature templates will be indistinguishable from the point of view of the Logistic Regression model.

The logistic classifier associates each combination of a feature template and class with a unique feature and a corresponding real valued parameter. Intuitively, the logistic classifier models correlations of feature templates and classes by changing the parameter values of the associated features. For example, it might associate the feature template *Tumor diameter* $\geq$ *5 cm* more strongly with the class MALIGNANT than the class BENIGN if large tumors are cancerous more often than smaller ones. This could be reflected in the parameter values of the model that correspond to the features $f =$ *Tumor diameter* $\geq$ *5 cm and class is MALIGNANT* and $f' =$ *Tumor diameter* $\geq$ *5 cm and class is BENIGN* so that the parameter value for f is greater than the parameter value for f'. In general parameter values, however, also depend on other features and feature correlations in the model. Therefore we can say that the parameter value of, f will be guaranteed to be greater than the parameter value of f' when f is the sole feature template and the model accurately reflects the original distribution of class labels among examples. In the general case, where there are several feature templates, this might fail to hold.

Formalizing the notation used in Section 8.3, let F be a finite set of feature templates and Y a finite set of classes. Each combination of feature template $f \in F$ and class

$y \in Y$ corresponds to a unique feature. Therefore, the model will have $|F \times Y|$ features in total. Let θ be a real valued parameter vector in $\mathbb{R}^{|F \times Y|}$ and let I be a 1-to-1 index function which maps each combination of feature template and class onto the indexes of θ, that is $1 \leq \mathrm{I}(f, y) \leq |F \times Y|$.

For each example x, let F_x be the set of feature templates that x activates and let $y \in Y$ be a class. Then the feature vector associated with x and y is $\phi(x, y) = \{0, 1\}^{|F \times Y|}$ defined by

$$\phi(x, y)[i] = \begin{cases} 1 & \text{iff } i = \mathrm{I}(f, y) \text{ for some } f \in F_x, \\ 0 & \text{otherwise.} \end{cases}$$

Now the conditional probability $p(y \mid x)$ defining the Logistic classifier is given by Equation (8.1). The equation defines a probability distribution over the set of classes Y because each quantity $p(y \mid x; \theta)$ is a positive real and the quantities sum to 1.

$$p(y \mid x; \theta) = \frac{\exp(\theta^\top \phi(x, y))}{\sum_{z \in Y} \exp(\theta^\top \phi(x, z))} \tag{8.1}$$

Inference Inference for a Logistic Regression Model means finding the probability of each class label $y \in Y$ given example x. The full computation of the probability is, however, not needed when the model is used as a classifier. For simply finding the class y_{max} which maximizes the conditional likelihood in equation 8.2 given fixed parameters θ it is sufficient to maximize the numerator of $p(y \mid x; \theta)$.

$$y_{max} = \arg\max_{y \in Y} p(y \mid x; \theta) = \arg\max_{y \in Y} \frac{\exp(\theta^\top \phi(x, y))}{\mathrm{Z}(x; \theta)} = \arg\max_{y \in Y} \exp(\theta^\top \phi(x, y)) \tag{8.2}$$

To avoid underflow when using finite precision real numbers (such as floating-point numbers), the maximization is usually rephrased as the minimization of a cost-function in Equation 8.3

$$y_{max} = \arg\min_{y \in Y} \theta^\top \phi(x, y) \tag{8.3}$$

From a practical implementation perspective, the minimization in Equation 8.3 boils down to computing one inner product $\theta^\top \phi(x, y)$ for each label $y \in Y$ and finding the minimum. Using a suitable sparse approach each of the inner products can be computed in $\mathrm{O}(|F_x|)$ time, where F_x is the set of feature templates activated by example x. Therefore, the worst-case complexity of classification is dependent on the size of the label set Y and the number of feature templates $f \in F$, that is the complexity is $\mathrm{O}(|Y||F|)$.

8.4.1 Estimation

The Logistic Regression Model is log-linear as demonstrated by Equation 8.4, which represents the model using a cost function $\mathcal{L}$.

$$\mathcal{L}(\theta; \mathcal{D}) = -\log p(y \mid x; \theta) = log(\mathrm{Z}(x; \theta)) - \theta^\top F_y(x) \tag{8.4}$$

Here $Z(x; \theta) = \sum_{z \in Y} \exp(\theta^\top F_z(x))$ is the partition function of the data point (x, y).

Given labeled training data $\mathcal{D} = \{(x_1,\ y_1),\ ...,\ (x_n,\ y_n)\}$, there exist several options for estimating the parameters θ. The most commonly used is the maximum likelihood, or equivalently minimum cost, estimation. The minimum cost estimate for the parameters θ using $\mathcal{D}$ is given by equation (8.5).

$$\theta = \arg\min_{\theta'} \mathcal{L}(\theta; \mathcal{D}) = \arg\min_{\theta'} \sum_{(x,\ y) \in \mathcal{D}} \mathrm{Z}(x; \theta) - \theta'^\top F_y(x) \tag{8.5}$$

The probability $p(y \mid x; \theta)$ has exponential form, which means that the probability is proportional to a product of factors of the form e^{ap}, where a is an activation (0 or 1) and p is a parameter. This has three important consequences:

1. The function $\theta \mapsto p(y \mid x; \theta)$ is smooth.

2. The function $\theta \mapsto p(y \mid x; \theta)$ is convex.

3. There exists a *unique* θ maximizing the likelihood of the training data $\mathcal{D}$.[1]

4. The model $p(y \mid x; \theta)$ is maximally unbiased.

Smoothness follows from the fact that each factor $a \mapsto e^{ap}$ is smooth and products and sums of smooth functions are smooth. Convexity of the likelihood follows by a

[1] Technically this requires that the possible values of θ are limited into a compact subset of the parameter space.

straightforward application of the Hölder inequality . Property 3 is a consequence of properties 1 and 2 and Property 4 follows from the discussion in Section 8.2.

Although the maximization in Equation 8.1 cannot be solved exactly in general, the convexity and smoothness of $p(y \mid x; \theta)$ mean that efficient numerical methods can be used for approximating the maximum.

Gradient based methods such as SGD (leading to online estimation) and L-BFGS (leading to batch estimatiom) require information about the partial derivatives of the cost function. Therefore the partial derivatives $\partial\mathcal{L}(\theta; \mathcal{D})/\partial$i need to be computed. Examining Equation 8.5, we can see that the cost consists of two terms $f(\theta; \mathcal{D}) = \log(\mathrm{Z}(\mathcal{D}; \theta))$ and $g(\theta; \mathcal{D}) = \sum_{(x,y)\in\mathcal{D}} \theta^\top F_y(x)$. The partial derivative of g w.r.t. parameter i can be computed in the following way

$$\frac{\partial g}{\partial i} = \sum_{(x,y)\in\mathcal{D}} F_y(x)[i]$$

This quantity represents the total activation of feature i in the training data and is called *the observed count* of feature i. Using the chain rule of derivatives, we get the partial derivative of f w.r.t. to param i, is

$$\frac{\partial f}{\partial i} = \sum_{(x,y)\in\mathcal{D}} \frac{\sum_{y\in\mathcal{Y}} F_y(x)[i] \exp(\theta^\top F_y(x))}{\mathrm{Z}(x; \theta)} = \sum_{(x,y)\in\mathcal{D}} \sum_{y'\in\mathcal{Y}} F_y(x)[i] p(y'|x; \theta)$$

This is the *expected count* of feature i which is the activation of feature i in the data set $x_1, ..., x_n$ predicted by the model given all possible label assignments.

Using the partial derivatives of the functions f and g, we see that the gradient of the cost function $\mathcal{L}$ is defined by

$$\nabla L[i] = \sum_{(x,y)\in\mathcal{D}} \Big(\sum_{y'\in\mathcal{Y}} F_y(x)[i] p(y'|x; \theta) \Big) - F_y(x)[i] \tag{8.6}$$

Equation 8.6 shows that the cost is zero when the expected and observed counts for each feature agree. The properties for the logistic regression model discussed above guarantee that this there is at most one θ_{ML} like this and, when it exists, θ_{ML} is the maximum likelihood estimate for the parameters.

Regularization methods such as L_1 and L_2 introduced in Chapte 3 can also be applied to the model. This naturally changes the gradient and also the properties of the model. Analysis of the regularized model falls outside of the scope of this thesis.

