GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

ISSN 1612-6793
Nr. ZFI-BM-2017-10

Bachelor's Thesis

submitted in partial fulfilment of the
requirements for the course "Applied Computer Science"

EnsembleForest: A Classifier Combination Method on the example of Part-of-Speech Tagging

David Steding

Institute of Computer Science

Bachelor's and Master's Theses
of the Center for Computational Sciences
at the Georg-August-Universität Göttingen

10. April 2017

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
🌍 www.informatik.uni-goettingen.de

First Supervisor: Dr. Marco Büchler
Second Supervisor: Prof. Dr. Florentin Andreas Wörgötter

I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Göttingen, 10. April 2017

Abstract

In this work, the classifier combination algorithm EnsembleForest is introduced. On the use case of part-of-speech (POS) Tagging models of this algorithm reach comparable good results. Compared to the best individual classifier combined the algorithm yields higher accuracy values. Combining a selection of algorithms performing POS Tagging an error reduction of 15.35% is achieved, compared to the best individual algorithm (TreeTagger). In addition, this work shows that it is beneficial to apply tagset mapping on a POS Tagging setup after classifications have been created, in order to reduce the number of misclassifications. By application of the EnsembleForest algorithm and succeeding usage of tagset mapping, an error reduction of up to 60.74% has been achieved, compared to the best individual algorithm results, which were gained beforehand tagset mapping.

Contents

List of Figures

List of Tables

List of Abbreviations

Chapter 1

Introduction

"Whoever has the choice has the agony." (A german aphorism)

How to make the right decision is the subject of investigation of the present work. In computer science many algorithms exist, which are ought to classify elements by a certain set of possible classifications. In some domains it is not trivial to always choose the correct classification. Hence, several distinct classifiers may select several different classifications for the same task. This work introduces the classifier combination method *EnsembleForest* to cope with this issue and to exploit the knowledge of other classifiers in order to create an improved classifier. The method is examined on the use case of part-of-speech (POS) tagging, at which, roughly spoken, words of natural languages as English or German have to be assigned with their corresponding word class.

This work focuses on the classifier combination task, but nevertheless investigates the POS tagging task as well. Since, POS tagging is not in the main focus, it is only of secondary interest, to reach state-of-the-art POS tagging performance results. Instead, the subject of the work is much more to obtain good classifier combination results.

At first, basic knowledge about the investigated topics is provided. Whereby definitions, issues and important examples are named for POS tagging and for classifier combination methods. Building on this, the *EnsembleForest* algorithm is introduced in chapter 4. In the chapter 5 the experimental setup, its goals and components are explained. The subsequent chapter 6 concerns all experimental executions and associated results. Afterwards, the results are discussed, and a final conclusion is made in chapter 7. Finally, some future work is suggested.

Chapter 2

Part-of-Speech Tagging

POS Tagging a subtask of statistical Natural Language Processing (NLP). Manning et al. define that statistical NLP "comprises all quantitative approaches to automated language processing, including probabilistic modeling, information theory, and linear algebra" [1]. In this context automated language processing has two main objectives, first the extraction of information from language, and second vice versa the generation of language from information. NLP involves many tasks as, voice recognition, tokenization, lemmatization, semantic analysis and POS Tagging. Voice recognition converts spoken language into written text. Tokenization splits texts into single words and punctuations which are referred to as tokens. Lemmatization reduces words to their base form. By semantic analysis the meaning, the information of a text is extracted. For this work, only POS Tagging is of interest, so all the other mentioned steps are not further examined.

2.1 Introduction to POS Tagging

POS Tagging is a part of syntactic analysis, the step of parsing a text into a more machine readable format. It is the process of assigning part-of-speech categories to the occurrences of tokens in a text. This is important due to the fact that other NLP tasks as *baseform-reduction* require these part-of-speeches to be assigned to the occurrences of each token. As described above, every single word and every punctuation of a text is a token, or with the words of Manning et al.; tokens are individual occurrences of something in a text (cf. [2]). For simplification, in this work a word or punctuation is also called token if not a specific occurence of it is meant. A POS category groups all tokens in a text with similar grammatical functions together in word classes. Abbreviations for these categories are called tags. The set of tags used for a POS Tagging task is known as the tagset. A standard English tagset like the *Brown* tagset (cf. [3]) contains categories as: nouns, verbs, adjectives, adverbs, pronouns, prepositions, conjunctions, interjections, numerals, articles or determiners. Such word classes can be either open or closed. An open word class accepts new

tokens while closed classes commonly have a fixed set of tokens as prepositions or determiners do. For example, the class of pronouns is closed, while the class of verbs is open. Obviously, it is more easy to assign closed word classes, since it is clearly defined if a token does belong to that class or not. Difficult are tokens being ambiguous in their grammatical function. Those tokens show their POS only within the context of the sentence they occur in. So for POS Tagging texts are split into sentences, hence a token part-of-speech may depend on other tokens from that sentence.

"It was a good race!" (2.1)

At the example 2.1, the token *race* clearly is a noun but in another context it could also be a verb. Furthermore, there are sentences which have several meanings depending on the assigned tags. Although the first interpretation in the next example 2.2 most probably will not fit the intent of the linguistic utterance nevertheless it is grammatically correct. The ambiguity in semantics of this sentence depends on the assigned tags.

"We saw her duck." (2.2)

1. Duck tagged as a noun. She has an animal, a duck. We saw that duck.
2. The verb to duck. We saw her moving her head or body downwards to avoid being hit or seen.

These ambiguities encourage to explain the correctness of POS Tagging in detail.

2.2 Correctness of POS Tagging

Defining correctness of POS Tagging is essential. A tag assigned to a token is correct if the tag corresponds to the token's part-of-speech. Whereby, the tokens part-of-speech is assumed to depend on the grammar of the used language, the surrounding sentence and the part-of-speeches of the surrounding sentence. Therefore, looking at the complete sentence and all the assigned tags can be required to check correctness for an assigned tag. A solution for the POS Tagging of a sentence $s \in Tokens^n$ is then a combination $c \in TagSet^n$ so that for every t_k, c_k is defined to be its part-of-speech (t_k = k-th token of s, c_k = k-th tag of c). Sometimes there is more than one grammatically correct combination of tags for a sentence. In this case, there are several combinations $c \in TagSet^n$ for a single sentence s. For modeling multiple such correct combinations a matrix C_s referred to a sentence s is defined:

$$C_s = \begin{pmatrix} c_1 & c_2 & \dots & c_m \end{pmatrix} \in TagSet^{n \times m}, where$$
$$n = \text{number of tokens in } s$$
$$m = \text{number of combinations } c_i,$$
$$c_i \in TagSet^n \text{ a correct combination of tags for } s,\ i = 1 \dots m$$

Then, a prediction $P \in TagSet^n$ of tags for a sentence s is correct ($corSen(P, C_s) = true$) if there is a $c_i \in C_s$ with $P = c_i$. Formally:

$$corSen(P, C_s) = \begin{cases} 1 \text{ or } \textit{true}, & \exists c_i \in C_s : P = c_i \\ 0 \text{ or } \textit{false}, & \text{otherwise} \end{cases}$$

Now, imagine a P and C_s so that $corSen(P, C_s)$ is false. For example:

$$P = \begin{pmatrix} a \\ d \\ x \\ y \\ w \end{pmatrix}, C_s = \begin{pmatrix} c_1 & c_2 \end{pmatrix}, c_1 = \begin{pmatrix} a \\ c \\ x \\ y \\ z \end{pmatrix}, c_2 = \begin{pmatrix} b \\ d \\ x \\ y \\ x \end{pmatrix}$$

It might be of interest to know which tokens were tagged wrong, or to count the number of correct tags. Obviously, the tags x,y are correctly and w is tagged wrongly. The other tags depend on the solution they are compared to. It could be that a is correct and d wrong if you look at the solution c_1, or it could be the other way round comparing them with the solution c_2. Because of that, giving a binary answer of correctness (*true* or *false*) for a or d is impossible.

Instead, define a third label *unclear* for such classifications. Then, the function $corTok(t_k, P, C_s)$ marks a tag $t_k \in P$ either *true*, *false* or *unclear*:

$$corTok(t_k, P, C_s) = \begin{cases} \textit{true}, & corSen(P, C_s) = true \text{ or } c_{i,k} = t_k \text{ for all } c_i \in C_s \\ \textit{false}, & c_{i,k} \neq t_k \text{for all } c_i \in C_s \\ \textit{unclear}, & \text{otherwise} \end{cases}$$

Now the number of correct tags can be counted. Two approaches are proposed. First, count the number of *true* marked tags in P to get the minimum number of correct classifications in (P, C_s). Second, get the maximum number of correct classifications in (P, C_s). Therefore, compare P with the c_i in C_s which is most similar to P and count the number of equal classes in those two combinations. For the example of C_s and P this would be either c_1 or c_2, both are resulting in a maximum amount of three correct tags in (P, C_s):

In this work always only one possible solution for a sentence is provided, so the following applies:

- $C_s \in TagSet^{n \times 1}$
- Each token, tag combination can be individually defined correct or wrong. So, there are no classifications whose correctness is *unclear*.

This situation simplifies classification and implementation a lot but also creates issues. For example, imagine a prediction for the tags of a sentence is done. This prediction is then compared with the given solution. If they match, the prediction is defined to be correct. But, if they do not match, it can not be said that the prediction is definitely wrong. That is, because it might be that there are several solutions allowed for this particular sentence, but only one is given. It is conceivable, that a prediction matches one of the not provided correct solutions. Following, to keep things simple, predictions not matching a given solution are said to be wrong nevertheless.

2.3 Annotation of Part-of-Speeches

It is commonly settled how POS tags are annotated if there is only one solution sequence for a sentence allowed. For each token a single tag is assigned. Then, in a written text or sentence every token is followed by a / and its assigned POS-Tag. In the following example, *dog* is tagged NN (noun) and *likes* is tagged VBZ (verb, 3rd person singular present):

"My/PRP$ dog/NN also/RB likes/VBZ eating/VBG sausage/NN ./." ([4])

2.4 Tagset Mapping

For a language word classes are predefined. However, different tagsets for the same language exist. For example, these differences appear, because some tagsets allow a more detailed annotation than others. As a result, tagsets may have different sizes. Obviously, larger tagsets provide more information than smaller ones. It is more difficult to assign tags from a big tagset than from a small tagset, corresponding to the aphorism: "Who has the choice has the agony". In this thesis, difficulty is measured by the expected performance of a random guessing classifier. Such a classifier randomly selects a tag of the tagset as its prediction. In the most simple case the tagset is made up of exactly one tag. This tag would be predicted by a random guessing classifier always. Of course, this tag would be always correct. To be less abstractive, imagine the tagset only contains the tag *Token*. Definitely every token is a token so this tag is always correct. The more elements exist in a tagset the less likely it is for a random guesser to predict the correct tag. In sum, learning a dataset annotated with a big tagset might be helpful and classification on a small solution tagset seems beneficial for the overall accuracy. Both characteristics can be exploited. Depending on the use case a minimal level of detail is requested for the classified POS tags. In an optimal case the tagset on which the classifiers are trained can be clearly mapped to a smaller tagset providing the amount of information requested. Then, every element of the originating tagset maps to at least one element of the final tagset. Following, it is evidenced formally that in this case resulting accuracy cannot decrease and instead may even increase.

Four cases exist how tags from one tagset T_1 might be mapped to tags from another tagset T_2:

1. $1:1$ - A tag $a \in T_1$ refers to exactly one tag $\alpha \in T_2$
2. $1:m$ - A tag $a \in T_1$ refers to several tags of T_2
3. $n:1$ - Several tags $a, b \in T_1$ refer to exactly one tag $\alpha \in T_2$
4. $n:m$ - Several tags of T_1 refer to several tags of T_2.

For the first case overall accuracy Acc after mapping stays the same. This is trivial. In the second case, when more detailed information is missing, the single tag of T_1 is defined to be mapped to a single tag of T_2 although several are possible. For all occurrences of the single tag in the solutions and the predictions of POS Tagging task the same tag of T_2 is chosen as mapping. This reduces the case to case one. In the third case overall accuracy of a classifier stays the same or increases. This is evidenced by:

$$\text{w.l.o.g} \quad \forall \quad \alpha \in T_2 \quad \exists \quad a, b \in T_1,$$
$$\text{s.t} \quad Mapping(a) = Mapping(b) = \alpha$$

Say for the training dataset following applies:

$$\text{Number of times a tag is the solution (the POS to a token)} \ = sol(a)$$
$$\text{Number of times a is predicted, given b is the solution} \ = pred(a \mid b)$$

Then follows for the accuracy Acc_{T_1} on tagset T_1 of tags a, b:

$$Acc_{T_1} = \frac{pred(a \mid a) + pred(b \mid b)}{sol(a) + sol(b)}$$

Execution of tagset mapping results in:

$$\begin{aligned}
& Mapping(a) = Mapping(b) = \alpha \\
\Rightarrow \quad & sol(\alpha) = sol(a) + sol(b) \\
\Rightarrow \quad & pred(\alpha \mid \alpha) = pred(a \mid a) + pred(b \mid a) + pred(a \mid b) + pred(b \mid b) \\
\Rightarrow \quad & Acc_{T_2} = \frac{pred(\alpha \mid \alpha)}{sol(\alpha)} \\
& = \frac{pred(a \mid a) + pred(b \mid a) + pred(a \mid b) + pred(b \mid b)}{sol(a) + sol(b)} \\
& = Acc_{T_1} + \frac{pred(b \mid a) + pred(a \mid b)}{sol(a) + sol(b)} \\
\text{Hence,} \quad & pred(b \mid a), pred(a \mid b) >= 0 \\
\Rightarrow \quad & Acc_{T_1} <= Acc_{T_2}
\end{aligned}$$

□

Case 4: Several tags T of T_1 refer to several tags T_x of T_2. Again this case can be reduced. For every element t in T choose one tag t_x of T_x. Now, every time t occurs it is mapped to t_x. Applying this procedure, only case 1 and case 3 situations remain. Hence, for all cases it is shown, that they are either reducible to other cases, or the overall accuracy will be equal or higher than on the original tagset. Consequently, the assertion is proven, that tagset mapping cannot decrease

overall accuracy and instead may even result in an increasement. For usage this means, if a tagger classifies part-of-speeches using a certain tagset and a smaller tagset is appropriate for the use case as well, the bigger tagset could be mapped to the smaller tagset in order to yield an increased overall accuracy of correct predicted tags.

In [5] Slav Petlov et al. performed POS Tagging on 25 corpora and different tagsets from 22 languages. After the initial tagging they mapped the original tagsets to another (in most cases smaller) tagset. In all of these experiments the tagset mapping did not decrease overall accuracy. In fact, they achieved better accuracies in most cases. These results underline the effectiveness of tagset mapping.

Chapter 3

Classifier Combination Methods

Classifier combination belongs to the field of classification problem solving. "Classification refers to the assignment of a finite set of alternatives into predefined groups; this is a general description" by Michael Doumpos and Constantin Zopounidis in [6]. Whenever elements of a dataset need to be assigned to a certain category or class, one speaks of a classification problem. Here, algorithms that solve classification problems are referred to as classifiers. Now, if a particular problem needs to be solved, one could apply not only one classifier, but several. In such cases classifier combination methods come into play. Classifier combination methods as defined by Tulyakov et al. [7] take a set of classifiers of a certain problem and somehow form a prediction with them. These methods are useful because different classifiers tend to have different predictions although they all are to classify the same problem. Those combination methods can be distinguished by several aspects. Two of them are addressed here. First, what kind of data requires a combination method as input, and second, which types of methods for combining classifiers exist. Implementations for later experiments of this thesis are not addressed in this chapter.

