Utrecht University

Training a name-variant model using historical data

Author:
Jorik van Kemenade
4167783

Supervisor:
dr. ir. Gerrit Bloothooft

Second Supervisor:
dr. Marijn Schraagen

Bachelor Thesis Artificial Intelligence, 15 ECTS

17 August, 2016

Abstract

One of the main problems in the field of record linkage is the variation in names. A possible approach for dealing with this variation is to remove name variation. To remove this variation each name in the historical records has to be converted to a base form. In this study a model is presented that can convert Dutch first names to their base form. To build this model a subset of a dataset [5] containing 132.140 first names and their base form will be used to train three different multiclass classifiers: k Nearest Neighbours, Boosted Decision Trees and Support Vector Machines. Each of the classifiers is compared on accuracy, training time and classification speed. The best performing classifier, a boosted decision tree, is then selected for training and testing on the entire dataset. The final model is a boosted decision tree with a learning rate of 1.0 and 200 decision trees with a maximum depth of 17 levels. The validation error of the model, using 10-fold cross validation, is 84.56%. The accuracy of the final model on the test set, containing 24.576 names and 447 base forms, is 85.04% with a classification speed of more than 300 samples per second.

Contents

1 Introduction

From 1795 - 1813 the Netherlands was occupied by France. During this so called French period (Dutch: 'Franse tijd') the French officially introduced the 'Burgerlijke stand', the Dutch statutory registration service. Since the introduction of the 'Burgerlijke stand' a record of almost every birth, marriage, divorce and death in the Netherlands exists. All those records combined show us the complex relations between the Dutch citizens. However, combining all those records is not a trivial task. Many disciplines are involved in all steps that are necessary to solve this huge task.

One of the approaches of combining the civil records is matching the names of the participants on each certificate. However due to name variation and transcription errors this approach is not reliable. This problem can be solved by removing name variation from the record set. To remove this variation we propose to train a model which can classify a name to a base name. This model will be trained using a dataset containing name variants coupled to standardised base names. In this thesis three different methods for the design and training of this model are compared. Each model is assessed on accuracy, classification speed and training time.

In this thesis section 1.1 introduces the field of record linkage and section 1.2 elaborates on the problem of name variance. Section 2 is an introduction into multiclass classification problems and three methods which can potentially be used to solve the problem are proposed. The historical records that are used to train the model are discussed in section 3. In section 4 the three models are implemented and tested to select the most promising model. In sections 5, 6 and 7 the most promising model is evaluated on the entire dataset.

1.1 Problems in record linkage

In the early days of the statutory registration all registration was based on non unique information. A civil certificate comprises the event for which the certificate was drafted, the date on which the event happened and the participants in the event [22]. In order to link two certificates to each other there must be a certain confidence that the participants mentioned on the certificates are the same. Deciding if two certificates relate seems trivial. If the names of the participants on both certificates are the same and if the dates on the certificates are within a reasonable and logical time frame the two certificates are likely candidates for a match. However, three problems arise when using this technique for matching records.

The first problem is that the names on historical records are inconsistent. There are two main sources causing this inconsistency. The first source of inconsistency is an absence of standards and conventions in formatting the applicants names. Although every civil record had to comply to the standard data format of each civil form, the content of the form was not standardised. As a result a single person can have non matching mentions on different civil certificates. For example, on one certificate a full name containing three first names can be used and on another certificate only the first name or a variant of the first name is mentioned.

The second problem are transcription errors. There are two main classes of transcription errors: historical and modern. First the historical transcription errors. These errors are in the original civil certificates and they arise when the civil servant misspells a name. The second type, the modern errors, arise when the original certificate is digitalised. These errors are caused by a transcriber either misspelling or misreading a name.

The first and second problem are not that big of a problem for people who are skilled in linking records. Humans are capable of linking data despite some of the noise that is introduced due to human error. However due to the number of records that is digitally available it is almost impossible to verify all possible record combinations for candidate matches. And this is where the third, and biggest, problem arises. To process that many records computers are used to automatically link our civil records in a fast and reliable manner. However, in contrast to humans, computers are not equipped to deal with the noise and errors in the records. Enabling computers to reliably link records, despite some of the errors, is one of the bigger challenges in the field of record linkage.

1.2 Record linkage using name variants

A common approach of handling variation in records such as described in section 1.1 is to compute the difference between two names. There are several measures for computing this difference, but each measure is used in a similar way. If the difference between two names is below a certain threshold the two names are considered to be a match, if the difference is above the threshold the names are considered to be different.

Human record linkers don't perform a character by character comparison of two names. They consider certain parts and features of a name to decide whether two names are similar. One of the important factors in deciding if two names are likely to match is the origin of the name. If two names share the same base name they are more likely to be compatible than if two names have a different origin.

One approach of making automatic record linkage more human like is to convert all names to their base name. This way name variation is discarded without some of the problems that arise using threshold based techniques such as minimum edit distance. While this solves the problem of name variation new problems, or actually existing, problems arise. Besides handling transcription errors in the data, names have to reduced to their base form. Although there are some rules of thumb on how to reduce a name to it's base form this is a non trivial task.

In this study several approaches for training this model using historical data will be evaluated. Each of the methods has to solve the same problem: *Given any name, find the most likely base name which is associated with that name.* This requires a model that can classify noisy input into many classes and data which can be used to train the models.

1.3 The data: Dutch first names

For training, verifying and testing the model a data set containing 132.140 variant - base name pairs is used [5]. This dataset is constructed using data from the Genlias project. The Genlias project aims to index all Dutch civil certificates of the 19^{th} century. Due to legal restrictions preserving the privacy of Dutch citizens not all civil certificates can be used. Birth certificates have to be at least 100 years old, marriage records need to be 75 years old and death certificates must be 50 years or older. Together with other sources, such as the website www.WieWasWie.nl, 22 million civil certificates are digitally available.

Bloothooft and Schraagen [6] have found a method that can be used to identify name variants in this data. This method combines several clustering steps to find first name clusters. Using this technique they identified a total of 927 name clusters. Each of the 132.140 names is assigned to a cluster. This is done with different levels of *confidence*, the confidence of a name-variant pair depends on the procedure that is used to standardize the name. The dataset that is used in this study is a variant of the dataset from Bloothooft and Schraagen [5]. For more information on the characteristics of the data and the clustering procedure please refer to chapter 3.

1.4 Modelling name variation

In this study three multiclass classification techniques are used to solve the problems mentioned in sections 1.1 and 1.2. Each of the methods will be assessed on their ability to solve these problems in the context of record linkage. This means that both accuracy and speed are important for assessing the performance of each approach. A third, less important, factor is the training time of each approach.

The main goal of this study is to find a reliable approach for removing name variation from a dataset. However the underlying problem and the methods used to solve this problem relate to a larger and more general field: multiclass classification. Multiclass classifiers are classifiers which are able to divide samples into one of k-classes where $k > 2$. However when k increases, so for cases where $k >> 2$, it gets increasingly more difficult to train multiclass classifiers. Each of the approaches that is discussed in section 2 are tested on a maximum of 26 classes. For this specific case the domain consists of over 400 classes. Therefore the conclusions of this study might not only be interesting for the domain of record linkage but also for the domain of multiclass learning as well.

2 Multiclass classification methods

The underlying problem of standardising a given name to a basic form is a multiclass classification problem. The aim of a multiclass classification algorithm is to assign a single class from a range of classes to an input sample. A k-class multiclass problem can be defined as:

Definition 1. Given a training data set of the form $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i \in \mathbb{R}^n$ is the ith sample and $y_i \in \{1, ..., K\}$ is the ith class label, we aim at finding a learning model $\mathbb{H}$ such that $\mathbb{H}(\mathbf{x}_i) = y_i$ for new unseen samples [4]

There are several approaches for solving a multiclass problem. Aly [4] created an overview of the various multiclass classification algorithms which are available. Multiclass classification algorithms can be divided into two main groups. The first group of algorithms decomposes the problem into binary classification problems. The second group of algorithms are an extension of existing binary classification algorithms to multiclass classification algorithms.

In this chapter both classes of algorithms will be covered. First section 2.1 will cover the decomposition of the problem in binary problems. Then sections 2.2 - 2.5 will cover the extensions of binary classification algorithms to multiclass classification algorithms. Each algorithm will be introduced followed by a short review of performance and suitability for solving the problem of name standardisation.

2.1 Decomposition into binary classification

As mentioned earlier, solving a multiclass problem is more complex than solving a binary classification problem. If a multiclass classification problem can be decomposed in several binary classification problems, the binary problems can be solved using efficient proven classification techniques. There are several approaches to decomposing a multiclass classification problem to a set of binary classification problems. There are three main approaches for dividing a multiclass classification problem into binary problems, the *one-versus-all* (OVA) approach, the *all-versus-all* (AVA) approach and *encoding*. Both the OVA and AVA method are discussed in [4, 19], the AVA method is also discussed in [11] and different encoding strategies are discussed in [3, 4, 12].

In the all-versus-all approach each class is compared to every other class. This requires the training of binary classifiers that can discriminate between classes. So given an arbitrary input and two classes $y_i, y_j \in \{1, ..., K\}$ the binary classifier has to classify the input as either y_i or y_j. This classification is done for every combination of classes and the class which gained the most wins out of the $K - 1$ competitors is the overall winner. This approach requires $K(K - 1)/2$ different binary classifiers. With more than 400 different classes this approach requires over 80.000 binary classifiers thus making it impractical for this study.

The one-versus-all approach requires less binary classifiers. In this approach K binary classifiers are trained, each classifier has to decide if a sample belongs to class K or any of the other $K - 1$ classes. The class K with the classifier that has the highest confidence out of all K classifiers is the winner. Due to the relatively low number of binary classifiers, in this study approximately 400, this approach might be suitable. However Ou and Murphy [19] found that class imbalance between the number of positive samples of class K and the number of negative samples in classes $K - 1$ has a negative impact on the performance of the binary classifiers.

There are two possible solutions for this class imbalance problem. The first approach by Sun *et al.* [23] uses boosting which is discussed in section 2.3. The second approach by Daqi *et al.* [11] uses a construction called *economic learning sets*. An economic learning set is a dataset that only contains samples from the larger dataset that are most likely to be relevant. This approach is discussed further in section 2.6.

	f_1	f_2	f_3	f_4	f_5	f_6	f_7
Class 1	0	0	0	0	0	0	0
Class 2	0	1	1	0	0	1	1
Class 3	0	1	1	1	1	0	0
Class 4	1	0	1	1	0	1	0
Class 5	1	1	0	1	0	0	1

Figure 1: Example of an error-correcting output-coding schema. The schema consists of K rows and N columns. Each row represents a codeword and each column represents a binary classifier,

The third approach is based on encoding strategies. Dietterich and Bakiri [12] describe a strategy in which N binary classifiers can distinguish K classes.This is done using an output strategy with so called error-correcting output codes. Each of the K classes gets a codeword of length N, the classes and code words can be combined to a matrix M. An example of an error-code matrix can be found in figure 1.