8.5 The Perceptron Classifier

The perceptron algorithm (Rosenblatt, 1958) is an alternative to the MLE for learning the weights of a discriminative classifier. As seen above, the logistic classifier optimizes the conditional probability of gold standard classes given training inputs. In contrast, the perceptron rule directly optimizes the classification performance of the discriminative classifier.

Intuitively, the multi-class perceptron algorithm works by labeling each training example in order using a current estimate of the parameter vector θ and adjusting the parameter vector whenever training examples are incorectly labeled. Consequently, the perceptron algorithm is an online learning algorithm.

Inference Similarly as in the case of any linear classifier, each example x and class y receives a score $\theta^\top \phi(x, y)$ which is computed using the current estimate of the parameter vector. If the score of the gold standard class y_{gold} is not the highest one, that is there is a class y which scores higher or equally high, then the parameter vector θ is modified in a way which increases the score for the gold standard class and decreases the score for the highest scoring class y.

Estimation The perceptron algorithm is an error-driven online learning algorithm. When a classification error is encountered during estimation, that is $\theta^\top \phi(x, y) > \theta^\top \phi(x, y_{gold})$, the parameter vector θ is adjusted for relevant features. For every feature template f which is activated by the example x, the weights $\theta[I(f, y_{gold})]$ and $\theta[I(f, y)]$ are adjusted. Here $I(f, y_{gold})$ and $I(f, y)$ are the features corresponding to the template f and classes y_{gold} and y repectively. The perceptron rule for weight adjustment is the following:

$$\theta[I(f, y_{gold})] = \theta[I(f, y_{gold})] + 1 \text{ and } \theta[I(f, y)] = \theta[I(f, y)] - 1$$

The perceptron adjustment does not guarantee that example x is correctly classified. However, it does guarantee that the score difference between the gold class and erro-

Algorithm 8.1: The pass of the perceptron algorithm in Python 3.

```python
def infer(x, fextractor, theta, label_set)
    """
        x          - An obesrvation.
        fextractor - A vector valued function.
                     len(fextractor(x,y)) == len(theta).
        theta      - A parameter vector.
        label_set  - Set of potential labels.
    """
    sys_label = None
    max_score = -float('inf')

    for y in label_set:
        score = dot_product(theta, fextractor(x,y))
        if score > max_score:
            max_score = score
            sys_label = label

    assert(sys_label != None)
    return sys_label

def perceptron(data, fextractor, theta, label_set):
    """
        data       - data[i][0] is an observation, data[i][1] a label.
        fextractor - A vector valued function.
                     len(fextractor(x,y)) == len(theta)
        theta      - The parameter vector.
        label_set  - Set of potential labels.

        Run one pass of the perceptron algorithm.
    """

    for x, y_gold in data:
         y_system = infer(x, fextractor, theta, label_set)

         if y_system != y_gold:
             for f in fextractor(x, y_system):
                 theta[f] -= 1
             for f in fextractor(x, y_gold):
                 theta[f] += 1
```

neous class decreases[2]. Given training data consisting of just one example, it is easy to see that the perceptron algorithm will eventually classify the example correctly. If there are more examples, it may however happen that a correct parameter vector is never found.

The perceptron algorithm *converges* when no example in the training data causes a change in the parameter vector θ. Equivalently, the perceptron algorithm correctly classifies every example in the training data. It can be showed that the perceptron algorithm converges whenever there exists a parameter vector that correctly classifies the training data (Freund and Schapire, 1999). Such a data set is called linearly separable. The term originates from a geometrical interpretation of the 2-class perceptron algorithm, where the parameter vector defines a hyper plane in the feature space which divides the apce into two halves. A data set is called separable if there is a hyper space which separates the examples in each of the classes into their own half space.

When a data set is linearly separable, there are typically infinitely many parameter vectors that that classify the data set correctly. The perceptron algorithm will give one of these. Other algorithms exist which attempt to find an optimal parameter vector in some sense (e.g. Support Vector Machines (Cortes and Vapnik, 1995)). These, however, fall beyond the scope of the present work.

Even when the training set is not linearly separable, the perceptron algorithm will have good performance in practice. In the non-separable case, a held out development set is used. Training is stopped when the performance of the classifier on the development set no longer improves.

Voting and Averaging Because the perceptron algorithm makes fixed updates of size 1, the parameter vector tends to change too rapidly at the end of the training procedure. To avoid this, it is customary to use the average of all parameter vectors from the training procedure instead of the final parameter vector. This will give better performance during test time. Parameter averaging is an approximation of so called *voting perceptron*. In voting, each parameter vector is considered a separate classifier and the classification is performed by taking a majority vote of all of the classification results. This is impracti-

[2] There are refiniements of the perceptron algorithm, such as the passive-aggressive learning algorithm, which aim to make fewer updates by updating more aggressively when the difference in scores between the erroneous class and gold class is large (**?**)

cal, because there are thousands of classifiers. Therefore, averaging is used in order to achieve almost the same effect.

8.6 CRF – A Structured Logistic Classifier

This Section presents Linear Chain Conditional Random Fields (CRF)[3]. Just as the HMM is a structured equivalent of the NB classifier, the CRF is the structured equivalent of the LRM. Consequently, many of the algorithms required to build a CRF tagger are similar to the algorithms required to build an HMM tagger. Estimation of model parameters is, however, different because of the discriminative nature of the model.

Another major difference between an HMM classifier and a CRF classifier is that the CRF classifier typically employs a much larger set of features. This increases the size of the model. It also makes the discriminative tagger slow in comparison to the generative tagger. The slowdown is demonstrated by the experiments in Silfverberg et al. (2015). However, the accuracy of the discriminative model is significantly superior to the generative HMM tagger.

Intuitively, the CRF model resembles a sequence of LR classifiers with shared parameters. Given a sentence $x = (x_1,\ ...,\ x_T)$, label sequence $y = (y_1,\ ...,\ y_T)$ and parameters θ for the LR model, a score for the label y_i in position i $s(x, y_{i-2}, y_{i-1}, y_i, i; \theta)$ can be computed. Here the LR model utilizes the input sentence x as well as labels y_{i-2}, y_{i-1} and y_i to extract unstructured and structured features from the sentence and label sequence. The score s takes on a familiar form

$$s(x, y_{i-2}, y_{i-1}, y_i, i; \theta) = \theta^\top F_y(x, i, y_{i-2}, y_{i-1})$$

where F_y is a vector valued feature extraction function. Each feature associated to the label y corresponds to one element of the vector $F_y(x, i, y_{i-2}, y_{i-1})$. As in the case of the LR model, each entry of the vector can be an arbitrary real number but in this thesis they will always be either 0 or 1.

[3] More general CRF models can be formulated but these mostly fall beyond the scope of this thesis. See (Sutton and McCallum, 2012) for further details.

The probability of label sequence $y \in \mathcal{Y}^T$ given a sentence x of length T is

$$p(y|x;\theta) \propto \prod_{i=1}^{T} s(x, y_{i-2}, y_{i-1}, y_i, i; \theta) \tag{8.7}$$

In Equation 8.7, the labels y_{-1} and y_0 are special stop labels which do not belong to the label set $\mathcal{Y}$. The partition function of the sentence x is given by

$$\sum_{y \in \mathcal{Y}^T} \prod_{i=1}^{T} s(x, y_{i-2}, y_{i-1}, y_i, i; \theta) \tag{8.8}$$

It is noteworthy, that the probability in Equation 8.7 is normalized for *the entire sentence*, not in each position. A similar model, where normalization happens in each position is called the Maximum Entropy Markov Model (MEMM). It has been shown to give inferior performance in POS tagging of English (Lafferty et al., 2001)[4].

Inference Tagging of a sentence using the CRF model is very similar to HMM tagging. The major difference is that there are far more features in a CRF model which slows down inference compared to a typical HMM tagger. As in the case of an HMM, the Viterbi algorithm has to be used to find the MAP assignment of the label sequence because of the structured nature of the model. The forward-backward algorithm can be used to compute the marginal probabilities of labels.

Estimation Estimation of the CRF model parameters is more involved than the straightforward counting which is sufficient for HMM training. Estimation is instead very similar to estimation of the LR model parameters. However, the structured nature of the CRF model complicates matters slightly.