3.1 Combined Classifier Output Levels

Classifier combination methods expect a certain type of output from the classifiers to be combined. The output types can be categorized. This categorization helps to easily decide whether a classifier can be used by a particular method or not. The following categorization by Xu et al. [8] splits classifiers output into three levels. To determine a classifier's output level the quantity and type of its predictions per element is considered. Not decisive are additional meta information provided by the classifier.

- Abstract level: The lowest level with regard to information amount provided by the classifier. Output is a single class label or an unordered set of class labels.

- Rank level: An ordered queue of candidate classes. Called the n-best list at [7]. Top element of the sequence is the first choice for the classification.
- Measurement level: The level with the most information provided. Additional to the n-best list a confidence value for every label is returned. The value can be any number, depending on the classifier chosen classifier.

There is an information reduction process from measurement level towards abstract level. Every output of measurement level can be transformed into rank level by ignoring the confidence values. A mapping from rank level to abstract level is done by selecting the top element of the n-best list. Hence, classifier combination methods requiring only data of abstract level can be equipped with classifiers most easily since any classifier fulfills these conditions.

3.2 Examples of Combination Methods

Classifiers can be combined with each other by the usage of different approaches. A few basic and well known examples are presented here, followed by methods which have a more direct connection to this work.

3.2.1 Majority Voting

A very basic technique of combining methods is majority voting. It performs a combination of classifier outputs to create an output of rank level. From all combined classifiers only the best candidate class is picked. From these the most frequent one is chosen as result. To overcome ties a random guess among the tied candidates can be done. An extension to majority voting is averaged majority voting. Here, classifiers get a constant weight value. For example, by obtaining their accuracy on the training dataset. The class with the highest summed up weights is predicted by this method.

3.2.2 Bagging

Bootstrap aggregation (bagging) is a method to generate an ensemble of models from the same classifier architecture rather than a combination method. Nevertheless, it also defines how to combine the generated models. Bagging (as defined by Breiman [9]) takes a training dataset which is a set of elements belonging to the concerned problem and which includes solutions for every element. Several times a random subset of this training dataset is drawn. Next, the classifier is trained on each of these subsets to generate several classifier models. The method of combination for these models differs depending on the problem. For problems of numerical classifications for which the results are averaged over finite classification sets, a majority vote is established on

the results of each of these trained classifier models. Note; that the big difference of bagging to k-fold-crossvalidation is the combination method. Although they are similar, bagging is said to outperform crossvalidation [10]. In order to achieve any improvement with bagging, only unstable classifiers may be used. Here, unstable means that a small change in the training dataset results in a different behavior of the resulting model.

3.2.3 Boosting on the example of AdaBoost

Boosting deals with the question whether an almost randomly guessing classifier can be boosted into an arbitrarily accurate learning algorithm [7]. Boosting is just like bagging a method to generate an ensemble of models from the same classifier architecture. In comparison to bagging it is reported to achieve overall better results [11]. Again, subsets of a training dataset are extracted to generate different models of the same architecture. In contrast to bagging the subsets can not be generated in parallel but have to be computed incrementally. A very famous example of boosting is the *AdaBoost* algorithm (introduced by Freund and Schapire [12]). At first, it will attach every element of the training data with a weight value so that all weight values are the same. A subset is then build considering the weights such that elements with higher weights are more likely to be included in the subset. After training a model, the weights of elements which were classified wrong are increased. Repeating this procedure again and again, many more models can be obtained. After training is finished, the models are combined into a final predictor by aggregation as in bagging with one difference: Models are aggregated based on their accuracy on the training data. Freund et al. say that any unstable weak learner (classifier with accuracy just above 50%) can be boosted by *AdaBoost* to perform reasonably well [11].

3.2.4 Combinations on Measurement Level

As data output of measurement level is important, three different approaches on it shall be presented here. They are also mentioned in [7].

- Sum-rule: Add up the values provided by each classifier for all class labels and pick the one with maximum score.
- Product-rule: Multiply measurement values for every class and then output the class with the maximum product value.
- Max-rule: Pick the candidate class with the highest measurement value among all classifiers and labels.

The later introduced *EnsembleForest* algorithm utilizes the max-rule in one of its processing stages.

3.3 Stacking

Beforehand, methods were explained which directly combine outputs of classifiers to a new prediction. Output of classifiers were categorized and examples of concrete combination methods named. To proceed: every classifier combination method or rather a whole combination setup can be further categorized itself by its level of *stacking*.
Stacked Generalization (stacking) is a categorization of methods that belong to the field of classifier combination as defined by Wolpert in [13]. Stacking separates parts of classifier combination methods into levels. The lowest level (level-0) is made up of classifiers or generalizers, which make a guess on the classification problem without concerning any other guessers. Every other subsequent level consists of classifiers, that take as input the output of the previous level. The whole method is then said to be a k-leveled stacking method if it consists of k levels of classifiers. Any number of $k \in \mathbb{N}$ greater one is possible. For example, above mentioned classifier combination methods like majority voting, bagging or boosting are two-leveled stacking methods all.

Chapter 4

EnsembleForest

This chapter defines and explains the algorithm *EnsembleForest* introduced by this work. *EnsembleForest* is a new second staged classifier combination algorithm for classification problems. New, in the sense that no equal or similar algorithm was found in the literature research.

The method belongs to stacked generalization, because it uses the output of base classifiers plus additional classification models to create a new prediction. The final goal of the *EnsembleForest* algorithm is to yield a higher accuracy than each individual base classifier.

In the sense of stacking *EnsembleForests* perform on three levels, level-0, level-1 and level-2. Every level consists of generalizers with different tasks. Level-0 generalizers, called base classifiers, try to solve the classification problem itself individually. Their results are passed to the *EnsembleForest* algorithm, which is made up of the subsequent levels (level-1 and level-2). The level-1 generalizers predict performance measurement values for the base classifiers and eventually suggest another class. Finally, on level-2, the level-1 generalizer outputs are combined by a voting system to form the prediction of the *EnsembleForest* algorithm. In the following, *EnsembleForests* are explained in detail and a concrete realization is proposed.

4.1 Applicable Problems and Base Classifiers

In general, classification problems may be single-class or multi-class classification problems. For single-class classification problems the solution for an element is always exactly one single class. In contrast, for multi-class classification problems there may be many classes which form a correct solutions all together. Apart from that, classification problems can be distinguished by the scale of the allowed classification set. For example there are nominal or ordinal scales.

- Nominal - the classes differ only by name, they can be compared by $=, \neq$ operators
- Ordinal - the classes are ordered by rank; possible comparisons are: $=, \neq, <, >$

The proposed basic form of the *EnsembleForest* algorithm is designed for single-class classification problems with a nominal classification set, for example for POS Tagging. A design for multi-class classification problems and other output scales is possible, but remains a subject for a future work.

The *EnsembleForest* is a supervised learning method. Hence, it needs to be trained with a dataset containing solutions for the classification task. Dependencies between elements in the input data can be taken into account if the elements are grouped together in fixed orders. For example, at POS Tagging all tokens of a sentence may be dependent from each other. Hence, tokens are grouped by sentences in the order they appear in the sentence. In conclusion, all classification problems are applicable for this version of the *EnsembleForest* algorithm if they meet the following requirements:

- The problem is single-classed
- The solutions are of nominal scale
- Data elements are grouped by dependencies and ordered within a grouping

In order to use the *EnsembleForest* for a classifier combination task, not only a problem is required, but as well some basic classification algorithms (called base classifiers). These base classifiers must be able to predict solutions for the problem data. As long as they make at least one prediction on each input data, they are usable. To incorporate them as base classifiers into an *EnsembleForest* model, they need to be trained on the given problem itself or on a similar problem upfront.

A problem is said to be similar if following applies for a classifier trained on data of that problem: The classifier takes input data of the same structure as the original problem and the classifier predicts a set of classes to which the initially given problem is assigned to as well.

These are the restrictions for base classifiers of *EnsembleForests*. In every other aspect, the base classifiers can be diverse. They can differ in their training set, their parameter settings and in their architecture or method. Even their level of output data may be different. The output data can be of abstract, rank or measurement level and may even include additional meta information. All outputs of all the base classifiers could be used for the following processing steps of the *EnsembleForest*. Anyway, the implemented version of the algorithm does only cope with a single prediction passed over from every base classifier and does not handle additional meta information. These restrictions are implemented because they are sufficient for the use case of POS Tagging as executed in this thesis.

4.2 Level-1 Generalizers

An *EnsembleForest* has two types of level-1 generalizers, a set of basic generalizers (named base validators) and a special one (named prediction changer). Base validators measure the quality of base classifiers, while the prediction changer creates a new prediction, given all base classifiers misclassified an element.

The input for all the level-1 generalizers of the implemented version of the *EnsembleForest* algorithm is of equal structure. It is also referred to as level-1 input data. The feature set of an input element contains a single result from every base classifier. As mentioned above in section 4.1 in theory an *EnsembleForest* algorithm could take additional features as well, containing any output of the base classifier or other meta information values. This version of EnsembleForest is implemented for the use case of POS Tagging and therefore does not require to cope with such additional features. There is a special case in which the proposed version of *EnsembleForest* generates its own additional features. This special case is not further considered until section 4.4.

4.2.1 Base Validators

The aim of the basic level-1 generalizers is to predict a performance measurement value (called trust value) for every base classifier prediction. A value ranging from 0 to 1 expressing as good as possible how likely a prediction is correct. The behavior expressed by the following equation 4.1 would be optimal:

$$P(\text{ Prediction } y \text{ is correct }) = \text{ trust value for } y \tag{4.1}$$

Why could it be useful to take additional models for predicting that value? Instead, only base classifiers could be used which return a measurement value on their own. There are several reasons. First and most obvious, some (strong performing) classifiers do not output a measurement value. Second, values should be in the same range, for example a range of 2 to 15 would be inappropriate, and values should express the desired behavior (compare equation 4.1). Normalization could transform all ranges to the range $[0, 1]$, but still it would not be guaranteed that the measurement function would show the intended behavior. Imagine a measurement function which returns a trust value in range of $[0, 1]$, at which for every value below 0.5 the prediction is wrong with a probability of 1. The subsequent combinatorial level-2 generalizer would misinterpret the trust value. Third and most important aspect, dependencies between base classifiers should not be neglected. The prediction of a classifier could always be correct if another classifier predicts a particular class. Such dependencies would not be taken into account in a measurement function if the base classifiers were to measure their performance on their own. Finally, the meta information retrieved from all the base classifiers is meant to be incorporated into the measurement process

as well. These are the reasons for which the base validators, which predict a trust value for every level-0 prediction, are introduced into the *EnsembleForest* algorithm. Moreover, these base validators are the most important aspect of *EnsembleForests*.

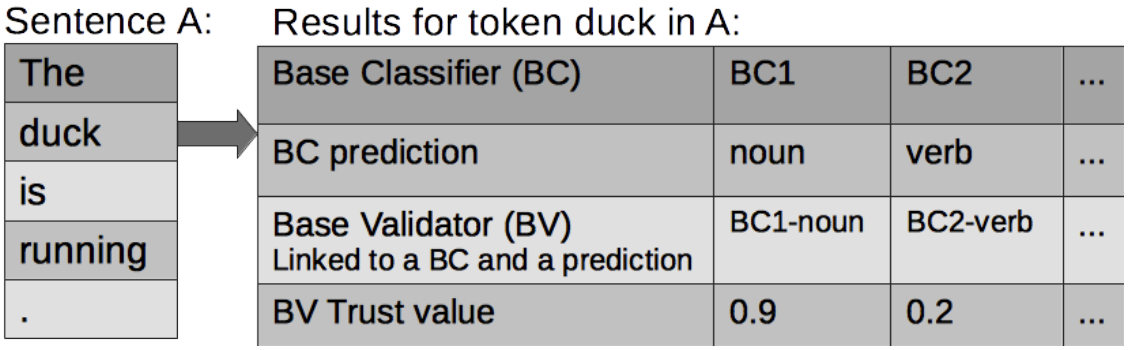

Figure 4.1: A hypothetical example of some base validators (BV) in action. Each returns a trust value, which measures how likely their linked base classifier (BC) correctly classifies the token *duck* in the sentence *A*.

Which algorithms could be used as base validators? Any classification algorithm which is able to take input data made up from a diverse set of features and which is capable to output a numerical value between 0 and 1. Only algorithms should be used which get close to the desired behavior (compare equation 4.1).

Binary decision trees serve as the base validators for the proposed version of the *EnsembleForest* algorithm. As described beforehand, the input features for these generalizers are made up of single predictions from every base classifier. So, they are of a nominal scale, which is easy to implement for decision trees. During the training of an *EnsembleForest* model, a binary decision tree model will be generated for every possible prediction of any base classifier.

That means, if there are three base classifiers which may predict classes from a set from 10 different classifications, the total amount of required validators is $3 * 10 = 30$. These base validators can be clearly distinguished by the base classifier and the classification they are linked to. All the base validators together form an ensemble of forests of base validators. Therefore, the algorithm is named *EnsembleForest*. To generate all those binary decision trees, they need to be trained. Therefore, each of them needs a training dataset. At first, the initial training dataset, the *EnsembleForest* algorithm is provided with, has to be classified by the base classifiers. Using their outputs the level-1 input dataset is generated (see figure 4.2).

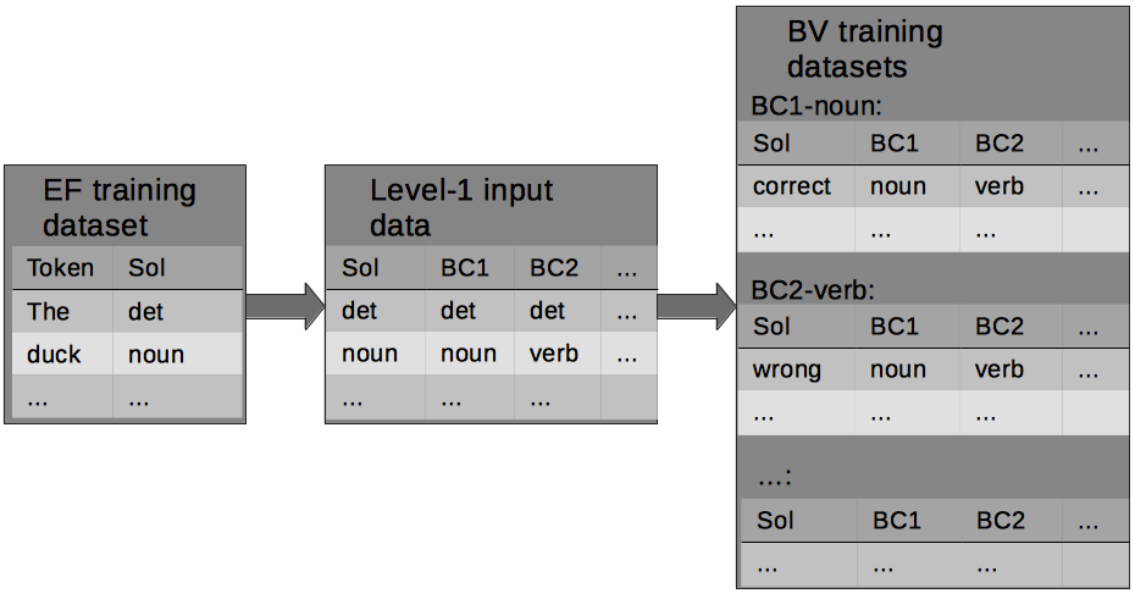

Figure 4.2: An example representation of an *EnsembleForest* (EF) training dataset at different stages. First, base classifiers (BC) classify elements of the initial training dataset to generate the level-1 input data. The base validators (BV) extract their training dataset from the level-1 input data. At BV stage the solutions (Sol) are adjusted to the needs of the BVs.