The code matrix from figure 1 is one of the possible encodings for a 5-class problem. Each row in M corresponds to a class K, each of the columns in M corresponds to one of the N binary classifiers. When an unseen sample is tested, each of the N classifiers classifies the sample as either 0 or 1. The classifications of each binary classification will be used to construct a codeword j. The winning class is the class with the codeword from M that has the shortest Hamming distance to j. The Hamming distance is computed using the definition in figure 2. For this technique to work it is important that the codewords are carefully selected. Each row needs to have a sufficient Hamming distance to other rows and the binary classifiers should be uncorrelated. Dietterich and Bakiri [12] have compared several methods for finding an optimal encoding.

Allwein *et al.* [3] created an extension to the error-correcting output-codes as proposed by Dietterich and Bakiri. Instead of constructing a matrix $M \in \{-1, +1\}^{K \times N}$, a matrix $M \in \{-1, 0, +1\}^{K \times N}$ is constructed. If $f_i = 0$ the classification of f_i is ignored. Experiments with this encoding scheme showed that it is a viable candidate for replacing other encoding schemes, but there is no clear improvement when compared to for example error-correcting output-codes by Dietterich and Bakiri.

2.2 k-Nearest neighbours

The most basic (multiclass) classification method is the k-nearest neighbours algorithm (kNN) [1]. The kNN algorithm is based on the *nearest neighbour rule*. The nearest neighbour rule is a very simple rule for which no actual training is required. In definition 1 the training set is defined as being of the form $(\mathbf{x}_i, u_i)$, where $\mathbf{x}_i \in \mathbb{R}^n$ is the ith sample and $y_i \in \{1, ..., K\}$ is the ith class label. Consider an unknown data point $\mathbf{x}$ for which a label $y \in \{1, ..., K\}$ has to be predicted. The nearest neighbour rule dictates that the label of the closest data point $\mathbf{x}_i$ in the training set is the most likely label for $\mathbf{x}_n$, so $y = y_i$. Finding the nearest neighbour is done by arranging the training set according to distance from $\mathbf{x}$. Define $(\mathbf{x}_{[n]}(\mathbf{x}), y_{[n]}(\mathbf{x}))$ as the nth pair with respect to $\mathbf{x}$. The entire rearranged data set will then be:

$$(\mathbf{x}_{[1]}(\mathbf{x}), \mathbf{y}_{[1]}(\mathbf{x})) \leq (\mathbf{x}_{[2]}(\mathbf{x}), \mathbf{y}_{[2]}(\mathbf{x})) \leq ... \leq (\mathbf{x}_{[N]}(\mathbf{x}), \mathbf{y}_{[N]}(\mathbf{x})) \tag{1}$$

The final hypothesis of the nearest neighbour algorithm will be:

$$g(\mathbf{x}) = y_{[1]}(\mathbf{x}) \tag{2}$$

Using the nearest neighbour rule the algorithm always returns the class of the datapoint in the training set which is closest to $\mathbf{x}$. Since only the nearest neighbour is considered this method is susceptible to noise in the training data. This can be solved by increasing the number of neighbours which will be taken into consideration. Instead of considering just one neighbour, kNN considers the first k-neighbours. Consider a vector $\mathbf{x}$ with an unknown class y. Similar to the single neighbour approach the training set will be arranged as in equation 1. The class y which is most common in $(\mathbf{x}_{[1]}(\mathbf{x}), \mathbf{y}_{[1]}(\mathbf{x})), ..., (\mathbf{x}_{[k]}(\mathbf{x}), \mathbf{y}_{[k]}(\mathbf{x})$ will be the predicted class for the unknown datapoint $\mathbf{x}$. Equation 3 is the final hypothesis of the kNN algorithm where [...] is the *Iversion bracket notation*: $[P]$ is defined to be 1 if P is true and 0 otherwise.

$$g(\mathbf{x}) = \underset{n \in 1, ...K}{\arg\max} \sum_{i=1}^{k} \left[y_{[i]}(\mathbf{x}) = y_n \right] \tag{3}$$

There are two factors which influence the performance of a kNN classifier. The number of considered neighbours and the distance measure. The first factor is the value of k. If $k = 1$ the resulting hypothesis is very complex, if k is very large the final hypothesis will have a much lower complexity. The optimal value for k is problem specific and has to be found by testing several values of k. The second factor is the distance measure that is used to measure the distance between $\mathbf{x}$ and $\mathbf{x}_i$. There are several approaches to measuring this distance, some popular choices can be found in figure 2. The optimal distance measure for a kNN classifier is problem specific, the best distance metric is selected during the model selection phase.

Braycurtis	$\frac{\sum_{i=0}^{d} \lvert x_i - y_i \rvert}{\sum_{i=0}^{d} \lvert x_i \rvert + \sum_{i=0}^{d} \lvert y_i \rvert}$	Canberra	$\sum_{i=1}^{0} \frac{\lvert x_i - y_i \rvert}{\lvert x_i \rvert + \lvert y_i \rvert}$
Chebyshev	$\arg\max_i \lvert x_i - y_i \rvert$	Euclidean	$\sqrt{\sum_{i=0}^{d} (x_i - y_i)^2}$
Hamming	$\frac{\sum_{i=0}^{d} [x_i \neq y_i]}{d}$	Manhatten	$\sum_{i=0}^{d} \lvert x_i - y_i \rvert$

Figure 2: Some popular distance measures as used in [20]. d is the dimension of the input vectors.

kNN has proven to be a simple yet effective method for classification. One of the main advantages is that kNN requires no training, however kNN is quite susceptible to noise in the 'trainging data'. Another disadvantage of kNN is the time and space complexity for larger problems, the memory requirement is Nd and the computational complexity is $O(Nd + N \log k)$ where d is the dimension of the input vectors [1]. However by preprocessing and storing the data in the appropriate data structures the memory requirements and computational complexity can be reduced.

2.3 Boosting

Boosting is a general method that can be used to "boost" the performance of any learning algorithm [21]. The concept of boosting is to take a weak learner, a learning algorithm which performs slightly better than random guessing, and combine many weak learners to create a *single* strong learner. There are several methods for using boosting to boost the performance of multiclass learning algorithms, these methods include: AdaBoost.M1, AdaBoost.M2, AdaBoost.MH and AdaC2.M1 [21, 23]. Each of these methods is an extension to the original AdaBoost algorithm as designed by Freund and Schapire [13] which can be found in algorithm 1.

Algorithm 1 The boosting algorithm.

Require: Training set $(x_1, y_1), ..., (x_m, y_m)$ where $x_i \in X,\ y_i \in Y = \{-1, +1\}$
Initialize: $D_1(i) = 1/m$
  for $t = 1, ..., T$ **do**
    Train a weak learner h_t using D_t
    $\epsilon_t = \Pr_{i \sim D_t} [h_t(x_i) \neq y_i]$
    $\alpha_t = \frac{1}{2} \ln(\frac{1-\epsilon_t}{\epsilon_t})$
    $D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$ ▷ Z_t is a normalization factor.
  end for
Return: $H(x) = \text{sign}(\sum_{t=1}^{T} \alpha_t h_t(X)))$

The basic idea of the AdaBoost algorithm is to train a set of weak learners. Each weak learner has a different distribution of weights over the training set. Initially the weight is distributed equally over all training samples. But on each round the weight distribution is updated. The weight on the incorrectly classified examples is increased, the weak learners are forced to focus on the 'harder' examples in the training set. The magnitude of the weight increase depends on the weak hypothesis h_t and the method that is chosen to compute α_t. Usually α_t is computed using the error ϵ_t of h_t, this error depends on the number of misclassified samples and their weights. The error e_t can be computed using equation 4.

$$\epsilon_t = \Pr_{i \sim D_t} [h_t(x_i) \neq y_i] = \sum_{i: h_t(x_i) \neq y_i} D_t(i) \tag{4}$$

In [13] Freund and Schapire proved some basic properties of the AdaBoost algorithm. The first property is the reduction of the training error. First the training error ϵ_t of h_t is defined as $\frac{1}{2} - \gamma_t$. The γ_t measures how much better the weak hypothesis h_t predictions are than random guessing. Using this notation Freund and Schapire have proven that the training error of the final hypothesis H it limited to:

$$\prod_t \left[2\sqrt{\epsilon_t(1-\epsilon_t)}\right] = \prod_t \sqrt{1-4\gamma_t^2} \;\leq\; \exp\left(-2\sum_t \gamma_t^2\right) \tag{5}$$

Equation 5 shows that if the weak hypothesis is slightly better than random, so $\gamma_t > 0$, then the training error drops exponentially. Freund and Schapire also showed that it is possible to bind the generalization error of the final hypothesis in terms of the training error, the sample size m, the VC-dimension d of the weak hypothesis space and the number of rounds T of boosting.

$$\hat{\Pr}\left[H(x) \neq y\right] = \tilde{\mathrm{O}}\left(\sqrt{\frac{Td}{m}}\right) \tag{6}$$

Equation 6 implicates that boosting will overfit if run for too many rounds T. However empirical observations showed that AdaBoost was not overfitting, even after thousands of iterations. Sometimes it was even observed that AdaBoost lowered the generalization error after the training error had reached zero. In response to these findings an alternative bound of the generalization error was given in terms of the margins of the training examples. The margin of an example (x, y) is defined in equation 7 [21].

$$\frac{y\sum_t \alpha_t h_t(x)}{\sum_t \alpha_t} \tag{7}$$

A margin is a number in $[-1, +1]$ and can be interpreted as the confidence of the prediction. It is proven that larger margins on the training set translate into a superior bound of the training error and that boosting is very effective in increasing the margins on the training set. Schapire [21] shows that boosting increases the margins even after the training error has reached zero.

As mentioned earlier the AdaBoost algorithm was originally designed to be used in classification problems with just two classes. There are several methods of using boosting for multiclass classification problems. The first, and simplest method, is to reduce the problem to a set of binary problems. Two popular candidates include the one-versus-all approach and the error-correcting output-codes, both approaches are described in section 2.1. The second method is to extent the original AdaBoost algorithm to support multiclass classification. Sun *et al.* [23] have compared two multiclass boosting algorithms, the 'straightforward' AdaBoost.M1 algorithm and the AdaC2.M1 algorithm.

AdaBoost.M1 differs slightly from the original AdaBoost algorithm as shown in algorithm 1. The main differences are a different weight update function and a different final hypothesis. The weight update formula in algorithm 1 will be replaced by equation 8.

$$D_{t+1}(i) = \frac{D_t(i)\exp(-\alpha_t I\left[h_t(x_i) = y_i\right])}{Z_t} \tag{8}$$

Where Z_t is the normalization factor and I is defined by equation 9.

$$I\left[h_t(x_i) = y_i\right] = \begin{cases} +1 & \text{if } h_t(x_i) = y_i \\ -1 & \text{if } h_t(x_i) \neq y_i \end{cases} \tag{9}$$

The final hypothesis $H(x)$ of algorithm 1 will be replaced by equation 10.

$$H(x) = arg\max_{C_i}\left(\sum_{t=1}^{T} \alpha + t\left[h_t(x) = C_i\right]\right) \tag{10}$$

The training error of AdaBoost.M1 has the same characteristics as the training error of AdaBoost. Therefore if a weak learner has a *better than random* performance the training error will drop exponentially. The AdaC2.M1 algorithm has a different approach to the multiclass problem compared to AdaBoost.M1. Instead of just adding a weight factor to individual samples, a cost function is also feeded into the weight update function.