Let $\mathcal{D} = \{(x_1, y_1), ..., (x_n, y_n)\}$ be a training data set consisting of n labeled sentences and let y_k^l be the label of the lth word in sentence l. Now the expected count of

[4] The inferior performance of the MEMM has been thought to be a result of the so called label bias problem (Lafferty et al., 2001) although *observation* bias may be more influential in POS tagging (Klein and Manning, 2002).

feature i in position k in sentence l is

$$\frac{\sum_{y \in \mathcal{Y}^T} F_y(x_l, k, y_{k-2}, y_{k-1}, y_k)[i] \exp(\theta^\top F_y(x))}{\mathrm{Z}(x; \theta)}$$

The quantities $\sum_{y \in \mathcal{Y}^T} F_y(x_l, k, y_{k-2}, y_{k-1}, y_k)[i]$ and $\mathrm{Z}(x; \theta)$ have to be computed using the forward backward algorithm because the number of computations is too large otherwise. The need of the forward backward algorithm is the most important difference between LR and CRF estimation. Commonly, the SGD algorithm and L-BFGS are used for the optimization of θ (Vishwanathan et al., 2006). As in the case of LR model, the CRF model can also be regularizaed using for example L1 or L2 regularization **?**.

Held out development data is commonly used to set the number of training epochs, model order and regularization hyper-parameters.

Approximative estimation In addition to the ML estimator, a variety of estimators are available for CRF models. These alternative estimators are mainly used due to the fact that ML estimation can be resouce intesive. The most commonly substitute for the ML estimator is the averaged perceptron estimator described in Section 8.7 and taggers trained using the averaged perceptron tagger are often called perceptron taggers (see for example Collins (2002)). Other examples mentioned by Sutton and McCallum (2012) are the pseudlikelihood Besag (1975) and piecewise pseudolikelihood estimators Sutton and McCallum (2007). Ruokolainen et al. (2014) investigate a pseudolikelihood inspired variant of the perceptron estimator.

Given a training example $(x, y) \in \mathcal{D}$ where $x = x_1, ..., x_T$ and $y = y_1, ..., y_T$, the pseudolikelihood of y given x is given by equation **??**

$$\prod_{t=1}^{T} p(y_t | x, i, y_{\neg t}; \theta) \tag{8.9}$$

where $y_{\neg t}$ refers to all the labels in y except label y_t. This is substantially more efficient than ML estimation because the labels can be maximized separately in each position. However, it may result in poor accuracy (**?**).

Piecewise-pseudo.

Pseudoperceptron.

- Hierarchical CRF Müller et al. (2013), Weiss and Taskar (2010) and Charniak and Johnson (2005).

8.7 The Perceptron Tagger

Whereas the perceptron classifier is a discriminative classifier similar to the LR model except that it uses perceptron estimation, a perceptron tagger (Collins, 2002) is a sequence labeling model similar to the CRF except that it uses perceptron estimation.

The model is formulated very similarly as the CRF model. It also uses a real valued parameter vector θ and the score of a label sequence y given sentence x is defined as

$$s(y|x;\theta) \propto \prod_{i=1}^{T} s(x, y_{i-2}, y_{i-1}, y_i, i; \theta) \qquad (8.10)$$

In Equation 8.10, $s(x, y_{i-2}, y_{i-1}, y_i, i; \theta) = \theta^\top F_y(x, i, y_{i-2}, y_{i-1})$, where F_y is the vector valued feature extraction function for label y.

Inference The Perceptron tagger uses the Viterbi algorithm for exact inference. Beam search can be used for faster approximate inference together with a label dictionary (Silfverberg et al., 2015).

Estimation Whereas, CRF estimation requires the forward-backward algorithm, perceptron estimation only requires the Viterbi algorithm. Exactly as in the case of the regular perceptron algorithm, each training example (i.e. sentence) is labeled and unstructured and structured features are updated accordingly. The number of training epochs is determined using held-out data. Parameter averaging is useful for improving the accuracy of the percetron tagger.

Beam search for estimation A high model order and large label set can result in a prohibitive runtime for the Viterbi algorithm which has a complexity that is dependent on the nth power of the label set size for an order n model. This problem can be avoided using beam search during estimation instead of the Viterbi algorithm. Because beam search is an approximative inference algorithm, it may however not give the correct

MAP assignment for a sentence. It may happen that beam search returns a label sequence y_{sys} for training sentence x whose score w.r.t. to model parameters is lower than the score of the gold label sequence y_{gold}. This lead to perceptron updates which are not necessary because the model already correctly labels sentence x if only exact inference is used. [WHY IS THIS SO BAD?]

Violation fixing To avoid superfluous perceptron updates, a technique called violation fixing can be used (Huang et al., 2012) (this is an extension of the early update technique suggested for incremental parsing in Collins and Roark (2004)). Huang et al. (2012) suggest several related violation fixing methods. The essence of the methods is to compute the score each prefix y^i_{sys} of the label sequence y_{sys} returned by beam search and y^i_{gold} the gold standard label sequence y_{gold}. One i is then selected so that the score of $y^i_{sys} > y^i_{gold}$ (if such an i exists). Updates are then performed for y^i_{sys} and y^i_{gold}. The choice of i depends on the violation fixing method. For example, the last i which exhibits a violation can be used.

In the experiments presented in Silfverberg et al. (2015), violation fixing did not result in consistent statistically significant improvements. It may be that it is more influential in parsing than POS tagging or morphological tagging.

Label guessing As in the case of the CRF tagger, a cascaded model can be used to shorten training time (Silfverberg et al., 2015). Instead of a cascade of discriminative classifiers, used in (Müller et al., 2013), Silfverberg et al. (2015) use a combination of a generative label guesser of the type presented in Section **??**. The number of guesses can be determined either beased on a probability mass threshold ot using a fixed number of guesses per word. Setting the threshold too low will result in a training task that is to easy. Consequently the model will overfit the training data. Higher thresholds will approximate the original training task more cloasely but will also lead to longer training times.

Like beam search, label guessing modifies the training task. It does, however, not require violation fixing because it does not influence the relative difference in scores of label sequences as long as the gold standard label sequence is never pruned out. Therefore, the gold standard label should always be added to the set of guesses given by the label guesser.

The combination of beam search and model cascading results in a fast training with a tolerable decrease in tagging accuracy even large label sets and for second order models as shown in Section **??**.

8.8 Enriching the Structured Model

As seen above, the feature set of a discriminative classifier with $|\mathcal{Y}|$ classes and $|F|$ feature templates has on the order of $|\mathcal{Y} \times F| + |\mathcal{Y}|^n$ features and parameters, where n is the model order. When $\mathcal{Y}$ is large, this gives rise to data sparsity because each feature template is seen with quite few labels $y \in \mathcal{Y}$. Large morphological label sets, however, are typically internally structured. For example, the noun label "noun+sg+nom" and adjective label "adj+sg+nom" share the same number "sg"and case "nom" .

One way to combat data sparsity, is to add a number of new feature extraction functions in addition to the feature extraction functions F_y for each label $y \in \mathcal{Y}$. For example, there can be a function $F_{+\text{sg}}$ which extracts features for each label that has singular number. We call the features speicifed by these astracted feature extraction functions, *sub-label features*.

For the label "noun+sg+nom", the activation which is extracted is by a standard CRF is

$$F_{\text{noun+sg+nom}}(x, i, y_{i-2}, y_{i-1})$$

When using sub-label features, we instead extract the activation

$$\sum_{y \in \{\text{noun+sg+nom, noun, sg, nom}\}} F_y(x, i, y_{i-2}, y_{i-1})$$

The features extraction function will also utilize the internal structure of the argument labels y_{i-1} and y_{i-2}. Examples of sub-label features include "A word dog is singular" and "a singular word follows another singular word". It is, however, best to do this in a restricted manner. The experiments in Chapter 10 demonstrate that in a second order CRF tagger, structured sub-label features of order one result in consistent improvement in accuracy but higher order sub-label features give no improvement.

Sub-label features can help combat data sparsity. Additionally, they are useful for modeling linguistic phenomena such as congruence. For example congruence in case and number can be modeled separately and independently from other inflectional and derivational information.

Restricted form of sub-label features have been utilized by for example Müller et al. (2013) and Smith et al. (2005).

Müller et al. (2013) uses sub-labels exclusively for unstructured features. While sub-labels seem to be most beneficial in combination with unstructured features in a morphological tagging setting where a morphological analyzer is not utilized, this is not the case in a morphological disambiguation setting as the experiemtns in Chapter 10 indicate. When using a morphological analyzer, sub-labels result in the greates improvement when combined with structured features.

? utilize structured sub-labels, however, only in a restricted way. They build separate structured chains modeling the sequence of main POS classes, cases, numbers, genders and lemmas of neighoring words. Their system cannot model cross category dependencies such as dependence between a verb and the case of its object.