Now, every base validator includes in his own training dataset only those data elements of the level-1 input dataset for which the particular linked base classifier predicted the class corresponding to the base validator's linked classification.

The details of the implemented binary decision trees yet have to be discussed. Input data features and content of a trees training dataset should now be clear. Besides that, as the binary decision trees belong to the group of supervised learning algorithms, they need a solution for every element of their training dataset in order to be able to learn. The solution $Sol(element)$ of an element is defined to be either *correct* or *wrong*, depending on the correctness of the linked base classifier prediction (see figure 4.2).

$$Sol(element) := \begin{cases} correct, & \text{prediction of linked classifier is correct} \\ wrong, & \text{otherwise} \end{cases}$$

At every decision node in the tree the question, which is asked to split data into two subsets, is always of the same kind. It is asked, whether a single feature is of a particular value or not. In addition, only those questions are allowed which result in subsets with at least a minimum amount of remaining elements. The decision trees are constructed so that at every node the

question with the maximum information gain is asked. The information gain of a question is the difference between the entropy of the current node and the sumed up entropies of the child nodes generated by this question (see equation 4.2 for the entropy of a node). For an easy construction of the trees the common top-down induction of decision trees (TDIDT) is used. So at first, the root of a tree is created and expanded by a question if possible. Then, each node is expanded until all nodes are expanded or are leaves. A node is defined to be a leave if there are no more allowed questions or the solutions Sol for all elements at this node are either *correct* or *wrong*.

$$
\begin{aligned}
H &= -\sum_{s \in Sol_n} q_s * log_2 p_s \\
Sol_n &= \text{Set of possible solutions for elements in node } n \\
p_s &= \text{Probability that the solution for elements in } n \text{ is } s
\end{aligned}
\tag{4.2}
$$

4.2.2 Expected Error of the Base Validators

In the forests of binary decision trees of an *EnsembleForest* two trees for different base classifiers might both return a trust value of 1 for a data element, although the base classifiers predict different classes. This implies that at least one of the predicted trust values does not behave as intended by the equation 4.1. So, a motivation exists to provide every tree with an expected average error value. The expected average error value shall give a quantified hint on the tree's error. So, the base validators output three values; the predicted class of their linked base classifier, a trust value measuring how likely the predicted class is correct, and an expected average error of the base validator itself.

The expected average error value is computed by counting the errors preformed by each tree on its training data. Therefore, the training set of the *EnsembleForest* model is split into two subsets, *train* 90% and *error detection* 10%. After constructing the *base validator* trees on the *train* dataset, the generated trees classify the *error detection* set. For every classification x on this set, the average error is summed up by the difference of a tree's trust value and its classification correctness $sgn(x)$.

$$
\begin{aligned}
AvgError(tree, X_{tree}) &= \frac{1}{|X_{tree}|} \sum_{x \in X_{tree}} |TrustValue(x) - sgn(x)| \\
sgn(x) &= \begin{cases} 1, & Classification(x) \text{ is correct} \\ 0, & \text{otherwise} \end{cases} \\
X_{tree} &= \text{Results of } tree \text{ for its error detection dataset}
\end{aligned}
$$

This calculation of the expected average tree error is disadvantageous. Instead, following different computation rule should have been used: First, sum up the number of correct classifications, then subtract the expected amount of correct classifications and finally divide this difference by the total number of classifications ($|X_tree|$).

$$
\begin{aligned}
AvgError_{Improved}(tree, X_{tree}) &= \frac{1}{|X_{tree}|} |SummedTrustValue(X) - \sum_{x \in X_{tree}} sgn(x)| \\
SummedTrustValue(X) &= \sum_{x \in X_{tree}} TrustValue(x) \\
X_{tree} &= \text{Results of } tree \text{ for its error detection dataset}
\end{aligned}
$$

Imagine, a tree predicts nine out of ten cases correct and $TrustValue(x) = 0.9$ for all of those cases. Then, one would expect the average error to be zero. Actually, $AvgError_{Improved}$ is computed to be zero but $AvgError$ is not. Table 4.1 shows some more examples, for which $AvgError$ behaves bad and $AvgError_{Improved}$ works properly. Because $AvgError$ does not return appropriate values, $AvgErr_{Improved}$ should have been utilized for the computation of the expected average tree error. Nevertheless, $AvgError$ is implemented for the introduced implementation of *EnsembleForest*.

$TrustValue(x)$	Cases Correct	#Cases	$AvgError$	$AvgError_{Improved}$
0.9	9	10	0.18	0
0.9	10	10	0.26	0.01
0.5	50	100	0.5	0
1	0	10	1	1

Table 4.1: Examples of $AvgError$ and $AvgError_{Improved}$ computations. In these particular situations $SummedTrustValue(X) = |X| \cdot TrustValue(x)$ is assumed.

4.2.3 Prediction Changer

Apart from the base validators, level-1 generalizers are made up of another system. The additional system copes with a special problem: Imagine all base classifiers misclassify an element. So, none

is correct. In this situation, base validators should return a small trust value and another class should be predicted by the *EnsembleForest* instead. Which class to be predicted is the question. The additional system coping with this situation is named prediction changer. The prediction changer is ought to do two things: First, detect whether all base classifiers are wrong or not. Second, given all of the base classifiers are wrong, predict another class. This behavior shall be achieved with algorithms returning the same output as the base validators do. So the prediction changer shall output a class, a trust value measuring how likely the class is correct, and an expected average error of the prediction changer itself. For simplification, the expected average error of the prediction changer is defined to be zero always.

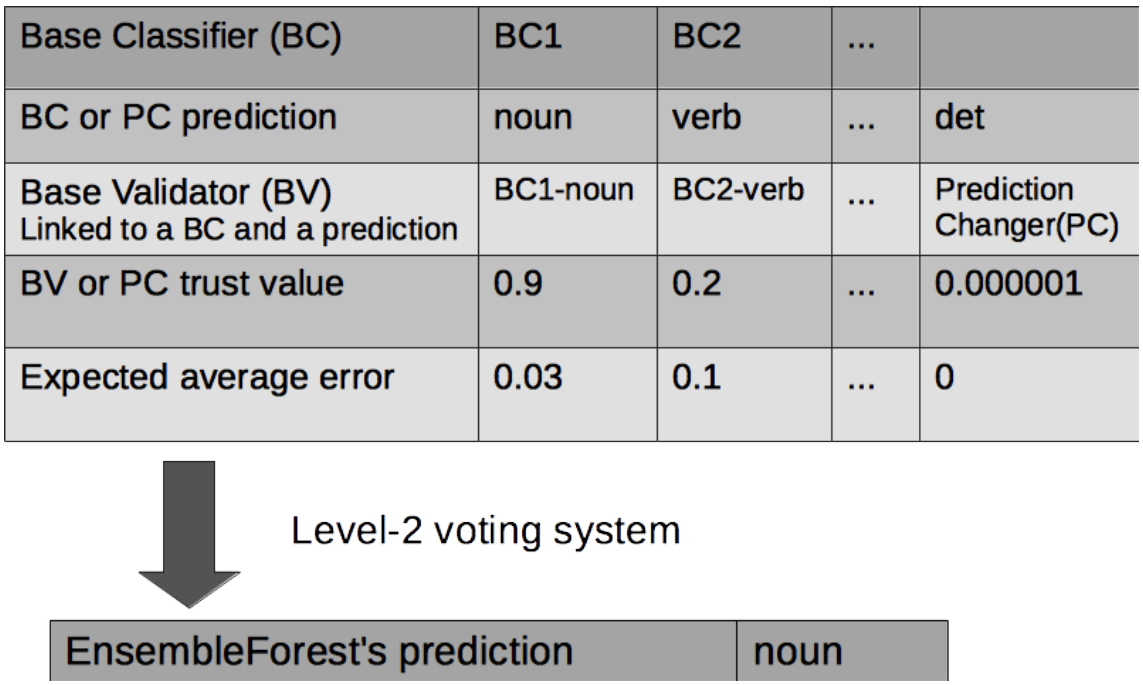

Figure 4.3: Example of results of some level-1 generalizers on a level-1 input data element. The prediction changer (PC) predicts a class non of the base classifiers (BC) predicted beforehand. Trust values are computed for each example. The expected average error of a base validator (BV) tree is computed once during the training of the BVs. The level-2 voting system combines the level-1 generalizer outputs (BVs and PC) to form a final prediction.

The prediction changer's desired behavior is achieved by the usage of two decision trees. The first decision tree is a binary decision tree as implemented before. This binary decision tree is ought to detect whether all base classifiers are wrong or not. The second decision tree is different. It is not a binary decision tree, because this one has to predict any class of the classification set, which should most likely be correct, given no base classifier predicted correctly. The tree is a

normal decision tree, constructed via the TDIDT method. At each node a feature is selected for this tree, which provides maximum information gain. The training data is split into subsets of joint feature values for this particular feature. To minimize the issue of overfitting, some decision tree algorithms prune the tree after construction. This implementation does not. Instead, a node will only branch into subsequent nodes, if a minimum size of elements from the training set per node is not undercut. To increase the trust values of the prediction changer the minimum size of elements per node is defined to be one nevertheless.

Yet, it has to be defined how trust values of the prediction changer are computed. They are calculated by multiplying the measurement value of the first and second tree. Why is this calculation appropriate? According to the Bayes theorem is $P(a \cap b) = P(a \mid b) \cdot P(b)$. In this context $P(a \mid b)$ equals the measurement value of the tree which predicts a new class, $P(b)$ refers to the measurement value of the other tree, and $P(a \cap b)$ is the probability or the trust value, for which the prediction changer system is correct.

4.3 Level-2: Combination of level-1 Generalizer Results

When all of the so far discussed generalizers (level-0 and level-1) computed their output, the resulting classification of the *EnsembleForest* for the given problem itself can be made on level-2. There, the following voting system (see figure 4.3 for an example) selects the class to be predicted using the output data of level-1 generalizers:

1. Choose a classification for which the trust value is highest, or rather, at which the predicted likelihood of its classification being correct is highest.
2. If there is a tie in the first point, choose the classification, which has the lowest expected error rate of all the classifications that were tied before.

Not only the class itself is part of the *EnsembleForest*'s output but as well the linked trust value. So for every prediction made by an *EnsembleForest* model one can see how sure the algorithm is, that its prediction is correct.

4.4 Positional Features

Whenever a classification decision for an element has to be made, a classifier somehow investigates the given features linked to that element. In addition, a classification algorithm may have access to positional features. Then, the algorithm can also investigate features of the elements from other positions. Whereby, positions are defined relative to the element's position in its group.

For the implemented version of the *EnsembleForest* algorithm, a parameter is defined such that, the algorithm can be granted access to certain positional features. Taking advantage of this on the POS Tagging problem, it is possible for an *EnsembleForest* model to read the results of the base classifiers and the meta information of a predecessor token or any other positional token in order to predict a correct tag. In addition, for this implementation, if the algorithm is granted access to positional features, every element is provided with another meta information feature. This additional feature contains the most often predicted class of the base classifiers for the corresponding element. The feature is intended to reduce the size of generated level-1 generalizer decision trees and is intended to reduce the potential overfitting of a level-1 generalizer.

4.5 Use case: Applying EnsembleForest on a single Base Classifier

Exploiting positional features, the *EnsembleForest* implementation can be applied on a single base classifier in order to increase the base classifier's performance. Normally, many base classifiers are needed to combine their outputs to a better final classification. If the parameters of the *EnsembleForest* implementation are selected such that the algorithm is granted access to many positional features, enough information could be available so that the level-1 generalizers of an *EnsembleForest* model are able to detect whether the single base classifier made a mistake and additionally suggest a better classification.

4.6 Comparability to AdaBoost

Comparing the *AdaBoost* algorithm with the *EnsembleForest* algorithm is not that easy. An *AdaBoost* model improves a single learner and performs an averaged voting on all created instances of that learner, while an *EnsembleForest* model takes any set of learners and aims on performing the best possible voting among these, or creating a new prediction if none of the learners seems to be correct. So, the *AdaBoost* algorithm and the *EnsembleForest* algorithm are no algorithms competing on the same task.
However, both methods could be combined. First, a set of diverse base learners could be boosted individually by an *AdaBoost* model. The resulting set of models could than be used as base classifiers for an *EnsembleForest* model. This hypothetical setup might yield in an improved performance higher than each individual performance, which can be reachable by the *AdaBoost* algorithm or the *EnsembleForest* algorithm. A second setup might be a good approach to empirically compare the performance of both methods. Therefore, take a single learner and apply the *AdaBoost* algorithm on it. Afterwards, take all models generated by this *AdaBoost* model as base classifiers for an *EnsembleForest* model.

Or third, a setup the other way round. Take a set of diverse learners as input for an *EnsembleForest* model. Than apply an *AdaBoost* algorithm on this *EnsembleForest* model. Like that, the *AdaBoost* model creates many instances of *EnsembleForest* models and combines them.

Chapter 5

Experimental Setup

For POS Tagging there are several well known reasonable good classification algorithms (called taggers). Depending on the use case some of them are scratching on an accuracy of 97.55% on english texts [14]. Although some taggers perform reasonably well, there is still an improvement potential. A better working algorithm could be constructed from scratch or by exploiting a combination of existing algorithms. In this work some well known reasonable good taggers are combined to yield a higher accuracy than the best performing individual tagger. For combination, among others, the proposed version of the *EnsembleForest* algorithm is applied. It is from great interest to test its performance and also to validate in detail if the algorithm works properly. Apart from that, the performance of *EnsembleForests* in contrast to comparable classifier combination methods is tested. The POS Tagging task is considered in this setup as well. Difficulties regarding the problem are examined and the effect of tagset mapping is evaluated. In conclusion, an experiment is executed on different part-of-speech data sets. Apart from the data, the experiment consists of a set of base classifiers performing POS Tagging and a set of classifier combination algorithms (called combiners).

5.1 Data

For the experimental setup, the German Text Archive (Deutsches Text Archiv (DTA)) [15] and the *Brown* corpus are used. Both corpora contain tagged texts in natural language. The *Brown* corpus is an English corpus with 500 samples from 1961 [16], summing up to 1,014,312 tokens, which are grouped together in form of sentences. The texts are splitted into 15 categories such as *press reportage and reviews*, *skills and hobbies*, *popular lore*, *miscellaneous* and *learned*. The original version was created by W. Nelson Francis and Henry Kučera at *Brown* University during 1963 and 1964. For tagging, the *Brown* corpus uses an own tagset. It includes in total 226 tags (cf. [3]). After an initial tagging round, incorrect tags are excluded by several proofreading and checking

procedures. Yet it is inevitable that error and inconsistencies remain [17]. Embodied version of the *Brown* corpus for this thesis is taken from the NLTK package [18]. An additional tagset is available for the *Brown* corpus. This is the *universal* tagset [5] provided by the NLTK package [18]. It is much smaller than the original *Brown* tagset. It consists of twelve tags, and is designed so that part-of-speeches tagsets of at least 22 languages can be modeled by it (cf. [5]). Practically, the NLTK package also includes a function to map the *Brown* tagset to the universal tagset. This mapping is used later in an empirical study on tagset mapping, which shows that a smaller tagset can increase but not decrease overall accuracy.