Consider a k-class classification problem consisting of m samples. Let $c(i, j)$ be the cost for misclassifying an example of class i to class j. By definition $c(i, j) = 0.0$ if $i = j$. Now define the cost of misclassifying samples of class i as $c(i)$. There are many possible rules for defining $c(i)$, a possible form is given in equation 11.

$$c(i) = \sum_{j}^{k} c(i,j) \tag{11}$$

However in AdaBoost the weight factor is sample based and not class based, so the class based cost function has to be expanded to a sample-based cost model. Suppose the i^{th} sample is misclassified. If the i^{th} sample belonged to class j than the misclassification costs would be the costs for misclassifying class j so $c_i = c(j)$. The updated weight update formula 8 for AdaC2.M1 will be:

$$D_{t+1} = \frac{c_i D_t(i) \exp(-\alpha_t I\left[h_t(x_i) = y_i\right])}{Z_t} \tag{12}$$

To lower the training error of the AdaC2.M1 algorithm the objective is to minimize Z_t [23]. Minimizing Z_t is done by finding a positive α_t. α_t is directly dependent on the error ϵ_t. In AdaC2.M1 α_t is computed using a cost altered version of the α_t in algorithm 1,

$$\alpha_t = \frac{1}{2} \ln \left(\frac{\sum\limits_{i, y_i = h_t(x_i)} c_i D_t(i)}{\sum\limits_{i, y_i \neq h_t(x_i)} c_i D_t(i)} \right) \tag{13}$$

To get a positive α_t equation 14 should hold. This is similar to the AdaBoost requirement of 'better-than-random' performance. But the main difference between AdaBoost.M1 and AdaC2.M1 is not the upper bound of the training-error, it is how the weight update functions influence the resampling effect of both algorithms.

$$\sum_{i, y_i = h_t(x_i)} c_i D_t(i) > \sum_{i, y_i \neq h_t(x_i)} c_i D_t(i) \tag{14}$$

As mentioned earlier the AdaBoost strategy is to increase the weight of false predictions and decrease the weight of true predictions. The AdaBoost.M1 algorithm increases the weights of all true and false predictions equally. Therefore large classes with many misclassified samples get relatively the same weight increase as smaller classes with less misclassified samples. However since AdaBoost.M1 already trains on the many misclassified samples from the larger class, the weight increase for the larger class is not important compared to the weight increase of the smaller classes. By altering the ratio of the weight increase the weight of the smaller classes can be increased while the weight of the larger classes is unaltered. This is implemented by AdaC2.M1. AdaC2.M1 introduces a cost function that can change the weight ratio between classes. Sun *et al.* [23] showed that this method performs better in imbalanced class situations than AdaBoost.M1. However finding the optimum cost function was a time consuming process.

2.4 Decision trees

As mentioned in chapter 2.3 AdaBoost requires a weak learner. Decision trees are known to be used in classification [16], mutliclass classification [4] and as weak learner in boosting [21, 23]. In the experiments in chapter 4 decision trees will be used to train boosted classifiers.

Decision trees are trees that classify instances by sorting them on feature values [16]. Each node in the decision tree represents a feature, each branch in the tree represents a possible node value and each leaf represents a class. Classifying an unknown sample using a decision tree is a simple process. Start at the root of the tree and evaluate the feature of the root node. Traverse the tree via the correct branch and repeat until a leaf node is reached. The value of the leaf node is the class of the unknown sample.

Although the concept of a decision tree is straightforward creating an optimal decision tree is not. Constructing an optimal decision tree is a NP complete problem. A lot of research has been done on finding heuristics for creating the optimal decision tree. However the basic idea is similar for most heuristics and can be found in algorithm 2.

Algorithm 2 The Decision Tree Algorithm (DTA).

Require: Samples **S**, AttributeList A
  create node N
  if all samples in S are of class C **then**
    label N as C
    return N
  end if
  if $A = \varnothing$ **then**
    find most common class C in S
    label N as C
    return N
  end if
  select $a \in A$ with the highest information gain
  label N with a
  for each v in a **do**
    select subset S_1 samples from S where $a = v$
    subtree = DTA(S_1, $A \setminus v$)
    attach subtree to N
  end for
  return N

An important factor in decision tree performance is tree depth. When a decision tree is allowed to reach an unlimited depth the decision tree is very sensitive to overfitting. By limiting the depth, and therefore the number of possible decisions, this can be prevented. Take for example a decision stump, a decision tree with just 1 level, because of the limited depth only one feature can be considered. When adding more levels, the number of possible decisions grows exponentially, creating more opportunity for overfitting. The ideal tree depth is problem specific and is set during the model selection phase.

2.5 Support vector machines

Support vector machines (SVM) are a popular choice for both binary and multiclass classification [2, 4, 7, 14]. As with many classification methods the SVM is originally designed as a binary classification method, however there are several ways to extend the SVM to a multiclass setting. This section will introduce the concept of the SVM and will briefly mention several methods of extending it to multiclass classification.

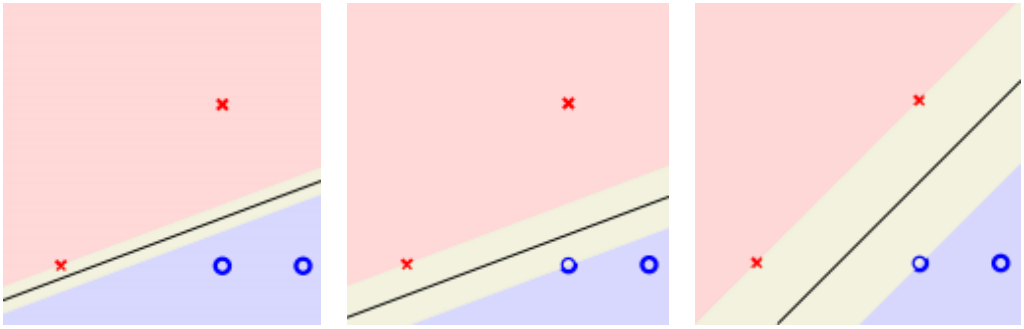
Huang *et al.* [14] explained the basic concept of a SVM. Figure 3 shows three of many possible decision boundaries separating two classes. The question that has to be answered is which decision boundary is preferred over the others. For this particular configuration the decision boundary in figure 3b is preferred over the boundary in figure 3a. The optimal decision boundary is the boundary in figure 3c. That particular boundary is optimal because it maximizes the distance between the classes and the decision boundary. As a result of this distance the classifier is less susceptible to noise, the margin for error is maximized. This boundary is called the maximum-margin separating hyperplane, the goal of the SVM is to find the maximum-margin separating hyperplane. If such a hyperplane exists the final hypothesis of a two class SVM can be defined as:

$$H(\mathbf{x}) = \text{sign}(\mathbf{w}^T\mathbf{x} + b) \tag{15}$$

Where $\mathbf{w}$ is the weight vector of the hyperplane and b is the bias.

Finding the maximum-margin separating hyperplane is done by solving equation 16. In this equation $y_n \in \{-1, +1\}$, $\mathbf{w}$ is the weight vector of the hyperplane and b is the bias. Solving equation 16 is not trivial for bigger data sets. However equation 16 can be converted to a quadratic programming problem and solved using already available solvers.

$$\begin{aligned} \text{minimize:} \quad & \frac{1}{2}\mathbf{w}^T\mathbf{w} \\ \text{subject to:} \quad & y_n\left(\mathbf{w}^T\mathbf{x}_n + b\right) \geq 1 \end{aligned} \tag{16}$$



(a) Small separator (b) Medium separator (c) Optimal separating plane

Figure 3: Three possible planes linearly separating the four data points [2].

The maximum-margin separating hyperplane as defined in equation 16 has one critical problems: it is assumed that the data is linearly separable. By assuming linearly separable data the SVM is limited to only a small class of problems. Both noisy problems, which are not separable due to data points at the 'wrong side' of the decision boundary, and problems with a non linear decision boundary are problematic for this SVM.

The problem with the noisy data can be solved by defining a soft-margin SVM. A soft-margin SVM is similar to the hard-margin SVM as defined in equation 16 however it allows violations of the margin and even classification errors. A soft-margin SVM is defined as:

$$\begin{aligned} \text{minimize:} \quad & \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{n=1}^{N}\xi_n \\ \text{subject to:} \quad & y_n\left(\mathbf{w}^T\mathbf{x}_n + b\right) \geq 1 - \xi_n \end{aligned} \tag{17}$$

Where $y_n \in \{-1, +1\}$, $\mathbf{w}$ is the weight vector of the hyperplane, b is the bias, ξ_n is the allowed margin violation and C is the penalty parameter. The margin violation ξ_n is the amount of margin violation for a data point, $\xi_n = 0$ if a point is classified correctly, $0 < \xi_n \leq 1$ if the point violates the margin and $\xi_n > 1$ if the point is misclassified. By minimising $\frac{1}{2}\mathbf{w}^T\mathbf{w}$ the maximum margin is found, by minimising $\sum_{n=1}^{N}\xi_n$ the least violating margin is found. The parameter C can be used to compromise between finding a large margin and allowing errors. If C is large equation 17 is similar to a hard-margin SVM. If C is small violations of the margin are less important and the margin will be larger. Finding the right balance between a large margin and allowing boundary or classification errors is crucial for the performance of soft-margin SVM's.

There still is a problem which isn't addressed by either equation 16 or 17, the linear margin. By transforming the data into a higher dimension it is possible to create non-linear maximum-margin separating hyperplanes. An efficient method for transforming to a higher dimensional space is using a kernel function. A kernel function uses a non-linear transform $\boldsymbol{\phi} : \mathcal{X} \rightarrow \mathcal{Z}$ such that $\mathbf{z} = \boldsymbol{\phi}(\mathbf{x})$. There are several types of kernels which can be used to train a SVM. As with all previous techniques the optimal kernel function is problem specific and is selected during the model selection phase. Figure 4 shows some of the popular kernel functions which are commonly used in SVM classification.

linear	$\gamma\ \langle x, x^{'} \rangle$
polynomial	$(\gamma\ \langle x, x^{'} \rangle + r)^d$
radial based function	$\exp(-\gamma \left\lvert x - x^{'} \right\rvert^2$ where $\gamma > 0$
sigmoid	$\tanh(\gamma\ \langle x, x^{'} \rangle + r)$

Figure 4: Populair kernel functions as used by [20].