However, **?** extends this approach to fully take into account structured features. As the experiments in Chapter 10, structured sub-labels have a substantial impact on tagging accuracy both in morphological tagging setting without a morphological analyzer and in morphological disambiguation setting. For Finnish, the impact of sub-label features on tagging accuracy seems to be on par with the impact of higher model order.

Another approach which utilizes sub-labels is the tiered tagging approach (Tufis, 1999, Ceausu, 2006). In tiered tagging, the label set is first projected onto a smaller set of coarse labels. A tagger is first used for labeling text with coarse labels. Subsequently another tagger is used to convert coarse labels into full morphological labels.

8.9 Model Pruning

Discriminative models for morhpological tagging can often grow quite large in terms of parameter count. For example, the model learned from the FTB corpus by the FinnPos tagger has more than 4 million parameters.

A large number of parameters is problematic because it causes over-fitting of the model to the training data. Moreover, large models can be problematic when memory foot-print is an issue: e.g. on mobile devices.

Different methods have been propsed for pruning of perceptron models. Goldberg and Elhadad (2011) prune the models based on update count. Parameters that receive less than a fixed amount of updates during training will be omitted from the final model. Another approach is to pruning by feature count. For example Hulden et al. (2013) prune out features for words occurring less than a fixed amount of times in the training data. More generally, fetures that are activate less than a fixed amount of times may be pruned out.

Some regularization techiques can also be used to learn sparse perceptron models. L1-regularization yields sparse models similarly as for logistic regression. Zhang et al. (2014) investigate L1-regularization for structured perceptron. They gain accuracy but do not report results on model size.

I have explored two different pruning strategies

- Pruning based on update counts (Goldberg and Elhadad, 2011).
- Pruning based on parameter value.

Goldberg and Elhadad (2011) show that update count based pruning beats feature count based pruning in dependency parsing and POS tagging. Therefore, I decided not to compare those approaches. Instead I compare update count based pruning to pruning based on final parameter value.

Update Count Pruning When using this strategy, each parameter which did not receive at least n updates during training, is omitted from the final model. Here n is a hyper-parameter which is set using held-out data. In practice, this pruning strategy requires that one maintains a update count vector where each element corresponds to one model parameter. Whenever a parameter is updated during training, the update count is increased.

As stated before, the perceptron algorithm labels a training example and then performs updates on the model parameters. When labeling during training, only those parameters that already received at least n updates are used. However, updates are performed on all parameters. When the update count of a parameter exceeds n, the parameter value will therefore already be of similar magnitude with the rest of the parameter values in the model.

(Goldberg and Elhadad, 2011) do not explore early stopping. Preliminary experiments showed that it is best to first set the number of training passes without parameter pruning and then set the pruning threshold n separately using develpment data. If the number of passes and the update count threshold are set at the same time, the model parameters converge quite slowly resulting in many training epochs and consequently many parameter updates. This has an adverse effect on the number of parameters that can be pruned from the final model.

Value Based Pruning A very simple strategy for parameter pruning is to prune based on the parameter value. The model is trained in the regular manner. After training, all parameters whose absolute value does not exceed a threshold κ are omitted from the model. Remaining parameter values remain unchanged. The hyper-parameter κ is determined using a development set.

In the experiment chapter, I show that value based pruning outperforms update count based pruning on the data-sets that I have used. In some settings, the difference is substantial.

Chapter 9

Lemmatization

9.1 Introduction

In this section, I will present the task of data driven lemmatization. I will examine different approaches to data driven lemmatization and present the lemmatizer used in the FinnPos toolkit Silfverberg et al. (2015).

A lemmatizer is a system which takes text as input and returns the lemma of each word in the text. Lexical resources such as dictionaries or morphological analyzers are very helpful for the lemmatization task. In fact, lemmatization is often seen as one of the sub tasks of morphological analysis. Another task which is closely related to lemmatization is *morphological paradigm generation* (**??**). Here the task is to generate all, or a selection, of the inflectional forms of a word form. Therefore, lemmatization is a sub-task of morphological paradigm generation.

I will treat lemmatization as a follow up task to morphological labeling. That is to say, the lemmatizer has access to the morphological labels in the text. The morphological label is extremely important for lemmatization. For example, in Finnish a word ending “-ssa” could be a noun or verb form. If it is a noun form it could be either a nominative or an inessive. All of these analyses produce different lemmas.

A morphological analyzer can be used for lemmatization of the words in a text where words have morphological labels. First, analyze each word using the morphological analyzer. This produces a set of morphological labels and associated lemmas. Then simply pick the lemma which is associated with the correct morphological label.

Problems arise when word forms are not recognized by the morphological analyzer. There are several approaches to solving these problems. One approach is to utilize the morphological analyzer (for example a finite-state analyzer) to produce a guess for a lemma even though the word form is not recognized. The guess is based on orthographically similar words which are recognized and lemmatized by the morphological analyzer. As an example of this approach, see Lindén (2009).

The main advantage of basing a data driven lemmatizer on an existing morphological analyzer is that large coverage morphological analyzers have to model most if not all morphotactical and the morphophonological phenomena that occur in a language. Therefore, it is likely that the analyzer models the inflectional paradigm of most word forms even though it would not have seen the specific word forms themselves.

Most existing work on analyzer based lemmatizers has used rather simple statistical models. For example, Lindén (2009) uses plain suffix frequencies.

In contrast to lemmatizers based on morphological analyzers, classifier based lemmatizers Grzegorz Chrupala and van Genabith (2008) are learned from data without an existing model. The general approach is based on the observation that word forms can be transformed into lemmas using an *edit script*. For example, the English noun "dogs" has the lemma "dog". To convert "dogs" into "dog" one needs to remove the suffix "s". This is a very simple example of an edit script which I will denote $[-s \rightarrow \varepsilon]$. All edit scripts cannot be applied on all word forms. For example the edit script which removes a final "s" cannot be applied on past English participle form ending "ed".

Classifier based lemmatizers frame the lemmatization task as a as classification task. The lemmatizer will use edit scripts as labels. Subsequently to labeling a word form with an edit script class, the lemmatizer will apply the edit script thus giving a lemma.

The advantage with using a classifier based lemmatizer is that the classifier can use a feature based discriminative model. In contrast to analyzer based lemmatizers, classifier based lemmatizers can therefore use richer information sources such as prefixes and word shapes expressed as regular expressions [1] – not exclusively information about word suffixes.

Although it would be very interesting to combine these approaches, it falls beyond the scope of this thesis. Therefore, I have used classifier based lemmatizers. I decided

[1] An example of a word shape expressions in POSIX syntax is `[A-Z][a-z]+` which matches capitalized English words.

upon classifier based lemmatizers partly because the work of Lindén (2009) already investigates analyzer based lemmatization for Finnish. When performing morphological disambiguation based on the output of a morphological analyzer, the current system does use the morphological analyzer for lemmatization of all word forms which it recognizes. For all other words, the data driven lemmatizer is used.

In the field of morphological paradigm generation, there exists work which in a sense combines the analyzer and classifier based approaches Hulden et al. (2014). However, the starting point is not a morphological analyzer. Instead a list of morphological paradigms is used. It would be interesting to explore this but it falls beyond the scope of the current work.

Joint tagging and lemmatization has also been explored and yields some improvements Müller et al. (2015).

9.2 Framing Lemmatization As Classification

A classification based lemmatizer reads in an input form, identifies the set of edit scripts that can be applied to the input form and scores the candidate scripts using the input form, its morphological label and a feature based classifier. Finally, the winning edit script is applied on the input form and the lemma is recovered.

Extracting Edit Scripts Given a word form such as “dogs” and its lemma “dog”, several edit scripts can be extracted. For example, $[-s \to \varepsilon]$, $[-gs \to -g]$, $[-ogs \to -og]$. The current system extracts the shortest script which adequately recovers the lemma.

The FinnPos system only extracts edit scripts which delete a suffix and appends another suffix such as the script $[-s \to \varepsilon]$. This is mostly sufficient for Finnish where only numerals exhibit inflection at the end of words . Naturally, this would not be sufficient in general. More general edit scripts can be used, for example Grzegorz Chrupala and van Genabith (2008).

