Master in Artificial Intelligence (UPC-URV-UB)

# Master of Science Thesis

# Robust Part of Speech Tagging

Eva Martínez Garcia

Advisor: Lluís Màrquez

Barcelona, January 2013

# Contents

# 1

# Introduction

## 1.1 Motivation

People usually make mistakes when they write something on a computer. This is specially true when people write in informal contexts in the Internet, where they tend to be less careful with their spelling and grammar and use a non-standard language. For instance, it is easy to notice how the number of errors found in blogs, social networks webpages like Twitter or Facebook, or even e-mails, is usually higher than the ones found in more formal documents. This is challenging for the Natural Language Processing (NLP) tasks, because nowadays many data come from the Internet and from the direct interaction with the user.

Generally, NLP tools use well-formed and annotated data to learn patterns by using machine learning techniques. However, in this work we will focus on the language used in an on-line platform for machine translation. In this area it is usual to have a framework such the following: a web-page which offer a service of translation between pairs of languages. The problem is that the casual users utilize the service to translate any type of text (cut and paste, single words, bad formatting, snipets, informal language, pre-traductions, etc.). Hence, in this situation we will find very often words with mistakes that make the system provides a bad translation because it is not able to understand the input. For example, we can find inputs like the following ones:

```
I'm going to take it slooow
how r u
court,, shaved
```

```
 U r suffering from lack of Vitamin
 I'm go KICK ASS Ming to take it slooooow
 how r  damn itu
 court, Oh fun ^^, shaved
 U r suffering from lack of Vitamin
 KICK ASS MOVES
 damn it
 Oh fun ^^
 I miss u baby' and
 civil engineer
 Powdered paint
 . In the current study
 preeminently , combined , vulnerability , caused
software. This chapter contains the following sections:
 , but not the madness of people"
 becaus the woman ist so nice
 your funny
 use.
```

where it is used a very informal language even with abbreviations and we can observe also spelling errors and emoticones. But the system not only will be affected by the typographical mistakes. There are also structure errors. We can find sentences with a wrong gramatical structure, incomplete sentences, lists of unrelated words or single words and a bad use of the punctuation marks. Obviously, this kind of mistakes affect the quality of the translation that the system will give as output to the user. In *Chapter* 3 we will analyze and cathegorize in depth the kind of errors that we can find in this framework. We use data from the FAUST project (Byr13)1 as representation of the texts that one can find through the Internet, in particular in the framework of a machine translation on-line service that we described before.

We will focus our work in one of the first steps in a NLP processing system: the Part–of–Speech (PoS) tagging task. The PoS tagging task is a well known task in the Natural Language Processing world. It consists in tagging every word in an input data with its syntactical cathegory (i.e., noun, verb, adjective, etc.). There are lots of PoS taggers based on different approaches: Hidden Markov Models-based TnT tagger(Bra00), several variants of Maximum Entropy approach (Rat96), transformation-based learning

tagger (Bri95), and Support Vector Machines based tagger SVMTool (GM04). Every approach has its advantages and its disadvantages, and they achieve a state-of-the-art accuracy for English PoS tagging (between 96.4% and 96.7%). But when we try to use the taggers with a non-standard input their performance drops significantly, as we will see in *Chapters* 4, 5 and6. This happens mainly because, in general, the tools use a dictionary of the words they see in the learning process, so that, these dictionaries are incomplete and miss words from specific domains. Hence, a non-standard input become a group of unknown and/or ambiguous words for the taggers. This kind of words, the unknown for the tool, are one of the most difficult ones because the tagger has to guess the tag of a target word using only some statistical information from the training data (but this data may not be in the same domain of the input), the word's context available in tagging time (previous word, following word, etc.), and, of course, the word itself (prefixes, suffixes, number of letters, etc.). We can say then that the PoS task is also naturally affected by the mistakes made by users' writting.

But this is not the only problem in nowadays PoS tagging task. Highly related to the topic we explained before, we think that is remarkable the small amount of supervised data available for training the tools using data that come from the Internet. This lack of annotated data makes more difficult to learn from the wrong written texts. One can think that it is useless to try to learn from this kind of data because it seems that users do not follow a clear pattern to make errors. However, there are a lot of mistakes which are more probable and more usual than others to appear. The probability of a mistake may depend on the key distribution of the keyboard or it could be induced by phonetical confusions also. For example, a user can write *borther* instead of *brother*, and this error would be more probable than writing *brohter*. It is useful to have as many annotated examples of this kind because we can extract statistical information of the errors or to just manually analyze the data to try to get some patterns of users' mistakes or to extract usual users' expresions that are a priori unknown for the tagger because they do not belong to the domain of the training data of the tool. Those ideas (mistakes probabilities, phonetical confusions, etc.) could be good starting points to develop approaches to improve the performance of a NLP tool when dealing with a non-standard input.

One question that arises is how to deal with non–standard or noisy input. In NLP literature we can find several works addressing this problem with different approaches.

We revisit them in *Chapter* 2.
In general, we can tackle the problen thinking in two ways: correcting the input before applying the tool or adapting the tool/task to this kind of input.
Our work is focused on dealing with non-standard data (written data with mistakes, halted sentences, sentences without structure, etc.) by developing several strategies of domain adaptation, other ones to try to correct the input, etc., in order to make robust a particular PoS tagger: The SVMTagger (GM04). The SVMTool is a tagger based on Support Vector Machines, this is, it follows a statistical approach to learn from the data. The SVMTool has been already successfully applied to English and Spanish PoS tagging, exhibiting state–of–the–art performance (97.16% and 96.89%, respectively).

The first step in our work, regarding to the absense of annotated data, is generating an annotated corpus containing real data from real users. We get data from online automatic translation systems in the framework of the FAUST project (Byr13) [1]. Although the data files are not very large (1421 sentences, 22401 words), it is enough to help us to make an idea of the errors and mistakes that users can making translation queries on the Internet. We do not only make a manual systematic tagging, we also make an analysis of the mistakes along the data to take them in account to design the strategies to improve our tagger.

Then, we continue with the strategy of correcting the input before trying to tag the data. We do not want to change the input, so that, what we want to do is to teach the tagger somehow how to tag unknown words that actually are known words but badly written, this is, somehow detect false unknown words. We make an exhaustive manual analysis of the mistakes in the data from the FAUST project and design different ways to deal with them. For example, introducing common proper nouns in the dictionary of the tagger (i.e. Google, Yahoo!,etc.), including common expresions like: a.k.a, i.e., a.m., u, r, lol, wtf?, etc. We also try to guess if an input word is a known word but written in a wrong way. Recalling the previous example, we can find *borther* in a text but where it is in fact the word *brother* which could be a known word for our tagger. In short, we also study how can help to the tagger a guessing module. This module will be a preprocessing step to the input data to try to recognise known words badly written by developing several heuristics and implementing the Levenstein's distance.

[1] the FAUST project is developed in the European Communitys Seventh Framework Programme (FP7/2007-2013) under grant agreement number 247762 (FAUST, FP7-ICT-2009-4-247762)

Afterwards, we start developing the task of adapting the tool to the non-standard input. First, we do that by retraining the SVMTagger using also the data from the FAUST project (Byr13) 1 to let the SVMTagger learn from it. Then, we use a transformation-based learning system (*fnTBL*(FN01b)) to extract rules that could fix the tagging errors made by the SVMTool tagger in both scenarios, before and after retraining it with the FAUST data. Those rules are learnt from the tagged FAUST data (manual tagged and tagged by the SVMTool).

Finally, we try to put it all together, this is, make a mixture of all the strategies we have developed and implemented to take advantage of the strong points of each one. What we expect and want to observe is that the methods we implemented and developed to improve the robustness of the SVMTagger work in an accumulative way and allow us to obtain a significant improvement in the results of the SVMTagger over non–standard–input or noisy data.

## 1.2 Goals of this Thesis

The main goal of our work is, once we have identified the problem of dealing with non-standard-input is to develop a robust PoS tagger from the SVMTagger. But we can define several "subgoals".

- Build an annotated corpus with real data. In particular, we use the data from the automatic translation system weblog in the framework of the FAUST project.
- Analyze and study real data to learn from users' errors in order to be able to design good strategies to improve the tagger.
- Retrain the tagger using also real data to let the SVMTool tagger learn statistical information from non–standard–input.
- Take advantage of a transformation-based learning system to improve the accuracy achieved by the tagger correcting tagging errors.
- Design and implement a module to expand the dictionary used by the tagger that takes in account the input data.
- Design and implement a guessing module to identify "false unknown words" (i.e., real known words but wrongly written).

- Test the new tagger and obtain results. Analyze the results and make conclusions. If necessary, design other experiments or repeat the ones already done.

Through this "subgoals", we expect to reach our main goal and obtain a robbust PoS tagger. Furthermore, we also want to provide a useful tool to the NLP comunity making the new tagger available through the open-source SVTool repository.

## 1.3 How to Read this Document?

This document is organized as follows. *Chapter* 2 contains background information. It discusses previous and related work. We mention there all the papers we found which are related to our work and that we used as inspiration to deal with our problem. Particularly, we explain the different tools available to implement the PoS tagging task, this is, we briefly report several PoS taggers and we explain in detail the SVMTagger, that is the tool we want to make robuster. We also explain the Transformation-based learning algorithm, the one we used to try to improve the SVMTagger by learning rules. Of course, we explain afterwards the particular implementation of the algorithm we use in our experiments: the *fnTBL* system. In short,in this section we explain the related work we have used as background and describe the tools we are going to use in our work.

*Chapter* 3 describes the corpora we are going to use in our experiments. We make a brief report of the Wall Street Journal corpus and we also explain its structure and which sections we are going to use in every task. Afterwards, we introduce and analyze there our corpus from the FAUST project.

After analyzing the FAUST corpus data, we try to reproduce artificially the mistakes that we observed in order to improve the performance of the tagger on noisy data. We retrain the tagger with this artificial data. In *Chapter* 4, we show the results obtained in these experiments.

Another way to learn from a non-standard input is doing *domain adaptation*. We developed several methods to do that following two strategies: with or without training step. *Chapter* 5 and *Chapter* 6, respectively, explain the approaches developed to deal with the *domain adaptation* challenge, explain the experiments and discuss the results

obtained.

Finally, in *Chapter* 7, the document ends with a general discussion of the work analyzing the achieved goals, drawing conclusions and describing further work.

# 2

# Background

The PoS tagging task is a well known task in the Natural Language Processing community. There is a lot of literature related to it. There are also many tools developed to deal with this task.

Through this section we will explain the related work that we used as a starting point of our work. We will separate the papers by the described techniques or the implemented algorithms as follows: Domain adaptation methods/techniques, non-standard input treatment, error correction algorithms or robust tagging approaches. We present the most popular taggers in the NLP community.

Afterwards, we explain in detail the algorithms and tools we will use in our work. The TBL algorithm, the *fnTBL* toolkit and, finally, the SVMTool system.

## 2.1 Related Work

Since we want to make more robust the SVMTagger, we need to learn from non-standard data somehow. We focused in the following sets of techniques: *domain adaptation*, *error correction*, and *robust tools* or methods to deal with *non-standard input*.

In the following subsections, we report the most interesting papers for us.

### 2.1.1 Non-Standard Input

The first important step was to recognize that there exists a problem with the input received by the NLP tools. It is not usual to have a well formed input from the real on-

line world. There are some works that claim the importance of recognizing, analizing and dealing with this kind of input. In (Kwa82)(KH82) the authors study where is the origin of the non-standar inputs. They say that it is normal to have deviations from the standard written language and they also propose some approches that they think can work well over noisy data like meta-rules based approaches or systems that can take into account users' feedback.

We are also concerned with the treatment of the non-standard input for a general task, as is described in (Mar82) or in (Geh83), searching for robustness among different tools. But in this work we are focused in the part–of–speech tagging problem.

When we try to obtain a robust tool which be able to deal with non-standar input, we can think in two ways of obtaining that: correcting the errors or adapting the tool to that input.

### 2.1.2 Error Correction

There are many work done in error correction tasks. For example, the spell and grammar checkers included in the word processor programs are well known. In (QBB$^+$08), the authors describe the design and implementation of a methodology for user-centered error correction applications. Another work which describes a framework to deal with the spelling correction task is (IEL$^+$93). We can find also description of spell correctors like in (MWM00), where the authors describe several methods to correct misspellings attending, for example, the word frequency or the character distance. The distance between words algorithm is one of the ideas that we developed to improve the SVM-Tagger without retraining it. Preprocessing the input applying a distance algorithm in order to guess a known word candidate for an unknown word in the input.
Finally, in (PKHC98) it is shown that not only English is the language of study. It is also interesting to study non-latin languages as Arabic, Chinese or, in the case studied in this paper, Korean. But the authors also show that one of the difficulties in this area is to find large annotated corpora. The important hint from this paper is that one of the clues to success in our experiments is to focus on improving the performance of the tagger over the unknown words for the system, this is, these words in the input (test set) that the tool do not recognize because they do not appear in the training data.

When we try to correct errors we also have to take into account that a native person does not make the same mistakes than a person who has the English as a

second language (ESL). Users will make errors attending to their mother tongue. In (RR10), the authors show the importance of this kind of errors to motivate their work in generating candidate corrections for the task of correcting errors in a text. They study there the usefullness of use error-tagged data in training. We will report similar results in one of our first approaches to improve the SVMTagger, when we tried to reproduce artificially the mistakes made by users (*Chapter* 4). The authors of this paper focused their work in building candidate sets (lists of confusable words) taking into account the context of the word. In particular, they talk about prepositions showing that the best approach is the one which takes into account the likelihood of each preposition confussion. That made us to think in designing a probability distribution of common errors or mispellings. We go in depth with that idea in the *Chapter* 5.
In (RR11), the authors analyze and revisit this problem and compare some algorithms in order to decide which one fits better for this kind of tasks (it results that the best one is an Averaged Perceptron). They also present how to adapt a model to the source language of the writer without the necessity of retraining the model. Their method is computationally cheap and performs better than other approaches.
An interesting application of the error correction methods is described in (Hig00). They work with a Bayesian statistics approach in the *Encyclopédie* project framework. What they want is to detect and correct all the errors in the electronic version of the $18th$ century French encyclopedia of Diderot and d'Alembert.

### 2.1.3 Domain Adaptation

Anther way to deal with non-standard input is to adapt the tool to the input trying to generalize as much as possible. With that we wanted to use the tool over other kind of non-standar texts.

There are lots of papers describing the *domain adaptation* problem. Our first approximation to the topic was by means of the Dan Roth talk (Rot11). In the talk, Dan Roth presents the necessity of adaptation, some problems of the domain adaptation and the necessity of combining adaptation methods. This ideas are developed in (CCR10). This paper claims the necessity of combining labeled and unlabeled adaptation algorithms.
Many works describe methods to make a good domain adaptation. Ones study the

problem from the instance weighting perspective (JZ07)(we will apply this idea in our experiments retraining the SVMTagger with real non-standard texts in *Chapter* 6). In other works, the authors describe methods to adapt texts instead of the model, like in (KR11), in order to be able to apply models across domains without modification. In (BMP06), the authors make correspondences among features from different domains. In (FM09) they described models based on a hierarchical Bayesian prior. One can stay in one of the simplest cases and only take care of the capitalization of the text, like they did in (CA04). Or, on the contrary, we can design a really easy way to do domain adaptation just augmenting the feature space in a fully supervised framework like it is explained in (DIKS10).

A special method of doing domain adaptation is the Transformation–Based Error–Driven Learning (TBL) algorithm (Bri95). This approach is rule based, in opposite of the statistical approaches we mentioned before. The advantages of this approach are that it captures linguistic information in simple and understandable rules. There are also no many rules obtained as oppposed to large numbers of lexical and contextual probabilities. We will apply an implementation of this method in some of our experiments in *Chapter 6*. The *fnTBL* toolkit and the TBL algorithm are described in detail later in this section.
We can see an application of this algorithm in (MBTVS06). Also, in (GMTJGM06) we can see an attempt to improve the efficiency of the method.

There are many applications of the domain adaptation to a specific domain or task. For example, in (MCJ10) they do domain adaptation for parsing, in (JR10) they do domain adaptation in medical context and in (MTVS07) they adapt PoS tagging for BioMedical domains.

### 2.1.4 PoS Tagging

We chose the SVMTagger to develop our work because we know it well and it has an state–of–the–art performance. Nevertheless, we also read and investigated about other PoS taggers. In particular, we also studied the tagger described by Brill in (Bri92). This tagger is cited in almost all the works related to the part of speech tagging problem, mostly because of its simplicity and good performance.
We found works concerned in the importance of the unknown words regarding to improving the performance of the taggers across domains, like the authors describe in

(UPRR10).
There are taggers that use a different statistical method from the one used in the SVM-Tagger, the Support Vector Machines. For example, Hidden Markov Models-based TnT tagger (Bra00), several variants of Maximum Entropy approach (Rat96), or variable Memoriy Markov Models (VMM)-based tagger described in (Sch94) .
We have to keep in mind that there are other ways to afford the PoS tagging task. For instance, we can use rule-based systems that implements the TBL algorithm (Bri95) like the *fnTBL* (see (FN01a), (FN01b) and (FHN00) )or the $\mu$ *TBL* (see (Lag99a), (Lag99b)), which are able of achieve a results in the state–of–the–art performance.

## 2.2 The Transformation-Based Error-Driven Learning Approach

Another way to learn from the non-standard input is using a rule-based approach to learn from the data. In order to do that, we use the approach described in Brill (Bri95). Particularly, we use the *fnTBL* system.

### 2.2.1 Transformation-based error-driven learning algorithm

The TBL approach consists in extracting linguistic information from the training data in the shape of a set of rules in order to be able to apply them afterwards to another inputs. The resulting set of rules is an ordered list of rules which goal is to correct the mistakes made after an initial assignment usually based on simple statistics. The rules are greedily learned until the selected transformation does not modify the data in enough places or there are no more rules to be selected. We can see a diagram of the algorithm in Figure 2.1.

In comparison to a statistical learner, in this case we will be able to understand and analize better the acquired knowledge. Furthermore, the set of rules usually is not very big and the learned rules use to be useful and meaningful.

We can describe the TBL algorithm as follows:

**Figure 2.1:** Transformation-Based Error-Driven Learning

1. Initialize each sample in the training data with a classification (most likely classification, output of other classification system, ...). This would be the starting training set $T_0$.

2. Considering all the transformations (rules) to the training data in this step $T_k$, select the rule with the highest score and apply it to the training data to obtain the training set modified by the rules $T_{k+1}$, this is, the samples of the training data after applying to them every rule in the set of rules.
   If there are no more possible transformations (if we applied all the rules), or the score of the best rule is below a threshold, stop.

3. Update the state of the training set. We continue with the algorithm from the last state of the data obtained in the previous step: $T_{k+1}$ (we can see this as an update of the $k$, $k = k + 1$).

4. Repeat from 2.

The main idea in our experiments will be to take advantage the TBL-based systems to learn rules from the FAUST corpus in order to correct the mistakes made by the

SVMTagger or by the TBL-based PoS tagger itself when tagging the test FAUST set.

We chose a well known and well documented TBL-based system: *the fnTBL*,and developed several experiments to test the system in the POS-tagging task and also to observe how the *fnTBL* toolkit can help us to improve the performance of our SVMTagger when working with non-standard input.

### 2.2.2 The fnTBL toolkit

As we said before, the *fnTBL* is a toolkit which implements a transformation based learning technique (the approach developed by Brill in (Bri95)). It is mainly based in transforming the data in order to correct the mistakes that make the biggest increasement of the error rate. Its developers define the *fnTBL* as a customizable, portable and free source machine-learning toolkit primarily oriented towards Natural Language-releated tasks (not only POS tagging). It is remarkable that it is also a well documented and easy to install tool.

*fnTBL* was created at the Johns Hopkins University NLP group by Radu Florian and Grace Ngai.See **??**, **??** and **??**.

Although a TBL-based system has a lot of advantages or, at least, a lot of attractive characteristics, it also have a big drowback: the learning time is usually large. However, *fnTBL* is a fasten version of this kind of systems since it only takes care of classification tasks, although the TBL approach is more general. These are some of the reasons to choose the *fnTBL* system, because of its capability of speed up the learning step and its orientation to the classification tasks.

While the most expensive step in the TBL-based systems is the computation of the rules' score, the *fnTBL* implements the following steps to speed up the learning process in the TBL algorithm.
The tool store the rules into memory jointly with the good and bad counts instead of regenerating the rules. The main idea is to store the rule counts (number of good counts - when the rule fixes a mistake- or bad counts -when the rule change a good classification-) and to recompute these values as necessary, after applying a new rule to the corpus. One can think that it could be more difficult to identify the rules that need to update their counts. The advantage is that only samples in the vicinity of the application of the best rule need to be examined and the rules that apply on them are

those which need to update their counts. The update process is also easy because we are generating the rules just using the set of templates to identify these ones which apply on those positions in the vicinity and update their counts if necessary. In the Figure 2.2 it is shown the algorithm in more detail. The developers report that this approach can obtain up to 2 orders of magnitude speed-up relative to the approach present in Brill's tagger (Bri95).

#### 2.2.2.1 Settings for the fnTBL

The *fnTBL* uses a set of files to adjust the different parameters of the algorithm (threshold to learn the rules, rules' templates,...). Although in the first experiments we used the files with the values given by default with the distribution, we changed several of these files through our experimets.
We explain briefly the different files and parameters we can change to tune the system in order to perform better on our non-standard data.

First of all, the system has to know the form of the files it is going to deal with. We can define this in the *file.templ* file. Since we are concerned in the PoS tagging task, our file template will look as the following:

```
word pos => tpos
```

this is, from that, the *fnTBL* would expect files that contain in every line, a word, the part-of-speech guessed related to it and the true part-of-speech that corresponds to that word.

Regarding to the rules' templates, we used the default files provided by the distribution of the *fnTBL*. We have to remark that the templates of rules are divided in two different files. *lexical_rules.templ* collects those templates that will generate rules focused only in the word itself (prefix, suffix, etc.) We can see an example of that file in Figure 2.3.

On the other hand, *rule.pos.templ* collects the rules that take care of the local context of the word. In Figure 2.4 we can see an example of that file.