The soft-margin SVM as defined in equation 17 is a two-class classification algorithm. However several extensions from a two-class to a multiclass SVM have been proposed. The most basic extension is to use a decomposition to binary one-against-one or one-against-all problems as discussed in chapter 2.1. Cheong *et al.* [7] propose a multiclass SVM with binary tree structure requiring $N - 1$ SVMs to be trained and a classification complexity of $\log_2 N$, making this approach faster than one-against-all classification strategies while maintaining the same accuracy. Another approach is expanding the SVM algorithm to include multiclass support as done by Crammer and Singer [10] and Huang *et al.* [14].

2.6 Neural networks

Due to problems as described in sections 1.1 and 1.2 the model that is used to standardise names has to deal with incomplete and noisy data. One of the main sources of noise are transcription errors and spelling mistakes. Karen Kukich published a survey on techniques that automatically record words in text [17]. According to Kukich "Neural nets are likely candidates for spelling correctors because of their inherent ability to do associative recall based on incomplete or noisy input".

In the same survey Kukich refers to her own research in which she used a standard back-propagating neural network to correct spelling mistakes in 183 surnames. The output layer of that network consisted of 183 nodes, one for each name. The input layer was constructed using 450 nodes in 15 sequential blocks of 30 nodes each. Each 30-node block contained one node for each character in a 30-character alphabet. As a result of that names with a maximum length of 15 characters could be corrected. The net was trained using artificially single-error misspellings of the 183 names. The final neural network achieved a near perfect accuracy on names with single spelling errors.

Experiments performed by Cherkassky and Vassilas confirmed Kukich's findings [8, 9]. Cherkassky and Vassilas trained different neural nets with different input strategies on a 24-100 name lexicon and they too found a near 100% correction rate. However they found that the learning rate and the number of hidden units made significant differences in the performance of the networks.

Ou and Murphey [19] endorse the findings of Cherkassky and Vassialas. According to Ou and Murphey many classification systems were developed for two-class classification problems and the extension from two-class to multiclass classification is non-trivial. In line with the experiments of Cherkassky and Vassialas this often leads to unexpected complexity or weaker performance. In their research Ou and Murphey compare the performance of different system architectures on a variety of multiclass problems. The two major system architectures used in their research are a single neural network system and a system of multiple neural networks. The multiple neural network approach can be divided into three different competition schemes: one-against-one, one-against-all and P-against-Q. The OvO and OvA schemes are discussed in section 2.1, the PvQ scheme lies somewhere between the OvO and the OvA scheme. In the PvQ scheme P classes are tested against Q other classes. All neural networks are trained using the back propagation algorithm. Besides measuring the performance of each network on multiclass datasets the network's learning capabilities are also evaluated with respect to imbalanced training data, the number of classes and the size of the training set.

Ou and Murphey concluded that performance wise the one-against-one approach is very effective on data sets with a large number of classes and a large number of training samples. However due to the quadratic increase with K in the number of networks that have to be trained this method seems to be impracticable for over 400 classes. The next best approach was the one-against-all approach. However this approach suffers greatly from imbalanced class distribution. Daqi *et al.* [11] introduce two concepts which, when combined, can solve the problem of class imbalance in multiclass classification problems.

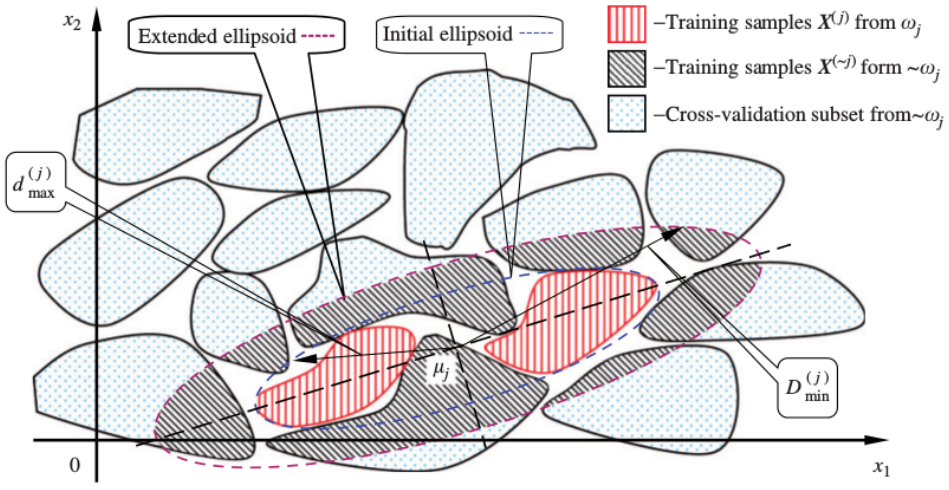


Figure 5: Creating an economic subset containing samples from w_j and a subset of the samples from $w_{\sim j}$ [11].

The first concept are so called *economic learning subsets*. In a one-against-all situation the set containing the negative examples has many distant samples. These distant samples are far away from the decision boundary and are less relevant for constructing the decision boundary. By removing the more distant samples the negative class gets smaller while maintaining the interesting samples. Selecting the closer samples instead of the more distant samples is done drawing an initial ellipsoid around the training samples. Then this ellipsoid is expanded as shown in figure 5. As the ellipsoid extends the training sets contains more and more samples from class $w_{\sim j}$. The ellipsoid is expanded until the number of samples from w_j and $w_{\sim j}$ is at an acceptable ratio.

The second improvement proposed by Daqi *et al.* is the reinforcement of thin distribution regions. The samples of class w_j and $w_{\sim j}$ have a different density throughout the feature space. If the density of class w_j is locally much lower than the density of class $w_{\sim j}$ a local imbalance between the number of samples in w_j and $w_{\sim j}$ is formed. Due to the local class imbalance and the use of the back propagation algorithms the decision boundaries in those regions tend to move away from class $w_{\sim j}$ and towards w_j. By adding virtual samples to those regions this problem can be solved. A similar problem arises when the number of misclassified samples is much smaller than the number of correctly classified samples. As a result of that the update function of the back propagation algorithm will change if there are to many correctly classified samples compared to the number of incorrectly classified samples.

The improvements proposed by Daqi *et al.* have shown to improve training times as well as performance in datasets with 10 and 26 classes. And therefore the improvements proposed by Daqi *et al.* might be viable for the training one-against-all neural nets for the standardisation of names.

The research above had one conclusion in common. Increasing the number of classes, increases the complexity of the networks, increases the training time, increases the classification time and increases the amount of data required for training the networks. Due to the availability of computational resources, the availability of data and the performance requirements of record linkage Neural Networks will not be considered in this study.

3 The data

In chapter 1.3 a dataset created by Bloothooft and Schraagen was mentioned. The process of creating this dataset as well as some of the data's characteristics are discussed in [5, 6]. Section 3.1 will briefly cover the creation of the dataset and the quality of the data, section 3.2 covers the cleaning procedure, section 3.3 will elaborate on the conversion of names to feature vectors and the final datasets and characteristics will be discussed in section 3.4.

3.1 The Genlias dataset

In the early 90s the Genlias project was founded. The goal of the Genlias project was to digitalise all the records of the Dutch civil archive from the 19^{th} century. Thanks to the efforts of all the volunteers that entered the names of all newborn children and their parents, married couples and their parents, and the names of deceased citizens and their partners, Genlias is now the biggest database of Dutch names from that period. This data is publicly available on WieWasWie.nl and the 2011 version consists of 4.170.416 birth certificates, 3.039.236 marriage certificates and 7.657.298 death certificates, 611.650 pleadings of succession, 173.917 baptismal entries, 8.976 marriage listings and 579 funeral listings.

All certificates in the Genlias database contain 49.990.511 first name fields resulting in 1.368.070 unique, gender dependent, first name references. There are 593.200 male references, 665.489 female references and for 109.362 references the gender could not be determined. Since a first name reference can consist of multiple single first names, *e.g. Johannus Wilhelmus Franciscus*, the total number of unique first names, *e.g. Johannus, Wilhelmus, Franciscus*, is less than the number of unique first name references. To create a dataset with unique first names Bloothooft and Schraagen cleaned the first name references in several steps which will be discussed shortly in this section.

The first step consisted of cleaning the first name references, the details of this cleaning procedure are discussed by Bloothooft [5], and resulted in 1.055.195 cleaned first name references. These first name references were divided into 189.672 singleton names. A second cleaning iteration was initiated to clean the singleton names and this resulted in 187.000 different singleton names. However most of the 187.000 singleton names were very rare. This means that the singleton name is almost never used at the first position of a combined name. From the singleton names only 40% is found more than once on the first position of a first name reference, 39% only once and 21% is never found on the first position of a first name reference. If the names which are found more than once on the first position of a combined name are considered to be commonly used names the Genlias project has a total of 76.104 first names.

Since this study focusses on standardising names for record linkage the 189.672 singleton names, the dataset before the second cleaning iteration, is more relevant than the cleaned 76.104 commonly used names. Bloothooft and Schraagen [6] propose a method for finding name variants using civil records. Bloothooft and Schaagen identify name variants by matching the name and date fields of civil records. Whenever they encounter an inexact match this match is labeled as a name-variant pair. The matching method they used for finding inexact matches is the so called 4/5-matching. If 4 out of 5 names and the dates on a group of certificates are a direct match the non-matching pair is considered to be a name-variant pair. Then the name-variant pairs are clustered and each cluster was labelled with the most common name in that cluster.

The dataset that is used to create the clusters contained 189.176 singleton names. From the singleton names 132.140 names were extracted with different procedures and different levels of confidence related to that procedure. Information regarding the exact procedures and levels of confidence can be found in figure 6. Using these procedures each of the 132.140 names could be reduced to any of the 927 name clusters.

3.2 Cleaning procedure

In section 3.1 a dataset containing 132.140 variant - base name pairs was introduced. Before this dataset can be used to train and test the models the dataset has to be prepared. This preparation consists of two stages. The first stage is cleaning and preparing the data for modelling first names. The second stage is creating the feature vectors which are used for the actual machine learning.

The goal of the cleaning procedure is to remove noise from the data which might cause overfitting. This can be done by simplifying the singleton names. This simplification consists of three steps. Step 1 is replacing all upper case characters with lower case characters. Then all accents are removed from the letters, *e.g.* à is replaced by

Level	Procedure
1	The name is found using inexact matching of records.
2	The name has a semi phonetic form similar to a name from 1.
3	The name has to meet all of the following requirements: **a)** The name is longer than 5 characters. **b)** At least two names from type 2 have the same initial with Levenstein distance of exactly 1. **c)** There is only one name as defined by 3b with frequency ≥ 100. **d)** There is only one name as defined by 3b but the base form is similar to one found by 4.
4	The beginning of a semi phonetic form of a name has at least four characters in common with a name from 1 or 2.

Figure 6: Procedure for finding name variants. Level 1 has the highest confidence, level 4 the lowest [5].

a. The last step consists of replacing all non letter characters by a single non letter character, e.g ! and : are replaced by #.