The second dataset comprehended is the DTA corpus. The DTA contains German texts from the 17th to the 20th century. The corpus covers 2,599 texts separated in the categories *fiction*, *science* and *usage literature*. In total 144,582,415 tokens are included. These are about 144 times as many as the *Brown* corpus contains. It took eight years from 2007 until 2015 to create the corpus. Tags are assigned utilizing the *Stuttgart/Tübinger Tagset (STTS)* tagset, which was created for German text annotation in 1999 (compare [19] for the STTS). It encompasses in total 54 tags. These are derived from a set of eleven higher ordered main word classes. Tagging in DTA was done by a classification algorithm. Later, humans corrected misannotated tags. This remains an ongoing process. In consequence, tags in the corpus version, which is used for this work, are considered to be more erroneous than tags in the *Brown* corpus. Additionally, a classifier generalizing from the DTA is expected to learn the contained mistakes. To appropriately meet the scope of this thesis, two small subsets of the DTA are considered. Both contain the texts of a certain time period. The first one is from the early 18th and the second one from the early 19th century. For the experiments, the fist 57,339 sentences from both these subsets are selected as datasets. This decision is made because the *Brown* dataset also contains exactly 57,339 sentences. Thus, all datasets are of the same size and are better comparable with each other.

Name	Time period	#Sentences	#Tokens	#UsedSentences
Brown	1961	57,339	1,014,312	57,339
DTA_18th	1700-1729	72,360	2,181,557	57,339
DTA_19th	1800-1829	150,391	3,822,116	57,339

Table 5.1: Datasets incorporated for this thesis.

5.2 Base Classifiers

In 1992 van Halteren combined many different versions of the same tagger algorithm as stated in [20]. Later, when a more diverse variety of algorithms existed, several different taggers were combined. For example, van Halteren et al. executed POS Tagging experiments using four different

classifiers, hoping to yield good results (compare [14] and [20]). Actually, they achieved accuracies up to 97.92% and 98.14% for the included combiners, while their best implemented taggers reached accuracies of up to 97.43% and 97.55%. To stay comparable to the scientific works mentioned, a similar set of classifiers is searched. Obviously, in the best case, the same algorithms would be used. Nevertheless, it is also important that the algorithms are available and easy to integrate. In conclusion finally six taggers were found (explained in sections 5.2.1 to 5.2.5).

For simplification of the classifier combination process, all of the taggers outputs are transformed to abstract level (compare section 3.1 for definition of abstract level). After transformation, they do return only the best single class label for each token if it exists, or *None* if nothing is predicted. Furthermore, only one model of each tagger is incorporated. However, it is feasible to include several models in order to maximize the overall performance of classifier combination methods.

5.2.1 Unigram

The first and most simple algorithm utilizes a unigram data model, therefore it is named *Unigram*. For every token (in the case of a *Unigram* algorithm they should rather be called type, cf. [2]) in the training set, the most frequent observed tag is predicted. During the classification process, *None* is predicted, if a token did not appear in the training set. Obviously, the *Unigram* algorithm should benefit from large training sets. In spite of its straightforwardness, high precision values are to be expected, because many words only belong to one single word class. Tokens with ambiguous word classes however, are difficult to classify.

5.2.2 Brill

The second algorithm, being part of the base classifier set, is the rule based transformation *Brill* tagger, which was invented by Eric Brill in 1994/1995 [21]. The tagger requires a base tagger itself, whose predicted tags are corrected with the help of certain selected transformation rules. These rules transform tags conditioned on certain positional tags and words. The rules are selected during training. First, a new annotated corpus is created using the initial tagger's predictions. Afterwards, transformation rules fitting predefined parameter settings are selected. Here, the NLTK implementation of the *Brill* tagger is used. The set of possible transformation rules is generated using the templates presented in the paper in which the *Brill* tagger was first published [21]. Parameters are defined so that the *Brill* tagger produces up to 2000 rules, each of which reduces the net number of errors in the corpus annotated from the initial tagger by at least 2, and each of which has an accuracy not lower than 0.99 on the training set.

Results should show that the probability of *Brill* being correct given the predictions of *Unigram* and *Brill* are different is close to 0.99.

5.2.3 Hidden Markov Model and Conditional Random Field

The next algorithms are a *Hidden Markov Model (HMM)* and a *Conditional Random Field (CRF)*, both implementations are from the NLTK package. The *CRF* implementation utilizes the Python *CRF*Suite [22], and the *HMM* implementation is based on a description of Huang, Acero and Hon [23]. Both methods represent the data they learn in a graph model. They take advantage of positional information. So, in the context of POS Tagging they require data grouped by sentences holding tokens ordered by position. *HMM* and *CRF* do not predict tags sequentially. Instead, the algorithms assign all tags for a complete sentence at once. They learn, by looking at a sentence and the corresponding possible tag sequence, to estimate a value for a sentence's combination. The tag sequence yielding the highest value is predicted. In greater detail: First, a *CRF* learns weights for a set of features. A feature is defined so that it returns a real valued number for a position in a sentence based on positional tag conditions. For example, a feature could output 1 if the tag from two positions behind is a determiner tag, the tag on the tokens position is a noun, and the token itself equals "dog", otherwise 0. During the training of *CRFs*, the set of features is generated and their importance weighted based on the training set. In this regard *HMM* equals a linear chain *CRF*. That is a *CRF*, which can only take restricted features into account. These are restricted by the positional tag they consider. So, for HMM and linear chain CRF models, only predecessor tags with a fixed maximum distance (for example, a distance of 1) can be taken into account. In other words, a *HMM* and a linear chain *CRF* are extensions to the Markov chain. For both the implementations of *HMM* and *CRF* the default parameter settings are used.

5.2.4 Perceptron

Last incorporated tagger algorithm of NLTK is the *Perceptron* tagger. The *Perceptron* tagger is given a template on how to represent a positional token in a sentence with a certain feature set. Each of theses features severs as a kind of predictor. A feature learns normalized weights or scores for each class of the classification set. At training, for every tuple $(\text{feature}, \text{word class})$, which is observed in the training set, scores or weights for the feature are adjusted. After training, the tagger sequentially predicts a word class for a token by grabbing its linked features, summing up the weights for all possible tags and returning the tag with the maximum final score. The implementation is orientated on Matthew Honnibals suggestion from 2013 which recorded up to 97.1% accuracy on 130,000 words from texts of the Wall Street Journal [24].

5.2.5 TreeTagger

In addition to all the NLTK implementations, another algorithm, the *TreeTagger* is included in the setup. The algorithm was developed by Helmut Schmid in 1994 [25]. Thereby, he achieved a percentage of 96.36% correctly assigned tags. In a later paper he even reaches up to 97.5% with a slightly modified version [26]. For this thesis, the implementation of *TreeTagger* provided by the University of Munich is used [27]. The algorithm is made up of automatic generated lexicons and a composition of algorithms.

The training corpus is transformed into several formats: first, into a lexicon with a priori tag probabilities; second, into a fullform lexicon containing words and their assigned tags; last, into a lexicon in form of a suffix tree, where the leafs contain probability vectors. The *TreeTagger* creates a single decision tree by recursive trigrams using a modified ID3 algorithm, which was introduced by Quinlan in [28]. Finally, predictions are made using a vitberi-like algorithm on the trigrams. In this regard a *TreeTagger* is similar to an *HMM*. There, with the help of the viterbi algorithm, the prediction is calculated based on bigrams. Hence, the *TreeTagger* exploits more complex dependencies then an *HMM*, it is expected to achieve better results.

5.3 Combiner algorithms

5.3.1 Majority Voting

The first and most simple classifier combination method is a self implemented majority vote system. If two tags are tied the one first observed is picked. The order in which base classifiers predictions are observed is always the same. Its overall accuracy is expected to be below the best performing tagger. Nevertheless, it is still of interest if the algorithm can reach the random baseline pick (compare table 6.3). The algorithm is called *MajVote*.

5.3.2 EnsembleForest

Of course, models of the *EnsembleForest* algorithm are included. They need some parameter settings to be accurately defined. In an best-parameter choosing experiment on a small subset covering 7733 sentences of the *Brown* corpus, several different parameter settings are tested. These are mostly base validator tree parameters; the minimum number of elements required per node, trees are grown full or use decision stumps instead, and the expected average error for every tree is computed or just set to be zero. The experiment shows merely similar performance results for the tested models. It would be out of the scope to show results in detail. Ultimately, due to

this best-parameter choosing experiment following parameter settings are chosen to generate all further *EnsembleForest* models. All base validators are fully generated with a minimum number of 10 elements per node and the expected average error for each base validator is computed. Besides those settings, the number of positional features is kept variable to study their effect in more detail. After all, a total number of four *EnsembleForest* models is examined in the following experiments. They differ in the distance of which positional features are allowed to be included. The distance ranges from zero to three. This means, a model with distance value of three (*EnsembleForestPF-3*) includes feature elements of tagger predictions, which are up to three positions before and after the elements corresponding position in a sentence. If the distance value is greater zero, the model will compute additional meta information for every element in a dataset as described in section 4.4.

5.3.3 J48 and AdaBoost on J48

The next classifier combination method is the *J48*, an implementation of the C4.5 decision tree from the Weka tool [29]. Originally, the C4.5 algorithm was introduced by Ross Quinlan in 1993 [30]. A decision tree classification algorithm is feasible to be repurposed as a classifier combination method. Therefore, predictions of base classifiers are selected as input features so that the decision tree creates a new prediction by extracting the information, which is provided. If the *J48* algorithm is applied in this way, it essentially differs from *EnsembleForest* models by the amount and purpose of the embedded trees. *J48* models exploit a single tree while *EnsembleForest* models take advantage of an ensemble of forests of trees. Accordingly, the *J48* algorithm is included in the experimental setup as an combiner. The implementation is run with default parameter settings. In 2006 Quinlan stated that the popular *AdaBoost* algorithm can be applied on a C4.5 algorithm to yield a higher performance. He evidenced this assumption on several classification problem domains [31]. On average, his boosted C4.5 version got 0.847 as many misclassifications as the regular C4.5 algorithm. In total, misclassification ratio was between 0.389 and 1.361. This is a motivation to include the Weka implementation of *AdaBoost* applied on the *J48* algorithm as well.

5.3.4 Hoeffding Tree

Another of Weka's incorporated combiner implementations is the Hoeffding Tree (originally named *VFDT*). A decision tree algorithm developed by Pedro Domingos and Geoff Hulten in 2000 [32]. It is described as an "incremental, anytime decision tree induction algorithm that is capable of learning from massive data streams, assuming that the distribution generating examples does not change over time. Hoeffding trees exploit the fact that a small sample can often be enough to

choose an optimal splitting attribute. This idea is supported mathematically by the Hoeffding bound, which quantifies the number of observations [...] needed to estimate [...] the goodness of an attribute" [33].

5.3.5 Random Forest

The so far introduced tree based combiners are single tree algorithms. An *EnsembleForest* though, makes use of multiple trees. For a good empirical comparison of these methods, another combination method based on multiple decision trees seems profitable. Hence, Wekas implementation of a random forest classifier is included. The implementation is based on Leo Breimans definition of a random forrest classifier. He defines the classifier as a method generating a predefined number of decision trees, which are trained on distinct random subsets of the training dataset (cf. [34]). The random forest implementation and all the other tree combiners in the setup are tested on their accuracy and time complexity. Weka's random forest implementation was thereby tested twice: with 100 and with 1 tree generated for its forest. Both models reached nearly the same performance values. The increased number of random generated trees did not positively affect the accuracy. In fact, the models caused an accuracy of about one percentage point lower than the other combination methods. Because of that, the random forest implementation is not used in further experiments.

5.3.6 Tag-Pair-Similar

The last included classifier combination algorithm is a self implemented similar version of the *TagPair* algorithm (called *TagPairSimilar*). It works similar as the original version, for which H. v. Halteren and others executed an experimental setup just like here [20]. There, they describe the algorithm, which they call *TagPair*. It outperforms every other classifier combination method they use. On the LOB corpus [35] they achieve up to 97.92% accuracy. *TagPairSimilar* is different from *TagPair* as described by H. v. Halteren et al. [20]. The implementation equally computes all solution tag scores for a quadruple of two taggers and their both predictions (such quadruples are named tagpair). Then, to classify an element, first, all tagpairs fitting that element are selected. In the classification step the *TagPairSimilar* algorithm searches for a tagpair having the highest score value from all tagpairs, and predicts the best tag of that single tagpair. In contrast, the original *TagPair* sums up all scores of all selected tagpairs and outputs the tag with maximum resulting score.

Chapter 6

Results

6.1 General Tagger and Combiner Results

The first set of experiments are executed on a small subset of the *Brown* corpus containing all text samples categorized as *learned*. In total this subset includes 7,733 sentences resulting in 181,830 tokens. The set was randomly split by 80% and 20% into training and test set. Different combinations of the six base classifiers are selected (41 out of 63 possible). Among these are all possible combinations of two, five and six taggers and some combinations of three and four taggers. The combiners *MajVote* and *EnsembleForest* are trained on each of these selections. *MajVote*, as it was already implemented and *EnsembleForest* to find a probably best fitting selection of base classifiers. Both, the base classifiers and the combiners are trained on the same training set. The table 6.1 shows the overall accuracy values, which each classifier reached under these conditions. The top row shows the accuracies reached by the base classifiers. The following rows show the performance of the combiners given a certain selection. To keep the table short, not all of the tested tagger combinations are included. For every number of selected base classifiers, the combination yielding the highest *EnsembleForest* accuracy is displayed. When sorting base classifiers by accuracy, the following is observed: *Treetagger*, reaching 94.92%, performs best, followed by *CRF* with 94.01%, *Perceptron* with 93.76%, *Brill* with 90.88%, *HMM* with 90.73% and finally *Unigram* with 88.34%. As expected, *CRF* and *Perceptron* show better results than *HMM*, and *Brill* shows better results than *Unigram*. The values are probably below the benchmark values of the taggers referenced performances, because the training set is very small. The accuracy of *MajVote* is always in between the selected tagger's accuracies. Its performance never breaks out of that restriction in the results. In contrast, *EnsembleForest* sometimes outperforms all selected base classifiers. The table shows that it performs best on the selection *Unigram*, *Brill*, *HMM*, *Perceptron* and TreeTagger. Therefore, and because of the time complexity of *CRF*, the *CRF* implementation is not longer used for further experiments. Each of these different training and testing settings were though only performed

once. There may be significant variances that are not taken into account. So results only show a tendency that using as many base classifiers as possible is beneficial.