Finally, only remains to explain the files which define the parameters that use the *fnTBL*. These are *tbl.lexical.train.params* and *tbl.context.pos.params*. These files specify

**Figure 2.2:** FastTBL Algorithm

```
word => pos
# This are the rules using 5 characters
# Suffix/prefix identity rules
pos word::5 => pos
pos word::5 => pos
# Suffix addition only at this level
pos word::++5 => pos
# Suffix/prefix subtraction rules
pos word::-5 => pos
pos word::5- => pos
# Rules at level 4, etc.
pos word::4 => pos
pos word::4 => pos
pos word::4++ => pos
pos word::++4 => pos
pos word::-4 => pos
pos word::4- => pos
pos word::3 => pos
pos word::3 => pos
pos word::++3 => pos
pos word::3++ => pos
pos word::3- => pos
pos word::-3 => pos
pos word::2 => pos
pos word::2 => pos
pos word::-2 => pos
pos word::2- => pos
pos word::++2 => pos
pos word::2++ => pos
pos word::1 => pos
pos word::1 => pos
pos word::++1 => pos
pos word::1++ => pos
pos word::-1 => pos
pos word::1- => pos
pos word::1<> => pos
pos word1 => pos
pos word-1 => pos
```

```
# The same rules as the preceding ones,
# but without conditioning on the POS
word::5 => pos
word::5 => pos
word::-5 => pos
word::5- => pos
word::4 => pos
word::4 => pos
word::4++ => pos
word::++4 => pos
word::-4 => pos
word::4- => pos
word::3 => pos
word::3 => pos
word::++3 => pos
word::3++ => pos
word::3- => pos
word::-3 => pos
word::2 => pos
word::2 => pos
word::-2 => pos
word::2- => pos
word::++2 => pos
word::2++ => pos
word::1 => pos
word::1 => pos
word::1<> => pos
word::++1 => pos
word::1++ => pos
word::-1 => pos
word::1- => pos
# Bigram cooccurrence predicates:
# Test on the previous word
word-1 => pos
# Test on the next word
word11 => pos
```

**Figure 2.3:** Sample of lexical rule template file

| | |
|---|---|
| pos_0 word_0 word_1 word_2 => pos<br>pos_0 word_-1 word_0 word_1 => pos<br>pos_0 word_0 word_-1 => pos<br>pos_0 word_0 word_1 => pos<br>pos_0 word_0 word_2 => pos<br>pos_0 word_0 word_-2 => pos<br>pos_0 word:[1,2] => pos<br>pos_0 word:[-2,-1] => pos<br>pos_0 word:[1,3] => pos<br>pos_0 word:[-3,-1] => pos<br>pos_0 word_0 pos_2 => pos<br>pos_0 word_0 pos_-2 => pos<br>pos_0 word_0 pos_1 => pos<br>pos_0 word_0 pos_-1 => pos<br>pos_0 word_0 => pos<br>pos_0 word_-2 => pos<br>pos_0 word_2 => pos<br>pos_0 word_1 => pos<br>pos_0 word_-1 => pos<br>pos_0 pos_-1 pos_1 => pos | pos_0 pos_1 pos_2 => pos<br>pos_0 pos_-1 pos_-2 => pos<br>pos_0 pos_1 => pos<br>pos_0 pos_-1 => pos<br>pos_0 pos_-2 => pos<br>pos_0 pos_2 => pos<br>pos_0 pos:[1,3] => pos<br>pos_0 pos:[1,2] => pos<br>pos_0 pos:[-3,-1] => pos<br>pos_0 pos:[-2,-1] => pos<br>pos_0 pos_1 word_0 word_1 => pos<br>pos_0 pos_1 word_0 word_-1 => pos<br>pos_-1 pos_0 word_-1 word_0 => pos<br>pos_-1 pos_0 word_0 word_1 => pos<br>pos_-2 pos_-1 pos_0 => pos<br>pos_-2 pos_-1 word_0 => pos<br>pos_1 word_0 word_1 => pos<br>pos_1 word_0 word_-1 => pos<br>pos_0 pos_1 pos_2 => pos<br>pos_0 pos_1 pos_2 word_1 => pos |
| | |

**Figure 2.4:** Example of contextual rule templates

the different files the tool will use to generate the statistical information needed to the learning process, the values of the thresholds and the directory where the tool has to work. There are two different files to define this because the tool learn the *lexical rules* using information from the training data focusing in the unknown words. Afterwards, following the manual guide of the *fnTBL*, the *contextual rules* can be learnt using the entire training data set. We think that this is useful if we have a text whithout any guessing tagging process made previously, but if we have a file which contains an already tagged text we considered more coherent just learn all the rules in only one step.

After setting the previous files, only remains to prepare the input and the training files. In our case, these would be the FAUST corpus. Basically, we only need to separate the sentences by an empty line to indicate the *fnTBL* that there was an end of a block, for the POS task this is the end of a sentence.

After these preprocess steps, and knowing welll the different files that the *fnTBL* is going to use, we are ready to run our experimets and work with the *fnTBL* toolkit.

## 2.3 SVMTagger

We talked before about several good POS-taggers that follow different strategies to deal with the POS tagging task.

We present through this section the POS-tagger we have chosen to improve and make it robust over a real non-standard input: the *SVMTagger*.

### 2.3.1 Presenting the SVMTagger

The *SVMTool* is a simple and effective generator of sequential taggers based on Support Vector Machines (SVMs).

The *SVMTool* has been applied to a number of NLP problems, such as Part-of-speech Tagging and Base Phrase Chunking, for different languages. The proposed SVM-based tagger is robust and flexible for feature modelling (including lexicalization), trains efficiently with almost no parameters to tune, and is able to tag thousands of words per second, which makes it really practical for real NLP applications.
Regarding accuracy, the SVMTagger achieves a very competitive accuracy of 97.2% for English on the Wall Street Journal corpus, which is comparable to the best taggers reported up to date.

The *SVMTool* software package consists of three main components, namely the model learner (*SVMTlearn*), the tagger (*SVMTagger*) and the evaluator (*SVMTeval*), which are described below.

Previous to the tagging step, SVM models (weight vectors and biases) are learnt from a training corpus using the *SVMTlearn* component. Different models are learned for the different strategies. Then, at tagging time, using the *SVMTagger* component, one may choose the tagging strategy that is most suitable for the purpose of the tagging. Finally, given a correctly annotated corpus, and the corresponding *SVMTool* predicted annotation, the *SVMTeval* component displays tagging results.

#### 2.3.1.1 Support Vector Machines

We can define the Support Vector Machines (SVM) as a machine learning algorithm for binary classification. This algorithm has been used to deal with several practical

problems, also for NLP tasks with good results. A good review of the algorithm is (CST00).

Describing the classification scenario, let $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$ be the set of $N$ training examples, where each instance $\mathbf{x}_i$ is a vector in $\mathbb{R}^N$ and $y_i \in \{-1, +1\}$ is the class label. In their basic form, a SVM learns a linear hyperplane that separates the set of positive examples from the set of negative examples with *maximal margin* (the margin is defined as the distance of the hyperplane to the nearest of the positive and negative examples). This learning bias has proved to have good properties in terms of generalization bounds for the induced classifiers.

The linear separator is defined by two elements: a weight vector $\mathbf{w}$ (with one component for each feature), and a bias $b$ which stands for the distance of the hyperplane to the origin. The classification rule of a SVM is:

$$sgn(f(\mathbf{x}, \mathbf{w}, b)) \tag{2.1}$$

$$f(\mathbf{x}, \mathbf{w}, b) = \langle \mathbf{w} \cdot \mathbf{x} \rangle + b \tag{2.2}$$

where $\mathbf{x}$ is the example to classify. Learning the maximal margin hyperplane $(\mathbf{w}, b)$ can be stated as a convex quadratic optimization problem with a unique solution in the linearly separable case. This is: *minimize* $||\mathbf{w}||$, *subject to the constraints* (one for each training example):

$$y_i(\langle \mathbf{w} \cdot \mathbf{x}_i) + b) \geq 1 \tag{2.3}$$

There is an equivalent dual formulation for the SVM model. It is characterized by a weight vector $\boldsymbol{\alpha}$ and a bias $b$. The weight vector $\boldsymbol{\alpha}$ contains one weight for each training vector, indicating the importance of this vector in the solution. Vectors with non null weights are called *support vectors*. Hence, the dual classification rule can be defined as follow:

$$f(\mathbf{x}, \boldsymbol{\alpha}, b) = \sum_{i=1}^{N} y_i \alpha_i \langle \mathbf{x}_i \cdot \mathbf{x} \rangle + b \tag{2.4}$$

Furthermore, one can calculate the $\boldsymbol{\alpha}$ vector as a quadratic optimization problem. Given the optimal $\boldsymbol{\alpha}^*$ vector of the dual quadratic optimization problem, the weight vector $\mathbf{w}^*$ that defines the maximal margin hyperplane is calculated as follows:

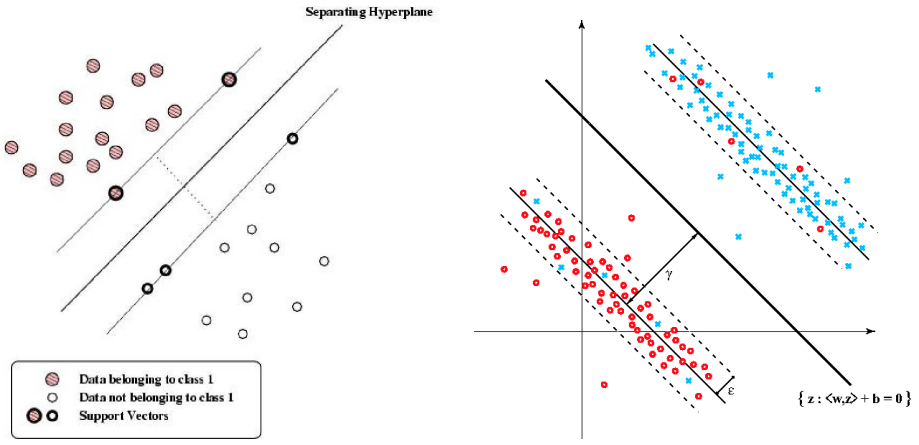


**Figure 2.5:** SVM example: hard margin (left) vs. soft margin (right) maximization in $\mathbb{R}^2$.

$$\mathbf{w}^* = \sum_{i=1}^{N} y_i \alpha_i^* \mathbf{x_i} \tag{2.5}$$

The $b^*$ has also a simple expression in terms of $\mathbf{w}^*$ and the training examples $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$. See (CST00) for details.

The dual formulation has its advantages. It allows an efficient learning of non–linear SVM separators, by introducing *kernel functions*. A kernel function calculates a dot product between two vectors that have been (not linearly) mapped into a high dimensional feature space. Since there is no need to perform this mapping explicitly, the training is still feasible although the dimension of the real feature space can be very high or even infinite.

To avoid overfitting, it may be useful to allow some training errors when there are outliers or wrongly classified training examples. This goal can be reached with a variant of the optimization problem, also called *soft margin*. In this case, the contribution to the objective function of margin maximization and training errors can be balanced through the use of a parameter called $C$. In Figure 2.5 rightmost representation is shown this variation of the optimization problem.

#### 2.3.1.2 Binarizing the POS-tagging task and Feature Codification

The PoS-tagging task is a multi-class classification problem. We explain how we can use the SVM binary classification learning algorithm to do the task. Before applying the SVMs to the problem, the SVMTool do a binarization of the problem. A simple *one-per-class* binarization is applied. This is, for every PoS tag a SVM is trained to distinguish between examples of this class and all the rest. Finally, when we tag a word, the system select the most confident tag according to the predictions of all binary SVMs.

However, the system do not consider all training examples for all classes. From the training corpus a dictionary is extracted with all possible tags for each word, so that, when ocurr a training word $w$ tagged with $t_i$ is used as a positive example for the $t_i$ class and as a negative example for all other classes considered as possible tags for $w$ in the dictionary. With this strategy, the developers of the SVMTool avoid the generation of too much negative examples and also fasten the trainning step.[1]

Regarding to the feature codification, every example has to be represented somehow internally in the system to train the SVMs. Now we are going to explain how the SVMTool do this.
Every word in the input of the system, words that the system tries to determine a tag (output decision), is represented using its *local context*. That context will help the tagger to make a decision even when the word did not occur in the training data.
In the SVMTool it is used a window of 7 tokens. In this window there are some basic and $n$–gram patterns evaluated to form binary features. For example: "previous_word_is_`the`", "two_preceeding_tags_are_`DT_NN`", etc. In Figure 2.6 we can see the list of all patterns considered by the system.

#### 2.3.1.3 SVMTool toolkit

We will describe how works the SVMTool. We present every component of the toolkit and try to show how we will use them in our experiments.

[1]The developers of the SVMTool aims to see (ASS07) for a discussion on the efficiency problems when learning from large PoS training sets.

| | |
|---|---|
| word features | $w_{-3}, w_{-2}, w_{-1}, w_0, w_{+1}, w_{+2}, w_{+3}$ |
| PoS features | $p_{-3}, p_{-2}, p_{-1}, p_0, p_{+1}, p_{+2}, p_{+3}$ |
| ambiguity classes | $a_0, a_1, a_2, a_3$ |
| may_be's | $m_0, m_1, m_2, m_3$ |
| word bigrams | $(w_{-2}, w_{-1}), (w_{-1}, w_{+1}), (w_{-1}, w_0)$ $(w_0, w_{+1}), (w_{+1}, w_{+2})$ |
| PoS bigrams | $(p_{-2}, p_{-1}), (p_{-1}, a_{+1}), (a_{+1}, a_{+2})$ |
| word trigrams | $(w_{-2}, w_{-1}, w_0), (w_{-2}, w_{-1}, w_{+1}),$ $(w_{-1}, w_0, w_{+1}), (w_{-1}, w_{+1}, w_{+2}),$ $(w_0, w_{+1}, w_{+2})$ |
| PoS trigrams | $(p_{-2}, p_{-1}, a_{+0}), (p_{-2}, p_{-1}, a_{+1}),$ $(p_{-1}, a_0, a_{+1}), (p_{-1}, a_{+1}, a_{+2})$ |
| sentence_info | punctuation ('.', '?', '!') |
| prefixes | $s_1,\ s_1s_2,\ s_1s_2s_3,\ s_1s_2s_3s_4$ |
| suffixes | $s_n,\ s_{n\text{-}1}s_n,\ s_{n\text{-}2}s_{n\text{-}1}s_n,\ s_{n\text{-}3}s_{n\text{-}2}s_{n\text{-}1}s_n$ |
| binary word features | initial Upper Case, all Upper Case, no initial Capital Letter(s), all Lower Case, contains a (period / number / hyphen ...) |
| word length | integer |

**Figure 2.6:** Rich feature pattern set used in experiments.

### SVMTlearn

First of all we present the part of the tool that takes care of the training process .

The SVMTlearn is the responsible of training a set of SVM classifiers from a training set of (annotated or unannotated) examples. It uses the SVM–light[1] to do that training process. In particular, it is an implementation of Vapnik's SVMs in C, developed by Thorsten Joachims (Joa99).

Training data must be in column format, i.e. a token per line corpus in a sentence by sentence fashion. For the SVMTool, the column separator is the blank space. The first column of the line will be the token. The second column, the tag to predict and the rest of the line may contain additional information.

To indicate sentence separation, sentence punctuation is used , i.e. [.!?] symbols which are taken as unambiguous sentence separators. Moreover, there is a special symbol to do this task too: '`<s>`'. We will find this useful to indicate where incomplete sentences end.

The configuration settings for the system will be set in a configuration file: *config.svmt*[2]. In that file we will be able of fix several parameters of the system.
In particular, we could adjust the size of the *sliding window* for the feature extraction. Also, we can specify what kinds of feature types can be collected from the sliding window, this is, define the *feature set* that the SVM classifiers will use.
Furthemore, we can say how to *filter the features* in order to maintain the feature space in a convenient size. Features appearing just once are ignored by default.
The system also filter out some weight vector components lower than a given threshold to improve the efficiency and decreasing the model size, this is, it is made a *SVM model compression*.
Moreover, the authors tried to make the tagger robust to corpus errors and allowed that the lexicon extracted from the training corpus could be repaired, so, we can indicate in the configuration file a heuristic dictionary to repair the lexicon using frequency heuristics or a list of corrections.

[1] The $SVM^{light}$ software is freely available (for scientific use) at the following URL: `http://svmlight.joachims.org`. It is necessary to download it prior to start using the *SVMTlearn* component. $SVM^{light}$ is not LGPL licensed.

[2] We show an standard configuration file config.svmt in A.2.

In the same fashion, we can specify a *backup lexicon*, this is, a morphological lexicon with words that do not appear in the training set.
We can specify the list of PoS which present *ambiguity* in order to make easier the training process to the system. However, this list is automatically extracted from the corpus by default. The *open PoS classes* can be treated in the same way.
Finally, we can indicate how to tune the $C$ parameter of the soft-margin version of the SVM learning algorithm. The tool can optimize the $C$ value. A local maximum is found exploring accuracy on a validation set for different $C$ values at shorter intervals.

The most important feature of the system for us is the possibility of authomatically adjusting the $C$ value of the soft margin version of the SVM algorithm that allow us to obtain better results. On the other hand, the possibility of playing with a backup lexicon and a repairing dictionary is very useful for us to start dealing with such a wide an open domain as the non-standard input.

**SVMTagger**

Once we have trained some SVM, we wanted to start testing our classifiers. We want to tag some data. When tagging a corpus (where we have one token per line as we explained before) we have to specify the path to a previously learned SVM model, this is, a path to the SVM classifiers obtained after the training process. Applying the *SVMTagger* we will obtain a PoS tagging output of a sequence of words. The output will be presented as follows: the first column will be the token, in the second column the predicted tag and the rest of the line will remain as it was in the input. Notice that lines beginning with '## ' are ignored by the tagger.

The tagging is on-line based on a sliding window which gives a view of the feature context to be considered at every decision.Calculated part–of–speech tags feed directly forward next tagging decisions as context features.

There was several options to set the tagging task. First of all, we can set the *tagging scheme*: greedy or sentence level. This is, if every tagging decision is made based on a reduced context or if the function to maximize is the global sentence sum of SVM tagging scores. By default, and in our experiments, it's used the greedy scheme.
We can set the *tagging direction*. It can be "'left-to-right"', "'right-to-left"' or a combination of both. By default, and in our experiments, it is used the "'left-to-right"'

direction.
We can achieve robustness by tagging in two passes. This is another setting that we can select for the system. We can say to the system that we want to make predictions for all possible parts-of-speech.
Furhtermore, we can set the threshold to do the *SVM Model Compression* in the same way as in the learning process; and indicate a *backup lexicon* to help the system to deal with unknown words. Or to lemmatize the output, indicate a *lemmae lexicon*. We can say also if the EOS tag is used (¡s¿).

When running the SVMTagger, we can chose an strategy to do the tagging. A strategy is a combination of the different options of the tagger. There are developed 7 different strategies. In our experiments we used the strategy used by default: it makes use of Model 0 in a greedy one pass on-line fasion.[1].

**SVMTeval**

Finally, we will want to evaluate how good is a tagging output. To obtain an evaluation resume we will use this part of the toolkit. We need to have a *SVMTagger* predicted tagging output and the corresponding gold-standard, and the *SVMTeval* will evaluate the performance in terms of accuracy.

The *SVMTeval* can present the results for different sets of words (known words vs unknown words, ambiguous words vs unambiguous words), not only the overall ones. Moreover, the results can be presented from the point of view of the ambiguity (words sharing the same kind of ambiguity may be considered together). Furthermore, words sharing the same degree of disambiguation complexity,can be grouped.

Using the SVMTeval component we can obtain a different report results depending on our interests.

[1]For further information about the tagger options and settings see **??** or the manual of the tool available on-line in the offitial web page of the tool: `http://www.lsi.upc.edu/~nlp/SVMTool/`

# 3

# Corpora

In all of our experiments we work with two types of data: clean and noisy . To represent or learn from clean or standard texts we use the Wall Street Journal corpus. On the other hand, we use data files from the FAUST project as a sample of real data collected from the Internet in the framework of a translation on-line system.

First of all, we describe the Wall Street Journal (WSJ) corpus. We use this corpus as a representation of clean and well-formed data. This corpus is built from news articles.

Finally, we introduce the corpus that represent the real texts that we can collect from the Internet: the FAUST data. We explain how we did the manual annotation of the examples from the FAUST project. We also analyze this information giving details of the common errors we find through the corpus.

## 3.1 The Wall Street Journal Corpus

The Wall Street Journal corpus (from now on we will refer to it as WSJ corpus) is a well known corpus in the natural language processing comunity. This corpus consists of WSJ articles that have been tagged for their part-of-speech. This corpus has $1,173$ Kwords. It is part of the Penn Treebank (Mar) [1], and it is tagged using the Penn Treebank taggset[2].

[1] and, in particular, this corpus is as well parsed. We can find the resources of the Penn Treebank project in its official webpage (`http://www.cis.upenn.edu/~treebank/` ).

[2] The Penn Treebank taggset is presented in the appendix A.1

The corpus is divided in 24 sections. We use a usual distribution of the sections for our tasks: sections 0-18 for. training (912 Kwords), 19-21 for validation (131 Kwords), and 22-24 for test (129 Kwords), respectively. About 2.81% of the words in the test set do not appear in the training set.

It's worth saying that in some of our experiments we will modify the content of the sections, this is, we modify the data itself in particular to try to reproduce artifitially some errors seen in the real data files, as we will explain later on. But we will maintain this separation of the sections of the corpus every time we use it in our experiments.

## 3.2 Non-Standard Real Data: the FAUST Corpus

The sample that we used in our experiments come from the Feedback Analysis for User adaptive Statistical Translation (FAUST) project. The FAUST project (Byr13)1 is concerned to develop machine translation (MT) systems which respond rapidly and intelligently to user feedback and has as main goal to develop high-volume translation systems capable of adapting to user feedback in real-time. In particular, the data came from the Reverso.com translation web service [1].

The FAUST data consist of two files, *development* file and *test* file. We divided the information in two files in order to have examples for the training steps and others to test the developed systems through our experiments. The texts are bilingual, there are texts in English and also in Spanish[2]. Since we are using an English corpus with clean data, we only used the English side in our experiments. The data sets are available on-line in the official web page of the project[3] These files contain unnanotated data, so that it was necessary to do a manual annotation of the data. In the following section we will explain how it was done.

Regarding to the FAUST files, the *development* file consist of $1,217$ sentences ($11,377$ words) and we mainly use it in the training process of our experiments. The *test* file has $1,204$ sentences ($11,024$ words), we use this file in the testing phase of our experiments. However, as long as we do not have a large quantity of data, in some

[1] The web service is available in `http://www.reverso.net/text_translation.aspx?lang=ES`

[2] Although actually there are more examples in more languages available in the web site of the project, when we did our experiments there was only bilingual data.

[3] In particular, the data sets are publised in `http://www.faust-fp7.eu/faust/Main/DataReleases`.

experiments we treat these files as a whole entire file and use it in a 10-fold and 100-fold cross-validation processes to training and testing the tagger simulating a larger number of examples. We will specify in every experiment if we obtain the results by a cross-validation process or not.

### 3.2.1 Manual annotation

First of all, we made a quick look to the FAUST data files and one of the first things we observed is that many of the sentences introduced in the system by the users are not finished by a end-of-sentence mark, this is, by a colon '.', a question mark '?' or an exclamation mark '!'. Hence, the first step to adapt the data to our tool was to separate the sentences introducing an end–of–sentence mark '<s>' at the end of every sentence of the file, nevermind if it has or not a puntuaction mark of end of sentence. This is a necessary process because the tagger needs to know when and where a sentence ends to do the tagging process.

Once we had the sentences separated, we started a manual tagging process of the examples from both of the FAUST files, (*development* and *test* files.