After simplifying the given names the base forms are analysed. It turns out that there are several single letter classes: w, f, b, m, h, s, j and a class where every other characters is followed by a period. Because each of the classes contains 10 or less entries these classes are removed from the dataset.

The last step of the cleaning process is to remove the classes that have to few samples to be used for machine learning. For this series of experiments the threshold was set to 50 samples. All classes containing less than 50 samples, 468 in total, are removed from the dataset. The cleaning and removing of minority classes from the dataset resulted in 447 classes with 122.403 singleton names.

3.3 Feature vectors

After the cleaning procedure the singleton names have to be converted to feature vectors. The first problem is the length of the singleton names. Each of the singleton names had a different length, however feature vectors have to have an identical length. Lewellen [18] has tested several implementations for converting words to equally sized feature vectors. Lewellen proposes three methods and each of the methods is tested using an artificial neural network. The three methods tested by Lewellen can be found in figure 7.

Left	l	u	c	a	s			
Split	l	u	c				a	s
Bi-D	l	u	c	a	c	u	a	s

Figure 7: Example of name representations in a feature vector with length 8 [18].

The first method is quite simple, the name is extended to the correct length by adding extra trailing whitespace characters. The split method also adds additional whitespace characters. However instead of adding the whitespace to the end of the name, the whitespace is added in the middle. As a result of that the letters at the beginning of the name get the same emphasis as letters at the end of the name. Only the middle letters of long names get less emphasis. The third and last approach, the bi-directional approach, tries to solve this problem by filling the middle part with letters from the beginning and end of the name. However by shorter names in long feature vectors the name might be repeated several times within one feature vector. As a result, the emphasis will move towards the middle part of the names instead of the begin and end. Since for the classification of Dutch first names the beginning and ending of a name are more important than the center this is an undesirable effect.

Lewellen found that *Split* and *Bi-D* outperformed the *Left* method on all types of position-altering and position-maintaining errors. However *Split* showed the best overall performance. Since the longest name in the dataset consists of 20 characters, all feature vectors will be 20 characters long and the *Split* method will be used to create a distributed representation.

Since the Scikit-learn [20], that is used to perform all experiments, has native support for feature vectors containing integers or boolean-values, two feature strategies will be tested. The first feature strategy converts letters to integers. A white space is converted to 0, a-z is represented by the integers 1-26 and the non-alpha character is converted to 27. So *estien* with base form *esther* has feature vector: $[5, 19, 20, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 9, 5, 14]$.

The structure of the boolean-based vectors is similar to the structure used by Lewellen [18]. The feature vector will consist of 20 groups of 28 booleans. Each boolean in a group of 28 represents either the empty character. one of 26 letters, or the special character.

3.4 The datasets

With the data cleaned and the feature vectors formed it is time to construct the datasets. For this study two datasets will be constructed. A dataset that will be used for model selection, and a dataset containing all the variant - base name pairs used for experiments with the final model.

The dataset that will be used for model selection is ideally not too large, but also not too small. Model checking involves trying a lot of different settings and models. A smaller dataset reduces the computational time required for the model verification. But the dataset must consist of enough classes and samples to see which models are likely candidates.

Figure 8 shows the distribution of class sizes. The first thing that is apparent when looking at the class distribution is the unequal distribution of class size. The smallest class consists of just 50 samples while the largest class contains 3520 samples. For the smaller subset that is used for model selection the 51 classes in the 200-300 range will be used. Using these classes has three advantages.

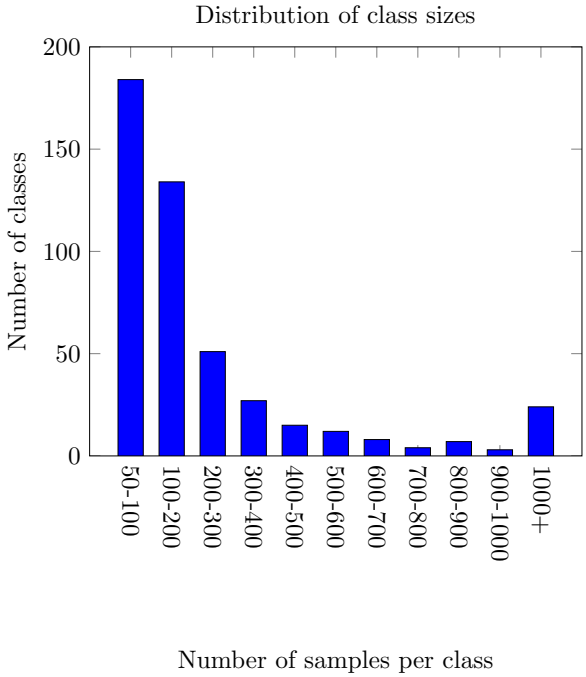


Figure 8: The distribution of class sizes over the dataset. The majority of the classes contains less than 100 samples. The largest class, *hendrik* consists of 3520 samples.

The first advantage is the number of classes, with 51 classes this group is unquestionably in the multiclass classification range. The second advantage is the class size, with class size ranging from 200-300 samples it is possible to reserve 50 samples for verification and 150-250 samples for training. The 150-250 sample range is a common range in the distribution of class sizes and the total number of items used for training is just 9495. The third and final advantage is that by taking an entire range the data has a class imbalance. This imbalance has a negative impact on the performance of most classification algorithms, but is inherently present in this dataset. By performing model selection with imbalanced data the chosen model is more likely to be able to handle the imbalanced data of the entire dataset.

4 Model selection

In chapter 2 several techniques for multiclass classification were discussed. To select the most suitable technique for training the model, each of the multiclass classification techniques from chapter 2 will be tested on the 51 class subset which is discussed in chapter 3. All experiments will be performed on a 64-bit Linux computer with a 3.1 GHz quad-core and 16GB of RAM using Python 2.7 and Scikit-learn 0.17. Methods will be evaluated on both training time and performance and the most promising techniques will be used for training on the entire data set.

4.1 k-Nearest neighbours

All kNN experiments are performed using the nearest neighbours algorithms of Scikit-learn [20]. As mentioned in chapter 3.3 there are two main strategies for converting the data into features. The integer-based strategy and the boolean-based strategy. For kNN both strategies were tested on training and testing time and accuracy for different distance measures and $k = 10$. The integer-based kNN classifiers scored upwards of 90% with a train and test time of less than 1 second. The boolean-based strategy had similar, although lower, scores. But the train and test time of all classifiers exceeded the 15 seconds with a maximum of almost 30 seconds.

Based on these scores the preferred feature-strategy for kNN will be integer-based. But there is a third possible feature-strategy which is not mentioned in chapter 3.3. This feature-strategy is only applicable to kNN and uses an unedited version of the singleton names, *e.g.* 'estien' or 'estieu'. This feature-strategy is combined with the Levenshtein edit distance. The Levenshtein edit distance can be computed by counting the minimum number of *insert*, *delete* and *substitute* operations that is required to transform string a intro string b [15]. This strategy is not supported by Scikit-learn and is therefore tested on a custom, unoptimised kNN implementation. Due to the costs of computing the Levenshtein distance and the unoptimised kNN algorithm the train and test time was approximately 15 seconds. However the Levenshtein-strategy outperformed both the boolean- and integer-based strategies and therefore the integer- and Levenshtein-based strategies will be considered.

As mentioned in chapter 2.2 there are two factors that influence the performance of a kNN classifier, the number of neighbours k and the distance metric that is used. To find an optimal value for both k and the distance metric we will use a grid search with 3 parameters on the small data set with just 51 classes. k will be sampled with values ranging from 1 - 49, there will be 7 different distance measures (including Levenshtein) and two methods for weighing data points. The total number of evaluated kNN classifiers is 686.

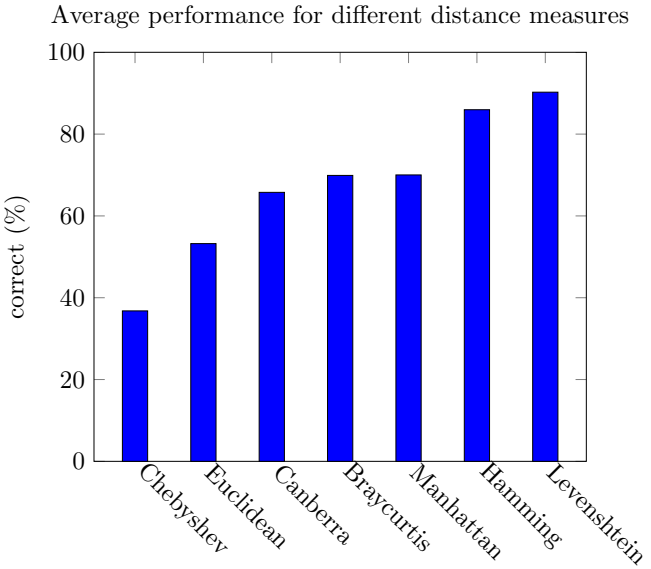


Figure 9: A comparison of different distance measures. The performance is the average performance of all kNN classifiers with that distance measure and with $k = [1...49]$ and two different weight metrics.

Figure 9 shows the average performance for all 686 kNN classifiers per distance measure. It is clear that the distance measure has a considerable effect on the performance of the classifiers. With almost 20% difference between the best and second best distance measure it is clear that the Hamming and Levenshtein distance are the best measures for this data set.

For the next experiment kNN classifiers with Hamming and Levenhtein distance have been tested with two different weight measures, 'uniform' and 'distance'. The difference between the 'uniform' and 'distance' weight measure is that with the 'uniform' weight measure all points in each neighbourhood are weighted equally and with the 'distance' weight measure points have a weighting which is the inverse of their distance to $\mathbf{x}$. Figure 10 shows the performance of the kNN classifiers for different k for each weight measure. For this dataset the 'distance' weight measure is clearly superior for both Hamming and Levenshtein distances and the optimum value for k is approximately 10.