For morphologically complex languages, there a large number of edit scripts may be extracted from training data. For example, Finnpos system extracts 4835 different edit scripts for the 145953 tokens in the training and development data of FinnTreeBank. Therefore, many of the classes occur few times in the training data. This leads to data sparsity. However, increasing the amount of the training data would probably alleviate the problem significantly because the inventory of inflectional paradigms is finite (maybe).

Features for Lemmatization For a word $w = (w_1...w_n)$ and a morphological label y, the lemmatizer in the FinnPos system currently uses the following feature templates:

- The word form w.
- The morphological label y.
- Suffixes (w_n), $(w_{n-1}w_n)$, ... Up to length 10.
- Prefixes (w_1), (w_1w_2), ... Up to length 10.
- Infixes $(w_{n-2}w_{n-1})$, $(w_{n-3}w_{n-2})$ and $(w_{n-4}w_{n-3})$.

For each feature template f (except the morphological label template y), FinnPos additionally uses a combination template (f, y) which captures correlations between morphological labels and the orthographical representation of the word form.

The infix templates are useful because they model the environment of where an inflectional suffix like “-s” is removed and a lemma suffix is added. They aim at preventing phonotactically impossible combinations.

Estimating the model The lemmatizer can be implemented using some disrciminative model. For example as an averaged perceptron classifier or a logistic classifier. In the FinnPos system, the lemmatizer is an averaged perceptron classifier.

The estimation of the lemmatizer model differs slightly from standard averaged perceptron estimation presented in Chapter 3. Even though the number of edit scripts can be very large (in the order of thousands), the subset of edit scripts applicable for any given word form is much smaller. Moreover, it is always known in advance because it is completely determined by the suffixes of the word form. Therefore, the classifier is only trained to disambiguate between the possible edit scripts associated to each word form. This speeds up estimation considerably.

Inference In the FinnPos system, words which were seen during training time, are lemmatized based on a lemma dictionary which associates each pair of word form and morphological label with a lemma. For words which were not seen during training or which received a label not seen during training, are lemmatized using the data driven lemmatizer. Additionally, a morphological analyzer can be used to assign lemmas to those words which it recognizes.

For word forms which cannot be lemmatized using the lemma dictionary or morphological analyzer, the data driven lemmatizer is used. For each word form, the set of applicable edit scripts is formed and scored. The highest scoring edit script is subsequently applied to the word form to produce a lemma.

Chapter 10

Experiments

FinnPos (Silfverberg et al., 2015). is a discriminative morphological tagger based on the CRF model and averaged perceptron estimation. It is especially geared toward morphologically rich languages and incorporates features which make it especially suited for use with these languages

- Structured and unstructured sub label dependencies (see Section **??**) are used to improve accuracy with large structured label sets.
- Adaptive beam search is used to speed up estimation (see Section **??**).
- A generative label guesser is used to speed up estimation (see Section **??**).
- A morphological analyzer is used during tagging and training (see Section **??**).

This section is intended to augment the treatment Silfverberg et al. (2015). It presents several experiments on morphological tagging using the FinnPos tagger toolkit which were omitted from the paper in favor of clarity.

Because FinnPos incorporates a variety of optimizations, governed by different hyper-parameters, both to accuracy and speed, it is impossible to conduct exhaustive experiments (a grid search would simply include too many experiments). Instead I have chosen the settings presented in Silfverberg et al. (2015) as vantage point and examined the impact of changing one hyper-parameter at a time. These settings were chosen because they give state-of-the-art accuracy as presented in the paper.

The basic setting for all experiments is

- A second order model.
- First order sub-label dependencies.
- 99.9% mass for the generative label guesser.
- 99.9% mass for the adaptive beam search.

A morphological analyzer is only used when separately indicated. A label dictionary is used in all experiments both to speed up decoding and improve accuracy. The label dictionary is also used when the morphological analyzer is used. The reason for this is that a liberal compounding or derivation mechanism (such as the one implemented in the OMorFi morphological analyzer) can result in improbable analyses. The analyses that have been attested in the training corpus should be preferred when possible.

The features for the tagger and lemmatizer follow Silfverberg et al. (2015).

Let $x = (x_1 ... x_T)$ be a sentence, $y = (y_1 ... y_T)$ a label sequence and t and index. Then the unstructured features templates for the morphological tagger are the familiar Ratnaparkhi features (Ratnaparkhi, 1998) augmented with a few additional features. For all words, FinnPos uses the following feature templates

- The word form x_t and the lower cased version of x_t.
- The length $|x_t|$ of word form x_t.
- Word form x_{t-2}, when $t > 2$.
- Word form x_{t-1}, when $t > 1$.
- Word form x_{t+2}, when $t + 1 < T$.
- Word form x_{t+1}, when $t > T$.

For rare words[1], it additionally extracts the following features

- `DIGIT` when x_t contains a digit.
- `UC` when x_t contains an upper case letter.

[1] The list of common words is user defined but in these experiments I have defined common words to be words having frequency 10 or higher in the training corpus

- Prefixes and suffixes of x_t up to length 10.

When a morphological analyzer is used, each morphological label given to word x_t is also used as a feature template.

Let $x = x_1 ... x_n$ be a word form of length n and y its label. Then the features for the lemmatizer are

- The lower cased variant of x.
- Prefixes of x up to length 5 and suffixes up to length 7.
- The infixes $x_{n-2}x_{n-1}$, $x_{n-3}x_{n-2}$ and $x_{n-4}x_{n-3}$.
- The label y and its main part-of-speech.
- `DIGIT` when x contains a digit.
- `UC` when x contains an upper case letter.

Additionally a combination of each feature template and the morphological label y is used as a feature template. Naturally, the combination of y with itself is omitted.

Some baseline runs are impossible to run: FinnPos uses a second order model. With a label set size of 1,000, trellises used during inference become very large and inference is prohibitively slow. Therefore, it was not feasible to run experiments without label pruning during training.

The results of the experiments presented here differ minutely from the results presented in Silfverberg et al. (2015) due to added features (lower cased word form and word length) and some bug fixes related to the lemmatizer.

10.1 Using a Cascaded Model

This subsection presents experiments on using different settings for the generative label guesser included as a pre-pruning step during training as explained in Section **??**.

For both Turku Treebank and FinnTreeBank, I compare a fixed amount of label guesses to choosing a varying amount of guesses per word using a generative label guesser. It was not possible to run experiments without any form of label pruning during

Adaptive Guess Count			
Guess Mass	Tagging Accuracy (%)	Training time	Dec. Speed (KTok/s)
0.9	93.21 (OOV: 77.68)	3 min, 3 epochs	7
0.99	93.11 (OOV: 77.14)	3 min, 3 epochs	7
0.999	93.23 (OOV: 78.49)	4 min, 4 epochs	8
0.9999	93.41 (OOV: 78.55)	2 min, 2 epochs	7

Fixed Guess Count			
Guess Count	Tagging Accuracy (%)	Training time (min)	Dec. Speed (KTok/s)
1	91.48 (OOV: 69.81)	1 min, 3 epochs	8
10	93.23 (OOV: 77.56)	2 min, 2 epochs	7
20	93.18 (OOV: 77.89)	3 min, 3 epochs	7
30	93.22 (OOV: 77.62)	4 min, 3 epochs	6
40	93.43 (OOV: 78.49)	4 min, 2 epochs	5

Table 10.1: Different label guesser settings for FinnTreeBank

training because of prohibitive runtime and memory requirements. The most important point of these experiments is that using some kind of label guesser during training is almost necessary if one wants to train a second order model for label sets of several hundreds or thousands of labels.

In general, using a larger amount of label guesses improves accuracy. For both FTB and TDT, the accuracy levels off already at 40 guesses per word.

For FTB, the mass 0.9999 yields approximately the same accuracy than as 40 guesses. For TDT, however, 40 label guesses results in 0.2%-points better accuracy than using the mass 0.9999.

The training time per epoch is clearly related to the amount of label guesses, however, the number of epochs seems to fluctuate somewhat from two to four for FTB and from five to eleven for TDT. Therefore, it is difficult to see a clear trend.

The amount of label guesses influences decoding speed to some degree because the same setting is used for the label guesser during decoding. Because the label guesser is only used for OOV words, the exact setting of the guesser does however only have a moderate impact.