Before we started the annotating process, we noticed that in the data there are several new punctuation marks that do not appear in the clean training file: '¿', '; '[', ']', etc. We treated some of them with an existing tag and we created a new tag called *OSYM* (other symbol) for those new ones. In particular, we tagged the square braquets with the tags for the parethesis, and we tagged the '>', ' ', '—', '*', and the composed puntuation marks ( '!?!?!?', '–¿', ':-)', '¡3' etc.) with the *OSYM* tag. When the word is a repetition of a particular punctuation mark, we tagged it with the same tag of the single symbol. For example, '!!!!' and '????' were taged as maks of end of sentence, '.....' and ',,,,' were tagged as ':', the tag of '...' and colon respectively.
We also observed that many words have attached at the beginning a punctuation mark. We did not change this kind of errors. we considered that words as it were single words ingnoring the symbols at the beginning to choose the tag. If the punctuation marks are attached at the end of the word, we only separate the symbol from the word if it is an end-of-sentence mark, a colon, a semicolon, a comma or ellipsis. In those cases, the symbols appear attached and indicate a change in the sentence so, we had to tokenize them to pass this information to the tagger. If we found a different symbol attached at the end of a word we maintained it in the same way as it appeared in the

original sentence and tagged it as we explained in the case where the symbols are at the beginning of the word. We did that in order to keep some representations of this kind of mistake to let the tagger learn it, after all, they have to be treated as a type of mispelling errors. For example, when we have '....you' we treated it in the same way as it was only the pronoun, or if we find in the input 'air*' or '-*Sales' or similar words we treat them as if they appeared in its single form: 'air' or 'Sales' in these particular examples. In Figure 3.1 we show more examples of these kind of problems.

We tagged the data by a semiauthomatic procedure. After the preprocessing steps that we have described before, we tagged the data with the SVMTagger using a model trained with the Wall Street Journal corpus, which will be our baseline/clean model. We used the Penn Tree Bank tagset A.1 extended with our new tag *OSYM*. Then, we obtained the FAUST data tagged by the SVMTool tagger.

The following step was to manually revise the corpus to correct the mistakes made by the tagger in order to obtain an annotated real data sample.
The cost in time of doing the manual annotating procedure was about a pair of weeks per file. In total we spent about a month in building the annotated corpus. Notice that the procedure did not take too many time because of the size of the files. The generated corpus is also available in the web to public usage.

We classify and analyze these and more kind of mistakes in the following section.

### 3.2.2 Analysis of the data: Frequent mistakes

During the process of manual annotation we observed many common mistakes made by the users and also by the tagger. We will describe now the errors we have seen and we will take them in account to develop the experiments to improve the SVMTool tagger and make it more robust.

As we said before, we are going to use the development file of the FAUST data in the training process of our experiments and the test file in the testing step. Hence, all the study of the mistakes are done using the development file. We did that in order to not bias the adaptation process towards the test set and obtaining unfairly high results.

One of the most common errors is related to the **capitalization** of the words. We can find words that the tagger knows but appear with a different capitalization and become unknown. For example, with a model trained with clean texts from the WSJ

**Figure 3.1:** Examples of sentences in the FAUST corpus. For every sentence, it is shown first the sentence as it appears in the corpus. Then, we show the well tokenized version of the sentence tagged by the basic SVMTagger. Finally, the manually annotated sentence is shown.

corpus, the tagger can know that “America” is a proper noun but it does not know what class “america” belongs to. Since this error is very common along the development file from the FAUST project, we will study models trained with uppercased or lowercased examples, or also designing an extension of the dictionary including input known words but for its capitalization.

Following with the data analysis, we find that there are many other unknown words for the tagger which are very common in the FAUST corpus like **URLs** from websites (www.facebook.com, yahoo.com, Google.com, xxxx@gmail.com, etc.). How can the tagger learn this words? one by one? We want to make the tagger learn this kind of words in order to make it able to tag them well. For example, we can expand the dictionary of the tagger by adding a list of the most common URLs. However, we observed that in the data also appear e-mail addresses, and we cannot make a list of “common e-mail addresses”. Instead of making a list, we can analyse the input in a previous step to the tagging in order to recognise somehow (using a regular expresion for example) if a word of the input is an URL, an e-mail address, etc., or not and in an affirmative case just add this word as a proper noun to the dictionary used by the tagger. This is one of the ideas that we will revisit and develop along the design of our experiments.

Furthermore, there are also common expressions like abbreviations which the users use a lot, like: “i.e”, “a.m.”, “a.k.a”, “u r”, “u”. And many **onomatopeias**: “ssss”, “hufff”, etc. These words, in the same way as in the case of the URLs, are unknown for a tagger trained using clean data, and a good idea is to follow the same strategy as the one we described for the URLs, just design a clever extension of the dictionary of the model used to tag by the SVMTagger looking at the word in the input.

We also found that a lot of words are unknown because they have a particle added at the end or at the beginning, usually a punctuation mark like “*”,“-”,“.”,“–”,etc. Moreover, it is usual that the user repeat a letter of a word to emphasize the meaning of it, like in ”‘slooooow”’ or “I loooove u”. An idea to deal with this kind of noise is to design a script that could recognize the known word without the added particle or the repeated letter, for example, just looking at a distance measure among words,

comparing the input words with the words known by the tagger.

Another common mistake made by the users are typos or misspellings in the words. We observed many confusions in the tagging task because there are many words in the input taken as unknown words because of the misspelling they have although they are, in fact, a known word. We have several ideas to deal with that kind of errors. One idea is to try to reproduce artifitially this mistakes and let the tagger learn from this artifitial data. Another one is to try to correct the misspellings in the input data looking at the dictionary of known words used by the tagger. This corrections can be made by using an algorithm that calculates the distance among words. We also can identify an error in a word looking for it in a list of common mistakes. So, we can identify the wrong word with a correct one and, at the end, add it to the dictionary. By doing this we can change that word in a known one for the system.

We also observed several other kinds of errors that could influence the performance of the tagger that are not related to the words forms. Instead of that, these errors are related to the structure of the sentences: word swapping, halted sentences, sentences that only are noun phrases (so, they do not have a verb with its information), etc. We will try to deal with this mistakes reproducing them artificially and train the tagger over the data with the artificial errors. We do not go deeper in that type of errors because, alghough they are very common, they do not have a significantly impact in the tagger's performance.

It's worth noting that the data we are using is not very large. And although it is representative of our use case (the framework of an on-line machine translation system), it is not enough to give us more clues about the data that can produce a common user on the Internet but, on the other hand, the ideas that we can extract from here will be good and useful.

# 4

# Using an artificially generated noisy training set

Through this section we present the first attempt to improve the performance of the *SVMTool* tagger over non-standard real texts. After the analysis of the errors in the *FAUST* corpus in the *Chapter* 3, we identified those ones that could be somehow replicated in an artificial way to create noisy training data to our tagger. Hence, our tool could learn from these mistakes and we expect that it would be able to generalize from them to the real errors observed in the FAUST corpus.

Finally, we show at the end of the section the results that we obtained through our experiments and after evaluating this approach using the FAUST test set.

## 4.1 Common error types

In *Chapter* 3 we presented an exhaustive analysis of the FAUST corpus that we will use to develop and test the robust tagger. Taking this analysis as an starting point, we gather the most relevant and usual mistakes, and also those ones that we can reproduce artificially are the following ones:

- **lc:** case-insensitive text (lack of capital letters)
- **uc:** upper cased text (all capital letters)
- **noEOS:** lack of end of sentence mark

- **noP:** lack of punctuation symbols
- **typos:** misspellings
- **ktypos:** keyboard-related misspellings
- **swap:** misspellings made by swapping characters of the words
- **ngrams:** sentence fragments
- **NP:** phrases made only of noun phrases
- **swapW:** swap contiguous words in a sentence

We tried to reproduce this mistakes by changing the WSJ corpus. In the following section is explained how these errors are reproduced.

## 4.2 Simulating errors

Our following step in this set of experiments is building a corpus that collect the most often mistakes we've seen in the samples of real data.

We want to use this data to train the SVMTool and see if we can learn from these modificated data in order to perform better over a non-standard input.
We tried to reproduce the different mistakes along the WSJ data. We modified the data from the WSJ corpus without changing the partition made in training, development and test data that we explained in *Chapter* 3 when detailing the WSJ corpus.

In general, all the mistakes are introduced in the WSJ data by means of a Python[1] module [2]. We chose to use Python because its facilities to deal with strings and also because it is quite fast in execution.

Now, we explain how we reproduced the errors one by one:

***Lowercased (lc):***
This error is easy to reproduce. All we want to have is text without any capital letter,

[1] `http://www.python.org/`
[2] We reported the code of this module in the digital version of this thesis

this is, lowercased words. Hence, all we did was generating a version of the WSJ without any capital letter (lowercased text).

***Uppercased (uc):***
This error is just the opposite to the previous. It is also easy to generate a version of the WSJ corpus with all the letters capitalized.

***No end of sentence marks (noEOS):***
As we said before, we observed in the FAUST data that usually users do not mark the end of a sentence. In order to reproduce this error, we removed the marks of end of sentence from the WSJ corpus. This is, removing the symbols *'.', '?' and '!'*. Instead of them, we put an end of sentece mark (`<s>`) to indicate to the tagger where a sentence ends and starts another but without any more semantic information. This is needed for the tagger to process the corpus sentence by sentence.

***No punctuation marks (noP):***
It is also very common that the users do not use punctuation in sensible way or even eliminate completely punctuation marks. In this case, we generated a new version of the WSJ corpus by removing all punctuation symbols [ *',!,",#, $, %, &, \, (, ), *, +, −, ., /, :, ;, ¡, =, ¿, ?, @,[,], ^,_,* {,} and ,], but maintaining the end of sentence mark (`<s>`) as we explained before.

***Misspellings (missp):***
Usually, users confuse letters when writting. Our first attempt to reproduce the typographical errors made by the users is to change randomly a character by another in a word. The second letter will be generated according to a previously fixed probability. In our experiments, we modified the WSJ data changing a character with a probability of 0.001, this is we change one of every thousand characters. We chose this probability since we observed through several experiments that represents well the usual distribution of typos in a real text.

***Typos taking into account the keyboard distribution and other more general typos (nktypos):***
Following a more realistic approach, we generated misspellings taking in account the

key distribution in the QWERTY keyboard. We simulated that by changing randomly in every word a character by another according to a prefixed probability, but we only changed a letter by other of the characters that are to the left or right of it in the keyboard. Going further with that idea, we introduced misspellings taking not only taking in account those keys to the left or right of the letter, also looking at those ones above and below the target letter. We also introduced typical typos like missing characters or swapping 2 contiguous characters or duplicate a letter, After several tests, we chose as a realistic probability of finding a word with a mistake to be one out of every 20 words. When we modify a word, we introduced only one typo in the word, chosen randomly among all the possible wrong spelling reachable from the original word.

***Swapping characters (swap):***

Among all the mispelling errors, one of the more common mistakes is swapping contiguous characters in a word. So, we want to test this case alone. This is why we consider this specific case.

We tried to reproduce this error swaping randomly two contiguous characters in a word acording to a fixed probability. After several tests, we considered that a realistic probability would be 1 swap out of 100 characters, this is, we swap two contiguous characters every 100 characters of the input.

***Sentence fragments (ngrams):***

As we noticed in the analysis of the data, many sentences are incomplete, like if they were halted, they are unfinished. To simulate this error, what we do was to introduce a mark of end of sentence (<s>) randomly, in order to fragment the sentences of the whole text.

***Noun Phrases (NP):***

Looking to the unfinished sentences more closely, we noticed that most of the syntactically correct ones respond to noun phrases. Taking this into account, we considered the parse trees of the Wall Street Journal to extract only the noun phrases of the corpus and build with them a new noun-phrase-corpus to train and test the SVMTagger. We also maintain here the distribution of the data in training, development and test sets.

***Swaping words (swapW):***
The last error we tried to reproduce is the one where the writer swap contiguous words of a sentence. We simulate this noise with a probability of make the swap among words. We chose to swap 1 out of 50 words.

## 4.3 Retraining the SVMTagger

After we built the different WSJ versions representing the different types of errors, we trained the SVMTagger using them. In short, we built the following models: *uppercased (uc)*, *lowercased (lc)*, *no end of sentences mark (noEOS)*, *no punctuation marks (noP)*, *random misspellings* with a probability of 0.001 (missp001), *mixture of typos* with a probability of 0.05 (nktypos05), *swapping contiguous characters* with a probability of 0.01 (swap01),*sentence fragments* (ngrams), *only noun phrases (NP)*, *swapping contiguous words* with a probability of 0.02 (swapW02).

We also built *extended tagging models* adding the clean WSJ text to the noisy training corpus in order to have robust models that perform well over clean and non-standard data. We build them concatenating the training section of the WSJ, which we modified introducing the error we want to study, to the clean training section of the WSJ. We will refer to those as *"clean+noise_name"* models.

Finally, we built *mixed tagging models* taking the clean training data from the WSJ and appending to it the training data from the most robust models representing errors. According to the results that we show in the following section, we chose to built two mixed models: one made from the clean corpus, the corpus with the no end of sentence noise and the corpus with the nktypos noise (*mixed1*) ; and another made from the clean corpus, the corpus with the nktypos noise and the corpus with the swapping word noise (*mixed2*).

In the following sections (Section4.4 and Section4.4.2) we show and discuss the results we obtained in the experiments.

## 4.4 Results using the new models with errors

First of all, we will present and analyze the results over the clean data.

### 4.4.1 Results on the WSJ

We took as baseline the results of the *SVMTagger* over the Wall Street Journal corpus.

The very first row of the Table4.1 shows our baseline result, the result of tagging the test set from the WSJ corpus (test *clean*) using the SVMTagger trained with WSJ corpus (model *baseline*). This results will be our upper bound results. We want to obtain a similar results in our experiments.

If we look at the first rows of the Table 4.1, the one for the lowercased (lc) experiments, we observe a droping in the overall accuracy of almost 10 points from the baseline when we tag the lowercased data with the baseline model(test *lc* data using model *baseline* row 2).
After training the SVMTagger with lowercased data (rows with model *lc*)we improve the accuracy over the FAUST texts (we achieve a 89.52% from 82.46% overall accuracy), but we want to perform better over clean data, where we only obtain a 78.72% of overall accuracy. So that, we think of training a combined model: *lowercase+clean*.
If we look at the Table 4.2, at the *lc+clean* rows, we can see that we recover almost all the overall accuracy over clean text (96.37%)and that the tool performs as well as it did with the noisy model over lowercased text (95.37%).

Dealing with uppercased texts (test *uc* and model *uc*), we observe a drooping in the overall accuracy of almost 10 points from the baseline when we tag the uppercased data with the baseline model, like if we noticed in the previous type of error.
After training the SVMTagger over uppercased data (*uc* model rows), we can see that we recover almost 8 points of overall accuracy over upercased data (from 57.88% to 96.27% of accuracy) and that we also notice that it do perform so bad over the clean model, achieving only 27.22% of accuracy.(Table 4.1)
In short, we can say that in this case we improve a lot over the data with errors. Trying to obtain better results, we follow the idea we used with the lowercased data: training the SVMTagger using a bigger set of training: the training examples from the WSJ

| Tagging Model | Test | known | unamb | amb. | unk. | overall | MFT | %unk.w. |
|---|---|---|---|---|---|---|---|---|
| **baselineWSJ** | **clean** | **97.32** | **99.42** | **93.04** | **88.46** | **97.07** | **91.30** | **2.81** |
| baselineWSJ | lc | 93.81 | 95.86 | 89.70 | 18.86 | **87.23** | 82.46 | 8.78 |
| lc | lc | 96.03 | 98.77 | 91.94 | 85.06 | **95.76** | 89.52 | 2.50 |
| lc | baselineWSJ | 96.61 | 99.50 | 92.23 | 65.94 | **91.77** | 78.72 | 15.78 |
| baselineWSJ | uc | 87.00 | 91.96 | 63.37 | 25.82 | **57.88** | 47.31 | 47.60 |
| uc | uc | 96.55 | 99.32 | 93.16 | 85.22 | **96.27** | 89.52 | 2.50 |
| uc | baselineWSJ | 96.54 | 99.92 | 90.69 | 12.42 | **27.22** | 16.89 | 83.01 |
| baselineWSJ | noEOS | 96.98 | 99.38 | 92.42 | 87.42 | **96.70** | 90.93 | 2.94 |
| noEOS | noEOS | 97.15 | 99.38 | 92.90 | 87.18 | **96.85** | 90.93 | 2.93 |
| noEOS | baselineWSJ | 97.14 | 99.38 | 92.89 | 35.24 | **92.83** | 87.15 | 6.97 |
| baselineWSJ | noP | 96.54 | 99.30 | 91.92 | 86.68 | **96.22** | 90.17 | 3.18 |
| noP | noP | 96.80 | 99.30 | 92.61 | 87.09 | **96.49** | 90.18 | 3.18 |
| noP | baselineWSJ | 96.79 | 99.30 | 92.58 | 21.62 | **86.00** | 79.77 | 14.35 |
| baselineWSJ | msp0.001 | 97.30 | 99.42 | 93.01 | 84.94 | **96.91** | 90.96 | 3.17 |
| msp0.001 | msp0.001 | 97.29 | 99.41 | 93.03 | 86.30 | **96.95** | 90.96 | 3.16 |
| msp0.001 | baselineWSJ | 97.31 | 99.42 | 93.05 | 87.94 | **97.04** | 91.29 | 2.81 |
| baselineWSJ | nktypos0.05 | 96.90 | 99.21 | 92.18 | 57.52 | **94.29** | 87.54 | 6.64 |
| nktypos0.05 | nktypos0.05 | 96.90 | 99.05 | 92.66 | 82.60 | **96.24** | 89.32 | 4.62 |
| nktypos0.05 | baselineWSJ | 97.29 | 99.38 | 93.18 | 86.77 | **96.99** | 91.25 | 2.85 |
| baselineWSJ | swap0.01 | 97.31 | 99.43 | 92.97 | 78.36 | **96.36** | 89.30 | 5.01 |
| swap001 | swap001 | 97.28 | 99.32 | 93.05 | 87.17 | **96.90** | 90.40 | 3.79 |
| swap001 | baselineWSJ | 97.30 | 99.41 | 93.00 | 88.32 | **97.04** | 91.25 | 2.87 |
| baselineWSJ | swapW0.02 | 97.06 | 99.42 | 92.25 | 87.67 | **96.79** | 91.30 | 2.81 |
| swapW002 | swapW002 | 97.18 | 99.42 | 92.62 | 87.67 | **96.91** | 91.30 | 2.81 |
| swapW002 | baselineWSJ | 97.31 | 99.42 | 93.03 | 88.08 | **97.05** | 91.30 | 2.81 |
| baselineWSJ | ng0.15 | 96.98 | 99.38 | 92.42 | 87.42 | **96.70** | 90.93 | 2.94 |
| ng0.15 | ng0.15 | 97.15 | 99.38 | 92.90 | 87.18 | **96.85** | 90.93 | 2.94 |
| ng0.15 | baselineWSJ | 97.14 | 99.38 | 92.89 | 35.24 | **92.83** | 87.15 | 6.97 |
| baselineWSJ | NounPhrases | 95.94 | 99.39 | 88.66 | 86.73 | **95.59** | 91.54 | 3.79 |
| NounPhrases | NounPhrases | 97.05 | 99.35 | 91.34 | 88.83 | **96.72** | 91.82 | 3.97 |
| NounPhrases | baselineWSJ | 95.96 | 98.94 | 88.89 | 66.16 | **94.72** | 89.26 | 4.16 |

**Table 4.1:** Results of the models simulating mistakes over WSJ data, i.e., models trained with WSJ modified data to simulate errors. The first column represents the input of the tagger. The second is the tagging model used in the experiment. Then, the central columns are related to the accuracy over different kind of words attending to its ambiguity. The *known* column shows the accuracy achieved over known words. The *unamb.* column is the accuracy over unambiguous known words and the *amb.* column is the accuracy over ambiguous known words. The *unk.* column is the accuracy achieved over unknonw words. Finally, in the *overall* column we see the overall accuracy achieved through the experiment. The *MFT* column shows the results of making the tagging using the most frequent tagg approach. The last column, the *unk.w.* column shows the percentage of unknown words in every experiment.

| Tagging Model | Test | known | unamb | amb. | unk. | overall | MFT | unk.w. |
|---|---|---|---|---|---|---|---|---|
| **clean** | **baseline** | **97.32** | **99.42** | **93.04** | **88.46** | **97.07** | **91.30** | **2.81** |
| lc+clean | clean | 97.03 | 99.36 | 93.30 | 73.24 | **96.37** | 91.18 | 2.77 |
| lc+clean | lc | 95.93 | 98.77 | 91.69 | 73.46 | **95.37** | 89.39 | 2.50 |
| uc+clean | clean | 96.62 | 99.02 | 92.36 | 80.73 | **96.18** | 91.32 | 2.77 |
| uc+clean | uc | 96.48 | 99.30 | 93.15 | 84.54 | **96.18** | 89.52 | 2.50 |
| noEOS_ext+clean | clean | 97.26 | 99.42 | 92.89 | 87.39 | **96.99** | 91.30 | 2.81 |
| noEOS_ext+clean | noEOS | 97.14 | 99.38 | 92.90 | 87.18 | **96.85** | 90.93 | 2.94 |
| noP_ext+clean | clean | 97.19 | 99.42 | 92.65 | 84.60 | **96.82** | 91.22 | 2.90 |
| noP_ext+clean | noP | 96.80 | 99.30 | 92.61 | 87.09 | **96.49** | 90.17 | 3.18 |
| msp0.001+clean | clean | 97.30 | 99.41 | 93.08 | 84.17 | **96.93** | 91.30 | 2.80 |
| msp0.001+clean | msp0.001 | 97.16 | 99.3 | 92.81 | 82.50 | **96.31** | 88.46 | 5.80 |
| swap001+clean | clean | 97.31 | 99.42 | 93.01 | 85.99 | **96.99** | 91.31 | 2.81 |
| swap001+clean | swap001 | 97.28 | 99.33 | 93.04 | 85.24 | **96.83** | 90.45 | 3.73 |
| nktypos+clean | clean | 97.31 | 99.40 | 93.22 | 82.40 | **96.89** | 91.31 | 2.80 |
| nktypos+clean | nktypos | 96.90 | 99.05 | 92.66 | 79.29 | **96.10** | 89.38 | 4.57 |
| ngrams+clean | clean | 97.14 | 99.38 | 92.89 | 35.24 | **92.83** | 87.15 | 6.97 |
| ngrams+clean | ngrams | 97.15 | 99.38 | 92.90 | 87.18 | **96.85** | 90.93 | 2.94 |
| NP+clean | clean | 97.10 | 99.42 | 92.36 | 48.29 | **95.72** | 91.23 | 2.81 |
| NP+clean | NP | 97.01 | 99.39 | 91.93 | 43.03 | **94.96** | 91.82 | 3.79 |

**Table 4.2:** Results of the extended models simulating mistakes over WSJ data, i.e., models trained with WSJ clean data jointly with modified data to simulate errors. Every row of the table represents a tagging scenario using the tagging model to tag the Input. The columns have the same meaning as in the Table 4.1.

in its original form concatenated to the uppercased texts used to build the uppercased model. With this model we obtain a significantly better results because we achieve a great accuracy over clean data as well as when tagging the modified data, a 96.18%, ( Table4.2).

Regarding to the error of the lack of end of sentence mark (test *noEOS* and model *noEOS*), we observe in the Table 4.1 that the accuracy drops a little when we tag the noisy test text using the baseline model (96.70% ). We also see that we recover a little of the overall accuracy by training the SVMTagger with the artificial noisy data (we achieve a 96.85% of accuracy).