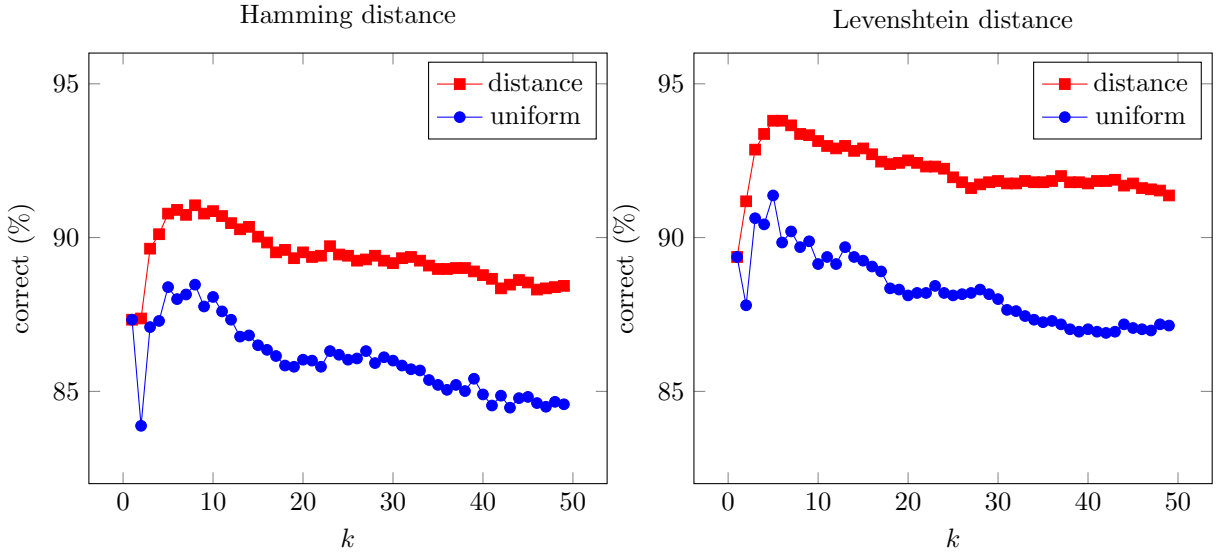


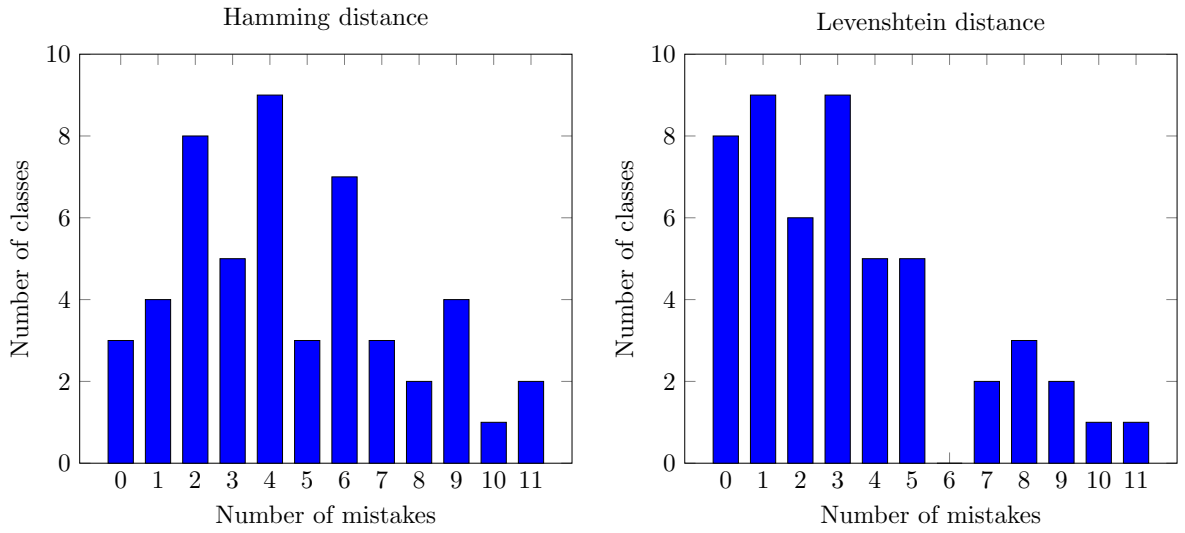
Figure 10: Comparison of the performance of two different distance measures. Each distance measure is tested with varying values of k and with uniform and distributed weight for neighbours.

When analysing the mistakes made by the Hamming distance classifier the first thing that is noticable is that in 48 of the 51 classes the classifier made one or more mistakes. Figure 11 shows the distribution of the number of errors made per class. The classes with the most mistakes: *wine, siebren* and *wieger* contribute with each class to approximately 12% of the mistakes. This is approximately twice as much as expected, however the mistakes seem to be distributed evenly across the classes.

Looking at the actual mistakes there seem to be three types of mistakes. The first type of mistake is a mistake you could expect a human to make, for example is classifying Lucun as Lucius instead of Lucas. The second type is when the classifier goes entirely wrong, for example classifying Esterdiena as Sibren instead of Esther. The third type is a mistake that is to be expected from a kNN algorithm with Hamming distance but is not human like. An example is classifying Hensterina as Rens instead of Esther. In both cases all letters are present, thus making it likely that the Rens and Esther examples are all close to Hensterina in Hamming distance. So although the two names are not alike for humans, they are similar for the mechanisms used for classification.

The Levenshtein distance based classifier performed slightly better than the Hamming distance based classifier. As a result of that, only 43 out of 51 classes had one or more errors and there are fewer classes with a high number of mistakes. When looking into the actual mistakes only two types of mistakes are clearly recognisable: the human like errors and the kind of errors where the classifier is entirely wrong.

Ultimately the kNN classifier gained a score of approximately 90% and 95% on a problem containing 51 classes for two different feature strategies. The errors were almost uniformly distributed across all 51 classes and the training and classification time was in the order of 1 and 15 seconds for two different feature strategies. Due to the reasonable performance and training times for both candidates the kNN classifier seems to be a viable candidate for testing on the entire dataset.



(a) 48 out of 51 classes created 1 or more errors.

(b) 43 out of 51 classes created 1 or more errors.

Figure 11: The distribution of number of errors over each of the 51 classes for two different weight measures. The Levenshtein distance based classifier has a better performance and fewer high error classes than the Hamming distance based classifier.

4.2 Boosted decision trees

All AdaBoost experiments are performed using the AdaBoost and decision tree algorithms of Scikit-learn [20]. There are several factors that influence the performance of AdaBoost. The most important factor is to find a base learner which has sufficient performance to be boosted. Figure 12 shows the performance of 100 boosted decision tree classifiers with varying tree depth for two different feature strategies. Both feature strategies seem to have similar performance. The integer-based strategy is a stronger performer on shallow trees. The boolean-based strategy ultimately outperforms the integer-based strategy using deeper trees. The same figure shows the time needed for training each of the classifiers. The boolean-based strategy clearly requires more training time to get an increase in accuracy of less than 1%. So for this experiment decision trees with a integer-based feature strategy and a maximum depth 6 - 20 are the preferred boosted classifiers. Therefore the rest of the model selection uses decision trees with a integer-based feature strategy and depths ranging from 6 - 20.

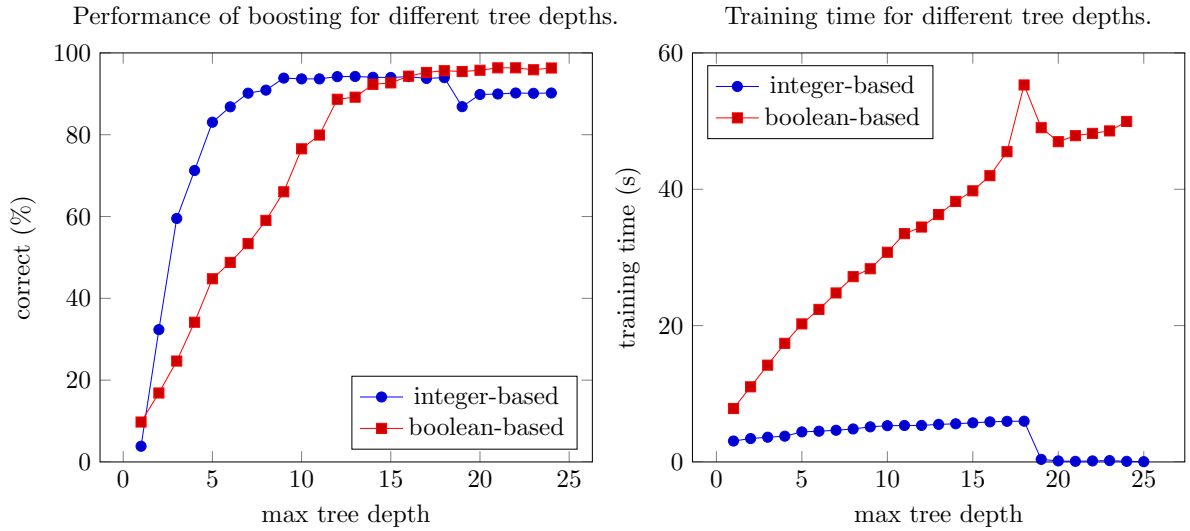


Figure 12: The performance of the boosted classifier for different tree depths and feature strategies. Each boosted classifier consisted of 100 decision trees with varying maximum depth and a learning rate of 1.0

A second factor in AdaBoost performance is the number of weak learners. In chapter 2.3 the number of weak learners was briefly discussed. When more weak learners are introduced the classifier might be more susceptible for overfitting. However empirical experiments have shown that AdaBoost is not really susceptible for overfitting. However, many weak learners are a higher computational burden, increasing both training and classification time. So the second AdaBoost experiment focusses on the error of the boosted classifier in relation to the depth of the decision trees and the number of decision trees. Figure 13 shows the results of this experiment. A decision tree with a depth of 8 outperforms the deeper decision trees in all possible configurations scoring approximately 95%.

Performance for different maximal tree depths and number of classifiers.

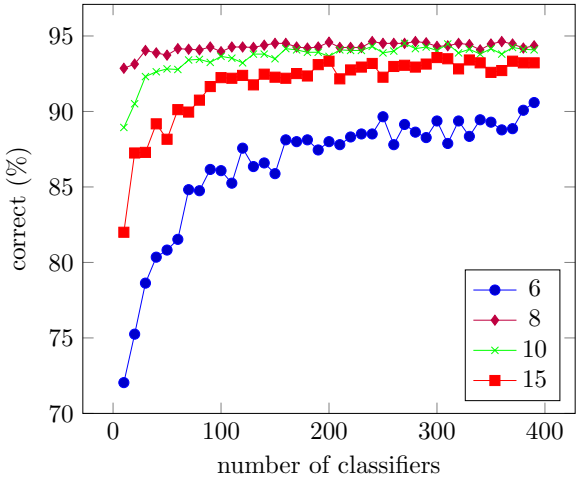


Figure 13: The performance of boosted decision tree classifiers with a different number of classifiers and a varying maximum depth. The learning rate for each classifier is 1.0.

The last experiment of the AdaBoost model selection is used to investigate the relation between the learning rate and the number of classifiers. The learning rate is a factor which shrinks the contribution of each additional weak learner. Figure 14 shows the results of this experiment. As expected a low learning rate suppresses the additional weak learners to much, the new specialised learners have too little influence. A high learning rate has the opposite effect, the new specialised weak learners have a lot of influence, this results in over fitting. The optimum learning rate is approximately 0.3.