Unigram	Brill	CRF	HMM	Perceptron	TreeTagger	MajVote	EF
88.34%	90.88%	94.01%	90.73%	93.76%	94.92%	-	-
x	x	-	-	-	-	89.17%	90.90%
x	-	-	x	-	-	88.77%	92.08%
-	x	-	x	-	-	90.32%	92.32%
x	-	x	-	-	-	89.24%	93.25%
-	-	x	x	-	-	92.26%	94.06%
x	-	-	-	x	-	89.32%	94.27%
-	-	-	x	x	-	92.36%	94.39%
-	x	x	-	-	-	90.83%	94.48%
-	-	x	-	x	-	94.22%	94.59%
-	x	-	-	x	-	90.98%	94.63%
-	-	-	x	-	x	92.90%	94.83%
x	-	-	-	-	x	89.97%	94.94%
-	-	x	-	-	x	94.77%	95.01%
-	-	-	-	x	x	94.35%	95.08%
x	-	-	-	x	x	94.60%	95.19%
x	x	-	-	x	x	91.51%	**95.50**%
x	x	-	x	x	x	92.81%	95.45%
x	x	x	x	x	x	**94.84%**	95.40%

Table 6.1: Effect of tagger selection on the combiner methods overall accuracy. The taggers and combiners are trained on the *Brown learned* data. Each tagger's accuracy is written in the first row. (X) - the tagger is selected, (-) - the tagger is not selected.

In many machine learning experiments, algorithms are trained with 10-fold-crossvalidation. Thereby, the training set is randomly divided into ten equally sized partitions and 10 training iterations are performed. In one iteration nine of these partitions are selected, which were not selected before in that composition. Training is performed on these partitions and the resulting model tested on the remaining partition. Finally, the model returning the best accuracy is retained. Crossvalidation is generally said to be a good training method to improve accuracy in comparison to regular training. However, it can be time consuming, since training has to be performed ten times. Computation time is limited. So, it is evaluated, if 10-fold-crossvalidation is needed for further experiments. Therefore, the previous setup using all six taggers is picked and executed again with 10-fold-crossvalidation for training of the base classifiers and combiners. Table 6.2 shows the results of the runs explained above. The first two columns show the actual accuracies of

the classifiers, the third column gives the absolute difference between a classifier trained with and without 10-fold-crossvalidation. It can be seen, that in most cases the regularly trained taggers perform even better than their counterparts. The maximum absolute difference is 0.35% (less then one percentage point) for the *Brill* tagger. As there is no significant difference in these values, 10-fold-crossvalidation is not further used.

Classifier	Regular	CrossVal	AbsDiff
Unigram	**88.3423**	88.0114	0.3309
Brill	**90.8773**	90.5272	0.3500
CRF	**94.0057**	93.8334	0.1723
HMM	**90.7323**	90.4179	0.3145
Perceptron	93.7568	**93.8553**	0.0984
TreeTagger	**94.9218**	94.7386	0.1832
MajVote	**94.8370**	94.6511	0.1860
EnsembleForest	95.4031	**95.5590**	0.1559
Average	**92.8595**	92.6992	0.2239

Table 6.2: The effect of 10-fold crossvalidation on the classifier's overall accuracy in %. The taggers and combiners are trained on the *Brown learned* dataset.

Now, the setup for main experiments is determined. In summary, an experiment is split into three processing parts. First, training of the base classifiers, second the training of the combiners and third the testing of the trained models. The training is performed regularly without crossvalidation. Unless otherwise stated, the data is split into a training set of 80% and a test set of 20% of the data, and combiners are trained on the same set as the base classifiers are. The selected base classifiers are *Unigram*, *Brill*, *HMM*, *Perceptron* and *TreeTagger*. As the general aim is to evaluate the classifier combination methods, some baselines giving a measure on their usability, are defined. The lowest baseline shall be the minimum performance tagger. If an combiner performs below that baseline, improving is definitely senseless; since, even if one would have the ability to use only the worst classifier, he would get better results. The second baseline is a random pic of all base classifiers (randPicBaseline). If a combiner is below that performance value, even a random picking combiner would result in a better performance. The second baseline is always at least as high as the first baseline. Table 6.3 displays the baselines for all the three corpora datasets. Baselines for the *DTA_19th* dataset are slightly higher than for the other datasets. Interestingly, the *DTA_18th* dataset seems to be more difficult to tag for POS taggers than the *DTA_19th* dataset, although it contains the same language.

Baseline	Brown	DTA_18th	DTA_19th
minPerformTagger	89.53%	89.08%	90.27%
randPicBaseline	93.22%	92.43%	93.87%

Table 6.3: Classifier combination method baselines

The baselines for the classifier combination on POS Tagging depend on the selected classifiers to be combined and on the selected dataset. Because of that, the performance of combiner systems can not be compared directly if the base classifiers and the dataset are not comparable as well. To cope with that issue, it is of interest to state the base tagger agreement on tokens tags. Therefore, the same patterns of agreement other authors already used (cf. [20] and [14]) are computed. The table 6.4 shows the inter tagger agreement probabilities of the base taggers on the test dataset. It is assumed that cases, matching higher sorted row patterns, are more easily combinable than others. In [14], the authors say that the agreement patterns enable to determine levels of combination quality. Experiments show that the cumulative probability of an agreement pattern level correlates with the overall accuracies achieved by combiners. In the setups of [14] and [20], the combiners and the best base classifier, which are performing POS Tagging on different datasets than the datasets used in this work, always settling between the cumulative pattern probabilities of *A majority is correct* and *Correct tag is present but is tied*. Whereby, in [20], the tagger agreement is computed on the training set, and in [14] it is computed on the test set. In both works an even amount of taggers is used. This cumulative inter tagger agreement pattern level (from now on called benchmark level) hopefully will be achieved by the combiners of this thesis as well.

Pattern	Brown		DTA_18th		DTA_19th	
	%	%Cum	%	%Cum	%	%Cum
All taggers agree and are correct	85.53	85.53	85.51	85.51	87.17	87.17
A majority is correct	8.82	**94.36**	7.68	**93.19**	7.64	**94.80**
Correct tag is present but is tied	1.35	**95.70**	1.93	**95.12**	1.79	**96.61**
A minority is correct	2.54	98.24	2.88	98.00	2.13	98.73
All taggers are wrong & don't agree	0.70	98.95	1.09	99.09	0.73	99.47
All taggers agree & are wrong	1.05	100.00	0.91	100.00	0.53	100.00

Table 6.4: Tagger agreement patterns for the three datasets. Percentages of observed cases for each pattern are listed (%). In addition, the cumulative percentage is displayed (%Cum). The interval of the benchmark level is highlighted for every dataset.

Now, the base tagger and the combiner accuracies are evaluated. It is measured, how likely a classifier is correct, given a certain base tagger agreement pattern is fitted. In addition, the overall accuracies of classifiers are measured on the whole dataset. Results of that evaluation for the experiments on the *Brown* corpus are displayed in the table 6.5. For clarity reasons, the

corresponding result tables for the *DTA_18th* and the *DTA_19th* corpora datasets are moved to the appendix (see tables A.1 and A.2). On the *Brown* corpus one can see, the *TreeTagger* is the best overall performing tagger with 95.5%. The *TreeTagger* just hits the upper border of the benchmark level. It can be seen that the expected good taggers (*TreeTagger* and *Perception*) perform best compared to the other taggers for the patterns *Correct tag is present but is tied* and *A minority is correct*. Yet, the weaker taggers predict correctly for these situations in a range of 2.08% to 20.41%. As to be expected, the *MajVote* algorithm does always predict correctly, if a majority of the taggers is correct. Interesting is, if the correct tag is tied with another one, the *MajVote* model only reaches 28.53%. That is, because in the worst case, all taggers predict a distinct word class. In that case, the chance for the *MajVote* algorithm being correct, by a random guess on the tied word classes, is only 20%. Out of all combiners the *TagPairSimilar* combiner performs worst. The algorithm's performance of 91.36% is between 89.53% for the first and 98.22% for the second baseline. Also, the model does not reach the benchmark level. All other combiners yield higher performance values than the baselines, and at least they reach the benchmark level. All models of the *EnsembleForest* algorithm outperform the other combiners and taggers, and even reach the subsequent higher level than benchmark. They yield an overall accuracy of between 96.35% and 96.36%. For agreement patterns, of which the correct tag is at least tied, *EnsembleForest*'s models outperform their direct counterparties: the *J48*, the *AdaBoostOnJ48* and the *HoeffdingTree* model. All the counterparty tree methods have problems to be correct, if all base classifiers are predicting correctly. In this situation, they achieve 99.12% correctness, while the *EnsembleForest* models reach exactly 100% correctness. If a minority of taggers is correct, the counterparty combiners and *EnsembleForest* models still predict correctly in more then 50% of the cases. For the nearly most difficult situation, at which the correct tag is not present and the taggers do not agree on a single tag, these models still are sometimes able to find the correct tag. Counterparty combiners find the solution with a probability of 0.25% to 0.36%, and *EnsembleForest* models are able to find solutions with a probability of about 0.08%. These were between 8 an 9 cases out of 11430 total cases, dependent from the number of positional features allowed for the *EnsembleForest* models. In all cases, the *AdaBoostOnJ48* setup does not increase or decrease the performance of the *J48* models, which are not boosted by the *AdaBoost* algorithm.

	All correct	*Majority correct*	*Correct is tied*	*Minority is correct*	*All wrong & don't agree*	*All wrong & agree*	*P(classifier correct)*
Overall Pattern	0.8553	0.0882	0.0135	0.0254	0.007	0.0105	-
Unigram	**1.0**	0.4273	0.0208	0.0792	0.0	0.0	0.8953
Brill	**1.0**	0.7872	0.0599	0.1439	0.0	0.0	0.9293
Hmm	**1.0**	0.7616	0.1312	0.2041	0.0	0.0	0.9295
Perceptron	**1.0**	0.7894	0.9162	0.5032	0.0	0.0	0.9501
Treetagger	**1.0**	0.8917	0.8713	0.4446	0.0	0.0	0.957
MajVote	**1.0**	**1.0**	0.2853	0.0	0.0	0.0	0.9474
TagPairSimilar	**1.0**	0.4771	0.849	0.1829	0.006	0.0	0.9136
J48	0.9912	0.9204	0.9078	**0.5658**	0.0025	0.0	0.9557
AdaBoostOnJ48	0.9912	0.9204	0.9078	**0.5658**	0.0025	0.0	0.9557
HoeffdingTree	0.9912	0.8922	0.8725	0.5477	**0.0036**	0.0	0.9522
ensembleForest	**1.0**	0.9286	**0.9289**	0.5401	0.0008	0.0	0.9635
EnsembleForestPF-1	**1.0**	0.9284	0.9284	0.5439	0.0007	0.0	**0.9636**
EnsembleForestPF-2	**1.0**	0.929	0.9288	0.5397	0.0008	0.0	0.9635
EnsembleForestPF-3	**1.0**	0.928	0.9268	0.5449	0.0008	0.0	0.9635

Table 6.5: Classifier performances on the base classifier agreement patterns in range $[0, 1]$. All results are averaged over seven experiment runs on the *Brown* dataset.

Results on the DTA corpora show similar tendency as on the *Brown* corpus. The *TagPairSimilar* model scratches on the randPicBasline with 92.35% on the *DTA_18th* dataset and 93.71% on the *DTA_19th* dataset. So, the *TagPairSimilar* algorithm performs slightly better on DTA data than on *Brown* data. Surprisingly, the counterparty combiners decrease drastically in performance. On the *DTA_18th* dataset they achieve accuracies in range of $[90.64\%, 91.18\%]$ (see tables A.1). On the *DTA_19th* dataset they all reach about overall 88% accuracy, which is even less than the worst performing tagger. Their most significant drop in prediction ability is for cases, in which all taggers predict correctly. These cases make up 85.51% of all *DTA_18th* and 87.17% of all *DTA_19th* cases. The counterparty taggers *J48*, *AdaBoostOnJ48* and *HoeffdingTree* tag 94.99% of the *DTA_18th* and 89.99% of the *DTA_19th* of these cases correctly. In contradistinction, *EnsembleForest* models still yield 100% of these cases correctly and reach about 95.5% and 96.94% overall accuracy for the *DTA_18th* and the *DTA_19th*.

In summary, independent from the data set *EnsembleForest* models yield the subsequent better agreement level than benchmark level.

In section 5.2 it was stated that the *Brill* tagger is expected to achieve a probability close to 99% for $P($*Brill* is correct given the predictions of *Unigram* and *Brill* are different$)$ on the training set. This probability is quickly investigated. The table 6.6 shows that the *Brill* tagger is in about 91.2% to 93.7% of the cases correct, given the *Brill* algorithm and the *Unigram* algorithm predicted something different from each other. This proofs that the *Brill* algorithm is able to reduce errors made by a *Unigram* algorithm.

Corpus	P(*Brill* correct | *Brill* != *Unigram*)
Brown	93.65%
DTA_18th	93.37%
DTA_19th	91.24%

Table 6.6: Probability of the *Brill* tagger being correct given that the *Unigram* and the *Brill* predictions are different from each other.

How strong does the overall accuracy depend on the training size? Therefore, experiments are executed on subsets containing 500, 1000, 2000, 5000, 7733, 20,000 and 57,339 (all) sentences of the *Brown* corpus. On each of these subset sizes the experiment is executed seven times, including retraining of combiners and taggers every time. Every time first, the current subset is split into 80% training data and 20% testing data. Then the taggers and combiners are trained, and finally, the trained models are tested on the test set. Averaged overall accuracies and variances are displayed in the figure 6.1.

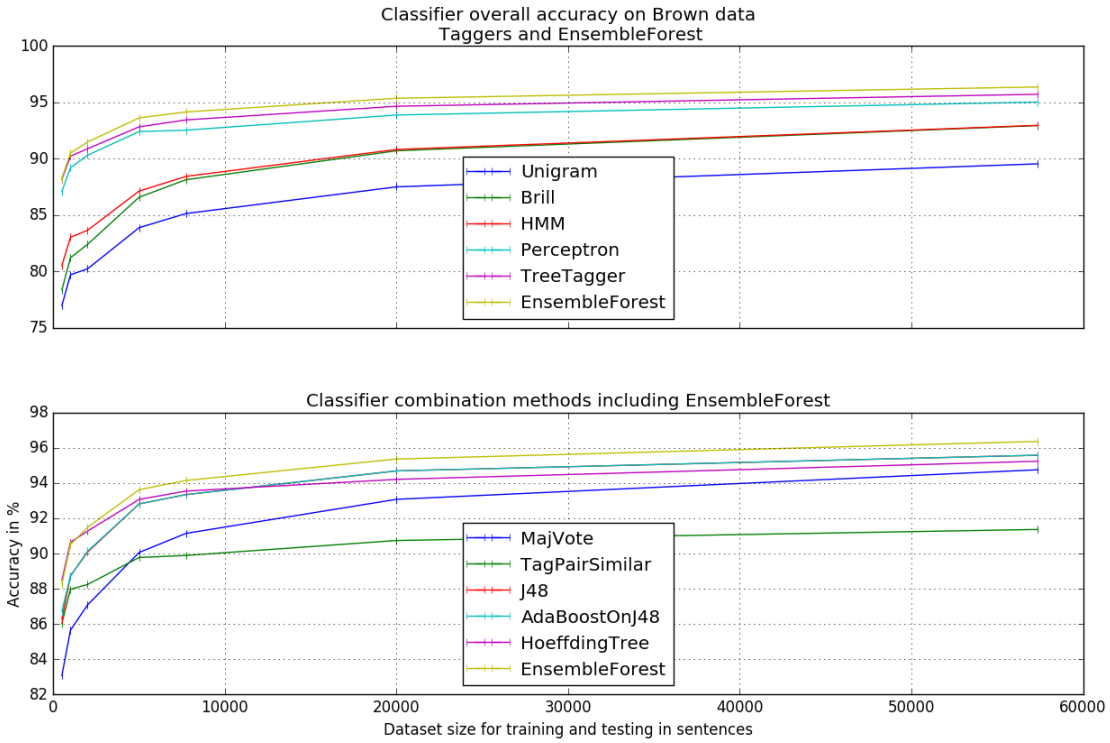

Figure 6.1: The classifier's overall accuracies and variances on the *Brown* data. All results are averaged over seven experiment runs. The graph of the *EnsembleForest* algorithm visualizes the same data in both plots.