Nevertheless, we notice that tagging clean texts with this noisy model we obtain a low accuracy over unknown words and that the percentage of unknown words increase. That is because we do not have the end of sentence marks as known words. So that, in this case could be useful to extend the dictionary adding this marks. We do not lose much in accuracy (96.70%) with respect to the baseline model (97.08%).(Table 4.1)

If we use also clean data also to train the SVMTagger, as we can see in the Table 4.2 in the rows *noEOS+clean*, the SVMTagger is able to perform better over well formed data (96.98%) and even better over the modified data (96.85%). So that, in this case it seems that it is useful to train the tagger also using modified data.

Studying the lack of punctuation marks (test *noP* and model *noP* rows), we see that the overall accuracy drops a little when we tag noisy data with the model trained with clean data (96.22%).

As well as in the no-end-of-sentence model, we observe that the accuracy over unknown words when tagging clean data drops quite a lot so, to improve the accuracy, here could be interesting introducing the punctuation marks in the dictionary. Regarding to the overall accuracy, we obtain a 96.49%, so we can recover a little bit of the lost accuracy with this model (Table 4.1).

It is interesting to analyze the behaviour of the tagger trained using also clean texts or extending the dictionary with the punctuation marks. If we look at the Table4.2, we can see that in this case the SVMTagger improve its accuracy until achieve a 96.82% of accuracy tagging clean data and maintain a good performance over the modified corpus (96.49%).

We move on to the analysis of the models which deal with the different kind of misspelling errors. We are going to analyze now the results obtained with the several

models we developed that try to reproduce the different typographical errors observed in the FAUST corpus.

First of all, we start studying the error that reproduce random misspellings. We see the results in the rows with the *msp0.001* in the test or model column. The data we are going to use have a random typo introduced every 1000 characters of the input. Tagging noisy data with the baseline system we see that the accuracy drops to 96.91%. It happens because we introduce not very much noise in the data in order to be as realistic as we can when simulating the errors.
After training the SVMTagger over this kind of data, we observe that the tool recover almost all the accuracy lost over noisy text and we don't lose a signifiant accuracy over clean data obtaining 97.04 and 96.95 overall accuracies respectively.

Looking at Table 4.2, we can see that the overall accuracy is more or less the same as the one achieved with the previous model. This happens because we changed very little the WSJ corpus, only one character every 1000. So that, it does not make great differences to attach clean data to the modified training set.

Following the idea of increasing the realistic character in the simulation of the errors, we analyze the experiments with the modified data using a mixture of possible and commons typos: the *nktypos* scenario.
The overall accuracy drops 3 points when dealing with modified texts representing this noise (96.99% of accuracy tagging the modified test data using the baseline model). After training the SVMTagger over data modified introducing the *nktypos* errors with a probability of 0.05 of changing a word, we observe that the tool recover the accuracy lost over noisy text (from 87.11 to 96.24) and we do not lose a signifiant accuracy over clean texts (from 97.07 to 96.99).
Once again, we improve the results over the corpus with the artifitial errors, but we want to perform better over clean examples and be able to achieve at least a 97% of accuracy.(Table 4.1).

Looking at the results of the extended model in the Table4.2, we can see in this case a robust behaviour of the tool. It achieves a 96.89% of accuracy tagging clean examples and a 96.10% of accuracy over the modified data. Once again, we observe that it is useful to use jointly modified clean data to train the tagger, controlling better the quantity of unknown words and also helping the tagger to deal with them and, in consecuence, obtaining better results.

We study one of the most common typos observed in the real data, swapping two contiguous characters in a word. Here we observe that the overall accuracy drops almost a whole point when we try to tag this kind of noisy examples with the baseline system (96.36%). We observe that the tool recover the lost accuracy over texts with errors (from 96.36 to 96.90) and we do not lose a significant accuracy over clean data (from 97.07 to 97.04). In this scenario, we obtain a quite good performance of the tool and maintaining at the same time a good performance over well formed data. This is the kind of behaviour we expect and hope to obtain in our tool after our experiments. Nevertheless, we cannot forget that here we are testing the tagger only with data modified artificially by our Python module and not using actual real data.
Once again, we obtain a robust behaviour of the tool. In this case, we obtain a 96.99% of the accuracy when tagging clean data and 96.83% when tagging modified data. (Table4.2).

In general we can see that, regarding the models that deal with misspelling errors, and particularly when we talk of its extended version, we recover almost all the overall accuracy over clean text and that the tool performs as well as it did with the noisy model over the modified data achieving also a good accuracy between 96% and 97% in most of the cases.

We continue analyzing the modification of the data that focuses on a word or set of words as unity of change and not on a letter.

We start the analysis of this set of experiments with the error of swaping words. In Table4.1 rows *swapW002*, we observe a little dropping of the accuracy when tagging noisy data with the baseline system (96.79%). After training the SVMTagger over text with swapped words generated randomly with a probability of 0.02, this is, 1 out of 50 words are swapped, we recover almost all the overall accuracy (it increases to 96.91) and it doesn't lose a significant accuracy over clean data (97.05%).(Table 4.1).

We try to reproduce the incomplete sentences of the users creating *ngrams*. In this experiment, we can see how the overall accuracy drops when using the baseline system to tag the modified data. This fact shows the influence of the end of sentence mark in the process of tagging and the importance of the context in tagging time.
We observe that the tool recover part of the accuracy lost over noisy text (from 96.70 to 96.85) although we lost a signifiant accuracy over clean data (92.83%).

The following step with this kind of mistakes is training the tagger using also clean examples, as we did in the previous experiments. In 4.2 we observe that, although we obtain the same results over the clean text, we gain a little bit when dealing with the modified corpus. That is because we have now more general information of the data because the tagger have seen all the data in training time.

However, we think that it would be more interesting doing this kind of noise attending to the grammatical meaning of the phragments: noun phrases, prepositional clauses, etc., which are more common in the FAUST corpus than random parts of sentences.

The first step after studying the random partition of the sentences in a text is considering only the most frequent fragments with grammatical meaning: *the noun phrases*. We considered the parse trees of the Wall Street Journal corpus to extract only the noun phrases of the text and built with them a new noun-phrase-corpus to train and test the SVMTagger.
We observe here that the tool recover the accuracy lost over noisy text (from 95.59 to 96.72) although we lost a signifiant accuracy over clean data, it drops until 94.72.
We want to perform better over incomplete sentences without losing accuracy over clean text.
If we use also the well formed examples, as we can see in Table 4.2, we recover a whole point of accuracy when dealing with clean texts (95.72%) but we lost in accuracy over modified data (95.72%). This happens because we gain the information about the verbs and the verbal forms when introducing all the data in training time but we lose the particular information we had about the noun phrases.

### 4.4.2 Mixed corpus

Our next step working with artificial errors is building mixed tagging models using those kind of mistakes that are able to recover more overall accuracy when tagging data, this is, have a more robust behaviour. The most promising cases.

We built two mixed corpora. The first one (*mixed model*), what we did was building a new corpus concatenating the files of the corpus of the clean corpus (*baseline*), the corpus with a mixture of misspelling errors (*nktypos005*) and the corpus without end of sentence marks (*noEOS*). Finally, the proportion of every corpus in the new one is

| Tagging Model | Input | known | unamb | amb. | unk. | overall | MFT | unk.w. |
|---|---|---|---|---|---|---|---|---|
| mixed | clean | 97.32 | **99.41** | **93.19** | **81.78** | **96.89** | **91.31** | **2.80** |
| mixed | lc | 93.81 | 95.92 | 89.68 | 60.50 | **91.09** | 82.94 | 8.17 |
| **mixed** | **uc** | 87.56 | 91.95 | 66.88 | 25.28 | **57.91** | 47.32 | 7.60 |
| mixed | noEOS | 97.10 | 99.37 | 92.91 | 80.71 | **96.63** | 90.94 | 2.92 |
| mixed | noP | 96.70 | 99.28 | 92.48 | 79.31 | **96.15** | 90.19 | 3.17 |
| mixed | missp-0.001 | 97.31 | 99.41 | 93.15 | 80.27 | **96.77** | 90.98 | 3.15 |
| mixed | nktypos0.05 | 96.89 | 99.05 | 92.57 | 76.45 | **95.91** | 89.15 | 4.83 |
| mixed | swap001 | 97.30 | 99.38 | 93.13 | 78.69 | **96.44** | 89.64 | 4.63 |
| mixed | ngrams-0.15 | 97.11 | 99.37 | 92.91 | 80.71 | **96.63** | 90.94 | 2.92 |
| mixed | NounPhrases | 96.16 | 99.37 | 89.58 | 80.16 | **95.55** | 91.56 | 3.77 |
| mixed | swapW002 | 97.05 | 99.41 | 92.39 | 80.79 | **96.60** | 91.31 | 2.80 |

**Table 4.3:** Resuls of the first mixed model (trained with clean data and those simulating the noEOS and the nktypos005 errors). Every row of the table represents a tagging scenario using the tagging model to tag the Input. The columns have the same meaning as in the Table 4.1.

the following: 50% is taken from the *clean* corpus, 25% from the *noEOS* corpus and the remaining 25% from the *nktypos005* corpus. To build the files we just concatenate to the *clean* files the first half of the *noEOS* files and finally the second half of the *nktypos005* files. The second one (*mixed model2*)) was built concatenating the files of the corpus of the *clean* corpus (*baseline*), the corpus with a mixture of misspelling errors (*nktypos005*) and the corpus with swapping among words (*swapW002*). Finally, the proportion of every corpus in the new one is the following: 50% is taken from the *clean* corpus, 25% from the *nktypos005* corpus and the remaining 25% from the *swapW002* corpus. To build the files we just concatenate to the *clean* files the first half of the *nktypos005* files and finally the second half of the *swapW002* files.

In the Table 4.3 and Table 4.4 we summarize the results that we obtained using those new corpora by tagging the different test data modified with the different kind of artificial noises.

We notice that the most of the errors are well managed by the mixed models, with the exception of the lowercased and uppercased errors. These ones do not obtain a good performance when tagging with this mixed models because they are not represented in their training data.

| Tagging Model | Input | known | unamb | amb. | unk. | overall | MFT | unk.w. |
|---|---|---|---|---|---|---|---|---|
| mixed2 | baselineWSJ | 97.30 | 99.40 | 93.17 | 81.58 | **96.86** | 91.31 | 2.80 |
| mixed2 | lc | 93.73 | 95.88 | 89.58 | 62.67 | **91.11** | 82.58 | 8.45 |
| mixed2 | uc | 87.56 | 91.95 | 66.88 | 25.28 | **57.91** | 47.32 | 47.60 |
| mixed2 | noEOS | 96.97 | 99.36 | 92.57 | 80.95 | **96.50** | 90.93 | 2.92 |
| mixed2 | noP | 96.52 | 99.26 | 92.07 | 79.79 | **95.99** | 90.18 | 3.17 |
| mixed2 | missp-0.001 | 97.28 | 99.39 | 93.12 | 80.30 | **96.75** | 90.98 | 3.15 |
| mixed2 | nktypos0.05 | 96.92 | 99.07 | 92.64 | 76.77 | **95.94** | 89.15 | 4.84 |
| mixed2 | swap001 | 97.28 | 99.37 | 93.11 | 78.38 | **96.41** | 89.63 | 4.62 |
| mixed2 | ngrams-0.15 | 96.97 | 99.36 | 92.57 | 80.95 | **96.50** | 90.93 | 2.92 |
| mixed2 | NounPhrases | 96.01 | 99.36 | 89.19 | 78.76 | **95.36** | 91.55 | 3.78 |
| mixed2 | swapW002 | 97.11 | 99.40 | 92.61 | 80.53 | **96.65** | 91.31 | 2.80 |

**Table 4.4:** Resuls of the second mixed model (trained with clean data and those simulating the nktypos005 and de swapping words (swapW002) errors). Every row of the table represents a tagging scenario using the tagging model to tag the Input. The columns have the same meaning as in the Table 4.1.

| Test | Model | known | unamb | amb. | unk. | overall | MFT | unk.w. |
|---|---|---|---|---|---|---|---|---|
| **clean** | **baseline** | **97.32** | **99.42** | **93.04** | **88.46** | **97.07** | **91.30** | **2.81** |
| **faust_test** | **baseline** | 93.85 | 96.11 | 89.57 | 52.09 | **88.41** | 78.20 | 13.05 |

**Table 4.5:** Results of tagging FAUST project's data with our baseline SVMTagger. Every row of the table represents a tagging scenario using the tagging model to tag the Input. The columns have the same meaning as in the Table 4.1.

## 4.5 Results on the FAUST data

A natural question to address is how the SVMTagger performes over real noisy FAUST texts.

We tested the SVMTagger using the FAUST test corpus described in *Chapter* 3. We can see in the Table 4.5 the results we obtained by tagging the test file using the baseline model.

We realized that the overall accuracy drops from 97.07% to 88.41% significantly.

Now we want to see how the best robust tagging models trained on corpora with artificial noise (Sections ) perform on the FAUST corpus and whether they allow to recover from the baseline performance accuracy 88.41%.

| Tagging Model | Input | known | unamb | amb. | unk. | overall | MFT | %unk.w. |
|---|---|---|---|---|---|---|---|---|
| baselineWSJ | faust_test | 93.85 | 96.11 | 89.57 | 52.09 | **88.41** | 78.20 | 13.05 |
| lc | test_faust | 93.82 | 98.88 | 87.33 | 34.17 | **80.80** | 71.17 | 21.84 |
| uc | test_faust | 96.53 | 98.89 | 73.02 | 5.97 | **17.20** | 11.99 | 87.60 |
| noEOS | test_faust | 93.90 | 96.14 | 89.90 | 41.35 | **84.95** | 74.30 | 17.03 |
| noP | test_faust | 93.37 | 95.83 | 89.33 | 32.53 | **80.25** | 69.83 | 21.56 |
| missp-0.001 | test_faust | 94.12 | 96.34 | 89.92 | 53.66 | **88.88** | 78.33 | 12.97 |
| nktypos0.05 | test_faust | 93.87 | 97.67 | 87.37 | 52.48 | **88.72** | 78.43 | 12.44 |
| swap001 | test_faust | 94.21 | 96.37 | 90.04 | 52.91 | **88.70** | 78.19 | 13.10 |
| swapW002 | test_faust | 94.06 | 96.41 | 89.54 | 53.75 | **88.80** | 78.29 | 13.03 |
| ngrams-0.15 | test_faust | 82.34 | 94.98 | 59.77 | 13.31 | **70.59** | 74.30 | 17.03 |
| NounPhrases | test_faust | 91.80 | 95.48 | 83.95 | 47.15 | **85.00** | 75.13 | 15.24 |
| mixed | test_faust | 94.12 | 97.69 | 87.95 | 46.38 | **88.16** | 78.52 | 12.48 |
| mixed2 | test_faust | 93.98 | 97.65 | 87.67 | 46.11 | **88.03** | 78.53 | 12.43 |

**Table 4.6:** Results of tagging FAUST project's data with the different taggers trained with data simulating errors. Every row of the table represents a tagging scenario using the tagging model to tag the Input. The columns have the same meaning as in the Table 4.1.

In the Table 4.6 we can see the goodness of our artificial models dealing with real noisy data. In general, our artificial models do improve the results of the WSJ trained version of the tagger. This fact indicates that we are not reproducing very well the errors we observed in the FAUST data or the observed errors are not replicated in the test data.

## 4.6 Conclusions

We can conclude from the results that our approach with artificially created noisy training corpus do not allow to improve significantly over the results obtained by the original tagger. Probably, because of the inability to reproduce the actual errors and text type of the FAUST corpus, or proportion of phenomena observed. Therefore, we can assume that in order to think in other new strategies to improve the results we must observe the errors made by the baseline model of our tagger in tagging the FAUST corpus and not the particularities of this corpus.

We thought of different strategies to deal with non-standard texts, explained in *Chapters*ñefguessingmodule and 6. The first one consisted of training the SVMT using

the small development set from the FAUST project. Another one was to modify the dictionary of the original SVMTagger by means of rules and heuristics.

# 5

# Domain Adaptation without Retraining

We want to learn from the non-standard input data. In particular, from the texts from the *FAUST project*. We tried to do this by different ways. We developed two different type of approaches to make domain adaptation: doing adaptation without a training process or with it.

Through this section we explain the approach that develops the idea of introducing a guessing module in the SVMTagger pipeline to modify the dictionary. In that way we allow the tagger to learn from the unknown words found in the input.

In the *Chapter* 6 we will describe the strategy that uses the retraining process.

## 5.1 Adapting the dictionary to the input: designing the guessing module

Taking as an starting point the analysis of the errors found in the FAUST project files, we designed a module that implements several heuristics in order to improve the performance of the tagger. The guessing module pursues its goal just enhancing the dictionary of the model used in the tagging process, without the necessity of a training

step.

The first attempt of this module was only taking into account the capitalization problem. Given a new word in the input ,we look for a new version of this word in the dictionary of the model we are using to tag. This is, we try to find first the word without any change; if we do not find it (is an unknown word), we look for a lowercased version of the word. If we do not find the lowercased version of the word, we look for the uppercased version of the word. And if we do not find the uppercased version of the word, then we look for the capitalized version of the word. Finally, if we do not find this last version of the word we do not do anything. In that case we do not introduce any new information of the target word in the dictionary so, the word remains unknown to the tagger. If we find any version of the word in the dictionary (lowercased, uppercased or capitalized), we add the word in the dictionary attaching to the word the part of speech tags associated to the version of it that we found in the dictonary,

In this point, we observed that we lose many time looking for uppercased and lowercased versions of punctuation marks. Our next step was modifying the guessing module in order to, after checking if a word from the input is a known word, we check if it is a punctuation mark or a concatenation of punctuation marks. If it is a simple punctuation mark we associate to it its corresponding PoS tag in the Penn Treebank tagset. If the word is a repeated punctuation mark we associate to it the PoS tag corresponding to the single punctuation mark that appear in the word (i.e.: "!!!" we tag it as "!"). If the word is a concatenation of punctuation marks we tag it using a new tag: *OSYM*. We have explained this in more detail in *Chapter* 3.

We implemented a Levenshtein's distance function. If any of our previous assumptions over the word does not work, we look for reasonable candidates to be a known version of the word. We think of generate candidates which are similar to the target word.

In order to find good candidates we implemented a *diagonal* algorithm. In this algorithm, we prioritize the searching of candidates among the known words with the same length of the input word $+-1$ character since we want to find the set of words with the minimum distance to the input word.

The first attempt of this diagonal algorithm only took the first word of the set of candidates returned by the guessing module, take its PoS tags and add the target word to

the dictionary adding only the PoS tags of that word. Afterwards, we observed that there were many other candidates with the same minimum distance to the target word. So that, we decided to combine all the PoS information of those "minimum candidates" and introduced the input word to the dictionary with a combined PoS information from the PoS tags of all these candidates.

In this point, we observed several things by looking at the list of candidates we obtained applying our algorithm:

1. Usually, the tagger fails tagging the webpages URL's and the e-mail addresses as NNP.

2. Some well-written words are very near to another known words (words in the dictionary). That will introduce noise in the dictionary.

3. We made some research and saw that some mistakes appear frequently also in other texts.

4. It could be a good idea to perform a ponderated distance function to give more or less weight to a mistake attending to the probability of appearance.

Taking into account this observations, we considered a good idea to use the Peter Norvig's list of errors (`http://norvig.com/ngrams/`). This list is a collection of misspelling errors collected from Wikipedia and Roger Mitton (`http://www.dcs.bbk.ac.uk/~ROGER/corpora.html`).
For every input word, we check if it is one of the common errors. If the target word appears in the list, we try to find the related corrected word in the dictionary. If the correct words is a known word, we attach the PoS tags of the word in the list to the input word.
If all the suggestions are unknown words we do not add the input word to the dictionary to let the tagger guess its PoS tag(s) by itself and avoid the introduction of noise to the system.

Before checking if the word appears in the list of common errors, we expand the dictionary from the model by adding a list of common words and abbreviations. So that, we will treat them as known words if we find them in the input.

We also check if the input word is an URL, particularly a webpage or an e-mail address. If the input word starts by "http","www.", if it contains one of the most common e-mail servers extensions: "@hotmail.", "@gmail." or "@yahoo.", or if the word ends in one of the most common extensions: ".es", ".org", ".com", ".gov", we understand that the target word is an URL and we add it at the dictionary with the "NNP" tag.

Finally, we did not change every single unknown word of the input. We introduced a threshold in the distance in order to avoid introducing too many noise in the systen. So that, we tested two threshold versions. One only considered candidates with a distance of 2 or less to the target word (results of $dist\_th2$ models). Another with distance 1 (results of $dist_th1$ models).

Summing up, the pipeline of our guessing module is as follows:
Given a word in the input:

1. Look for the word in the dictionary of the model, extended with the list of common expresions. If it is a known word we go on with the next word. If it is unknown, we go on with step 2.

2. Look for the word in the list of most common errors of Peter Norvig. If the word is there, we try to find the related suggestion to it in our dictionary. If the suggestion is a known word, we attach the PoS tags of the suggested word to the input word and go on with the next input word. If the suggested word is not in the dictionary (also in its lowercased and uppercased version) we pass to the next input word.

3. If the word is a punctuation mark, we add to the dictionary the information we know about it like we explained before and we go on with the next word in the input. If it is not a punctuation mark, we go on with the step 3.

4. We check if the input word is an URL. If it is an URL we add it to the dictionary as an NNP. Otherwise, we go on with the step 4.

5. We look for an uppercased version of the word in the dictionary. If we find it, we add the target word to the dictionary with the related PoS tags and we go on with a new word. Otherwise, we go on with the next step.

6. We look for a lowercased version of the word in the dictionary. IIf we find it, we add the target word to the dictionary with the related PoS tags and we go on with a new word. Otherwise, we go on with the next step.

7. We look for a capitalized version of the word in the dictionary. If we find it, we add the target word to the dictionary with the related PoS tags and we go on with a new word. Otherwise, we go on with the next step.

8. We apply the diagonal-Levenshtein's-distance algorithm to the target word.

## 5.2 Applying the guessing module

Throught this section we will show and discuss the results obtained adding our guessing module to the pipeline of the SVMTagger. We study the experiments in an incremental way. Analyazing the influence of every developed heuristic in the improvement of the tagger when dealing with the FAUST corpus.

We started our experiments focusing only on the capitalization problem. We observe in Table 5.1 (dit_mays rows)that with this method we can recover over 2 points of accuracy without losing too much time in the preprocessing step.

Studying the influence of the punctuation marks in the performance of the tagger we observed that only introducing in the dictionary the new punctuation marks unknown for the model we can recover two points of accuracy (Table 5.1 *distPunct* rows.) up to 89.18%. This improvement comes from the fact that we are reducing the set of unknown words and because of the inambiguity of the punctuation marks.

If we only treat the words that we recognize as URL, we see the following improvements in the accuracy of the tagger (Table 5.1 (*dis_tURL* rows). Although it is not an important improvement, we observed that we substitute 8 words in that process, and only with those little changes we can recover some of the accuracy lost before (from 88.41 to 88.45).