Performance for different number of classifiers and learning rates

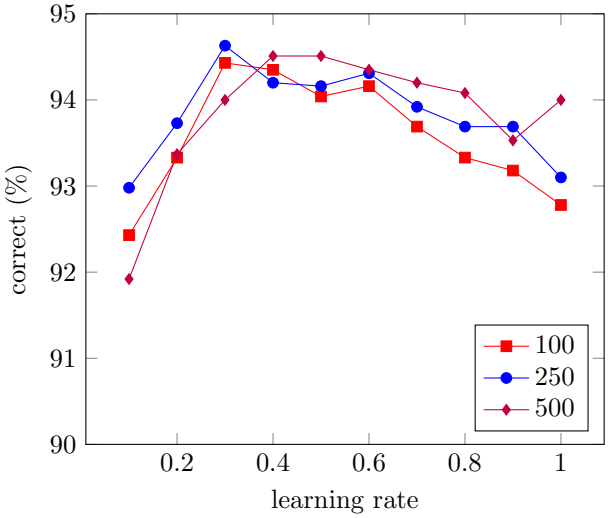


Figure 14: Performance of boosted decision tree classifiers for a different number of trees, a maximum depth of 15 and learning rates varying from 0.1 to 1.0.

When analysing the mistakes made by the classifier the first thing that is noticed is that in just 39 of the 51 classes the classifier made a mistake. Figure 15 shows how many mistakes were made by how many classes.

If 15 is compared to figure 11 it clearly shows that the boosted classifier performs better and has significantly more classes making just a few mistakes.

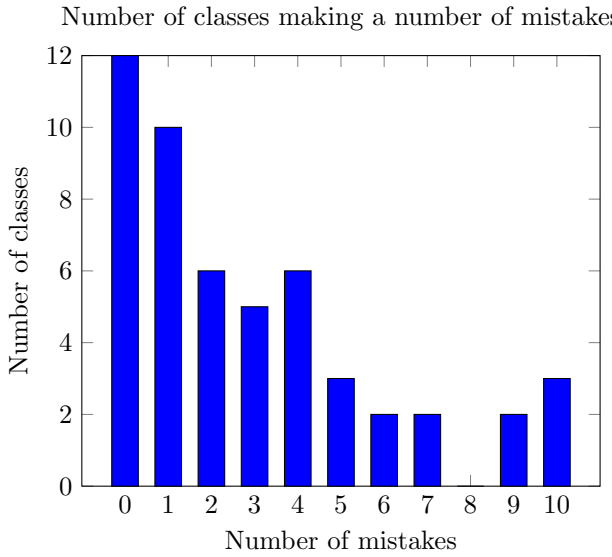


Figure 15: The distribution of number of errors over each of the 51 classes. 39 out of 51 classes created 1 or more errors.

When looking at the actual mistakes there are still the three types which were discussed in section 4.1. However, the distribution of the three types is different. There are more human like errors, for example classifying Cijbe as Seije instead of Sibren. And errors where there are a lot of matching characters such as classifying Lucias as Lucas instead of Lucius. There are less errors where the classification was entirely wrong, one of the few examples is #Aagje# which was classifued as Seije instead of Agatha.

After the first experiments it can be concluded that AdaBoost outperforms kNN on the number of correct classifications. The training time of AdaBoost compared to kNNm is longer. But since the training time of AdaBoost classifiers has been < 5 seconds per boosted classifier and the time complexity is lower compared to kNN, AdaBoost is also a viable candidate for the full experiments.

4.3 Support vector machines

All SVM experiments are performed using the SVM algorithms of Scikit-learn [20]. Scikit-learn supports two main SVM implementations, SVC and LinearSVC. LinearSVC is supposedly a faster implementation of the SVM algorithm than SVC. However the LinearSVC implementation had structurally lower accuracies than the SVC implementation, $< 30\%$. Therefore all experiments are performed using the SVC implementation of Scikit-learn.

The SVC implementation offers 4 different types of kernels: linear, polynomial, radial basis and sigmoid. Each of the 4 kernels is tested with two feature strategies. The sigmoid kernel is the worst performing kernel, for both feature strategies, with scores of approximately 2%. The other kernel types combined with the boolean-based feature strategy scored similarly to the integer-based feature strategy, however training times are significantly longer, exceeding 600 seconds.

Figure 16 shows the accuracy of different kernel types for different values of C on a dataset with an integer-based feature strategy. Although the polynomial kernel has a higher accuracy the training times, with a lower bound of 60 seconds and more than quadratic complexity, are to long to consider this technique for full scale experiments. The linear kernel has similar training times. The radial based kernel has a more reasonable training time, lower bound of 10 seconds, and is therefore a viable candidate for further optimisation.

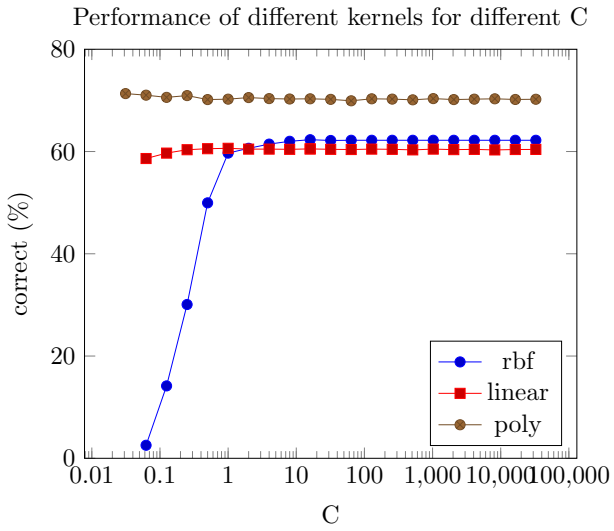
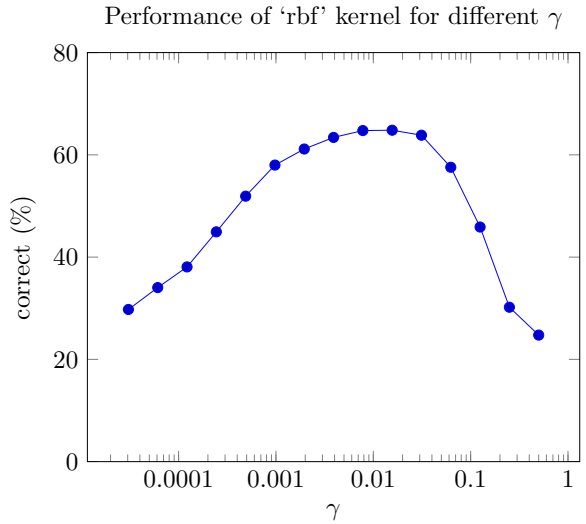
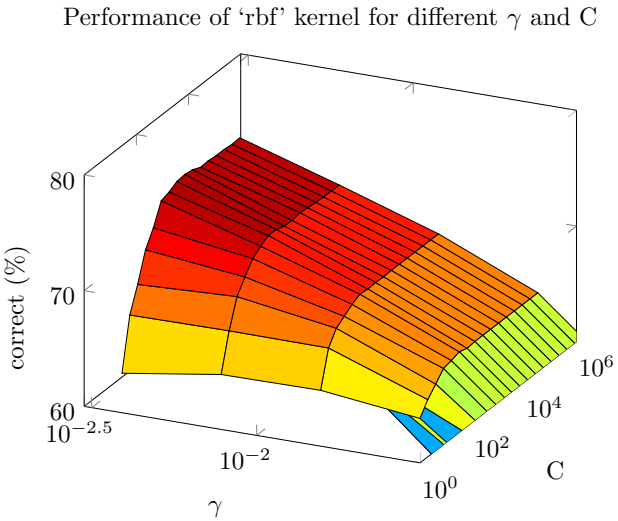


Figure 16: The performance of a SVM with different kernels, one-versus-all scheme for different values of C.

Figure 17 shows the results of further optimisation of the radial basis kernel SVM. The radial basis function kernel has a second value, γ, which can be optimized. γ is a weight factor influencing the range of influence of a training sample. Figure 17a shows the performance of the radial basis kernel for different values of γ with constant C. This clearly shows the influence of γ on the accuracy of the classifier. Figure 17b shows the optimisation process of the radial basis function kernel for different values of γ and C. Finding the right combination of γ and C resulted in accuracies approaching 80%. However an accuracy of 80% and training times of more than 10 seconds, with quadratic complexity, are insufficient when compared to kNN and boosted decision trees.



(a) The performance of SVM with C = 1.0 for different values of γ



(b) A γ - C optimisation. A low γ with higher C seems to be the best performer with scores of 75% or higher.

Figure 17: The performance of a SVM with rbf kernel, one-versus-all scheme for different values of γ and C.

5 Method

In chapter 4 various approaches for creating the name-variant model were examined. From the various approaches boosted decision trees have been the best performing approach, both on accuracy, training and testing time. As a result of that boosted decision trees will be used to test whether learning a name-variant model is feasible.

As discussed in chapter 3 using this dataset creates three problems. The first problem is the number of classes, with 447 different classes there are many classes to choose from. For each of the 447 classes the model has to learn the features that separate each class from the other classes. This requires disjoint classes and enough data to capture all variance within each class, and that is where the second and third problem come in. The dataset suffers greatly from class imbalance. AdaBoost has a natural mechanism for dealing with class imbalance, however AdaBoost does perform better on balanced classes. The third problem is probably the biggest problem, the availability of data. In order for the model to capture all variance in a name cluster, each name cluster has to provide enough samples to learn this variance. If a name cluster has only a handful of samples the model won't be able to learn all variance which will result in more classification errors.

The final experiments of this study will focus on different aspects of these three problems. The first experiment, section 5.1, will be on a subset of the data and is aimed at finding out how the method scales when more samples will be added to the training data. The second experiment, section 5.2 will be on the entire dataset and is aimed at finding the relation between class imbalance and learning variance.

5.1 Scaling

The first experiment is to measure the potential of this method. As discussed earlier two problems are expected. The first problem is the imbalance in data and the other problem is the number of different classes. As discussed in section 2.3 boosting has a natural mechanism for training many classes and dealing with class imbalance. However, the source of the imbalance is more of a problem than the balance itself. Most classes in the dataset are really small, they might be even to small to capture the variance in names. As a result of that the dataset might be insufficient for creating a true name-variance model. The goal of this experiment is to find out if the model can be scaled when more data becomes available.

The dataset for this experiment is contains all classes that consist of 200 samples or more. The total number of base names meeting this requirement is 151. From each of the 151 classes 150 samples will be randomly selected for training the model. The final training set contains 22.650 samples. The rest of the data will be used for testing the final model. Each class will have at least 50 test samples and the entire test set contains 69.162 samples.

The process of tuning the final model will have a similar structure to the procedure used in chapter 4.2. However instead of using a validation set, the model will be validated using 10-fold cross validation. For the parameter selection the same experiments as in section 4.2 will be used.

The first parameter optimisation will consist of indexing the influence of depth. This is done by performing a grid search with a maximum depth of 1 to 35 levels in alternating intervals of 2 and 3. The boosting algorithm uses 100 weak learners and a learning rate of 1.0. The second parameter optimisation is the number of classifiers. This is done by validating boosted classifiers with 25 to 250 weak learners. The learning rate of the boosting algorithm is set to 1.0 and the weak learners have a varying maximum depth which is based on the first optimisation. The last optimisation is the learning rate. The optimum learning rate will be decided by grid searching with learning rates of 0.1 to 1.5 in steps of 0.1. The grid search will be performed on boosted classifiers containing 100 weak learners with varying depth. After the optimal parameters have been selected the model will be tested on accuracy and classification time.