The graphs in the figure 6.1 show a common course. For small training sets accuracies increase rapidly while increasement stagnates for bigger datasets. However, the gradient stays positive for all graphs. This intends that an even more enlarged training set than the *Brown* corpus could further increase performance. Independent from training size, the *EnsembleForest* algorithm exceeds all other taggers, and for training set sizes of at least 2000 sentences also all combiners. For small datasets the *HoeffdingTree* implementation performs better than the *J48* and the *AdaBoostOnJ48* models, but for datasets about 10,000 sentences, *J48* models and *AdaBoostOnJ48* models outperform *HoeffdingTree* models. The visualization in the figure 6.1 is displayed in concrete numbers in the table A.3. There, variance values become clear. Variances are below one percentage point for the smallest dataset, and drop as the dataset size raises. On the complete *Brown* dataset (57,339 sentences) variances do not exceed 0.007 percentage points.

In order to evaluate overfitting of the taggers and combiners the main experiment is repeated once more on the *Brown* dataset, the *DTA_18th* dataset and the *DTA_19th* dataset. This time, the testing is not done on the 20% testing subset but on the same subset the classifiers are trained on. The table 6.7 shows the results of this overfitting experiment. Columns named *Train* hold accuracies on the

training data set, and columns named *Test* hold accuracies on the testing data set. Performances on the training data set are about 2%-5% points higher than on the testing dataset. Higher values for the training set indicate overfitting, which means that a classifier learned some cases *by heart* rather than generalizing the underlying system. Surprisingly, *EnsembleForest* models with access to positional features mostly do not have increased overfitting than *EnsembleForest* models without access to positional features. This is surprising, because it is said that providing a classification algorithm with additional features, does likely result in an increased overfitting. On the training sets, the *EnsembleForest* models classify up to 99.24% of all the data correct. This percentage seems tremendous, but it has to be compared to the pattern agreement level and the baselines.

Classifiers	Brown		DTA_18th		DTA_19th	
-	Train	Test	Train	Test	Train	Test
Unigram	0.9491	0.8953	0.9411	0.8908	0.9491	0.9027
Brill	0.9745	0.9293	0.9626	0.9082	0.9745	0.9242
HMM	0.9640	0.9295	0.9569	0.9266	0.9640	0.9392
Perceptron	0.9858	0.9501	0.9772	0.9467	0.9858	0.9611
TreeTagger	0.9823	0.9570	0.9729	0.9494	0.9823	0.9665
MajVote	0.9745	0.9474	0.9763	0.9366	0.9847	0.9543
TagPairSimilar	0.9250	0.9136	0.9411	0.9235	0.9491	0.9371
J48	0.9802	0.9557	0.9435	0.9118	0.9047	0.8815
AdaBoostonJ48	0.9802	0.9557	0.9435	0.9118	0.9047	0.8815
HoeffdingTree	0.9762	0.9522	0.9390	0.9064	0.9010	0.8756
EnsembleForest	0.9879	0.9635	**0.9873**	0.9552	0.9923	**0.9695**
EnsembleForestPF-1	**0.9924**	**0.9636**	**0.9873**	0.9552	**0.9924**	0.9694
EnsembleForestPF-2	0.9923	0.9635	**0.9873**	0.9551	0.9923	0.9694
EnsembleForestPF-3	0.9923	0.9635	0.9872	**0.9553**	0.9923	0.9694

Table 6.7: Overfitting evaluation of classifiers on all three datasets. Columns state the probability that elements of the training set (Train) or the testing set (Test) are classified correct. The test values computed on the *Brown* corpus data, are averaged over seven experiment runs. The test values on DTA subsets are averaged over four experiment runs. The *Train* values are all computed on one single experiment run.

The inter tagger agreement raises, when computed on the training set compared to the testing set. Hence, the base tagger agreement level changes as well. In the table 6.8 the agreement percentages can be seen. By them, it becomes clear that only the *Perceptron* tagger and *EnsembleForest* models reach the benchmark agreement level $[0.9841, 0.9854]$ on the training set. *EnsembleForest* models do reach, as on the testing set, the subsequent level of benchmark, which is bordered by the interval $[0.9854, 0.9977]$.

Pattern	%	%Cum
All taggers agree and are correct	91.81	91.81
A majority is correct	6.61	**98.41**
Correct tag is present but is tied	0.12	**98.54**
A minority is correct	1.23	99.77
All taggers are wrong & don't agree	0.03	99.80
All taggers agree & are wrong	0.20	100.00
RandPicBaseline	97.11	-

Table 6.8: The base tagger agreement in % on the training set extracted from the Brown corpus. The interval of the benchmark level is highlighted.

6.2 POS Results

It is good to know the overall accuracy of a classifier. For POS Tagging it is also of interest to know the performance of classifiers on a specific word class. As this question is not in the main focus of this thesis it shall be answered very roughly. Word class performances of the *EnsembleForest* models and the taggers on the *Brown* corpus are evaluated. For 15% of all tags the F1 score of *EnsembleForest* is perfect. Whenever these tags had to be predicted they actually were predicted. In the table 6.9 three word classes are picked as representatives. For every base classifier and for the *EnsembleForest* the precision, recall and F1 score on these word classes is computed. Is is visible, for determiners (*DT*) precision, recall and F1 score of all classifiers are mostly similar. Here, the *EnsembleForest* algorithm is able to maximize the recall. For this tag, the *TreeTagger* implementation has a better overall accuracy than the *Perceptron* tagger and *EnsembleForest* models seem to trust the *Perceptron* tagger for predictions of DT more likely. For the word class *DOZ-HL*, taggers perform very different. The *Unigram* algorithm does not predict *DOZ-HL* at all while the *Brill* algorithm learned to transform some of *Unigram*'s predictions to *DOZ-HL*, resulting in an F1 score as good as the one of the *Perceptron* algorithm. Here, the *TreeTagger* models do perform surprisingly bad. *HMM*s estimate the DOZ-HL tag in way to few cases, which causes a perfect precision but very weak recall and F1 score. In this particular case, *EnsembleForest* models have troubles to combine the taggers perfectly. The highest tagger's recall is achieved, but the precision value is only on intermediate level, which may be caused of to high trust values for this tag. Last example picked is the tag *VBG+TO*. Here, all taggers have perfect precision but, except of *TreeTagger* models, low recall. The taggers are predicting the *VBG+TO* tag to rarely. This time, the *EnsembleForest* algorithm could easily generalize how to combine the taggers, resulting in a perfect F1 score of 1. Probably, the algorithm did return a trust value of 1 for *VBG+TO* tag predictions always.

Classifier		DT	DOZ-HL	VBG+TO
Unigram	Pr	**0.9868**	None	**1**
	Re	0.7935	0	0.75
	F1	0.8797	None	0.8571
Brill	Pr	0.986	0.8	**1**
	Re	0.9093	**0.8**	0.75
	F1	0.9461	**0.8**	0.8571
HMM	Pr	0.9663	**1**	**1**
	Re	0.9133	0.2	0.25
	F1	0.939	0.3333	0.4
Perceptron	Pr	0.9652	0.8	**1**
	Re	0.9631	**0.8**	0.75
	F1	0.9641	**0.8**	0.8571
TreeTagger	Pr	0.9595	0.5	**1**
	Re	0.9278	0.6	**1**
	F1	0.9434	0.5455	**1**
EnsembleForest	Pr	0.9642	0.6667	**1**
	Re	**0.9636**	**0.8**	**1**
	F1	**0.9639**	0.7273	**1**

Table 6.9: Tagger and *EnsembleForest* word class performances. For each classifier its precision (Pr), recall (Re) and F1 score (F1) on a certain tag (column) is displayed.

Besides the word class performance analysis, another evaluation on performances for the changed tagset is done. The table 6.10 lists the tags of the changed tagset, sorted ascending by the F1 scores of the *EnsembleForest* algorithm. The corresponding tags are marked either closed, if the amount of tokens fitting that tag are countable finite and is fixed, or open otherwise. The tag *num* is defined as open because there are at least countable infinite numbers. The table indicates that closed word classes are easier to assign correctly than open word classes.

Tag	x	adj	adv	prt	num	noun	verb	adp	pron	conj	det	.
Open or Closed	o	o	o	o	o	o	o	c	c	c	c	c

Table 6.10: Tags of the universal tagset sorted ascending by the F1 scores an *EnsembleForest* model reached on the *Brown* dataset. Tags are categorized either as closed word class (c) or open word class (o)

So far, the POS analysis is restricted to word class evaluation. Now, a performance analyzation of taggers and *EnsembleForest* models on sentence length is added. In the figure 6.3 the probability a classifier predicts every token of a sentence correctly, is visualized depending on sentence length.

The data is sampled into ten buckets. Each bucket holds about 1000 sentences of similar length. The figure shows that longer sentences are less likely to be tagged completely correct than shorter sentences. This is expected, because long sentences obviously contain more tokens, which can be misclassified, than short sentences. Nevertheless, it can be seen that the shortest sentences (up to six tokens long) are harder to tag than slightly longer sentences. The *EnsembleForest* algorithm exceeds all taggers performances, assigning about 28% - 81% of sentences correct depending on the sentence length. An ordering of taggers by accuracy stays nearly the same for every sentence length, except one exclusion. For sentences equally long to or shorter than nine tokens the *TreeTagger* tagger outperforms the *Perceptron* tagger. In all other cases the *Perceptron* tagger tags more sentences completely correct than the *TreeTagger* tagger.

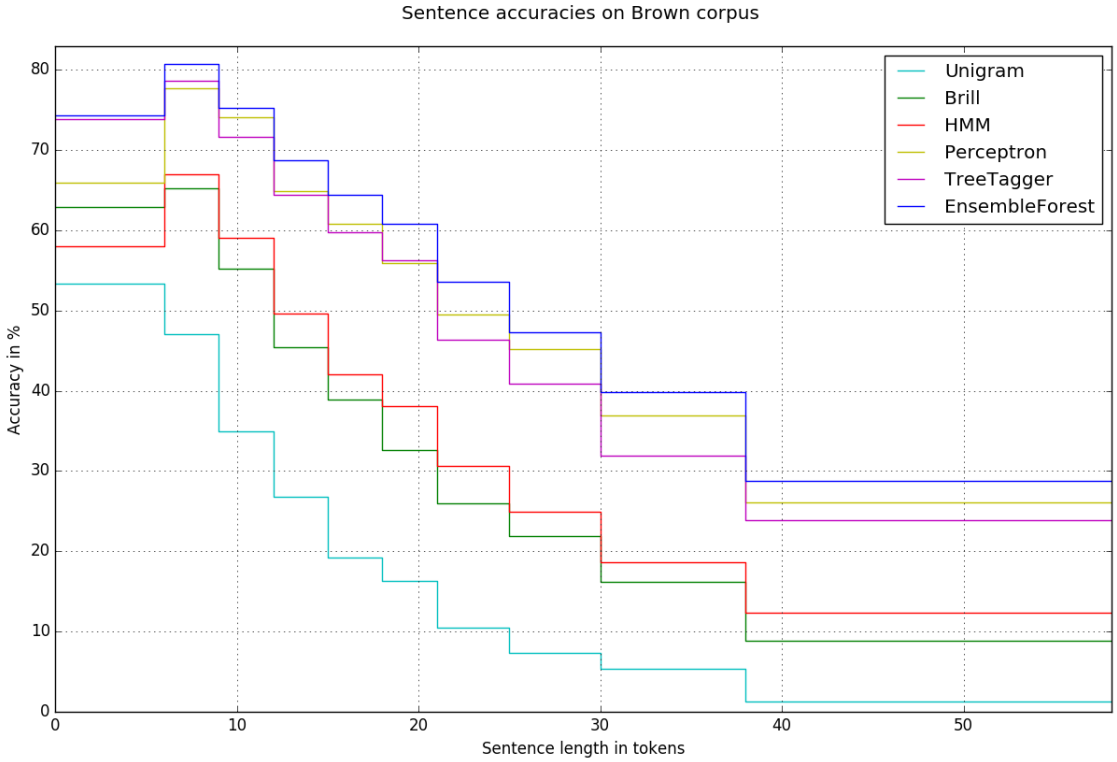

Figure 6.2: Accuracies that in the *Brown* corpus a sentence is tagged completely correct, dependent from the sentence length. Results are computed using one experiment run. Data is sampled into 10 parts, so that every part consists of an equally sized amount of data points.

The next figure 6.3 visualizes the average error per token of taggers and the *EnsembleForest* combiner on the *Brown* corpus. Here, it can be seen, that the probability a token is misclassified is mostly independent from the sentence length, with about 3% errors for the best classifier (*EnsembleForest*). Only very short (up to six tokens long) and very long sentences (having at least 38 tokens) fall apart. Very short sentences are more harder to classify, on average about 5% percentage points harder. Long sentences are easier to tag, on average about three times as easy as sentences of a intermediate length.

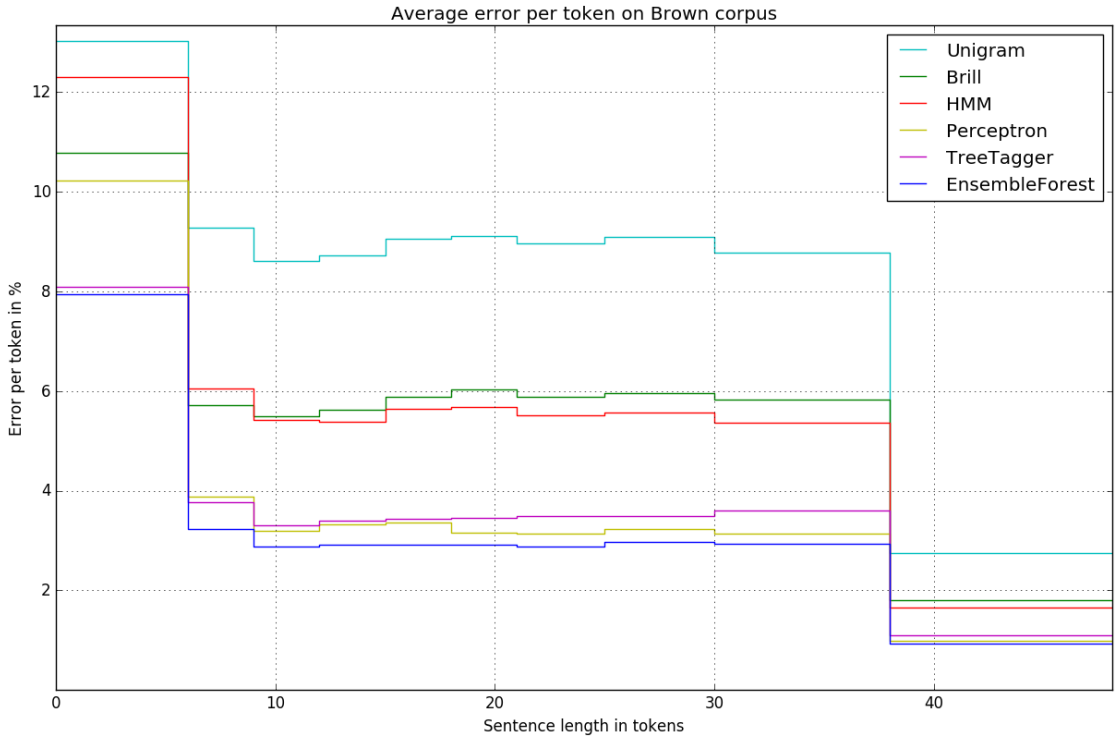

Figure 6.3: Average error per token on the *Brown* corpus, dependent from the sentence length. All results are computed using one experiment run. The data is sampled into 10 parts, so that every part consists of an equally sized amount of data points.