We observed that many times the tagger does not understand well the numercial expressions. Although this is not a common mistake we thought that would be a good

| Tagging Model | Input | known | unamb | amb. | unk. | overall | MFT | unk.w. |
|---|---|---|---|---|---|---|---|---|
| **baselineWSJ** | **clean** | **97.32** | **99.42** | **93.04** | **88.46** | **97.07** | **91.30** | **2.81** |
| baselineWSJ | test_faust | 93.85 | 96.11 | 89.57 | 52.09 | **88.41** | 78.20 | 13.05 |
| dist_mays | test_faust | 92.47 | 94.48 | 88.60 | 58.30 | **89.16** | 80.08 | 9.68 |
| dist_faust_mays | test_faust | 93.61 | 96.36 | 88.84 | 60.02 | **91.01** | 82.74 | 7.74 |
| dist_punct | test_faust | 92.92 | 94.81 | 89.21 | 58.00 | **89.18** | 79.75 | 10.72 |
| dist_faust_punct | test_faust | 94.71 | 97.62 | 89.73 | 52.26 | **90.27** | 81.15 | 10.45 |
| dist_URL | test_faust | 93.86 | 96.12 | 89.57 | 52.17 | **88.45** | 78.27 | 12.97 |
| dist_faust_URL | test_faust | 94.81 | 97.76 | 89.76 | 51.55 | **90.24** | 81.12 | 10.56 |
| dist_Num | test_faust | 93.72 | 95.88 | 89.60 | 51.62 | **88.42** | 78.54 | 12.58 |
| dist_faust_Num | test_faust | 94.80 | 97.76 | 89.76 | 51.62 | **90.21** | 81.05 | 10.63 |
| dist_Common | test_faust | 93.85 | 96.06 | 89.62 | 52.78 | **88.62** | 78.51 | 12.72 |
| dist_faust_Common | test_faust | 94.80 | 97.76 | 89.76 | 51.62 | **90.21** | 81.05 | 10.63 |
| dist_WordList | test_faust | 93.43 | 95.77 | 89.01 | 52.28 | **88.28** | 78.32 | 12.51 |
| dist_faust_WordList | test_faust | 91.72 | 95.32 | 85.59 | 58.36 | **88.28** | 81.16 | 10.31 |

**Table 5.1:** Results applying the guessing module implementing only the capitalization heuristic. Every row of the table represents a tagging scenario using the tagging model to tag the Input. The first column represents the input of the tagger. The second is the tagging model used in the experiment. Then, the central columns are related to the accuracy over different kind of words attending to its ambiguity. The *known* column shows the accuracy achieved over known words. The *unamb.* column is the accuracy over unambiguous known words and the *amb.* column is the accuracy over ambiguous known words. The *unk.* column is the accuracy achieved over unknonw words. Finally, in the *overall* column we see the overall accuracy achieved through the experiment. The *MFT* column shows the results of making the tagging using the most frequent tagg approach. The last column, the *unk.w.* column shows the percentage of unknown words in every experiment.

idea to deal separately with this kind of expressions. In this case, we changed 3941 words and we cannot see an improvement in the obtained accuracy (Table 5.1 *dist_Num* rows). The tagger makes error so unfrequently in this kind of expressions that it is not worthy to deal with them separately. For this reason, we don not consider this heuristic when we put all the heuristics together.

After the study of the FAUST files, we extracted a list of some common expressions that we cannot see in the WSJ corpus. Besides, we also investigated over the Internet to find some commonly used expressions and abbreviations. We used this list adding it to the dictionary of the model used to tag the input.

The list of words is in the appendix A.3.2. In the results we can observe that we do not improve the performance (Table 5.1 *dist_Common* rows), indeed we lost accuracy, but we consider interesting to maintain this set of common words to enhance the dictionary of a the tagger because it could be very useful depending on the nature of the input presented to the system.

One of the most common mistakes that we observed in the FAUST data were the misspelling errors. We tried to deal with them looking at a list of most common mistakes made by writters: the Peter Norvig's word list. In the Table 5.1 *dist_WordList* rows we can see the results obtained by looking at this list to decide if an input word has a misspelling and substitute it by its correct form.

We observe how the performance of the tagger does not improve. That is because there are many correct words that could be unknown to the tagger but not necessarily because they have a typo. For example, if we have the word "borther" in the input and identify it as an unknown word, we can say that a corrected version of the word is "brother". This kind of assumptions do not have any sense in general and introduce a lot of noise in the system. Hence, we do not include this heuristic in our final method to deal with the non standard input.

After those experiments, we integrated all the heuristics together and observed the improvement of the tagger when all of them work together. The Table 5.2 summarizes the results of these experiments and shows how changes the accuracy when we put incrementally all the heuristics working together. The results are shown first using the guessing module before using baseline system and then before using the tagger trained also using the FAUST corpus (as we explain in the *Chapter* 6).
We notice that we obtain the best results with the SVMTagger trained using FAUST

| Tagging Model | Input | known | unamb | amb. | unk. | overall | MFT | unk.w. |
|---|---|---|---|---|---|---|---|---|
| **clean** | **baseline** | **97.32** | **99.42** | **93.04** | **88.46** | **97.07** | **91.30** | **2.81** |
| **test_faust** | **baseline** | 93.85 | 96.11 | 89.57 | 52.09 | **88.41** | 78.20 | 13.05 |
| common_mays | test_faust | 92.56 | 94.59 | 88.64 | 58.89 | **89.38** | 80.39 | 9.45 |
| common_punct | test_faust | 93.58 | 95.54 | 89.76 | 57.00 | **89.29** | 79.18 | 11.73 |
| common_mays_punct | test_faust | 93.07 | 95.03 | 89.24 | 58.26 | **89.34** | 79.75 | 10.72 |
| common_mays_URL | test_faust | 92.56 | 94.59 | 88.64 | 59.05 | **89.42** | 80.46 | 9.37 |
| common_punct_URL | test_faust | 93.58 | 95.54 | 89.76 | 57.12 | **89.33** | 79.25 | 11.66 |
| mays_URL_punct | test_faust | 92.32 | 94.19 | 88.65 | 65.05 | **89.97** | 80.90 | 8.62 |
| common_mays_punct_URL | test_faust | 92.40 | 94.29 | 88.70 | 65.91 | **90.18** | 81.21 | 8.38 |
| common_WordList | test_faust | 91.49 | 94.24 | 86.10 | 52.73 | **86.59** | 76.98 | 12.65 |
| **test_faust** | **all_heuristics** | 91.98 | 93.92 | 88.17 | 66.35 | **90.04** | 81.63 | 7.58 |
| faust_common_mays | test_faust | 93.59 | 96.33 | 88.83 | 60.02 | **90.99** | 82.74 | 7.74 |
| faust_common_punct | test_faust | 94.43 | 97.20 | 89.40 | 52.01 | **90.02** | 81.05 | 10.40 |
| faust_common_mays_punct | test_faust | 93.95 | 96.71 | 88.93 | 53.24 | **90.08** | 81.56 | 9.51 |
| faust_common_mays_URL | test_faust | 93.59 | 96.33 | 88.83 | 60.00 | **91.02** | 82.82 | 7.67 |
| faust_mays_URL_punct | test_faust | 93.53 | 96.23 | 88.81 | 60.97 | **91.09** | 82.92 | 7.49 |
| faust_common_mays_punct_URL | test_faust | 93.51 | 96.20 | 88.81 | 60.97 | **91.07** | 82.92 | 7.49 |
| faust_common_WordList | test_faust | 92.60 | 95.89 | 86.80 | 51.42 | **88.25** | 79.58 | 10.55 |
| faust_all_heuristics | test_faust | 93.29 | 96.01 | 88.52 | 60.50 | **91.04** | 83.30 | 6.87 |

**Table 5.2:** Results applying the guessing module implementing several combinations of our heuristics. Every row of the table represents a tagging scenario using the tagging model to tag the Input. The columns have the same meaning as in the Table 5.1.

data with a combination of heuristics. In particular, we observe the best accuracy when the guessing script deals with the capitalization of the words, punctuation marks, recognizing URLs and using a list of common words.

Finally, we present and analyze the results obtained only by applying the Levenshtein distance algorithm to the input.
We can see how we recover in this case form 88.41% of accuracy over the baseline model to a 89.68%. We perform better if we try to control the introduction of noise using the thresholds achieving a 90.36% of accuracy.(Table 5.3). It is important to notice that in this experiment we only use the Levenshtein's distance algorithm.

On the one hand, we can say that our guessing Levenshtein's distance module helps the tagger with the unknown words when it deals with noisy input but, on the other hand, we have to be careful with the noise that we introduce in the dictionary which could be a reason of why we do not improve as much as we can expect.

| Tagging Model | Input | known | unamb | amb. | unk. | overall | MFT | unk.w. |
|---|---|---|---|---|---|---|---|---|
| **clean** | **baseline** | **97.32** | **99.42** | **93.04** | **88.46** | **97.07** | **91.30** | **2.81** |
| baselineWSJ | test_faust | 93.85 | 96.11 | 89.57 | 52.09 | **88.41** | 78.20 | 13.05 |
| dist_lev | test_faust | 86.94 | 92.55 | 77.58 | 0.00 | **86.94** | 83.81 | 0.00 |
| dist_lev_th1 | test_faust | 89.56 | 94.07 | 81.69 | 60.35 | **88.05** | 81.58 | 5.17 |
| dist_lev_th2 | test_faust | 88.15 | 93.30 | 79.39 | 57.68 | **87.41** | 82.79 | 2.42 |
| dist_faust_lev | test_faust | 89.68 | 94.33 | 82.25 | 0.00 | **89.68** | 86.15 | 0.00 |
| dist_faust_lev_th1 | test_faust | 92.10 | 95.79 | 86.00 | 54.42 | **90.36** | 84.15 | 4.62 |
| dist_faust_lev_th2 | test_faust | 90.73 | 95.07 | 83.71 | 51.65 | **89.88** | 85.17 | 2.20 |

**Table 5.3:** Results applying the guessing module implementing the Levenshtein's distance algorithm with different thresholds. Every row of the table represents a tagging scenario using the tagging model to tag the Input. The columns have the same meaning as in the Table 5.1.

### 5.2.1 Ponderate Levenshtein's distance module

Misspelling errors do not have the same probability to happen. There are many factors that influence the occurrence of a typo: character confusion, proximity of the characters in the keyboard, phonetics, etc. Having this in mind, it is a good idea to give a different weight to an error depending on the probability of its appearance in a text to calculate the distance among words.

We changed the distance function of our guessing module to make the distance algorithm implementation able to give a different weight to a possible misspeling attending to its appearance probability.

We introduced a matrix of weights which collect the probability of a substitution between two characters. We found that misspelling errors probabilities matrix in the notes of the Stanford online Natural Language Course of Daniel Jurafsky

**sub[X, Y] = Substitution of X (incorrect) for Y (correct)**

| X \ Y (correct) | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | 0 | 0 | 7 | 1 | 342 | 0 | 0 | 2 | 118 | 0 | 1 | 0 | 0 | 3 | 76 | 0 | 0 | 1 | 35 | 9 | 9 | 0 | 1 | 0 | 5 | 0 |
| b | 0 | 0 | 9 | 9 | 2 | 2 | 3 | 1 | 0 | 0 | 0 | 5 | 11 | 5 | 0 | 10 | 0 | 0 | 2 | 1 | 0 | 0 | 8 | 0 | 0 | 0 |
| c | 6 | 5 | 0 | 16 | 0 | 9 | 5 | 0 | 0 | 0 | 1 | 0 | 7 | 9 | 1 | 10 | 2 | 5 | 39 | 40 | 1 | 3 | 7 | 1 | 1 | 0 |
| d | 1 | 10 | 13 | 0 | 12 | 0 | 5 | 5 | 0 | 0 | 2 | 3 | 7 | 3 | 0 | 1 | 0 | 43 | 30 | 22 | 0 | 0 | 4 | 0 | 2 | 0 |
| e | 388 | 0 | 3 | 11 | 0 | 2 | 2 | 0 | 89 | 0 | 0 | 3 | 0 | 5 | 93 | 0 | 0 | 14 | 12 | 6 | 15 | 0 | 1 | 0 | 18 | 0 |
| f | 0 | 15 | 0 | 3 | 1 | 0 | 5 | 2 | 0 | 0 | 0 | 3 | 4 | 1 | 0 | 0 | 0 | 6 | 4 | 12 | 0 | 0 | 2 | 0 | 0 | 0 |
| g | 4 | 1 | 11 | 11 | 9 | 2 | 0 | 0 | 0 | 1 | 1 | 3 | 0 | 0 | 2 | 1 | 3 | 5 | 13 | 21 | 0 | 0 | 1 | 0 | 3 | 0 |
| h | 1 | 8 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 12 | 14 | 2 | 3 | 0 | 3 | 1 | 11 | 0 | 0 | 2 | 0 | 0 | 0 |
| i | 103 | 0 | 0 | 0 | 146 | 0 | 1 | 0 | 0 | 0 | 0 | 6 | 0 | 0 | 49 | 0 | 0 | 0 | 2 | 1 | 47 | 0 | 2 | 1 | 15 | 0 |
| j | 0 | 1 | 1 | 9 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| k | 1 | 2 | 8 | 4 | 1 | 1 | 2 | 5 | 0 | 0 | 0 | 0 | 5 | 0 | 2 | 0 | 0 | 0 | 6 | 0 | 0 | 0 | 4 | 0 | 0 | 3 |
| l | 2 | 10 | 1 | 4 | 0 | 4 | 5 | 6 | 13 | 0 | 1 | 0 | 0 | 14 | 2 | 5 | 0 | 11 | 10 | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| m | 1 | 3 | 7 | 8 | 0 | 2 | 0 | 6 | 0 | 0 | 4 | 4 | 0 | 180 | 0 | 6 | 0 | 0 | 9 | 15 | 13 | 3 | 2 | 2 | 3 | 0 |
| n | 2 | 7 | 6 | 5 | 3 | 0 | 1 | 19 | 1 | 0 | 4 | 35 | 78 | 0 | 0 | 7 | 0 | 28 | 5 | 7 | 0 | 0 | 1 | 2 | 0 | 2 |
| o | 91 | 1 | 1 | 3 | 116 | 0 | 0 | 0 | 25 | 0 | 2 | 0 | 0 | 0 | 0 | 14 | 0 | 2 | 4 | 14 | 39 | 0 | 0 | 0 | 18 | 0 |
| p | 0 | 11 | 1 | 2 | 0 | 6 | 5 | 0 | 2 | 9 | 0 | 2 | 7 | 6 | 15 | 0 | 0 | 1 | 3 | 6 | 0 | 4 | 1 | 0 | 0 | 0 |
| q | 0 | 0 | 1 | 0 | 0 | 0 | 27 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| r | 0 | 14 | 0 | 30 | 12 | 2 | 2 | 8 | 2 | 0 | 5 | 8 | 4 | 20 | 1 | 14 | 0 | 0 | 12 | 22 | 4 | 0 | 0 | 1 | 0 | 0 |
| s | 11 | 8 | 27 | 33 | 35 | 4 | 0 | 1 | 0 | 1 | 0 | 27 | 0 | 6 | 1 | 7 | 0 | 14 | 0 | 15 | 0 | 0 | 5 | 3 | 20 | 1 |
| t | 3 | 4 | 9 | 42 | 7 | 5 | 19 | 5 | 0 | 1 | 0 | 14 | 9 | 5 | 5 | 6 | 0 | 11 | 37 | 0 | 0 | 2 | 19 | 0 | 7 | 6 |
| u | 20 | 0 | 0 | 0 | 44 | 0 | 0 | 0 | 64 | 0 | 0 | 0 | 0 | 2 | 43 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 2 | 0 | 8 | 0 |
| v | 0 | 0 | 7 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 8 | 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| w | 2 | 2 | 1 | 0 | 1 | 0 | 0 | 2 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 7 | 0 | 6 | 3 | 3 | 1 | 0 | 0 | 0 | 0 | 0 |
| x | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| y | 0 | 0 | 2 | 0 | 15 | 0 | 1 | 7 | 15 | 0 | 0 | 0 | 2 | 0 | 6 | 1 | 0 | 7 | 36 | 8 | 5 | 0 | 0 | 1 | 0 | 0 |
| z | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7 | 5 | 0 | 0 | 0 | 0 | 2 | 21 | 3 | 0 | 0 | 0 | 0 | 3 | 0 |

Looking at the matrix, we observed that it seems to not take into account the keyboard distribution.

Therefore, we designed a new matrix of weights. In our matrix, a character confusion has asigned a weight regarding to how the characters are distributed in the keyboard. We saw the keyboard as a grid. Every pigeonhole of the grid is a key of the keyboard. We assigned coordenates to the keys setting the origin to the character 'A' as we can see in the following figure.

| Tagging Module | test-file | known | unamb | amb. | unk. | overall | MFT | unk.w. |
|---|---|---|---|---|---|---|---|---|
| **baseline** | **wsj_test** | **97.32** | **99.42** | **93.04** | **88.46** | **97.07** | **91.30** | **2.81** |
| baseline | faust_test | 93.86 | 96.11 | 89.60 | 52.09 | 88.41 | 78.21 | 13.05 |
| b_distPond1_th2 | faust_test | 86.80 | 93.93 | 75.90 | 49.11 | 86.42 | 82.31 | 1.02 |
| b_distPond1_th1 | faust_test | 88.69 | 94.33 | 79.53 | 49.69 | **86.98** | 81.24 | 4.38 |
| b+devfaust2 | faust_test | 94.81 | 97.76 | 89.78 | 51.62 | **90.22** | 81.05 | 10.63 |
| b+devfaust2_distPond1_th2 | faust_test | 89.33 | 91.24 | 85.90 | 54.29 | **89.09** | 84.65 | 0.89 |
| b+devfaust2_distPond1_th1 | faust_test | 89.73 | 91.53 | 86.49 | 53.17 | **89.69** | 83.71 | 3.32 |

**Table 5.4:** Results applying the guessing module implementing the first ponderate Levenshtein's distance algorithm with different thresholds. Every row of the table represents a tagging scenario using the tagging model to tag the Input. The columns have the same meaning as in the Table 5.1.

Now, we consider that the weight among two characters, $k_1$ and $k_2$, is the corresponding euclidean distance between its coordinates.

$$k_1 \to (x_{k_1}, y_{k_1}),\ k_2 \to (x_{k_2}, y_{k_2})$$

$$w(k_1, k_2) = \sqrt{(x_{k_1} - x_{k_2})^2 + (y_{k_1} - y_{k_2})^2}$$

The resulting matrix is A.3.1 and it is shown in the appendix A.3.

Hence, we implemented two metrics to ponderate the distance: the misspelling errors probabilities matrix and the keyboard distribution matrix. In the following section we show the obtained results.

### 5.2.2 Results using the ponderate Levenshtein's distances

We present in Table 5.4 the results obtained using the misspelling's probabilites. We observe how the adjustment of the threshold helps in the task of controlling the noise added to the dictionary of the tagger. Neverherless, with this kind of metric we cannot improve the results obtained only by the retraining process using FAUST data in the training step since we only achieve a 89.69% of accuracy.

We found that the second metric have a better behaviour when tagging real data. In Table 5.5 we can see how this metric is able to adjust the performance to almost achieve the same accuracy as if we retrained the tool using the FAUST corpus. Furthermore, we observe how we recover until a 91.64% of overall accuracy just only applying

| Model | test-file | known | unamb | amb. | unk. | overall | MFT | unk.w. |
|---|---|---|---|---|---|---|---|---|
| **baseline** | **wsj_test** | **97.32** | **99.42** | **93.04** | **88.46** | **97.07** | **91.30** | **2.81** |
| baseline | faust_test | 93.86 | 96.11 | 89.60 | 52.09 | **88.41** | 78.21 | 13.05 |
| b_distPond2_th2 | faust_test | 88.76 | 92.71 | 81.79 | 63.61 | **87.96** | 82.62 | 3.17 |
| b_distPond2_th1 | faust_test | 91.79 | 94.17 | 87.31 | 59.27 | **89.04** | 80.58 | 8.47 |
| b_distPond2_th075 | faust_test | 92.57 | 94.73 | 88.46 | 58.30 | **89.25** | 80.20 | 9.68 |
| b_distPond2_th05 | faust_test | 92.57 | 94.73 | 88.46 | 58.29 | **89.25** | 80.20 | 9.68 |
| b_distPond2_th025 | faust_test | 92.57 | 94.73 | 88.46 | 58.29 | **89.25** | 80.20 | 9.68 |
| b+devfaust2 | faust_test | 94.81 | 97.76 | 89.78 | 51.62 | **90.22** | 81.05 | 10.63 |
| b+devfaust2_distPond2_th2 | faust_test | 91.17 | 94.93 | 84.95 | 67.73 | **90.50** | 85.04 | 2.84 |
| b+devfaust2_distPond2_th1 | faust_test | 93.24 | 96.31 | 87.99 | 67.68 | **91.49** | 83.22 | 6.85 |
| b+devfaust2_distPond2_th075 | faust_test | 93.69 | 96.59 | 88.69 | 67.25 | **91.64** | 82.83 | 7.73 |
| b+devfaust2_distPond2_th05 | faust_test | 93.69 | 96.59 | 88.69 | 67.25 | **91.64** | 82.83 | 7.73 |
| b+devfaust2_distPond2_th025 | faust_test | 93.69 | 96.597 | 88.59 | 67.25 | **91.64** | 82.83 | 7.73 |

**Table 5.5:** Results applying the guessing module implementing the second ponderate Levenshtein's distance algorithm with different thresholds, i.e., the one that takes in account the QWERTY keyboard distribution to calculate the distance between words. Every row of the table represents a tagging scenario using the tagging model to tag the Input. The columns have the same meaning as in the Table 5.1.

the Levenshteins' distance algorithm with almost an 8% of final unknown words using a tagger trained with non-standard texts.

These results show that the second ponderate Levenshteins' distance is able of learning from the FAUST texts. It also can generalize well enough to take that heuristic in account when building a final experiment merging the heuristics with the best results.

In Table 5.6 we can see how the accuracies increase in an incremental way through the experiments since we add more heuristics to the guessing module. However, we find that when we achieve an accuracy over the $90 - 91\%$ it is difficult to improve it.

One of the best result obtained in this section is seeing how from a SVMTagger trained without real non-standard texts we can achieve more than 90% of overall accuracy. These results show that our heuristics and the distance algorithm make the SVMTagger know about this non-standard input.
We observed through our experiments that it is important to adjust the thresholds of the distances used in the guessing model in order to avoid introducing noise to the system.

| Tagging Model | Test | known | unamb | amb. | unk. | overall | MFT | unk.w. |
|---|---|---|---|---|---|---|---|---|
| **baselineWSJ** | **clean** | **97.32** | **99.42** | **93.04** | **88.46** | **97.07** | **91.30** | **2.81** |
| baselineWSJ | test_faust | 93.85 | 96.11 | 89.57 | 52.09 | **88.41** | 78.20 | 13.05 |
| common_mays_punct_URL | test_faust | 92.40 | 94.29 | 88.70 | 65.91 | **90.18** | 81.21 | 8.38 |
| faust_common_mays_punct_URL | test_faust | 93.51 | 96.20 | 88.81 | 60.97 | **91.07** | 82.92 | 7.49 |
| b_distPond2_th025 | test_faust | 92.57 | 94.73 | 88.46 | 58.29 | **89.25** | 80.20 | 9.68 |
| b+devfaust2_distPond2_th025 | faust_test | 93.69 | 96.597 | 88.59 | 67.25 | **91.64** | 82.83 | 7.73 |
| b_Heuristics+distPond2_th025 | faust_test | 92.40 | 94.29 | 88.70 | 65.91 | **90.18** | 81.21 | 8.38 |
| b+devfaust2_Heuristics+distPond2_th025 | faust_test | 93.52 | 96.21 | 88.83 | 68.93 | **91.68** | 82.87 | 7.48 |

**Table 5.6:** Summary of the best results obtained using the guessing module implementing several combinations of our heuristics. Every row of the table represents a tagging scenario using the tagging model to tag the Input. The columns have the same meaning as in the Table 5.1.