Adaptive Guess Count

Guess Mass	Tagging Accuracy (%)	Training time (min)	Dec. Speed (KTok/s)
0.9	92.69 (OOV: 74.10)	8 min, 8 epochs	5
0.99	92.66 (OOV: 74.13)	9 min, 8 epochs	5
0.999	92.76 (OOV: 74.65)	8 min, 7 epochs	5
0.9999	92.75 (OOV: 74.30)	6 min, 5 epochs	5

Fixed Guess Count

Guess Count	Tagging Accuracy (%)	Training time (min)	Dec. Speed (KTok/s)
1	89.85 (OOV: 61.33)	2 min, 5 epochs	6
10	92.35 (OOV: 72.55)	5 min, 7 epochs	6
20	92.61 (OOV: 73.88)	5 min, 4 epochs	6
30	92.81 (OOV: 74.87)	12 min, 9 epochs	5
40	92.91 (OOV: 75.35)	14 min, 9 epochs	5

Table 10.2: Different label guesser settings for Turku Dependency Treebank

10.2 Beam Search

This subsection presents experiments using different beam settings during training and decoding.

I compare fixed beam width to an adaptive beam presented in Section **??**. Additionally, I include training and decoding results when no beam is used.

It is difficult to see a clear relation between the beam width and tagging accuracy. The fixed beam of width one is clearly the worst for both TDT and FTB. However, all higher beams seem to give results in the same range. Moreover, increasing the beam from 10 to 20 results in a 0.1%-point drop in accuracy for FTB. Additionally, the system without any beam search performs worse than systems with adaptive or fixed beam width for TDT.

A small beam width results in a faster training time than a larger beam. When using an infinite beam, the training time for TDT is surprisingly large. The reason for this is that there are sequences of words in the training and development data which receive a large number of label guesses. When no beam is used, this results in very long tagging

Adaptive Beam

Beam Width	Tagging Accuracy (%)	Training time (min)	Dec. Speed (KTok/s)
0.9	93.08 (OOV: 76.99)	3 min, 3 epochs	6
0.99	93.14 (OOV: 77.44)	3 min, 3 epochs	8
0.999	93.28 (OOV: 80.49)	2 min, 2 epochs	8

Fixed Beam

Beam Width	Tagging Accuracy (%)	Training time	Dec. Speed (KTok/s)
1	92.32 (OOV: 75.07)	2 min, 2 epochs	7
10	93.33 (OOV: 78.28)	2 min, 2 epochs	7
20	93.11 (OOV: 77.29)	4 min, 3 epochs	7
∞	93.31 (OOV: 78.19)	20 min, 2 epochs	6

Table 10.3: Different beam settings for FinnTreeBank.

Adaptive Beam

Beam Width	Tagging Accuracy (%)	Training time (min)	Dec. Speed (KTok/s)
0.9	92.55 (OOV: 73.73)	5 min, 5 epochs	6
0.99	92.87 (OOV: 74.88)	7 min, 7 epochs	6
0.999	92.76 (OOV: 74.65)	8 min, 7 epochs	6

Fixed Beam

Beam Width	Tagging Accuracy (%)	Training time (min)	Dec. Speed (KTok/s)
1	91.80 (OOV: 71.26)	6 min, 9 epochs	6
10	92.83 (OOV: 74.85)	7 min, 6 epochs	6
20	92.60 (OOV: 73.98)	10 min, 7 epochs	6
∞	91.58?? (OOV: 69.46)	199 min, 8 epochs	6

Table 10.4: Different beam settings for Turku Dependency Treebank.

times for the sequences because a second order model is used.

Contrary to what the literature indicates (Huang et al., 2012, Collins and Roark, 2004), violation fixing gave no significant improvements in accuracy in preliminary experiments. As it only slows down training, it was not included in FinnPos.

10.3 Model Order

Without a Morphological Analyzer

Model Order	Tagging Accuracy (%)	Training time	Dec. Speed (KTok/s)
0	91.91	3 min, 3 epochs	8
1	92.49	3 min, 4 epochs	8
2	92.49	4 min, 5 epochs	7

Using a Morphological Analyzer

Model Order	Tagging Accuracy (%)	Training time	Dec. Speed (KTok/s)
0	95.35	5 min, 9 epochs	25
1	95.98	5 min, 10 epochs	23
2	95.96	5 min, 8 epochs	24

Table 10.5: Different Model Orders for FinnTreeBank

Model Order	Tagging Accuracy (%)	Training time (min)	Dec. Speed (KTok/s)
0	91.15	6 min, 4 epochs	6
1	91.17	8 min, 8 epochs	5
2	91.83	5 min, 3 epochs	5

Using a Morphological Analyzer

0	95.53	5 min, 4 epochs	21
1	96.05	5 min, 7 epochs	22
2	96.13	5 min, 5 epochs	20

Table 10.6: Different Model Orders for Turku Dependency Treebank

In this section I examine the impact of model order on tagging accuracy, training time and decoding time. The experiments in this section do not use sublabel features in order to clearly reveal the impact of model the order in isolation of other factors.

The accuracy on both FTB and TDT increases when going from order zero to a first order model. Further increasing the model order only gives an improvement for TDT.

The increase in accuracy when going from a unstructured (order zero) model to a second order model is approximately 0.5%-points for both FTB and TDT. This applies both when using a morphologial analyzer and when not using it. It is noteworthy that the increas in accuracy resulting from the morphological analyzer is substantially larger for both data sets.

10.4 Utilizing Sub-Label Dependencies

In this section I examine the impact of sub-label order order on tagging accuracy, training time and decoding time both using a morphological analyzer and without a morphological analyzer. The results in this section differ slightly from Silfverberg et al. (2015) because of minor bug fixes and improvements to the feature set of the tagger.

The total improvement in accuracy stemming from sub-label features is approximately 0.8%-points for both FTB and TDT when not using a morphological analyzer and around 0.3%-points when using a morphological analyzer. Contrasting these results with the previous section examining model order, we can see that the added improvement from sub-label features is larger than improvement given by incresing model from order 0 to 2 when not using the morphological analyzer. Moreover, the imporovement is approximately approximately the same as going from model order 0 to 1 when using a morphological analyzer.

Only for FTB do second order sub-label features give added accuracy compared to first order features and only when using the morphological analyzer. In other cases, second order sub-labels perform worse than first order sub-labels.

Training time increases and decoding speed decreases with incresing sub-label order. However, sub-labels seems to decrease the amount of training epochs needed to converge to the final model parameters.

For both FTB and TDT, unstructured sub-label features are more influential for accuracy than structured sub-label features when the morphological analyzer is not used. Then it is used, structured sub-labels, converesly, give a larger improvement.

Without a Morphological Analyzer

Sub-Label Order	Tagging Accuracy (%)	Training time	Dec. Speed (KTok/s)
None	92.49 (OOV: 74.68)	3 min, 5 epochs	6
0	93.05 (OOV: 77.74)	1 min, 2 epochs	5
1	93.29 (OOV: 78.40)	1 min, 2 epochs	4
2	92.68 (OOV: 75.22)	5 min, 4 epochs	6

Using a Morphological Analyzer

Sub-Label Order	Tagging Accuracy (%)	Training time	Dec. Speed (KTok/s)
None	95.98 (OOV: 91.41)	3 min, 8 epochs	25
0	96.08 (OOV: 91.98)	1 min, 3 epochs	22
1	96.24 (OOV: 92.28)	1 min, 2 epochs	21
2	96.31 (OOV: 92.58)	4 min, 3 epochs	19

Table 10.7: Different Sub-Label Orders for FinnTreeBank

Without a Morphological Analyzer

Sub-Label Order	Tagging Accuracy (%)	Training time	Dec. Speed (KTok/s)
None	91.89 (OOV: 70.63)	2 min, 3 epochs	5
0	92.59 (OOV: 73.98)	2 min, 4 epochs	5
1	92.69 (OOV: 74.35)	5 min, 7 epochs	3
2	92.31 (OOV: 72.31)	13 min, 8 epochs	5

Using a Morphological Analyzer

Sub-Label Order	Tagging Accuracy (%)	Training time	Dec. Speed (KTok/s)
None	96.12 (OOV: 91.12)	3 min, 5 epochs	19
0	96.17 (OOV: 91.39)	2 min, 5 epochs	18
1	96.39 (OOV: 91.84)	3 min, 5 epochs	16
2	96.29 (OOV: 91.69)	12 min, 8 epochs	16

Table 10.8: Different Sub-Label Orders for Turku Dependency Treebank

10.5 Pruning the Model

In this section, I examine two model pruning strategies: pruning by update count and pruning by parameter mass. The strategies are presented in 8.9.