Still missing results are the probabilities a classifier predicts all tokens of a sentence correct and the effect of tagset mapping. Therefore, results of experiments on the *Brown* corpus are analyzed. The table 6.11 contains the accuracies of most classifiers. For both, the sentence accuracy and the token accuracy the effect of tagset mapping is shown in columns named *changedTagset*. The

results in columns *changedTagset* are computed by mapping results of classifiers with the *Brown* tagset to the universal tagset after classification is performed with the initial tagset. As evidenced beforehand in chapter **??**, it is visible that mapping the original tagset to a smaller one yields in an increased overall accuracy. For example, *Treetagger*'s performance on tokens is increased from 95.67% to 97.94%, which is equal to an error reduction of 45.50%. Comparing the best performing combiner with the best performing tagger, the table allows to calculate an error reduction of 14.32% for *EnsembleForest* compared to TreeTagger, both on normal tagset. Most tremendous error reduction is gained by comparing the performance for tokens of *EnsembleForest* on the changed tagset with the performance for tokens of *Treetagger* on the normal tagset. There an error reduction of 60.74% is achieved. Overall probability that no mistakes are done, while classifying a sentence, increases a lot when the tagset is mapped. While the best tagger beforehand predicted 53.59% of all sentences correct, afterwards percentage is raised to 70.49%. There again, *EnsembleForest* outperforms all other classifiers. Without the tagset mapping the algorithm tags 57.98% of all sentences are correctly. The mapping of the tagset to the universal tagset increases accuracy of the *EnsembleForest* algorithm to 74.04%.

Brown corpus	Sentence Accuracy		Token Accuracy	
Classifier	normalTagset	changedTagset	normalTagset	changedTagset
Unigram	20.99%	33.54%	89.57%	93.12%
Brill	35.46%	47.00%	92.95%	95.29%
HMM	35.61%	47.56%	92.90%	95.50%
Perceptron	43.51%	67.50%	94.89%	97.76%
Treetagger	**53.59**%	**70.49**%	**95.67**%	**97.94**%
MajVote	45.23%	57.66%	94.69%	96.72%
TagPairSimilar	26.52%	44.60%	91.35%	95.18%
J48	50.29%	63.88%	95.52%	97.50%
AdaBoostOnJ48	50.29%	63.88%	95.52%	97.50%
HoeffdingTree	48.23%	62.43%	95.27%	97.37%
EnsembleForest	**57.98%**	**74.04%**	**96.29%**	**98.30%**

Table 6.11: The overall sentence and token accuracy of classifiers on the training set and the testing set. The *Brown* dataset is used for training and testing. The values are extracted from a single experiment run.

6.3 EnsembleForest's Special Results

The key aspect of the *EnsembleForest* algorithm is to measure the likelihood a base classifier is correct. These measurement values (trust values) are to be evaluated on their correctness. They are exactly correct, if they fit the behavior defined by the equation 4.1. Therefore, the trust values and

the corresponding predictions are analyzed. The figure 6.4 displays the expected correctness of predictions (blue) and the observed correctness of predictions (green). The result data had to be sampled by trust value intervals, to measure an average correctness for all predictions falling into a trust value interval. Each interval holds at least 5000 elements to secure a low variance for the observed accuracy. Expected correctness (ExpCor) for an interval $]a, b]$ is computed by averaging over all fitting trust values:

$$ExpCor(X)_{a,b} = \frac{1}{|X|} \sum_{x \in X} TrustValue(x) * sgn_{a,b}(x)$$

$$sgn_{a,b}(x) = \begin{cases} 1, & a < TrustValue(x) \leq b \\ 0, & \text{otherwise} \end{cases}$$

$$X = \text{Dataset of all level-1 generalizer outputs}$$

The figure 6.4 visualizes, that the trust values are not exactly fitting the expected accuracy. The probability a prediction is correct if the corresponding trust value is below about 0.6 is on average greater than the expected probability given by the trust value. For trust values close to 1 the distance of expectation and observed accuracy shrinks.

Although, the correctness of a base classifier prediction is not measured perfectly, the computed trust values are useful. To evidence this assumption, the results of *EnsembleForest*'s level-1 generalizers are evaluated. The resulting table 6.12 displays a lot of information concerning this issue. There, it is stated that in 85.53% of all cases all base classifiers are correct in their prediction, and in 1.76% of all cases all base classifiers are wrong in their prediction. For the other cases, the trust values are very important. There, the highest trust value (c) of the correctly predicting base classifiers should be higher than the highest trust value (w) of the wrongly predicting base classifiers. In 85.01% of all cases that is the case. In about 0.2% of the cases, at which at least one correct and one wrong prediction exist, c equals w. Than, the correct class is only predicted, if the expected average error related to c is below the expected average error related to w. As an unpractical calculation rule for the expected average tree error was applied, it is assumed this case is not resolved by *EnsembleForest* properly. The output of *EnsembleForest*'s level-1 generalizers is not only of base validators but as well of the prediction changer. To remind: The prediction changer aims on finding the correct prediction, given no base classifier is correct. According to the agreement pattern levels, these are the most difficult cases for a combination method. If the prediction changers trust value is higher than all the base validators trust values, it is defined that the *EnsembleForest* algorithm assumes all base classifiers to be wrong in their prediction (*no correct assumed*). In the setup of this thesis, the prediction changer achieves to assume *no correct* in 0.07% of the cases, at which indeed no base classifier is correct. He assumes *no correct* wrong in 2.8e-07% of all cases.

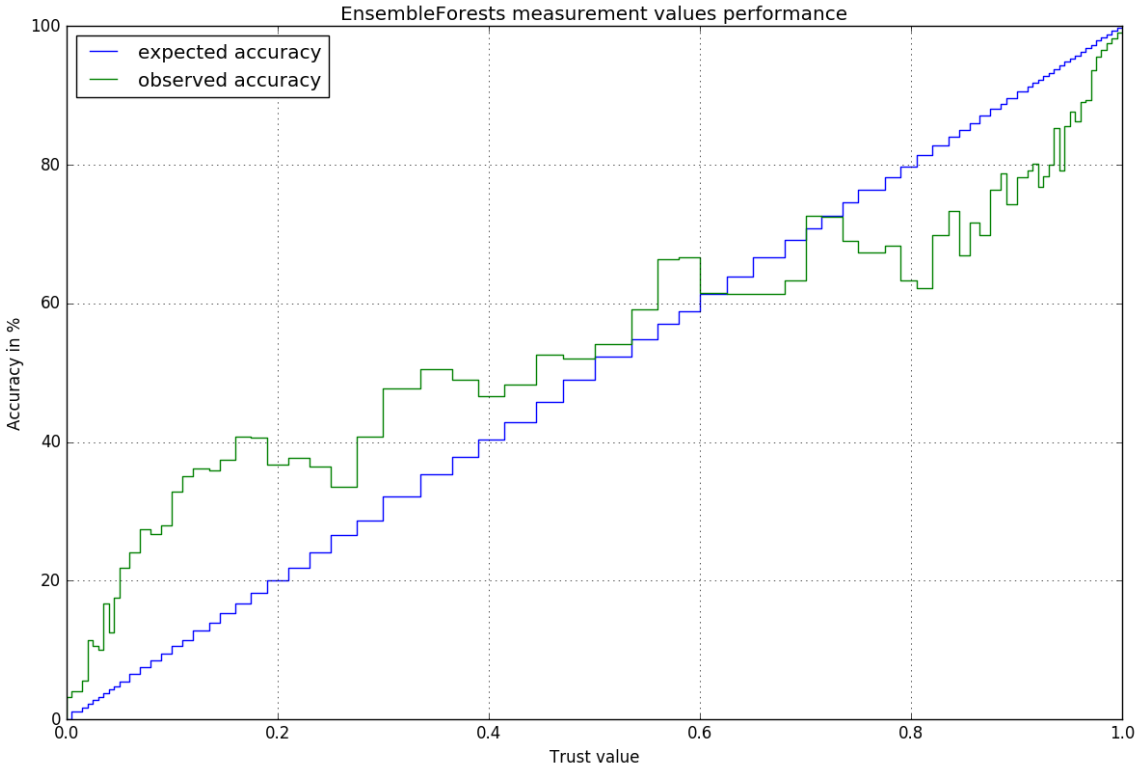

Figure 6.4: Trust value evaluation on the *Brown* dataset. An *EnsembleForest* (EF) algorithm tries to estimate the observed accuracy of its base classifiers. Therefore, the EF estimates an expected accuracy. The accuracy is displayed in %

Pattern	%
no wrong	85.53
c > w	10.81
c == w	0.03
c < w	1.88
no correct	1.76
vote correct | at least one correct and wrong	[85.01, 85.21]
no correct assumed | no correct	0.07
no correct assumed | at least one correct	2.8e-07

Table 6.12: Trust value performance evaluation. The second column holds percentages of cases at which a pattern is fitted. Abbreviations mean: *no wrong*- No tagger misclassified the case (all predictions are correct), *c* - The highest trust value of all correct predicting taggers, *w* - The highest trust value of all erroneous predicting taggers, *no correct* - No tagger misclassified the case, *vote correct* - Voting system of *EnsembleForest* picks a correct tagger prediction, *no correct assumed* - It is assumed, by the prediction changer, that no tagger predicted correctly

In general, an *EnsembleForest* model predicts a single class for every element to be classified. This single class is selected by the model's voting system from all the predictions of the model's level-1 generalizers (base validators and the prediction changer). Instead of returning a single prediction, a voting system might as well return an ordered list of the n-best predictions. For the table 6.13 such a list for each element of the testing dataset of an experiment on the *Brown* dataset is created. Elements of these lists are all the distinct predictions of the level-1 generalizers. For each distinct prediction only the highest trust value is considered. Elements in the list are sorted descending by this highest trust value. In contrast to the voting system of the *EnsembleForest* algorithm, predictions of equal trust values are not sorted by the average error of the corresponding level-1 generalizer. Instead, they remain in the order they appear. By this it is caused, that the first element of these n-best lists is not always the same as the prediction of the *EnsembleForest* algorithm. Of course, it would be possible to include a second stage ordering of the n-best list by the expected average error values, but the expected effort therefore did not stand in relation to the potential gain. For example, from the table 6.12 one can estimate that the first n-best list element and the *EnsembleForest* prediction will be different in about 0.03% of all cases at maximum. The table 6.13 does contain the probabilities the first n elements of these n-best lists contain the correct prediction for an element. There, the first element is correct with a probability of 96.35%. That is the same probability the prediction of an *EnsembleForest* model is correct (compare table 6.5). The probability the correct classification is contained in the first two elements of the n-best list of an *EnsembleForest* model is 98.64%, which is even two levels higher than the benchmark level (compare table 6.4 for the benchmark level).

n-best predictions, n=	P(At least one correct tag in n-best list of EF)
1	0.9635
2	0.9864
3	0.9888
4	0.9893
5	0.9893
6	0.9893

Table 6.13: The probability that the n-best list of an *EnsembleForest* (EF) model contains the correct solution. The EF model is tested on the *Brown* dataset. The n-best lists are ordered only by the highest trust value a prediction (tag) got from EF's level-1 generalizers. At maximum there are six predictions in the n-best list, because the EF model is trained on five base classifiers, so, there are five base validators and the prediction changer, which all may predict a different class for an element.

Chapter 7

Discussion and Conclusion

This thesis introduced a classifier classification method, called *EnsembleForest*, which is based on ensembles of decision trees. The corpus linguistic NLP task of POS tagging was investigated and issues, such as tagset mapping was discussed. Some classifier classification methods, especially the *EnsembleForest* algorithm were examined in experiments. There *EnsembleForest*'s capability to serve as a classifier combination algorithm was evaluated and its performance compared to some other similar algorithms and some other basic classifier combination methods.

Experiments showed that *EnsembleForest* outperformed all incorporated taggers. Hence, it fulfilled the aim of classifier combination, since it achieved an increased accuracy compared to any individual tagger. Furthermore, all included combiners were outclassed if a minimum amount of training dataset size was given. Classifier combination methods are said to be comparable by the cumulative inter tagger agreement pattern level they reach on a particular dataset. The *EnsembleForest* was the only classifier able to reach accuracies between inter tagger agreement accuracies for *correct tag is present but is tied* and *a minority of taggers is correct*. Compared works, which computed the pattern values as well, did not reach this level for any of their classifiers. Nevertheless, they achieved higher overall accuracies for their taggers and combiners. This might be caused either by usage of a different dataset and a partly different set of taggers. It might also be the case that the incorporated taggers and combiners in this thesis do in general not perform as well as classifiers of compared works.

Looking at error reduction, the *EnsembleForest* algorithm was able to reduce the error of the best individual tagger (TreeTagger) by 15.35% for averaged overall accuracies on the *Brown* data. When the tagset was mapped to a smaller tagset, the *EnsembleForest* algorithm could even reduce the error by 42.57% compared to the best tagger on the *Brown* data. By application of the *EnsembleForest* algorithm and succeeding usage of tagset mapping, an error reduction of up to 60.74% was

achieved compared to the best individual algorithm results, which was gained beforehand tagset mapping. Therefore, it is stated that whenever a classifier performs POS-Tagging utilizing a big tagset, an accuracy increase is most easily gained by mapping the results of that classifier to a another minimum sized tagset, which is still appropriate for a given use case.

Functionality of the *EnsembleForest* algorithm was evaluated. Therefore, trust values, expected average tree errors, and the prediction changer performance were analyzed. Trust values computed by *EnsembleForest* level-1 generalizers did not perfectly estimated the base tagger correctness probability. However, they achieved what they were intended for. In about 85.11% of all cases, for which at least one tagger prediction was correct and one tagger prediction was wrong, *EnsembleForest* models could correctly classify the situation. That is, because the highest trust value of all correct predicting taggers was greater than the greatest trust value of all wrongly predicting taggers. Whenever all predictions of taggers were correct, classification of *EnsembleForests* was correct either. The initially chosen computation of the expected average tree error was shown to be inapplicable. Therefore, another, better solution was suggested, which remains to be tested empirically. The prediction changer was sometimes able to detect that all taggers are predicting wrongly in 0.07% of the cases and extremely rarely it changed a correct prediction to a misclassification (2.8e-07%).