Using the guessing module combined with the SVMTagger trained with FAUST texts we can recover up to 91.68% of overall accuracy. This is our best result only using the retraining and the guessing techniques.

## 5.3 Applying a general spell checker/corrector

Our purpose through this experiments is to improve the performance over non-standard input without retraining the tagger but without modifying the input. However, there is another way to deal with that problem: modifying the input by correcting the errors.

If we want to correct the input, we need to be able to do two tasks: identify the errors, and correct the mistakes. As we have seen in the state of the art section of the *Chapter* **??**, there are many work done in this area.

In particular, we show the results obtained using the well known spell checker from the Microsoft Office 2007 package included in the Microsoft Word program.

### 5.3.1 The Microsoft Word 2007 checker: designing the experiment

Since we could not find an open source spell corrector to include it in the pipeline of our system, we decided to use a well known spell checker instead.
We used the MS Word 2007 version of the spell checker. Since it is not possible to include the checker in an automatic way in the SVMTagger pipeline, we preprocessed the input using the MS Word program.

The preprocessing step consisted of opening the input test file from the FAUST project with the MS Word program. Afterwards, we selected the "American English" as the language of the document and started the manual spell checking process. When we had already selected the language, we started to check the spelling with the option of the program: *Review: Spell and Grammar*.
When the program identifies a wrong word, it shows a list of candidates to correct it. We chose always the first option given by the checker if the tool did not split the word in two parts (i.e. from "darling.do " to "darling. do"). We made that decision because we need a golden standard to obtain results and in the manual annotated files this words would appear like a single word.

It is worth noting that most of the corrected errors are capitalization errors, mostly at the beggining of a sentence or in proper nouns detected by the checker. There were also some words in other language different from the English that the checker did not recognize as English words. So that, it also could not give any candidates to correct them.

In the following section we present the results obtained through this experiments.

### 5.3.2 Results

In this experiment we modified the FAUST test file correcting the spelling errors detected by the spell checker from the MS Word 2007 program. After that, we applied our models to tag the processed input: the baseline model and the model trained with faust data. The Table5.7 sumarizes the results obtained using both models.

## 5.4 Conclusions

We designed a module that modifies the dictionary of the SVMTagger. These modifications are based in recognizing the input unknown words like possible known words but somehow wrong written. We considered as possible modifications capitalizations, punctuation marks, common mistakes in common words, recognizing URLS and also applying several Levenshteins' distances.

| Tagging Model | Input | known | unamb | amb. | unk. | overall | MFT | unk.w. |
|---|---|---|---|---|---|---|---|---|
| baselineWSJ | faust_checked | 93.47 | 95.71 | 89.09 | 66.02 | **91.17** | 81.21 | 8.38 |
| faust model | faust_checked | 93.42 | 96.04 | 88.86 | 62.06 | **91.07** | 82.92 | 7.49 |

**Table 5.7:** Results after passing the MS Word 2007 spell checker to the FAUST data after tagging it with the SVMTagger. As in the other tables, the first column represents the input of the tagger. The second is the tagging model used in the experiment. Then, the central columns are related to the accuracy over different kind of words attending to its ambiguity. The *known* column shows the accuracy achieved over known words. The *unamb.* column is the accuracy over unambiguous known words and the *amb.* column is the accuracy over ambiguous known words. The *unk.* column is the accuracy achieved over unknonw words. Finally, in the *overall* column we see the overall accuracy achieved through the experiment. The *MFT* column shows the results of making the tagging using the most frequent tagg approach. The last column, the *unk.w.* column shows the percentage of unknown words in every experiment.

One of the best results of this set of experiments is that we are able to achieve a similar accuracy that the one achieved retraining the SVMTool with FAUST data (*Chapter*6). Hence, our heuristics are able to show the tagger useful information about the data. Without the drawback of being a time consuming task as it can be a training process. Furthermore, we see that these heuristics have an accumulative effect when combining them to the learning by means of the retraining process. We also designed a Levenshteins' distance that is able to reproduce a comparable results to those ones obtained with the rest of the heuristics or with the retraining process.

We hoped that we can have an accumulative behaviour of our good results: retraining the tools, applying our heuristics and using our distance algorithm. However, we observe that after retraining the tagger, using the heuristics we are not able to learn much more. The same happens when applying also the distance algorithm implementations. We observe that the results improve but not as much as we can expect.
This happens because the SVMTagger have already learnt many information of these that the heuristics implemented in the guessing module can contribute with the retraining process.

Finally, it is important to say that the experiment using the spell cheker also obtain a good results (around $90 - 91\%$ accuracy). We must take it into account in further work since we can combine this experiment with the rest of the methods implemented

in the guessing module. The challenge here will be obtaining a checker that allows us to correct texts in an automatic way.

# 6

# Domain Adaptation with Retraining

As we said, what we want is to learn from the texts from the *FAUST project*. We try to do this by different ways. We developed two different type of approaches to do domain adaptation: doing adaptation with a training process or without it. The first one consist in re-training our tagger using real data in the training files, or training a TBL algorithm to extract a set of rules to apply to the input.

We have already explained in *Chapter* 5 the strategy that uses a guessing module.

In this chapter we will describe the strategy that uses the retraining process. It consists in re-training our tagger using the FAUST corpus in the training files. We also present here the experiments designed using a TBL algorithm implementation.

## 6.1 Training the SVMTagger with FAUST data

In order to learn from real non-standard data we retrained the SVMTagger using the FAUST corpus. Through this section, we discuss the experiments we made and the obtained results.

| Input | Model | known | unamb | amb. | unk. | overall | MFT | unk.w. |
|---|---|---|---|---|---|---|---|---|
| **faust_test** | **baseline** | 93.85 | 96.11 | 89.57 | 52.09 | **88.41** | 78.20 | 13.05 |
| clean | base+dev | 97.32 | 99.42 | 93.11 | 82.89 | 96.92 | 91.33 | 2.79 |
| test_faust | base+dev | 94.81 | 97.76 | 89.78 | 51.62 | 90.22 | 81.05 | 10.63 |
| clean | base+devx2 | 97.28 | 99.42 | 93.00 | 88.31 | 97.03 | 91.33 | 2.79 |
| test_faust | base+devx2 | 94.74 | 97.76 | 89.59 | 59.64 | 91.01 | 80.99 | 10.63 |
| clean | base+devx5 | 97.31 | 99.42 | 93.08 | 88.54 | 97.06 | 91.31 | 2.79 |
| test_faust | base+devx5 | 94.65 | 97.76 | 89.34 | 60.41 | 91.01 | 81.05 | 10.63 |

**Table 6.1:** Results of concatenating the FAUST data to the WSJ data to build the training set. The first column represents the input of the tagger. The second is the tagging model used in the experiment. Then, the central columns are related to the accuracy over different kind of words attending to its ambiguity. The *known* column shows the accuracy achieved over known words. The *unamb.* column is the accuracy over unambiguous known words and the *amb.* column is the accuracy over ambiguous known words. The *unk.* column is the accuracy achieved over unknonw words. Finally, in the *overall* column we see the overall accuracy achieved through the experiment. The *MFT* column shows the results of making the tagging using the most frequent tagg approach. The last column, the *unk.w.* column shows the percentage of unknown words in every experiment.

### 6.1.1 Giving more weight to the FAUST data

The first attempt to learn from non-standard input was training the SVMTagger with a training set made by concatenating the clean training set from the WSJ corpus and the development (*dev*) file from the FAUST project.
In order to give more importance to the new words from the real non-standard texts, we concatenated once the *dev* FAUST file to the training file from the WSJ corpus (base+dev model), twice the *dev* file ( base+devx2 model) and concatenated the *dev* FAUST file 5 times (base+devx5 model). We can see the results in Table6.1.

From the results showed in the Table6.1 we observe that the tagger learn from the noisy data by the retraining strategy. We can reinforce such learning by concatenating more real data to the training file. We see how the performance increase up to a 91% accuracy.

Regarding to the results it is not worthy to concatenate too many times the *dev* FAUST file to the training set because it did not make the tagger learn more or improve the results. We need to check the performance of the tagger with a larger dataset of

| Input | Model | known | unamb | amb. | unk. | overall | MFT | unk.w. |
|---|---|---|---|---|---|---|---|---|
| test_faust | crossval | 94.78 | 97.76 | 89.67 | 51.41 | **90.12** | 80.98 | 10.73 |
| clean | crossval | 97.04 | 99.24 | 92.61 | 87.89 | **96.92** | 91.62 | 3.20 |

**Table 6.2:** Results of the 10-fold crossvalidation process to train the SVMTagger with FAUST data. The columns here have the same interpretation as in Table6.1.

| Input | Model | known | unamb | amb. | unk. | overall | MFT | unk.w. |
|---|---|---|---|---|---|---|---|---|
| test_faust | crossval | 94.79 | 97.76 | 89.73 | 51.47 | **90.18** | **81.05** | 10.64 |
| clean | crossval | 96.90 | 99.11 | 92.06 | **87.93** | **96.97** | 90.43 | 3.01 |

**Table 6.3:** Results of the 100-fold crossvalidation process to train the SVMTagger with FAUST data. The columns here have the same interpretation as in Table6.1.

non-standard real texts and see if we obtain similar results. Since we do not have access to a larger dataset, we made some cross validation experiments.

### 6.1.2 Experiments with FAUST data

In order to simulate a larger dataset, we trained the *SVMTagger* by 10-fold and 100-fold cross validation processes.
We concatenated the training WSJ corpus sections and the *dev* and *test* files from the FAUST project. After that, we made $k$ blocks and we do a cross validation procedure: we train the tagger using $k-1$ blocks and test over the remaining block.

#### 10-fold crossvalidation

In the experiment making 10 blocks we had in average the results showed in Table6.2.

#### 100-fold crossvalidation

In the experiment making 100 blocks we had in average the results showed in Table6.3.

We can say from the results obtained by the cross validation experiments that the results obtained by concatenating the development file to the WJS training file are consistent.

### 6.1.3 Using the fnTBL toolkit

As we said in *Chapter* **??**, *fnTBL* is a toolkit which implements a transformation based learning technique (the approach developed by Brill in (Bri95)) mainly based in transforming the data in order to correct the mistakes that make the biggest increasement of the error rate.

We describe in more detail the system and the TBL algorithm in *Chapter* **??**, when describing the state-of-the art and the background of our work.

Through this section we will show and describe the different experiments we developed using the *fnTBL* toolkit. We also discuss and analyze the results of our experiments.

### 6.1.4 TBL experiments

We describe now the experiments we designed and developed using the *fnTBL* toolkit.

First of all, we loked at the performance of the *fnTBL* as PoS tagger. Then, we tried to improve the tagging performance of our SVMTagger using the *fnTBL* system. We made several combinations of the *fnTBL* system combining it with the SVMTagger and all the different versions of the system that we have developed.

#### 6.1.4.1 fnTBL as standard POS tagger

We tested the *fnTBL* as a POS tagger in order to compare it with the SVMTool. In other experiments, we also tried to do domain adaptation using the output of this tagger.

It is important to remark that in these experiments we used the default configuration of the *fnTBL*. We took as input the tagged data with the most frequent tag approach implemented by the toolkit. We divided the learning process in two steps, as it is recommended in the manual: first learning the lexical rules and afterwards learning the context rules. We follow this steps in the following experiments because we do not have any additional tagging information as a starting point.

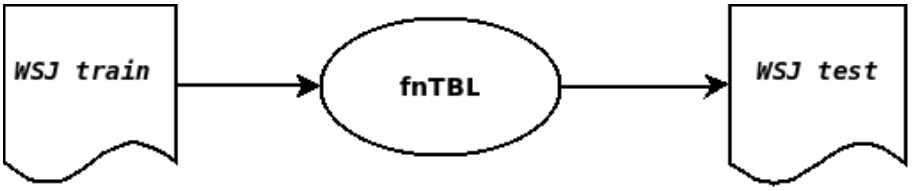


**Figure 6.1:** Settings of the experiment 0. The fnTBL as POSTagger using WSJ data.

| Experiment | fnTBL acc. | SVMTagger acc. |
|---|---|---|
| 0 | 96.64 | 97.07 |
| 1 | 86.74 | 88.41 |
| 2 | 89.46 | 90.22 |
| 3 | 88.38 | 90.22 |
| 4 | 91.49 | 90.22 |
| 5 | 91.06 | 90.22 |
| 6a | 91.39 | 90.18 |
| 6b | 90.97 | 89.25 |
| 6c | 91.48 | 90.18 |
| 6d | 91.73 | 91.67 |
| 6e | 91.69 | 91.64 |
| 6f | 91.74 | 91.68 |

**Table 6.4:** Results of the different experiments using the *fnTBL* toolkit. The first column indicates the experiment from that the results come from. The second column is the overall accuracy achieved by the experiment using the *fnTBL*. The third column is the accuracy achieved by the SVMTagger in a similar scenario.

### Experiment 0

First of all, we tested the *fnTBL* as a POS tagger. We trained the tool using the WSJ data (sections $00 - 19$ like we did with the SVMTagger). We can see a representation of the situation in Figure 6.1.

First we applied the *fnTBL*-based tagger the WSJ test set(sections $22 - 24$).In Table6.4, in the rows corresponding to the *Experiment 0*, we can see that the accuracy obtained by the *fnTBL* system is 96.64%, while the SVMTool in the same situation achieve a 97.7% of accuracy. Hence, we observe that the SVMTagger is a better PoS tagger. We compare the results when tagging FAUST data in the *Experiment 1*.

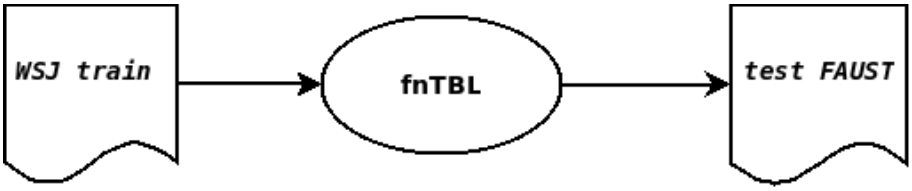


**Figure 6.2:** Settings of the experiment 1. The fnTBL as POSTagger using WSJ data and applied to FAUST data.

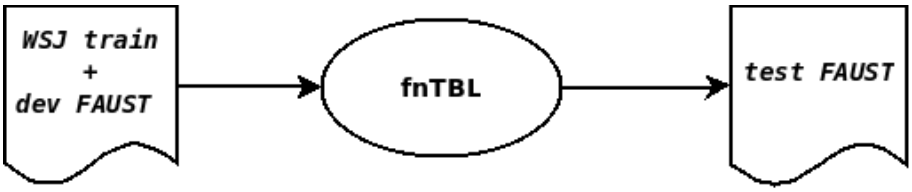


**Figure 6.3:** Settings of the experiment 2.Domain adaptation using the FAUST data and the WSJ train data to learn the rules.

### Experiment 1

In this situation we tested the performance of the *fnTBL* PoS tagger applied to FAUST texts. The results are shown in Table6.4, in the rows corresponding to the *Experiment 1.* In the same way as happened with the SVMTagger, the accuracy achieved drops significantly from 96.64% to 86.74%, it almost drops 10 points. Notice that the behaviour of the SVMTagger in this situation is similar: the SVMTagger accuracy achieved drops from 97.07% to 88.92%. However, we observe again that the SVMTagger obtain better results than the *fnTBL* tagger.

We can say that the SVMTagger is also a more robbust tool because its performance does not drop as many points as the performance of the *fnTBL* tagger when applying it to non-standard input.

In the *Experiment 2* and the *Experiment 3* we changed the starting point to the learning step of the *fnTBL*. We used the output of the SVMTagger to train the *fnTBL* since we have seen it is a better PoS tagger.

### Experiment 2

The first attempt to improve the *fnTBL* tagger was to use data from the FAUST project in the learning process. In particular, we used the development FAUST file

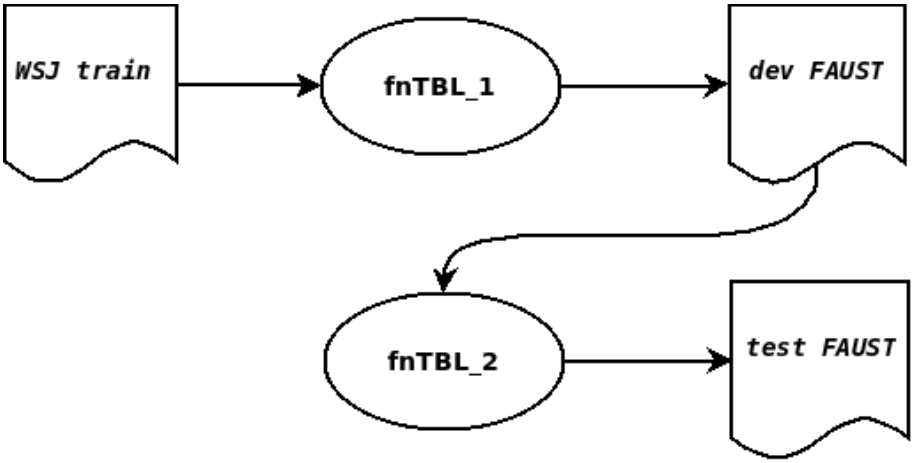


**Figure 6.4:** Settings of the experiment 3. Domain adaptation using the fnTBL POS tagger output.

jointly with the WSJ training set in order to obtain a more complet tagger, since the FAUST development file is very small. In Figure 6.3 we can see a representation of the experiment setting.

The results are sound with those we obtained in the previous experiments. Although the performance of the system improves, from 86.74% to 89.46%, the *fnTBL* tagger does not defeat the SVMTagger under the same conditions. As we can see in Table6.4, in the rows corresponding to the *Experiment 2*, the SVMTagger trained also with the development FAUST data achieves a 91% of accuracy over the FAUST test set.

### Experiment 3

In this experiment we try to do our first attempt to do domain adaptation using the *fnTBL*. What we do is trying to fix the errors made by the *fnTBL* PoS tagger using the TBL algorithm.
We first tagged the development FAUST file using the *fnTBL* PoS tagger described in *Experiment 2* and we used the output to train other *fnTBL* PoS tagger (the *fnTBL_2* in Figure 6.4).

We can see in Table6.4, in the rows corresponding to the *Experiment 3*, that we achieve 88.38% of overall accuracy. We can say then that the *fnTBL* does not learn so much about the domain in this framework. In other words, it is not able to learn rules from the development FAUST file general enough to be useful also when tagging the

FAUST test file. This result shows that when dealing with a non-standard-input it is difficult to extract rules that generalize well over other kind of data.

As a conclusion we can say that our SVMTool baseline tagger is a better POS tagger. It suggest that its tagging results would be a better starting point to the *fnTBL* system to try to do domain adaptation.
These first experiments allowed us to know the framework of using the *fnTBL*. We want to take advantage of all the properties of a TBL-based system to correct tagging errors made by the SVMTagger.

#### 6.1.4.2 fnTBL combined with SVMTagger

After testing the *fnTBL* as PoS tagger, we continued trying to improve the results of the TBL algorithm starting from a better starting point than the tagged data obtained by the most probable tag approach.
In the following experiments we use as a starting point the FAUST corpus tagged by our SVMTaggers designed and described in the previous chapters (*Chapter* 5 and the previous part of *Chapter* 6), this is, the SVMTaggers obtained by retraining the tool using the FAUST data and the SVMTaggers improved by the direct adaptation of the dictionaries using our guessing module to deal with unknown words.

In short, through our last set of experiments we will take advantage from the results of our improved SVMTaggers to train the *fnTBL* toolkit and try to improve the global performance of the system by learning to fix the errors made by our taggers.

**Experiment 4**

The first way to improve the tagging process is starting from a good tagged file. In our case, the first step here is tagging the test FAUST with the SVMTagger trained with clean data from the WSJ corpus, our baseline system.

The setting of the experiment is described in Figure 6.5. We expect to learn rules using the *fnTBL* toolkit to fix the errors made by the SVMTagger tagging the FAUST data, trying to learn from the development set and generalizing enough to obtain useful rules for tagging the test FAUST set.

We can see Table6.4, in the rows corresponding to the *Experiment 4*, that we can improve the performance of the SVMTagger when tagging the FAUST test data only looking at the development FAUST file. We observe that we gain more than an entire

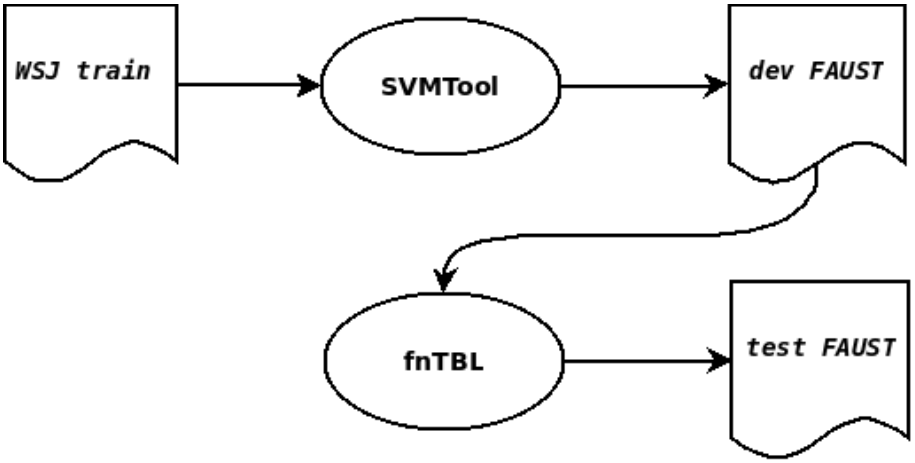


**Figure 6.5:** Settings of the experiment 4. Domain adaptation using the SVMTool POS tagger output to learn the rules.

point of overall accuracy, this is, we fix 1.20% of the errors made by the SVMTagger over the FAUST corpus with this setting.

We notice that we can learn many information from the FAUST corpus. We can learn statistical information through the SVM models used by the SVMTagger and we can learn rules with lexical/contextual information to tag better the FAUST texts.
In the following experiments we tried to learn more from the FAUST data using the improved SVMTaggers described before.

### Experiment 5

The first step in this experiment was a 10-fold cross validation process, training the tagger using the development FAUST data and the WSJ training set to tag the development FAUST file (Figure 6.6). Then, we used this development FAUST tagged file to train the *fnTBL* and try to learn rules general enough to be useful to tag the test FAUST file. Finally, we tag the test FAUST data to evaluate the system.