The value for the pruning parameter was set using development data. The range of parameter values was chosen so as to show the difference in pruning efficiency. The specific parameter values are not very important. The important thing is to show the relation of model size and accuracy. In these experiments, pruning has not been applied to the lemmatizer although that would be possible.

Clearly, mass based pruning is more effective than update count based pruning. For FTB, without a morphological analyzer, the full accuracy of 93.2% can be maintained even when pruning out 81% of model parameters. When using update count as pruning criterion, full accuracy cannot be maintained when pruning out more than 38% of model parameters. For TDT, the corresponding figures are 55% for mass based pruning and 23% for update based pruning.

When using a morphological analyzer, even further feature pruning is possible. For FTB, 84% of model parameters can be pruned while maintining full accuracy when using mass based pruning. When using update count based pruning, however, no parameters can be pruned without losing accuracy. For TDT, update count based pruning can prune out 72% of the features when using a morphological analyzer but mass based pruning can prune out even more – 81%.

	Update Count Threshold				
MA	None	< 2	< 3	< 4	< 5
no	93.2%, 4.8M	93.2%, 3.9M	93.2%, 3.6M	93.2%, 3.0M	93.1%, 1.0M
yes	96.3%, 4.3M	96.2%, 3.9M	96.2%, 3.7M	96.2%, 3.3M	96.1%, 1.0M

	Parameter Mass Threshold				
MA	None	< 2.0	< 2.5	< 3	< 3.5
no	93.2%, 4.8M	93.3%, 1.8M	93.2%, 1.4M	93.2%, 1.2M	93.2%, 0.9M
yes	96.3%, 4.3M	96.3%, 0.9M	96.3%, 0.7M	96.2%, 0.3M	96.1%, 0.1M

Table 10.9: Result of applying different pruning strategies on FinnTreeBank models.

10.6 Lemmatizer

Remember to check how often the correct edit script exists.

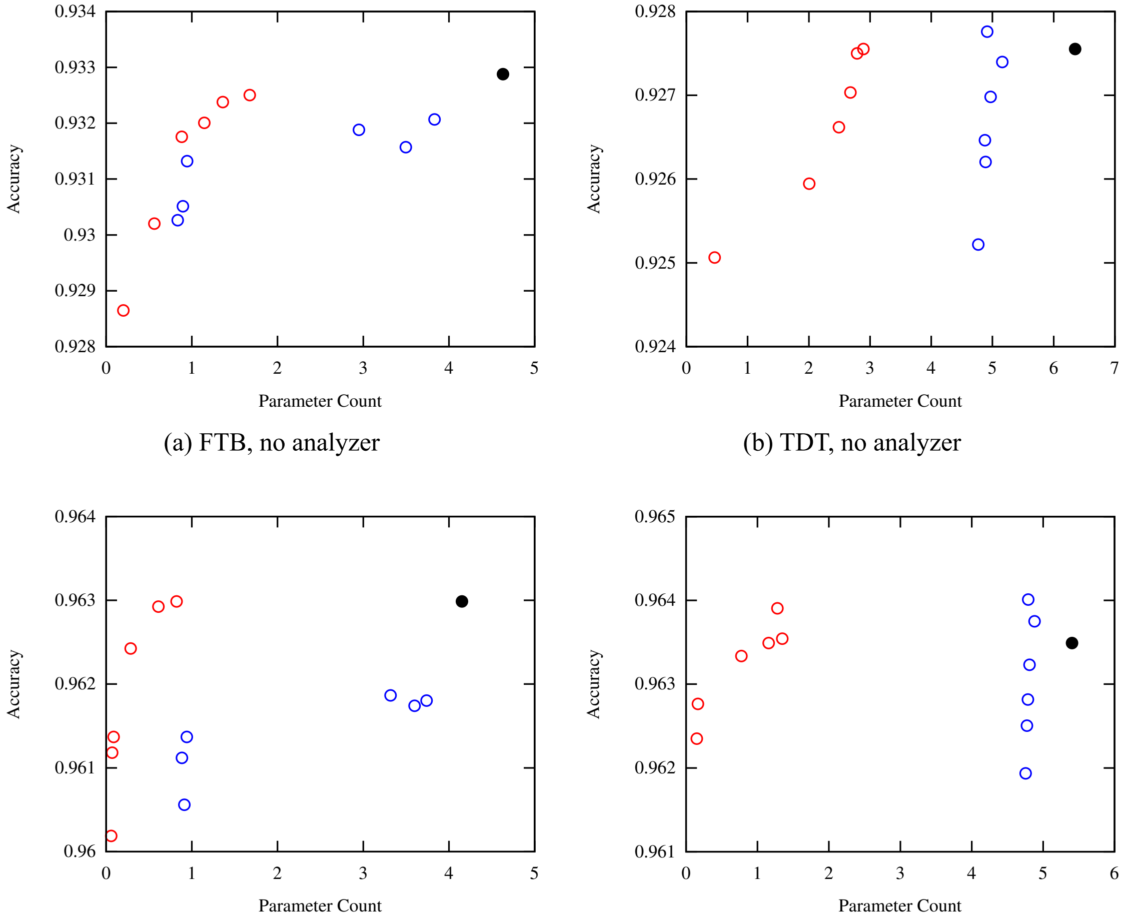
Table 10.10: Result of applying different pruning strategies on Turku Dependency Treebank models.

	Update Count Threshold				
MA	None	< 2	< 3	< 4	< 5
no	92.8%, 6.4M	92.7%, 5.2M	92.7%, 5.0M	92.8%, 4.9M	92.6%, 4.9M
yes	96.3%, 5.5M	96.4%, 5.0M	96.3%, 4.9M	96.4%, 4.9M	96.3%, 4.9M

	Parameter Mass Threshold				
MA	None	< 4.0	< 5.0	< 6.0	< 7.0
no	92.8%, 6.4M	92.8%, 2.9M	92.7%, 2.8M	92.7%, 2.6M	92.6%, 2.1M
yes	96.3%, 5.5M	96.3%, 1.2M	96.3%, 0.8M	96.2%, 0.2M	96.2%, 0.2M

Table 10.11: Result of applying different pruning strategies on Turku Dependency Treebank models.

Test leaving out the word form as a feature.



(a) FTB, no analyzer

(b) TDT, no analyzer

(c) FTB, analyzer

(d) TDT, analyzer

Chapter 11

Conclusions

References

Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., and Mohri, M. (2007). Openfst: A general and efficient weighted finite-state transducer library.

Besag, J. (1975). Statistical analysis of non-lattice data. *Journal of the Royal Statistical Society. Series D (The Statistician)*, 24(3):pp. 179–195.

Bilmes, J. (1997). A Gentle Tutorial on the EM Algorithm and its Application to Parameter Estimation for Gaussian Mixture and Hidden Markov Models.

Boyen, X. and Koller, D. (1998). Tractable inference for complex stochastic processes. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, UAI'98, pages 33–42, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Brants, T. (2000). Tnt - a statistical part-of-speech tagger. In *Proceedings of the Sixth Applied Natural Language Processing (ANLP-2000)*, Seattle, WA.

Ceausu, A. (2006). Maximum entropy tiered tagging. In *The 11th ESSLI Student session*, pages 173–179.

Charniak, E. and Johnson, M. (2005). Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, ACL '05, pages 173–180, Stroudsburg, PA, USA. Association for Computational Linguistics.

Church, K. W. (1988). A stochastic parts program and noun phrase parser for unrestricted text. In *Proceedings of the second conference on Applied natural language processing*, ANLC '88, pages 136–143, Stroudsburg, PA, USA. Association for Computational Linguistics.

Collins, M. (2002). Discriminative training methods for hidden markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing - Volume 10*, EMNLP '02, pages 1–8, Stroudsburg, PA, USA. Association for Computational Linguistics.

Collins, M. and Roark, B. (2004). Incremental parsing with the perceptron algorithm. In *Proceedings of the 42Nd Annual Meeting on Association for Computational Linguistics*, ACL '04, Stroudsburg, PA, USA. Association for Computational Linguistics.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Mach. Learn.*, 20(3):273–297.

Cutting, D., Kupiec, J., Pedersen, J., and Sibun, P. (1992). A practical part-of-speech tagger. In *Proceedings of the Third Conference on Applied Natural Language Processing*, ANLC '92, pages 133–140, Stroudsburg, PA, USA. Association for Computational Linguistics.

Dempster, A. P., Laird, N. M., and Rubin, D. B. (1977). Maximum likelihood from incomplete data via the EM algorithm. *Jornal of the Royal Statistical Society, Series B*, 39(1):1–38.