The POS Tagging evaluation achieved to detect that some word classes were definitely easier to assign then others. On the *Brown* data with the *Brown* tagset 15% of all tags from the tagset could be classified perfectly (recall and precision values of 1) by an *EnsembleForest* model. The results indicated that closed word classes are more easy to classify, tokens of short sentences are more likely classified wrong, in mid-long sentences (length of 6-38 tokens) tokens have on average (dependent on the classifier) the same probability to be correct and for longer sentences, tokens are more likely classified correct. The mapping of tagsets to other tagsets did perform as evidenced beforehand. It drastically increased overall accuracy.

To make a final conclusion: The *EnsembleForest* algorithm is applicable for classifier combination tasks in order to yield a higher accuracy compared to the best individual classifier. Results of experiments indicate that *EnsembleForest* models perform well compared to other combination method models. Yet, a lot of improvement possibilities exist. Some of them are named in following future work section.

7.1 Future Work

The implementation of *EnsembleForest* is not yet fully developed. There still is potential for an improvement. Following, some ideas are shortly mentioned. For example, one could apply the function $AvgError_{Improved}$, to compute the expected average error of *EnsembleForest*'s level-1 generalizers, because the other function was shown to perform worse. The prediction changer might be improved by letting it consider the base validator's output as well as level-0 output data. Besides those ideas, this work motivates to analyze the *EnsembleForest* algorithm in more detail. So, an *EnsembleForest* could be combined with an *AdaBoost* model as described in section 4.6. For the POS Tagging, the *EnsembleForest* should be further applied on corpora, for which studies on classifier combination exist already, in order to compare *EnsembleForest* more directly with other combiners. Furthermore, it could be tested, whether *EnsembleForest* is applicable for other domains as well. In addition, *EnsembleForests* could be used for error detection problems, for which an error not only should be detected, but as well suggestions for improved classifications should be made. These are only some possibilities for a future work on the *EnsembleForest* algorithm.

Bibliography

[1] C. D. Manning, H. Schütze *et al.*, *Foundations of statistical natural language processing*. MIT Press, 1999, vol. 999, p. XXXI.

[2] ——, *Foundations of statistical natural language processing*. MIT Press, 1999, vol. 999, p. 22.

[3] L. U. Eric Atwell. Language research group, School of Computing, "The brown corpus tag-set," http://www.scs.leeds.ac.uk/amalgam/tagsets/brown.html, Tech. Rep., accessed: 16.Jan.2017.

[4] "Stanford parser," http://nlp.stanford.edu:8080/parser/index.jsp, Tech. Rep., webservice of Stanford Parser, Update from: 12.Sep.2016, Accessed: 02.Feb.2017.

[5] S. Petrov, D. Das, and R. McDonald, "A universal part-of-speech tagset," *arXiv preprint arXiv:1104.2086*, 2011.

[6] M. Doumpos and C. Zopounidis, *Multicriteria decision aid classification methods*. Springer Science & Business Media, 2002, vol. 73.

[7] S. Tulyakov, S. Jaeger, V. Govindaraju, and D. Doermann, "Review of classifier combination methods," in *Machine Learning in Document Analysis and Recognition*. Springer, 2008, pp. 361–386.

[8] L. Xu, A. Krzyzak, and C. Y. Suen, "Methods of combining multiple classifiers and their applications to handwriting recognition," *IEEE Transactions on systems, man, and cybernetics*, vol. 22, no. 3, pp. 418–435, 1992.

[9] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[10] ——, "Stacked regressions," *Machine Learning*, vol. 24, no. 1, pp. 49–64, 1996. [Online]. Available: http://dx.doi.org/10.1007/BF00117832

[11] Y. Freund, R. Schapire, and N. Abe, "A short introduction to boosting," *Journal-Japanese Society For Artificial Intelligence*, vol. 14, no. 771-780, p. 1612, 1999.

[12] Y. Freund, R. E. Schapire *et al.*, "Experiments with a new boosting algorithm," in *icml*, vol. 96, 1996, pp. 148–156.

[13] D. H. Wolpert, "Stacked generalization," *Neural Networks*, vol. 5, pp. 241–259, 1992.

[14] H. Van Halteren, J. Zavrel, and W. Daelemans, "Improving accuracy in word class tagging through the combination of machine learning systems," *Computational linguistics*, vol. 27, no. 2, pp. 199–229, 2001.

[15] B. Jurish, "More than words: using token context to improve canonicalization of historical german." *JLCL*, vol. 25, no. 1, pp. 23–39, 2010.

[16] C. by W. N. Francis and H. Kučera, "A standard corpus of present-day edited american english, for use with digital computers (brown)," Providence, Brown University, 1964, 1971, 1979.

[17] W. N. Francis and H. Kučera, *Manual of Information to accompany A Standard Corpus of Present-Day Edited American English, for use with Digital Computers*, 1979th ed., Providence, Rhode Island: Department of Linguistics, Brown University, 1964, available: http://icame.uib.no/brown/bcm.html.

[18] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[19] A. Schiller, S. Teufel, and C. Thielen, "Guidelines f ur das tagging deutscher textcorpora mit stts," *Universit aten Stuttgart und T ubingen*, 1995.

[20] H. van Halteren, J. Zavrel, and W. Daelemans, "Improving data driven wordclass tagging by system combination," in *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics - Volume 1*, ser. ACL '98. Stroudsburg, PA, USA: Association for Computational Linguistics, 1998, pp. 491–497. [Online]. Available: http://dx.doi.org/10.3115/980845.980928

[21] E. Brill, "Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging," *Computational linguistics*, vol. 21, no. 4, pp. 543–565, 1995.

[22] "python-crfsuite 0.9.1 - python binding for crfsuite," https://pypi.python.org/pypi/python-crfsuite, Tech. Rep., accessed: 20.Jan.2017.

[23] X. Huang, A. Acero, H.-W. Hon, and R. Foreword By-Reddy, *Spoken language processing: A guide to theory, algorithm, and system development*. Prentice hall PTR, 2001.

[24] M. Honnibal, "A good part-of-speech tagger in about 200 lines of python," https://explosion.ai/blog/part-of-speech-pos-tagger-in-python, Tech. Rep., accessed: 10.Dez.2016.

[25] H. Schmid, "Probabilistic part-of-speech tagging using decision trees," in *New methods in language processing*. Routledge, 2013, p. 154.

[26] ——, "Improvements in part-of-speech tagging with an application to german," in *In Proceedings of the ACL SIGDAT-Workshop*. Citeseer, 1995.

[27] ——, "Treetagger - a part-of-speech tagger for many languages," http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/, Tech. Rep., accessed: 11.Dez.2016.

[28] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[29] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*. Morgan Kaufmann, 2016, vol. Fourth Edition.

[30] J. R. Quinlan, *C4. 5: programs for machine learning*. Elsevier, 2014.

[31] J. R. Quinlan *et al.*, "Bagging, boosting, and c4. 5," in *AAAI/IAAI, Vol. 1*, 1996, pp. 725–730.

[32] P. Domingos and G. Hulten, "Mining high-speed data streams," in *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2000, pp. 71–80.

[33] A. Bifet, G. Holmes, B. Pfahringer, R. Kirkby, and R. Gavaldà, "New ensemble methods for evolving data streams," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '09. New York, NY, USA: ACM, 2009, pp. 139–148. [Online]. Available: http://doi.acm.org/10.1145/1557019.1557041

[34] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[35] S. Johansson, "The tagged {LOB} corpus: User\'s manual," 1986.

Appendix A

Appendix

A.1 Implementation

The *EnsebmleForest* algorithm (see chapter 4) and the experimental setup as described in chapter 5 are implemented by the author of this work. The code is bundled as an executable tool, and made available online twice (see A.1). If you consider downloading or using the tool or the *EnsembleForest* algorithm, please mind to notice the author.

- Version ready to execute on Roedel server:
  roedel.gcdh.de:/roedel/dsteding/ensembleForestProject/EnsembleForestProject
- Tool ready for download and installation:
  roedel.gcdh.de:/roedel/dsteding/ensembleForestProject/download
  and
  git@vcs.gcdh.de:dsteding/ensembleForestProject.git
- Configuration files executed for experimental setup:
  /roedel/dsteding/ensembleForestProject/download/executedExperimentConfigsDuringThesisWriting

A.1.1 Reproducibility

The results of the executed experiments are accessible on the Roedel server. All the experiment runs processed to create the results for this thesis can be re-executed. Therefore, the configuration files for all those experiment runs are provided with the distributed versions of the tool.

A.1.2 Implementation Details

The implementation is mainly written in the programming language *Python*. Third party tools like the NLTK package [18], the Weka tool [29] and the *TreeTagger* [27] are required for the experimental setup. The tool includes a detailed README, which explains how to install and use the tool and how to incorporate additional datasets or additional algorithms. The tool is written for further usage. So, experiments with different configurations from the ones executed for this thesis are possible to create and execute.

A.2 Additional Tables

Three tables which did not fit in the main written part are presented here. Fist two tables A.1 and A.2 show classifier performances on the inter tagger agreement patterns on the *DTA_18th* dataset and on the *DTA_19th* dataset. The tables are structured as like the corresponding table 6.5 on the *Brown* dataset. Third table A.3 contains the concrete values, which are visualized in figure 6.1 in the overall results section 6.1.

	All correct	*Majority correct*	*Correct is tied*	*Minority is correct*	*All wrong & don't agree*	*All wrong & agree*	*P(classifier correct)*
Overall Pattern	0.8551	0.0768	0.0193	0.0288	0.0109	0.0091	-
Unigram	**1.0**	0.4341	0.0208	0.0682	0.0	0.0	0.8908
Brill	**1.0**	0.6431	0.0373	0.1034	0.0	0.0	0.9082
HMM	**1.0**	0.7826	0.1937	0.2672	0.0	0.0	0.9266
Perceptron	**1.0**	0.7907	0.8764	0.4853	0.0	0.0	0.9467
TreeTagger	**1.0**	0.8455	0.8716	**0.4366**	0.0	0.0	0.9494
MajVote	**1.0**	**1.0**	0.2454	0.0	0.0	0.0	0.9366
TagPairSimilar	**1.0**	0.6142	0.8541	0.1666	0.0006	0.0	0.9235
J48	0.9499	0.9188	0.87	0.4264	0.0006	0.0	0.9118
AdaBoostOnJ48	0.9499	0.9188	0.87	0.4264	0.0006	0.0	0.9118
HoeffdingTree	0.9499	0.9046	0.731	0.3667	**0.0088**	0.0	0.9064
EnsembleForest	**1.0**	0.9237	0.901	0.4079	0.0008	0.0	0.9552
EnsembleForestPF-1	**1.0**	0.9234	0.9011	0.4088	0.0001	0.0	0.9552
EnsembleForestPF-2	**1.0**	0.9232	0.8997	0.4093	0.0005	0.0	0.9551
EnsembleForestPF-3	**1.0**	0.9237	**0.903**	0.4108	0.0001	0.0	**0.9553**

Table A.1: Classifier performance on base classifier agreement patterns in range $[0, 1]$. Results are averaged over four experiment runs on the *DTA_18th* dataset.

	All correct	*Majority correct*	*Correct is tied*	*Minority is correct*	*All wrong & don't agree*	*All wrong & agree*	*P(classifier correct)*
Overall Pattern	0.8717	0.0764	0.0179	0.0213	0.0073	0.0053	-
Unigram	**1.0**	0.3832	0.0154	0.66	0.0	0.0	0.9027
Brill	**1.0**	0.6549	0.028	0.0935	0.0	0.0	0.9242
HMM	**1.0**	0.7764	0.1257	0.2772	0.0	0.0	0.9392
Perceptron	**1.0**	0.8293	0.9159	0.4489	0.0	0.0	0.9611
TreeTagger	**1.0**	0.8927	0.9147	**0.4768**	0.0	0.0	0.9665
MajVote	**1.0**	**1.0**	0.3427	0.0	0.0	0.0	0.9543
TagPairSimilar	**1.0**	0.5875	0.917	0.1885	0.0007	0.0	0.9371
J48	0.8999	0.9354	0.924	0.4226	0.0012	0.0	0.8815
AdaBoostOnJ48	0.8999	0.9354	0.924	0.4226	0.0012	0.0	0.8815
HoeffdingTree	0.8999	0.9147	0.6937	0.4061	**0.0183**	0.0	0.8756
EnsembleForest	**1.0**	0.9423	0.9414	0.4167	0.0007	0.0	**0.9695**
EnsembleForestPF-1	**1.0**	0.9428	0.9289	0.4197	0.0013	0.0	0.9694
EnsembleForestPF-2	**1.0**	0.943	0.9342	0.4157	0.0007	0.0	0.9694
EnsembleForestPF-3	**1.0**	0.9425	**0.9398**	0.4148	0.0006	0.0	0.9694

Table A.2: Classifier performance on base classifier agreement patterns in range $[0, 1]$. Results are averaged over four experiment runs on the *DTA_19th* dataset.

Classifier	500	1000	2000	5000	7733	20,000	57,339
Unigram	**76.96**	**79.69**	**80.22**	**83.87**	**85.12**	**87.49**	**89.53**
	0.290	0.585	0.191	0.065	0.084	0.034	0.003
Brill	**78.35**	**81.20**	**82.39**	**86.57**	**88.12**	**90.69**	**92.92**
	0.271	0.577	0.081	0.075	0.054	0.021	0.007
HMM	**80.52**	**83.01**	**83.65**	**87.11**	**88.42**	**90.81**	**92.94**
	0.743	0.329	0.363	0.095	0.037	0.019	0.001
Perceptron	**87.09**	**89.20**	**90.31**	**92.39**	**92.52**	**93.86**	**95.01**
	0.239	0.561	0.228	0.044	0.049	0.015	0.005
TreeTagger	**88.15**	**90.22**	**90.87**	**92.81**	**93.43**	**94.64**	**95.70**
	0.381	0.340	0.201	0.068	0.030	0.012	0.001
MajVote	**83.06**	**85.64**	**87.08**	**90.05**	**91.13**	**93.07**	**94.74**
	0.241	0.658	0.131	0.052	0.042	0.011	0.003
TagPairSimilar	**86.02**	**87.96**	**88.23**	**89.76**	**89.88**	**90.73**	**91.36**
	0.100	0.122	0.186	0.070	0.040	0.021	0.002
J48	**86.28**	**88.72**	**90.07**	**92.80**	**93.33**	**94.68**	**95.57**
	0.326	0.323	0.108	0.045	0.045	0.008	0.001
AdaBoostOnJ48	**86.73**	**88.72**	**90.12**	**92.80**	**93.33**	**94.68**	**95.57**
	0.385	0.323	0.108	0.0449	0.045	0.008	0.001
HoeffdingTree	**88.47**	**90.63**	**91.26**	**93.06**	**93.53**	**94.20**	**95.22**
	0.333	0.383	0.216	0.052	0.036	0.019	0.004
EnsembleForest	**88.25**	**90.49**	**91.48**	**93.61**	**94.13**	**95.35**	**96.35**
	0.630	0.400	0.130	0.035	0.051	0.006	0.002

Table A.3: Classifier's overall accuracies in % and variances in percentage points on *Brown* data per dataset size (columns). Results are averaged over seven experiment runs.