The results of the evaluation of the setting are shown Table6.4, in the rows corresponding to the *Experiment 5*. We observe that once again we recover almost a point of global accuracy. However, if we compare the results obtained here and the ones in the *Experiment 4*, we observe that here the rules learnt are less useful than those learnt in the *Experiment 4* attending to the results. Because there we achieved a 91.49% of accuarcy when here we only have a 91.06% of overall accuracy so, there is a difference

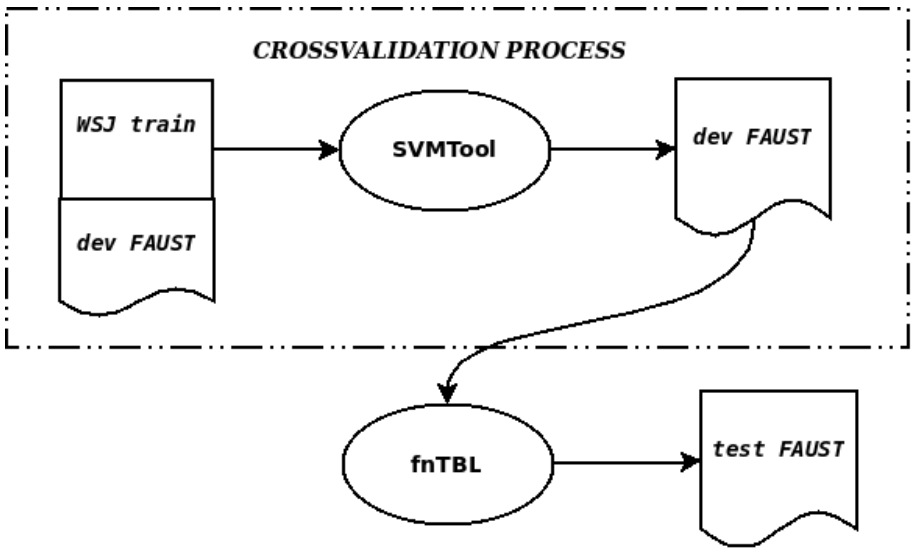


**Figure 6.6:** Settings of the experiment 5. Domain adaptation using the SVMTool POS tagger output to learn the rules.

of half a point of accuracy.
We can say that the *fnTBL* learns better from this particular domain in the setting of the *Experiment 4*.

### Experiment 6

Our last experiment is made by using the output of our SVMTagger using the guessing module described in *Chapter* 5.

The frameworks of the experiment are described in Figure*exp5* and Figure*exp5cx*. In the first one we can see how we took advantage of the setting used in *Chapter* 5 applied to tag the development FAUST file. Then we used these tagged data to train the *fnTBL* system. The second scenario, Figure*exp5cx*, shows the experimental setting for the situation of learning from the development FAUST file tagged using the guessing module combined with the SVMTagger, this is why there is a box representing a crossvalidation procedure (in particular, we do a 10-fold cross validation process in this kind of experiments).

We tested several kind of heuristics implemented in the guessing module. We can see in Table Table6.4, in the rows corresponding to the *Experiment 6x* the results obtained in every scenario. We will describe now the settings of every experiment.

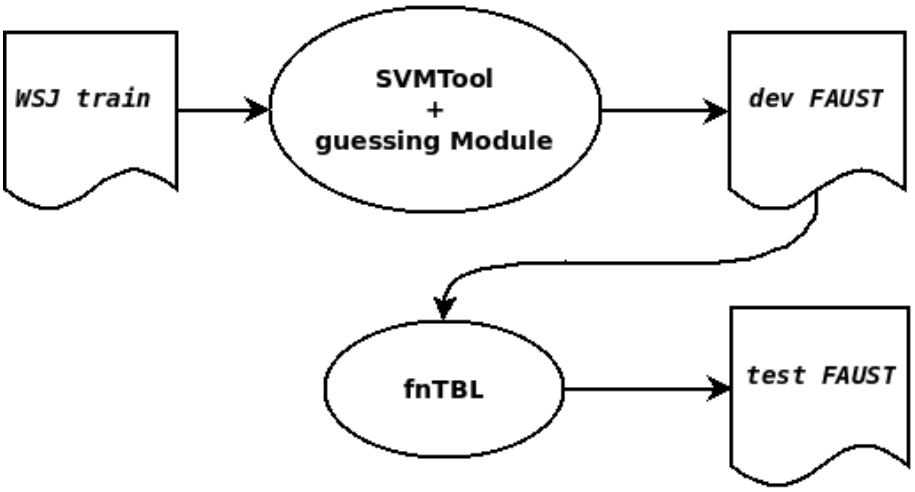


**Figure 6.7:** Settings of the experiment 6a,6b and 6c. Domain adaptation using the SVMTool POS tagger output to learn the rules.

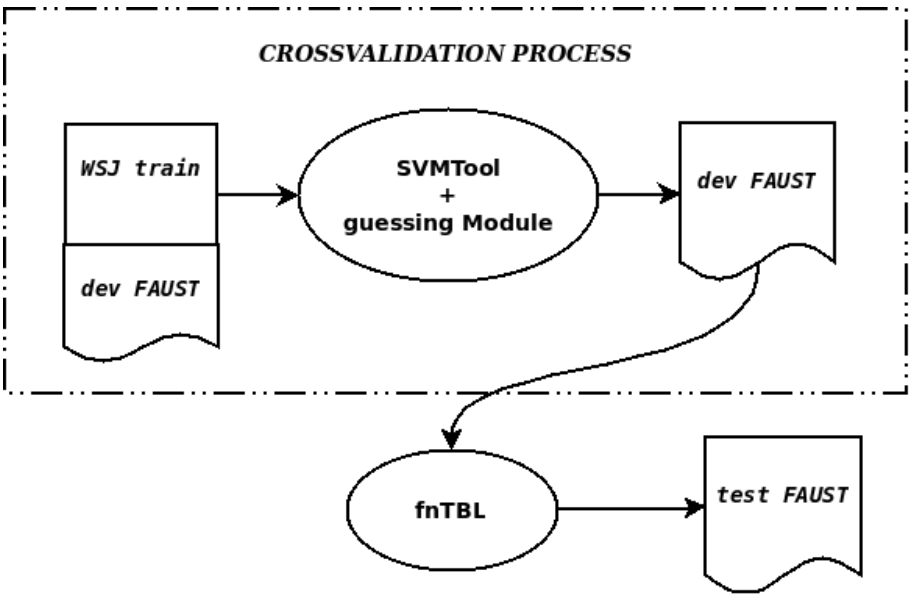


**Figure 6.8:** Settings of the experiment 6d,6e and 6f. Domain adaptation using the SVMTool POS tagger output to learn the rules.

First of all, in the experiment $6a$, we started applying the guessing module only dealing with the capitalization problem. With this setting, the SVMTool achieves a 90.18% of overall accuracy but we are able of improving the result achieving a 91.39% of overall accuracy, this is, the *fnTBL* can learn useful rules to fix the errors made by the SVMTagger trained with WSJ data and using the capitalization heuristics.

Then, we moved on applying a Levenshtein's distance algorithm to the SVMTagger trained with WSJ data to tag the development FAUST file. This $6b$ experiment shows that from this kind of tagging the *fnTBL* also is able to improve the results (from 89.25% to 90.97%). But we observe that it is easier to learn useful rules from a tagging only treating the capitalization problem than from a tagging using a Levenshtein's distance to try to recognize unknown words.

In the experiment $6c$ we used the guessing module with the best combination of heuristics that we tested in *Chapter* 5. This is, a combination of heuristics that deal with the capitalization problem, use a list of common errors to identify unknown words, identify URLs and also deal with punctuation marks. It also applies a Levenshtein's algorithm (the ponderate distance algorithm using the matrix that takes into account the keyboard distribution) to try to guess if an unknown word is actually a known word.
We observe how we recover more accuracy than in the other scenarios, this is, we have an incremental learning achieving 91.48% of overall accuracy, beating the performance of the system based only in the use of the SVMTagger ( 91%).

In the three next experiments, we used the SVMTagger trained also with development FAUST data. Hence, we used a 10-fold cross validation process to obtain the development FAUST file tagged with this models in order to be able to learn from it with the *fnTBL*.

We reproduced the same experiments as in $6a, 6b$ and $6c$ but this time, we use the SVMTagger models trained using also FAUST corpus. We can see the results in Table **??**, but this time we are going to talk about experiments $6d, 6e$ and $6f$.

The experiment $6d$ followed the same idea as the experiment $6a$ but, this time, we use the SVMTagger trained also with the FAUST corpus. We observe that, in the same way as in the experiment $6a$ ,we learn how to improve the accuracy of the system. But this time, the improvement is significantly lower (from 91.67 to 91.73). We observe the same kind of behaviour in the experiment $6e$. This experiment is the same as the

experiment 6*b* but using a SVMTagger trained with FAUST data. We also see that we learnt only a little bit more from the non-standard input using the *fnTBL*.
Finally, using the same setting as in the experiment 6*c* but with the tagger trained with the FAUST corpus, like in the models 6*d* and 6*e*, we observe that in the experiment 6*f* we can learn more from the tagging output of the SVMTagger adapted to the FAUST domain.

We can say, looking at the results, that there are many ways to learn from the data of the FAUST project using the *fnTBL*. Vut we observe that there is a boundary that is very difficult to pass and keep on improving the results of the system. We will discuss the possible reasons of the existence of this boundary in the *Conclusions* of this section.

### 6.1.5 Conclusions

Through this section we tried to learn from the FAUST corpus, actual real non-standard data collected from the Internet, to improve the performance of the SVMTagger in order to obtain a more robust PoS tagger.
In particular, we tried to learn directly from the data by mean of training processes, using a statistical approach (the SVM training) and using a rule based system based in the TBL algorithm (the *fnTBL* toolkit).

We observe that the SVMTo is able of learning useful statistical information from the FAUST data and combine it with the patterns learnt from the WSJ training data. We obtained an overall accuracy of 90.22% when using the development FAUST file to train the tagger using the FAUST test set to evaluate. We obtain also a 96.92% of overall accuracy when tagging the WSJ test data. We can say that we obtain a robust behaviour of the tool doing this retraining process, but we wanted to recover more accuracy when dealing with non-standard texts.

We tried to give more weight to the FAUST examples just adding the same data more times to the training set (models *base+devxN*) but we observed that this only makes a slight improvement ( up to 91.01%).

As we said in the description of the FAUST corpus, we do not have a great volume of real data, so that, we tried to simulate a larger amount of real data by means of cross validation processes. With the results of these experiments of cross validation we observed that the resuls are sound with those obtained just retraining the SVMTagger,

this is, we can say that the SVMTagger is able to generalize quite well from a little amount of data.

Finally, we tried to learn from the real data using a rule approach using the *fnTBL* toolkit. Through these experiments we tried to fix the tagging errors made by the SVMTagger by learning rules using the TBL-based system. We observe that the *fnTBL* can learn good general rules from the development FAUST set. These rules also are useful when tagging the test set from the FAUST projetc. The system achieve an overall accuracy of a 91.5% from the SVMTagger trained with the FAUST corpus and it achieves a performans of 91.74% accuracy using the SVMTaggers developed in the *Chapter* 5.

We notice that we always find a boundary that does not allow us to recover more accuracy. We explain this fact as a consequence of two main issues.
The first one is the amount of data. We are always working with a small number of sentences, both in test and development sets. It is difficult to make good generalizations in an statistical way with such a little set of data. On the other hand, we tried to simulate a larger set of data using cross validation processes and we obtained similar results. This shows that the quantity of data is not the only problem that we have when dealing with non-standard texts.
There is another problem: the diversity of the data. The sentences do not share a common theme or domain. In the FAUST corpus one can find sentences from a novel, conversation's fragments or a sentence from a technical manual.

In short, we can conclude that is difficult to adapt a POS tagger to real non-standard data (in particular, the data from the FAUST project), first because of the sparsity of this kind of data to train the tools and also because it is hard to make good generalizations through this kind of texts because of its topics diversity and the errors that this kind of texts usually contains.
It is remarkable then how we could recover performance from 88.41% to 91.5% only using the *fnTBL* system and the SVMTagger trained using FAUST data. And we can improve more the results using our guessing module up to 91.74% of accuracy. Since ther is a floor of improvement up to around 97% accuracy, we expected to get better results. However, we could not achieve better results with this techniques because part of the information from the FAUST input data handeled by the heuristics in the

guessing module are have been also handeled by the *fnTBL*. There is a lot of redundancy so, there is a small possible improvement.

# 7

# Conclusions

We can find lots of examples of non–input standard and noisy uses of language over the Internet. As long as many NLP tools are designed as web services or web aplications it is important to identify the necessity of adapting the classical tools, designed to run over clean text, to these new type of data. But there are some issues to take into account when dealing with these texts: the lack of annotated examples and the wide range of language variation and sources of errors that appear in the data, and usually there is not a common theme or domain in the texts collected from the Internet. These issues can be a problem when we try to learn information in a statistical way and also when we try to generalize for the annotation of new texts.

In this thesis we developed a robust PoS-tagger tailored for working with noisy texts coming from the FAUST project framework.We annotated manually the FAUST project corpus. We used also a well known one: the WSJ corpus. We studied and analized the texts to design several strategies to achieve the goals that we want to achieve at the end of our work.

In the analysis of the real non-standard data, we confirmed the difficulty posed by this type of texts to the tools. The texts have a lot of mistakes, of different type and with the quantity of examples that we handle it is difficult to preview how and when the mistakes are going to appear again, this is, that it is really hard to reproduce and also anticipate the mistakes.

We also identified and classified the type of mistakes that we observed in the texts. We noticed that the most common errors are related to the capitalization of the words. Then, the second most common type of errors are misspellings or typographical errors.

The typos are subdivided into different types also: swapping letters in a word, character confusions, phonetical confusions,etc. We also find that it is usual that the user only write a part of a sentence, like a noun phrase, and do not write a sentence with gramatical correction. We tried to study also this phenomena in our experiments. We observe also that in real data there are also are many punctuation marks that don't appear in published texts (e.g. news, articles, etc.).
We noticed that there is a general knowledge that could be useful to improve the tagging task, for example well known proper nouns of companies like Google, Yahoo!, etc., or URLs that are easily recognisable by a user but could become unknown words for a NLP tool.
Hence, we can say that we achieved our two first goals of building an annotated corpus with real non-standard texts and analyze these data to improve our tagger.

After the analysis of the FAUST data, and taking into account all the ideas of domain adaptation techniques and non–standard input treatment that we read in the papers that we commented in Section 2.1 state–of–the–art, we devised some techniques and methodes and we designed several experiments to achieve our goals.

First of all we tried to reproduce artificially the mistakes that we observed in the FAUST corpus. On the WSJ corpus, we reproduced several kind of typos, words case variations, sentences with only noun phrases, among other type of mistakes. What we did was to retrain the tagger with the modified corpus and try to tag clean texts, modified texts and finally data from the FAUST project. However, we did not obtain good results through this experiments. We observed that among the artificially modified data, the behaviour of the tagger is robust and it has a good performance on the modified test files, but the tagger does not make a good job when tagging the FAUST data set using the information learnt from the artifitially modified data. This fact tells us that we are not reproducing well the errors that we observed in the FAUST texts. Furthermore, it tells that it is hard to reproduce such kind of texts and learn from them. Ultimately, it confirms the importance of having available annotated real non-standard data.

Moving on with our experiments, we have seen that the best way to learn from real data is using it directly somehow to help the tool to improve its performance.
The training step is usually very time consuming. Taking this into account and considering that it takes around one day to train the SVMTagger with the WSJ corpus

($> 1Mw$ corpus), we started our experiments designing a guessing module to try to adapt and expand the dictionary of the models of the SVMTagger withot the necessity of the training step.
We noticed through our experiments that one of the biggest problems with the real non-standart data is the treatment of the unknown words, so that, we design several heuristics to try to recognize when we have in the input words that actually are known to the tagger but are badly written.
In particular, we dealt with words changing their capitalization, since we observed that many times there are proper nouns written without capital letter. We tried to recognize new punctuation marks, URLs and we tried to identify common errors made on common words by looking at a list of common errors. Moreover, we implemented a Levenstein's distance algorithm ponderating the errors taking in account the QWERTY keyboard distribution.
We significantly improved the performance of the tagger reaching an overall accuracy of 90.18% when tagging real non-standard test data using a SVM model trained only with the WSJ corpus, whereas the baseline system achieved a 88.41% accuracy when using the regular SVMTagger on FAUST corpus. And we achieved an overall accuracy of 91.68% when the training data contains also examples from the FAUST project, since the SVMTagger trained using also the FAUST corpus achieved a 91% of accuracy as its best result. This improvement on FAUST data is at no cost on other corpora in both cases. The resulting tagger is able to keep performance on WSJ corpus at 97%.

Finally, we changed the strategy to learn from the FAUST data. We moved on and made experiments retraining the SVMTagger adding FAUST texts to the training set. We gave more weight to the real data adding the FAUST data repeatidly to the training data. In this way we achieved an overall accuracy of 91.01% as a better result. We also verified our results simulating a bigger quantity of real data with 10-fold and 100-fold cross-validation processes.

After all these experiments, we tried to improve our results applying another learning technique: a TBL based tool. In particular, we used the *fnTBL* toolkit. This is a rule-based system that learn rules in order to fix the errors made by the SVMTagger at tagging time.
This tool helped us to understand better the texts that we are handling and also helped

us to achieve a slightly better results, reaching a 91.74% accuracy when training the *fnTBL* using the output of the SVMTagger tagging FAUST data with a model trained with FAUST data and applying our guessing module combining the heuristics and the Levensteins' distance algorithm and a 91.48% of overall accuracy in the same framework as before but using a SVMTool model trained only with the WSJ corpus.

We can say after analyzing our experiments and the results obtained with and without the retraining process are comparable but it is easier to learn from the output of the models that have FAUST data in the training set because the SVMTool has already learnt from these texts. This means that we have made a good analysis of the texts from the FAUST project and identified the main errors that one can found in real data as general guidelines, but that it is more efficient to learn from the real non-standard texts using directly a learning algorithm as the SVMs like in the SVMTagger or a rule-based one like the *fnTBL* and there is no need for for manually developing rules .

However, we find that when we achieve an accuracy over the 91% it is difficult to improve it. This value seems to be the ceiling with the current techniques. We could improve the performance of the tagger on real non-standard texts because we designed systems that can make good generalizations from noisy data to deal with this kind of texts. But, in this generalization process we lost the vision of the details. Those details can be the key of keep on improving the performance of the system. However, we also observed in our experiments that if we use techniques focused in the details we will introduce too many noise to the system and in the end they do not help us to achieve our goals.

## 7.1 Future work

We can think of many aspects worth being investigated in the future.

- First of all, we identified the necessity of collecting more real texts to build a larger corpus and make more experiments to be able to make an more extensive analysis of the data.

- We also can try to apply other machine learning techniques to do the classification task to implement the PoS-tagging task, in particular one can think of using one of the taggers that we mention in the *Section*2.1. We can think of using a Perceptron algorithm that have reported good results also in this kind of classification tasks(RR11).

- Moreover, we can design a more efficient guessing module by developing the ideas that we can obtain after the analysis of larger corpora.

- Another idea could be to take advantage of a spell checker or spell corrector to correct the input text before it is processed by the tagger. The problem is that usually those are interactive tools and not an automatic ones. Given an input, they do not return an output, they need the interaction with the user to build a final output. This implies a large time of data processing. Hence, it could be a good idea to develop an automatic or semi-automatic spell corrector to introduce it in the running pipeline of the SVMTagger. One can control, by means of several options, how to make the substitutions of the words identified as wrong written words.

- Another idea to be explored is to study the possibility of identifying and fixing not only spelling confusions that could be caused by the proximity of the characters in the keyboard but also looking at spelling errors caused by phonetic similarities. This approach implies the treatment of the texts from a phonetical point of view, and, of course, the treatment of phonetical representations of the words, which could be a large and costful process and we will also need the help of a professional linguistics.

Summarizing, we can say that we achieved good results through our experiments, although one can expect better ones because there is still room for improvement (up to 97% accuracy achieved by taggers tested on well formed data). We have the boundary of the lack of annotated real non-standard data that limits our experiments. However, we have learnt a lot from the texts we have handeled through our experiments and

also from other types of texts (blogs, reviews, Twitter, Facebook, e-mails, etc.) and other languages. We learnt not only extracting information about common vocabulary or common errors that appear in texts, also about the difficulty of this kind of texts for learning generalizations that correctlyl appply to new corpora. We need to be able to characterize the particular mistakes as well, and this is a trade off. Language variability is very large in non-standard texts. In short, that it is difficult to make good generalizations when one try to solve particular errors and also we do not have many data to be able to make statistical inferences.

# Appendix A

## A.1 Penn Treebank

We used the Penn Treebank set in our experiments. In Figure A.1 are shown the tags used in our tagging experiments.

## A.2 SVMTool files

A general configuration file to train the SVMTagger looks like in Table A.1.