Forney, G. D. J. (2005). The viterbi algorithm: A personal history.

Freund, Y. and Schapire, R. E. (1999). Large margin classification using the perceptron algorithm. *Machine Learning*, 37(3):277–296.

Geman, S., Bienenstock, E., and Doursat, R. (1992). Neural networks and the bias/variance dilemma. *Neural Comput.*, 4(1):1–58.

Goldberg, Y. and Elhadad, M. (2011). Learning sparser perceptron models. *Technical report*.

Grzegorz Chrupala, G. D. and van Genabith, J. (2008). Learning morphology with morfette. In Calzolari, N., Choukri, K., Maegaard, B., Mariani, J., Odijk, J., Piperidis, S., and Tapias, D., editors, *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, Marrakech, Morocco. European Language Resources Association (ELRA). http://www.lrec-conf.org/proceedings/lrec2008/.

Halácsy, P., Kornai, A., and Oravecz, C. (2007). Hunpos: An open source trigram tagger. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, ACL '07, pages 209–212, Stroudsburg, PA, USA. Association for Computational Linguistics.

Huang, L., Fayong, S., and Guo, Y. (2012). Structured perceptron with inexact search. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, NAACL HLT '12, pages 142–151, Stroudsburg, PA, USA. Association for Computational Linguistics.

Huang, Z., Eidelman, V., and Harper, M. (2009). Improving a simple bigram hmm part-of-speech tagger by latent annotation and self-training. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Short Papers*, NAACL-Short '09, pages 213–216, Stroudsburg, PA, USA. Association for Computational Linguistics.

Huang, Z., Harper, M., and Wang, W. (2007). Mandarin part-of-speech tagging and discriminative reranking. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 1093–1102, Prague, Czech Republic. Association for Computational Linguistics.

Hulden, M., Forsberg, M., and Ahlberg, M. (2014). Semi-supervised learning of morphological paradigms and lexicons. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 569–578, Gothenburg, Sweden. Association for Computational Linguistics.

Hulden, M., Silfverberg, M., and Francom, J. (2013). Finite state applications with javascript. In *Proceedings of the 19th Nordic Conference of Computational Linguistics (NODALIDA 2013)*, pages 441–446, Oslo, Norway. LiU Electronic Press.

Johnson, M. (2007). Why Doesn't EM Find Good HMM POS-Taggers? In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 296–305.

Kaplan, R. M. and Kay, M. (1994). Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378.

Klein, D. and Manning, C. D. (2002). Conditional structure versus conditional estimation in nlp models. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing - Volume 10*, EMNLP '02, pages 9–16, Stroudsburg, PA, USA. Association for Computational Linguistics.

Koller, D. and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, Cambridge, Massachusetts, USA.

Koskenniemi, K. (1984). A general computational model for word-form recognition and production. In *COLING-84*, pages 178–181, Stanford University, California, USA. Association for Computational Linguistics.

Lafferty, J. D., McCallum, A., and Pereira, F. C. N. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pages 282–289, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Lindén, K. (2009). Entry generation by analogy – encoding new words for morphological lexicons. *Northern European Journal of Language Technology*, 1:1–25.

Manning, C. D. and Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA.

Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.*, 19(2):313–330.

Merialdo, B. (1994). Tagging english text with a probabilistic model. *Comput. Linguist.*, 20(2):155–171.

Mohri, M., Pereira, F., and Riley, M. (2002). Weighted finite-state transducers in speech recognition. *Computer Speech & Language*, 16(1):69 – 88.

Müller, T., Cotterell, R., Fraser, A. M., and Schütze, H. (2015). Joint lemmatization and morphological tagging with lemming. In Màrquez, L., Callison-Burch, C., Su, J., Pighin, D., and Marton, Y., editors, *EMNLP*, pages 2268–2274. The Association for Computational Linguistics.

Müller, T., Schmid, H., and Schütze, H. (2013). Efficient higher-order CRFs for morphological tagging. In *Proceedings of 2013 Empirical Methods in Natural Language Processing (EMNLP 2013)*, pages 322–332, Seattle, Washington, USA.

Pal, C., Sutton, C., and McCallum, A. (2006). Sparse forward-backward using minimum divergence beams for fast training of conditional random fields. In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, volume 5, pages V–V.

Pearl, J. (1982). Reverend bayes on inference engines: a distributed hierarchical approach. In *in Proceedings of the National Conference on Artificial Intelligence*, pages 133–136.

Pirinen, T., Silfverberg, M., and Lindén, K. (2012). Improving finite-state spell-checker suggestions with part of speech n-grams. In *13th InternationalConference on Intelligent Text Processing and Computational Linguistics CICLing 2014*.

Rabiner, L. R. (1989). A tutorial on hidden markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, pages 257–286. IEEE.

Ratnaparkhi, A. (1996). A maximum entropy model for part-of-speech tagging. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP, Pennsylvania, USA. Association for Computational Linguistics.

Ratnaparkhi, A. (1998). *Maximum Entropy Models for Natural Language Ambiguity Resolution*. PhD thesis, Philadelphia, PA, USA. AAI9840230.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408.

Ruokolainen, T. and Silfverberg, M. (2013). Modeling oov words with letter n-grams in statistical taggers: Preliminary work in biomedical entity recognition. In *Proceedings of the 19th Nordic Conference of Computational Linguistics (NODALIDA 2013)*, pages 181–193, Oslo, Norway. LiU Electronic Press.

Ruokolainen, T., Silfverberg, M., kurimo, m., and Linden, K. (2014). Accelerated estimation of conditional random fields using a pseudo-likelihood-inspired perceptron variant. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics, volume 2: Short Papers*, pages 74–78, Gothenburg, Sweden. Association for Computational Linguistics.

Silfverberg, M. and Lindén, K. (2011). Combining statistical models for POS tagging using finite-state calculus. In *Proceedings of the 18th Conference of Computational Linguistics NODALIDA 2011*, NEALT Proceedings Series. Northern European Association for Language Technology.

Silfverberg, M. and Lindén, K. (2010). Part-of-speech tagging using parallel weighted finite-state transducers. In Loftsson, H., Rögnvaldsson, E., and Helgadóttir, S., editors, *Advances in Natural Language Processing*, volume 6233 of *Lecture Notes in Computer Science*, pages 369–380. Springer Berlin Heidelberg.

Silfverberg, M., Ruokolainen, T., Lindén, K., and Kurimo, M. (2014). Part-of-speech tagging using conditional random fields: Exploiting sub-label dependencies for improved accuracy. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 259–264, Baltimore, Maryland. Association for Computational Linguistics.

Silfverberg, M., Ruokolainen, T., Lindén, K., and Kurimo, M. (2015). Finnpos: Pos tagging toolkit for morphologically rich languages. *Language Resources and Evaluation*, VOL(NUM):XX–YY.

Sipser, M. (1996). *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition.

Smith, N. A., Smith, D. A., and Tromble, R. W. (2005). Context-based morphological disambiguation with random fields. In *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*, HLT '05, pages 475–482, Stroudsburg, PA, USA. Association for Computational Linguistics.

Sutton, C. and McCallum, A. (2007). Piecewise pseudolikelihood for efficient training of conditional random fields. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 863–870, New York, NY, USA. ACM.

Sutton, C. and McCallum, A. (2012). An introduction to conditional random fields. *Foundations and Trends in Machine Learning*, 4(4):267–373.

Tufis, D. (1999). Tiered tagging and combined language models classifiers. In *Proceedings of the Second International Workshop on Text, Speech and Dialogue*, pages 28–33.

Vishwanathan, S. V. N., Schraudolph, N. N., Schmidt, M. W., and Murphy, K. P. (2006). Accelerated training of conditional random fields with stochastic gradient methods. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 969–976, New York, NY, USA. ACM.

Weiss, D. J. and Taskar, B. (2010). Structured prediction cascades. In Teh, Y. W. and Titterington, D. M., editors, *AISTATS*, volume 9 of *JMLR Proceedings*, pages 916–923. JMLR.org.

Weiss, Y. (2000). Correctness of local probability propagation in graphical models with loops. *Neural Comput.*, 12(1):1–41.

Zhang, K., Su, J., and Zhou, C. (2014). Regularized structured perceptron: A case study on chinese word segmentation, pos tagging and parsing. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics*, pages 164–173, Gothenburg, Sweden. Association for Computational Linguistics.

Contributions

Articles