## A.3 Guessing module Additional information

### A.3.1 Weight Matrix for the Ponderate Levenshtein's distance

The A.3.1 matrix is the one used in the development of the ponderate Levenshtein's distance that take into account the distribution of the keyboard.

| SVMTool | *Penn Treebank Tagset* |
|---|---|
| **CC** | Coordinating conjunction<br>**e.g.,** and,but,or... |
| **CD** | Cardinal Number |
| **DT** | Determiner |
| **EX** | Existential *there* |
| **FW** | Foreign Word |
| **IN** | Preposition or subordinating conjunction |
| **JJ** | Adjective |
| **JJR** | Adjective, comparative |
| **JJS** | Adjective, superlative |
| **LS** | List Item Marker |
| **MD** | Modal<br>**e.g.,** can, could, might, may... |
| **NN** | Noun, singular or mass |
| **NNP** | Proper Noun, singular |
| **NNPS** | Proper Noun, plural |
| **NNS** | Noun, plural |
| **PDT** | Predeterminer<br>**e.g.,** all, both ... when they precede an article |
| **POS** | Possessive Ending<br>**e.g.,** Nouns ending in 's |
| **PRP** | Personal Pronoun<br>**e.g.,** I, me, you, he... |
| **PRP$** | Possessive Pronoun<br>**e.g.,** my, your, mine, yours... |
| **RB** | Adverb<br>Most words that end in -ly as well as degree words like quite, too and very |
| **RBR** | Adverb, comparative<br>Adverbs with the comparative ending -er, with a strictly comparative meaning. |
| **RBS** | Adverb, superlative |
| **RP** | Particle |
| **SYM** | Symbol<br>Should be used for mathematical, scientific or technical symbols |
| **TO** | *to* |
| **UH** | Interjection<br>**e.g.,** uh, well, yes, my... |
| **VB** | Verb, base form<br>subsumes imperatives, infinitives and subjunctives |
| **VBD** | Verb, past tense<br>includes the conditional form of the verb to be |
| **VBG** | Verb, gerund or persent participle |
| **VBN** | Verb, past participle |
| **VBP** | Verb, non-3rd person singular present |
| **VBZ** | Verb, 3rd person singular present |
| **WDT** | Wh-determiner<br>**e.g.,** which, and *that* when it is used as a relative pronoun |
| **WP** | Wh-pronoun<br>**e.g.,** what, who, whom... |
| **WP$** | Possessive wh-pronoun<br>**e.g.,** whose |
| **WRB** | Wh-adverb<br>**e.g.,** how, where why |

**Figure A.1:** Penn Treebank(Mar) tagset.

```
TRAINSET = /home/me/WSJ/WSJTP.TRAIN
SVMDIR = /home/me/soft/svmlight/
NAME = WSJTP.912k
R = /home/me/WSJ/WSJ.repair.DICT
W = 5 2
F = 2 100000
X = 3
Dratio = 0.001
REMOVE_FILES = 1

do M0 LRL
```

```
# M0 ambiguous-right [default]

# known word feature set definition
A0k = w(-2) w(-1) w(0) w(1) w(2) w(-2,-1) w(-1,0) w(0,1) w(-1,1) w(1,2)
w(-2,-1,0) w(-2,-1,1) w(-1,0,1) w(-1,1,2) w(0,1,2) p(-2) p(-1) p(-2,-1)
p(-1,1) p(1,2) p(-2,-1,1) p(-1,1,2) k(0) k(1) k(2) m(0) m(1) m(2)

# unknown word feature set definition
A0u = w(-2) w(-1) w(0) w(1) w(2) w(-2,-1) w(-1,0) w(0,1) w(-1,1) w(1,2)
w(-2,-1,0) w(-2,-1,1) w(-1,0,1) w(-1,1,2) w(0,1,2) p(-2) p(-1) p(-2,-1)
p(-1,1) p(1,2) p(-2,-1,1) p(-1,1,2) k(0) k(1) k(2) m(0) m(1) m(2) a(2)
a(3) a(4) z(2) z(3) z(4) ca(1) cz(1) L SA AA SN CA CAA CP CC CN MW
```

**Table A.1:** SVMTlearn configuration file.

| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **A** | 0.0 | 4.12 | 2.23 | 2.0 | 2.24 | 3.0 | 4.0 | 5.0 | 7.07 | 6.0 | 7.0 | 8.0 | 6.08 | 5.10 | 8.06 | 9.06 | 1.0 | 3.16 | 1.0 | 4.12 | 6.08 | 3.16 | 1.41 | 1.41 | 5.10 | 1.0 |
| **B** | 4.12 | 0.0 | 2.0 | 2.24 | 2.83 | 1.41 | 1.0 | 1.41 | 3.61 | 2.24 | 3.16 | 4.12 | 2.0 | 1.0 | 4.47 | 5.39 | 4.47 | 2.24 | 3.16 | 2.0 | 2.83 | 1.0 | 3.61 | 3.0 | 2.24 | 4.0 |
| **C** | 2.23 | 2.0 | 0.0 | 1.0 | 2.0 | 1.41 | 2.24 | 3.16 | 5.39 | 4.12 | 5.10 | 6.08 | 4.0 | 3.0 | 6.33 | 7.28 | 2.83 | 2.24 | 1.41 | 2.83 | 4.47 | 1.0 | 2.24 | 1.0 | 3.61 | 2.0 |
| **D** | 2.0 | 2.24 | 1.0 | 0.0 | 1.0 | 1.0 | 2.0 | 3.0 | 5.10 | 4.0 | 5.0 | 6.0 | 4.12 | 3.16 | 6.08 | 7.07 | 2.24 | 1.41 | 1.0 | 2.24 | 4.12 | 1.41 | 1.41 | 1.41 | 3.16 | 2.24 |
| **E** | 2.24 | 2.83 | 2.0 | 1.0 | 0.0 | 1.41 | 2.24 | 3.16 | 5.0 | 4.12 | 5.10 | 6.08 | 4.47 | 3.61 | 6.0 | 7.0 | 2.0 | 1.0 | 1.41 | 2.0 | 4.0 | 2.24 | 1.0 | 1.24 | 3.0 | 2.83 |
| **F** | 3.0 | 1.41 | 1.41 | 1.0 | 1.41 | 0.0 | 1.0 | 2.0 | 4.12 | 3.0 | 4.0 | 5.0 | 3.16 | 2.24 | 5.10 | 6.08 | 3.16 | 1.0 | 2.0 | 1.41 | 3.16 | 1.0 | 2.24 | 2.24 | 2.24 | 3.16 |
| **G** | 4.0 | 1.0 | 2.24 | 2.0 | 2.24 | 1.0 | 0.0 | 1.0 | 3.16 | 2.0 | 3.0 | 4.0 | 2.24 | 1.41 | 4.12 | 5.10 | 4.12 | 1.41 | 3.0 | 1.0 | 2.24 | 1.41 | 3.16 | 3.16 | 1.41 | 4.12 |
| **H** | 5.0 | 1.41 | 3.16 | 3.0 | 3.16 | 2.0 | 1.0 | 0.0 | 2.24 | 1.0 | 2.0 | 3.0 | 1.41 | 1.0 | 3.16 | 4.12 | 5.10 | 2.24 | 4.0 | 1.41 | 1.41 | 2.24 | 4.12 | 4.12 | 1.0 | 5.10 |
| **I** | 7.07 | 3.61 | 5.39 | 5.10 | 5.0 | 4.12 | 3.16 | 2.24 | 0.0 | 1.41 | 1.0 | 1.41 | 2.24 | 2.83 | 1.0 | 2.0 | 7.0 | 4.0 | 6.08 | 3.0 | 1.0 | 4.47 | 6.0 | 6.33 | 2.0 | 7.28 |
| **J** | 6.0 | 2.24 | 4.12 | 4.0 | 4.12 | 3.0 | 2.0 | 1.0 | 1.41 | 0.0 | 1.0 | 2.0 | 1.0 | 1.41 | 2.24 | 3.16 | 6.08 | 3.16 | 5.0 | 2.24 | 1.0 | 3.16 | 5.10 | 5.10 | 1.41 | 6.08 |
| **K** | 7.0 | 3.16 | 5.10 | 5.0 | 5.10 | 4.0 | 3.0 | 2.0 | 1.0 | 1.0 | 0.0 | 1.0 | 1.41 | 2.24 | 1.41 | 2.24 | 7.07 | 4.12 | 6.0 | 3.16 | 1.41 | 4.12 | 6.08 | 6.08 | 2.24 | 7.07 |
| **L** | 8.0 | 4.12 | 6.08 | 6.0 | 6.08 | 5.0 | 4.0 | 3.0 | 1.41 | 2.0 | 1.0 | 0.0 | 2.24 | 3.16 | 1.0 | 1.41 | 8.06 | 5.10 | 7.0 | 4.12 | 2.24 | 5.10 | 7.07 | 7.07 | 3.16 | 8.06 |
| **M** | 6.08 | 2.0 | 4.0 | 4.12 | 4.47 | 3.16 | 2.24 | 1.41 | 2.24 | 1.0 | 1.41 | 2.24 | 0.0 | 1.0 | 2.83 | 3.61 | 6.33 | 3.61 | 5.10 | 2.83 | 2.0 | 3.0 | 5.39 | 5.0 | 2.24 | 6.0 |
| **N** | 5.10 | 1.0 | 3.0 | 3.16 | 3.61 | 2.24 | 1.41 | 1.0 | 2.83 | 1.41 | 2.24 | 3.16 | 1.0 | 0.0 | 3.61 | 4.47 | 5.39 | 2.83 | 4.12 | 2.24 | 2.24 | 2.0 | 4.47 | 4.0 | 2.0 | 5.0 |
| **O** | 8.06 | 4.47 | 6.33 | 6.08 | 6.0 | 5.10 | 4.12 | 3.16 | 1.0 | 2.24 | 1.41 | 1.0 | 2.83 | 3.61 | 0.0 | 1.0 | 8.0 | 5.0 | 7.07 | 4.0 | 2.0 | 5.39 | 7.0 | 7.28 | 3.0 | 8.25 |
| **P** | 9.10 | 5.39 | 7.28 | 7.07 | 7.0 | 6.08 | 5.10 | 4.12 | 2.0 | 3.16 | 2.24 | 1.41 | 3.61 | 4.47 | 1.0 | 0.0 | 9.0 | 6.0 | 8.06 | 5.0 | 3.0 | 6.33 | 8.0 | 8.25 | 4.0 | 9.22 |
| **Q** | 1.0 | 4.47 | 2.83 | 2.24 | 2.0 | 3.16 | 4.12 | 5.10 | 7.0 | 6.08 | 7.07 | 8.06 | 6.33 | 5.39 | 8.0 | 9.0 | 0.0 | 3.0 | 1.41 | 4.0 | 6.0 | 3.61 | 1.0 | 2.24 | 5.0 | 2.0 |
| **R** | 3.16 | 2.24 | 2.24 | 1.41 | 1.0 | 1.0 | 1.41 | 2.24 | 4.0 | 3.16 | 4.12 | 5.10 | 3.61 | 2.83 | 5.0 | 6.0 | 3.0 | 0.0 | 2.24 | 1.0 | 3.0 | 2.0 | 2.0 | 2.83 | 2.0 | 3.61 |
| **S** | 1.0 | 3.16 | 1.41 | 1.0 | 1.41 | 2.0 | 3.0 | 4.0 | 6.08 | 5.0 | 6.0 | 7.0 | 5.10 | 4.12 | 7.07 | 8.06 | 1.41 | 2.24 | 0.0 | 3.16 | 5.10 | 2.24 | 1.0 | 1.0 | 4.12 | 1.41 |
| **T** | 4.12 | 2.0 | 2.83 | 2.24 | 2.0 | 1.41 | 1.0 | 1.41 | 3.0 | 2.24 | 3.16 | 4.12 | 2.83 | 2.24 | 4.0 | 5.0 | 4.0 | 1.0 | 3.16 | 0.0 | 2.0 | 2.24 | 3.0 | 3.61 | 1.0 | 4.47 |
| **U** | 6.08 | 2.83 | 4.47 | 4.12 | 4.0 | 3.16 | 2.24 | 1.41 | 1.0 | 1.0 | 1.41 | 2.24 | 2.0 | 2.24 | 2.0 | 3.0 | 6.0 | 3.0 | 5.10 | 2.0 | 0.0 | 3.61 | 5.0 | 5.39 | 1.0 | 6.33 |
| **V** | 3.16 | 1.0 | 1.0 | 1.41 | 2.24 | 1.0 | 1.41 | 2.24 | 4.47 | 3.16 | 4.12 | 5.10 | 3.0 | 2.0 | 5.39 | 6.33 | 3.61 | 2.0 | 2.24 | 2.24 | 3.61 | 0.0 | 2.83 | 2.0 | 2.83 | 3.0 |
| **W** | 1.41 | 3.61 | 2.24 | 1.41 | 1.0 | 2.24 | 3.16 | 4.12 | 6.0 | 5.10 | 6.08 | 7.07 | 5.39 | 4.47 | 7.0 | 8.0 | 1.0 | 2.0 | 1.0 | 3.0 | 5.0 | 2.83 | 0.0 | 2.0 | 4.0 | 2.24 |
| **X** | 1.41 | 3.0 | 1.0 | 1.41 | 2.24 | 2.24 | 3.16 | 4.12 | 6.33 | 5.10 | 6.08 | 7.07 | 5.0 | 4.0 | 7.28 | 8.25 | 2.24 | 2.83 | 1.0 | 3.61 | 5.39 | 2.0 | 2.0 | 0.0 | 4.47 | 1.0 |
| **Y** | 5.10 | 2.24 | 3.61 | 3.16 | 3.0 | 2.24 | 1.41 | 1.0 | 2.0 | 1.41 | 2.24 | 3.16 | 2.24 | 2.0 | 3.0 | 4.0 | 5.0 | 2.0 | 4.12 | 1.0 | 1.0 | 2.83 | 4.0 | 4.47 | 0.0 | 5.39 |
| **Z** | 1.0 | 4.0 | 2.0 | 2.23 | 2.83 | 3.16 | 4.12 | 5.10 | 7.28 | 6.08 | 7.07 | 8.06 | 6.0 | 5.0 | 8.25 | 9.22 | 2.0 | 3.61 | 1.41 | 4.47 | 6.32 | 3.0 | 2.24 | 1.0 | 5.39 | 0.0 |

### A.3.2 List of Common Words

The following list is the list of common words that we used in the experiments from the *Chapter*5 using the guessing module. We used that list to enhance the dictionary used by the tagger.

```
{
u  561 1 PRP 561
  U  561 1 PRP 561
  r  12 1 VBP 12
  R  12 1 VBP 12

  i.e.  1 1 FW 1
  I.e.  1 1 FW 1
  i.E.  1 1 FW 1
  I.E.  1 1 FW 1

  a.m.  1 1 RB 1
  A.m.  1 1 RB 1
  a.M.  1 1 RB 1
  A.M.  1 1 RB 1
  p.m.  1 1 RB 1
  P.m.  1 1 RB 1
```

```
p.M.  1 1 RB 1
P.M.  1 1 RB 1

a.k.a.  1 1 JJ 1
a.K.a.  1 1 JJ 1
a.k.A.  1 1 JJ 1
A.k.a.  1 1 JJ 1
A.K.a.  1 1 JJ 1
a.K.A.  1 1 JJ 1
A.k.A.  1 1 JJ 1
A.K.A.  1 1 JJ 1

Google  1 1 NNP 1
google  1 1 NNP 1
GOOGLE  1 1 NNP 1
yahoo  1 1 NNP 1
Yahoo  1 1 NNP 1
YAHOO  1 1 NNP 1
facebook  1 1 NNP 1
Facebook  1 1 NNP 1
FACEBOOK  1 1 NNP 1
Internet  1 1 NNP 1
internet  1 1 NNP 1
INTERNET  1 1 NNP 1

OMG  1 1 UH 1
oMG  1 1 UH 1
OmG  1 1 UH 1
OMg  1 1 UH 1
Omg  1 1 UH 1
oMg  1 1 UH 1
omG  1 1 UH 1
omg  1 1 UH 1

LOL  1 1 UH 1
LOl  1 1 UH 1
```

```
  LoL  1 1 UH 1
  lOL  1 1 UH 1
  Lol  1 1 UH 1
  lOl  1 1 UH 1
  loL  1 1 UH 1
  lol  1 1 UH 1

  WTF  1 1 UH 1
  wtf  1 1 UH 1
  Wtf  1 1 UH 1
  wTf  1 1 UH 1
  wtF  1 1 UH 1
  WTf  1 1 UH 1
  wTF  1 1 UH 1
  WtF  1 1 UH 1

  hi  1 1 UH 1
  Hi  1 1 UH 1
  HI  1 1 UH 1

}
```

# References


[ASS07] S. Abney, R. E. Schapire, and Y. Singer. Boosting applied to tagging and ppattachment. In *Proceedings of EMNLP/VLC 99*, 2007. 23

[BMP06] John Blitzer, Ryan McDonald, and Fernando Pereira. Domain adaptation with structural correspondence learning. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pages 120–128, Sydney, Australia, July 2006. Association for Computational Linguistics. 12

[Bra00] T. Brants. Tnt - a statistical part-of-speech tagger. In *Proceedings of the Sixth ANLP*, 2000. 2, 13

[Bri92] Eric Brill. A simple rule-based part of speech tagger. In *Speech and Natural Language: Proceedings of a Workshop Held at Harriman, New York, February 23-26, 1992*, pages 112–116, 1992. 12

[Bri95] Eric Brill. Transformation-based-error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics*, 21(4):543–564, 1995. 3, 12, 13, 15, 16, 72

[Byr13] Bill Byrne. Faust project ( european communitys seventh framework programme (fp7/2007-2013) under grant agreement number 247762 (faust, fp7-ict-2009-4-247762)). In *Develop interactive machine translation (MT) systems which adapt rapidly and intelligently in response to user feedback*, 2010-2013. 2, 4, 5, 30

[CA04] Ciprian Chelba and Alex Acero. Adaptation of maximum entropy capitalizer: Little data can help a lot. In Dekang Lin and Dekai Wu,

editors, *Proceedings of EMNLP 2004*, pages 285–292, Barcelona, Spain, July 2004. Association for Computational Linguistics. 12

[CCR10] Ming-Wei Chang, Michael Connor, and Dan Roth. The necessity of combining adaptation methods. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 767–777, Cambridge, MA, October 2010. Association for Computational Linguistics. 11

[CST00] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines*. Cambridge University Press, 2000. 21, 22

[DIKS10] Hal Daumé III, Abhishek Kumar, and Avishek Saha. Frustratingly easy semi-supervised domain adaptation. In *Proceedings of the 2010 Workshop on Domain Adaptation for Natural Language Processing*, pages 53–59, Uppsala, Sweden, July 2010. Association for Computational Linguistics. 12

[FHN00] R. Florian, J.C. Henderson, and G. Ngai. Coaxing confidence from an old friend: Probabilistic classifications from transformation rule lists. In *Proceedings of EMNLP-SIGDAT 2000*, Hong Kong, China, October 2000. Association for Computational Linguistics. 13

[FM09] Jenny Rose Finkel and Christopher D. Manning. Hierarchical bayesian domain adaptation. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 602–610, Boulder, Colorado, June 2009. Association for Computational Linguistics. 12

[FN01a] R. Florian and G. Ngai. Multidimensional transformational-based learning. In *Proceedings of the fifth Conference on Computational Natural Language Learning, CoNLL 2001*, pages 1–8, July 2001. 13

[FN01b] R. Florian and G. Ngai. Transformation-based learning in the fast lane. In *Proceedings of North Americal ACL 2001*, pages 40–47, June 2001. 5, 13

[Geh83] Manfred Gehrke. A multilevel approach to handle non-standard input. In *First Conference of the European Chapter of the Association for Computatlona Linguistics*, pages 183–187, 1983. 10

[GM04] Jesús Giménez and Lluís Màrquez. Svmtool: A general pos tagger generator based on support vector machines. In *Proceedings of the 4th LREC*, Lisbon, Portugal, 2004. 3, 4

[GMTJGM06] Jesus Gonzalez Marti, Jose Antonio Troyano Jimenez, and David Gonzalez Maline. Adaptación del método de etiquetado no supervisado tbl. In *Procesamiento del lenguaje natural*, pages 5–10. Sociedad Española para el Procesamiento del Lenguaje Natural, September 2006. 12

[Hig00] Derrick Higgins. The use of error tags in artfl's encyclopedie: Does good error identification lead to good error correction? In *Proceedings of the ANLP-NAACL 2000 Student Research Workshop*, pages 30–34, 2000. 11

[IEL+93] Aduriz I, Agirre E, Alegria L, Arregi X, Arriola J.M, Artola X, Diaz de Ilarraza A, Ezeiza N, Maritxalar M, Sarasola K, and Urkia M. A morphological analysis based method for spelling correction. In *Sixth Conference of the European Chapter of the Association for Computational Linguistics*, page 463, 1993. 10

[Joa99] T. Joachims. *Making large-Scale SVM Learning Practical*. MIT-Press, 1999. 25

[JR10] Prateek Jindal and Dan Roth. Using domain knowledge and domain-inspired discourse model for coreference resolution for clinical narratives. In *Journal of American Medical Informatics Association, JAMIA*, Urbana, Illinois, USA, July 2010. 12

[JZ07] Jing Jiang and ChengXiang Zhai. Instance weighting for domain adaptation in nlp. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 264–271, Prague, Czech Republic, June 2007. Association for Computational Linguistics. 12

[KH82] Anthony S. Kroch and Donald Hindle. On the linguistic character of non-standard input. In *Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics*, pages 161–163, Toronto, Ontario, Canada, June 1982. Association for Computational Linguistics. 10

[KR11] Gourab Kundu and Dan Roth. Adapting text instead of the model: An open domain approach. In *Proceedings of the Fifteenth Conference on Computational Natural Language Learning*, pages 229–237, Portland, Oregon, USA, June 2011. Association for Computational Linguistics. 12

[Kwa82] Stan Kwasny. Ill-formed and non-standard language problems. In *Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics*, pages 164–166, Toronto, Ontario, Canada, June 1982. Association for Computational Linguistics. 10

[Lag99a] T Lager. The mu-tbl lite : A small, extensible transformation-based learner,. In *Poster paper presented at the Ninth Conference of the European Chapter of the Association for Computational Linguistics (EACL'99)*, Bergen, June 8-12 1999. 13

[Lag99b] T Lager. The mu-tbl system: Logic programming tools for transformation-based learning. In *Proceedings of the 3rd International Workshop on Computational Natural Language Learning (CoNLL'99)*, Bergen, 1999. 13

[Mar] Mitchell Marcus. Penn treebank project, http://www.cis.upenn.edu/ treebank/. 29, 92

[Mar82] Mitchell P. Marcus. Building non-normative systems - the search for robustness an overview. In *Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics*, pages 152–152, Toronto, Ontario, Canada, June 1982. Association for Computational Linguistics. 10

[MBTVS06] John E. Miller, Michael Bloodgood, Manabu Torii, and K. Vijay-Shanker. Rapid adaptation of pos tagging for domain specific uses. In *Proceedings of the HLT-NAACL BioNLP Workshop on Linking Natural Language and Biology*, pages 118–119, New York, New York, June 2006. Association for Computational Linguistics. 12

[MCJ10] David McClosky, Eugene Charniak, and Mark Johnson. Automatic domain adaptation for parsing. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 28–36, Los Angeles, California, June 2010. Association for Computational Linguistics. 12

[MTVS07] John E. Miller, Manabu Torii, and K. Vijay-Shanker. Adaptation of pos tagging for multiple biomedical domains. In *Biological, translational, and clinical language processing*, pages 179–180, Prague, Czech Republic, June 2007. Association for Computational Linguistics. 12

[MWM00] Kyongho Min, William H. Wilson, and Yoo-Jin Moon. Typographical and orthographical spelling error correction. In *Proceedings of the Second International Conference on Language Resources and Evaluation (LREC-2000)*, Athens, Greece, May 2000. European Language Resources Association (ELRA). 10

[PKHC98] Junsik Park, Jung-Goo Kang, Wook Hur, and Key-Sun Choi. Machine aided error-correction environment for korean morphological analysis and part**of-speech tagging. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Volume 2*, pages 1015–1019, Montreal, Quebec, Canada, August 1998. Association for Computational Linguistics. 10

[QBB+08] Martí Quixal, Toni Badia, Francesc Benavent, Jose R. Boullosa, Judith Domingo, Bernat Grau, and Guillem Massó Oriol Valentín. User-centred design of error correction tools. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC-*

*08)*, Marrakech, Morocco, May 2008. European Language Resources Association (ELRA). 10

[Rat96] A. Ratnaparkhi. A maximum entropy part–of–speech tagger. In *Proceedings of the 1st EMNLP Conference*, 1996. 2, 13

[Rot11] Dan Roth. Adaptation without retraining. In *NIPS 2011 Domain Adaptation Workshop*, Sierra Nevada, Granada,Spain, December 2011. Neural Information Processing Systems Foundation. 11

[RR10] Alla Rozovskaya and Dan Roth. Generating confusion sets for context-sensitive error correction. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 961–970, Cambridge, MA, October 2010. Association for Computational Linguistics. 11

[RR11] Alla Rozovskaya and Dan Roth. Algorithm selection and model adaptation for esl correction tasks. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Techologies*, pages 924–933, Portland, Oregon, USA, June 2011. Association for Computational Linguistics. 11, 89

[Sch94] Hinrich Schfitze. Part-of-speech tagging using a variable memory markov model. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, pages 181–187, Las Cruces, New Mexico, USA, June 1994. Association for Computational Linguistics. 13

[UPRR10] Shulamit Umansky-Pesin, Roi Reichart, and Ari Rappoport. A multi-domain web-based algorithm for pos tagging of unknown words. In *Coling 2010: Posters*, pages 1274–1282, Beijing, China, August 2010. Coling 2010 Organizing Committee. 13