5.2 Effects of class imbalance

To get an idea of how the model behaves when interacting with many different base names and unbalanced and insufficient data the last experiment will be on the entire data set. The experiment will consists of two parts, each part using a different version of the entire dataset.

For the first experiment the data set used in section 5.1 will be extended with all the smaller classes. All samples of the smaller classes are added to training set. The final training set contains the first 50 to 150 samples of each class resulting in 51.947 training samples. The training set contains the remaining 70.456 samples from the classes that contain more than 150 samples. The goal of this experiment is to measure the effects of smaller

classes, a maximum of 150 instead of 200 samples, and to see what happens when the number of classes is increased from 141 to 447.

The second experiment uses the entire dataset, all 447 classes. From each class 80% of the samples is selected for training and 20% of the samples is selected for testing. This creates a training set of 97.836 samples and a test set containing 24.576 samples. By choosing this division the class imbalance from the dataset is maintained, but the larger classes have more material available for learning variance. The goal of this experiment is to measure the effects of increasing the class imbalance. AdaBoost has a natural mechanism for handling class imbalance, therefore it is expected that increasing the class imbalance does not negatively impact the performance of the classifier, however the extra training samples might positively impact the performance of the classifier.

The two training sets will be used to train a classifier which has the same configuration as the classifier used in section 5.1. First the boosted classifiers are validated using 10-fold cross validation. After that the classifiers are tested using the test set to measure accuracy and classification time.

6 Results

As discussed in chapter 5 there will be two different tests using the boosted classifier. Section 6.1 will discuss the potential of the boosted classifier and 6.2 will discuss the performance of the boosted classifier on the entire dataset.

6.1 Scaling

Figure 18 shows the influence of the maximum depth of the weak learners on classification accuracy. The results are similar to what has been concluded in the model selection stage. The main difference between the results of this experiment compared to the model selection stage is that the validation accuracy is lower and that deeper trees, maximum depth of 12-20, are preferred. The preference towards deeper weak learners can be explained by the increased complexity of the problem. By adding more base names and samples the number of choices that has to be considered increases.

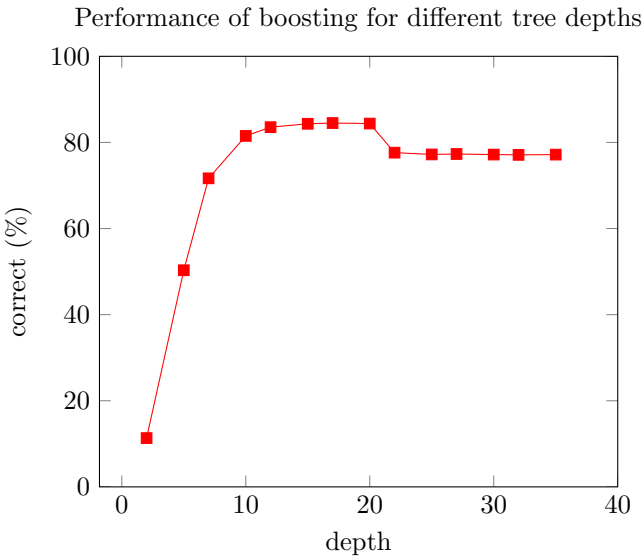
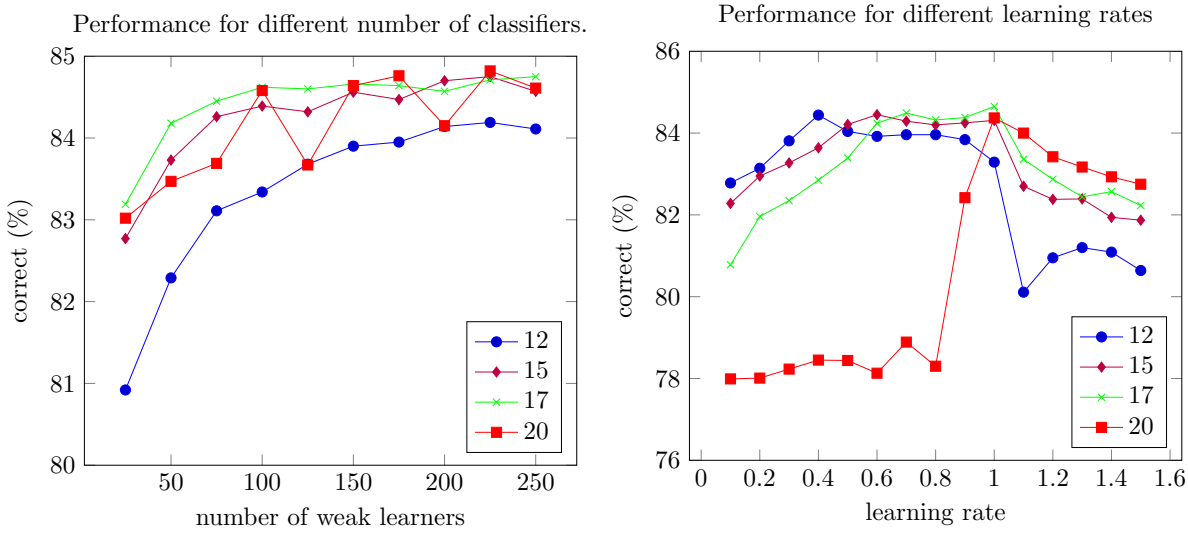


Figure 18: Performance of boosting for different tree depths. The performance is measured using 10-fold cross validation.



(a) Learning rate: 1.0.

(b) Number of weak learners: 100.

Figure 19: The performance of the boosted classifier for different number of weak learners and learning rates. The weak learners have different maximum levels of tree depth. The performance is measured using 10-fold cross validation.

Figure 19 shows the results of the second and third parameter optimisation step. The optimum number of weak learners is approximately 200 and the optimum learning rate is 1.0. In all cases the weak learner with a maximum depth of 17 was among the best performing classifiers.

The final configuration for this experiment was a boosted classifier consisting of 200 weak learners with a maximum depth of 17 and a learning rate of 1.0. The training time of the classifier with a training set of 22650 items was 49.76 seconds on a 64-bit Linux machine with a 3.1 GHz quad-core processor and 16 GB of RAM. The classification time was 69.82 seconds for 69192 samples. The validation accuracy was 84.72% and the accuracy on the test set was 82.62%. Figure 20 shows the fraction of the test set the classifier had wrong for each class. The errors are approximately equally distributed around the 18% mark. This suggests that the number of mistakes in bigger and smaller classes is also equally distributed.

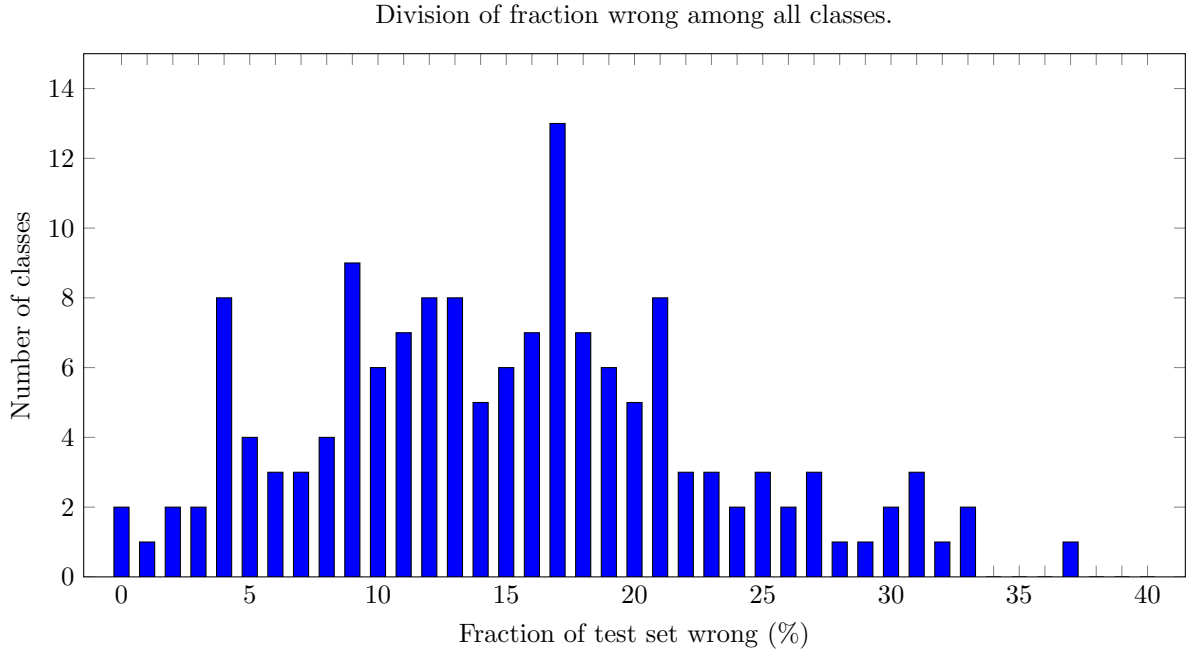


Figure 20: The distribution of errors over each of the 151 classes. 149 out of 151 classes created 1 or more errors.

6.2 Effects of class imbalance

For the first experiment with the full dataset the training set consists of 51947 training samples divided in 447 classes. Each of the classes contains a maximum of 150 samples. The rest of the samples, 70456 divided in 201 classes, are added to the training set.

The cross validation error of the boosted classifier, containing 200 weak learners with a maximum depth of 17 levels and a learning rate of 1.0, achieved a cross validation accuracy of 75.84%. This is lower than the scores in both sections 4.2 and 6.1. However considering the increase in classes, from 51 and 151 to 447, and the loss of samples, from at least 150 and 150 to at most 150, this is a reasonable score.

The final boosted classifier was trained with a training set of 51947 items in 370.09 seconds on a 64-bit Linux machine with a 3.1 GHz quad-core processor and 16 GB of RAM. The prediction time for 70456 samples was 206.67 seconds and the accuracy was 77.47%. Figure 21 shows how the fraction of the error is distributed across al 201 tested classes. The errors seem to be distributed around 15% mark, this is lower than expected. This is a result of the classifier making relatively more mistakes in the smaller classes compared to the mistakes made in larger classes. This combined with the lower overall accuracy suggest that the smaller classes require more samples.

For the second experiment the 80% of each class in the full dataset is reserved for training, the other 20% is used for testing. This results in a training set containing 97.836 samples and a test set containing 24.576 samples. The cross validation error of the boosted classifier, containing 200 weak learners with a maximum depth of 17 levels and a learning rate of 1.0, achieved a cross validation accuracy of 84.58%. This is comparable to the accuracy of the classifier in section 6.1.

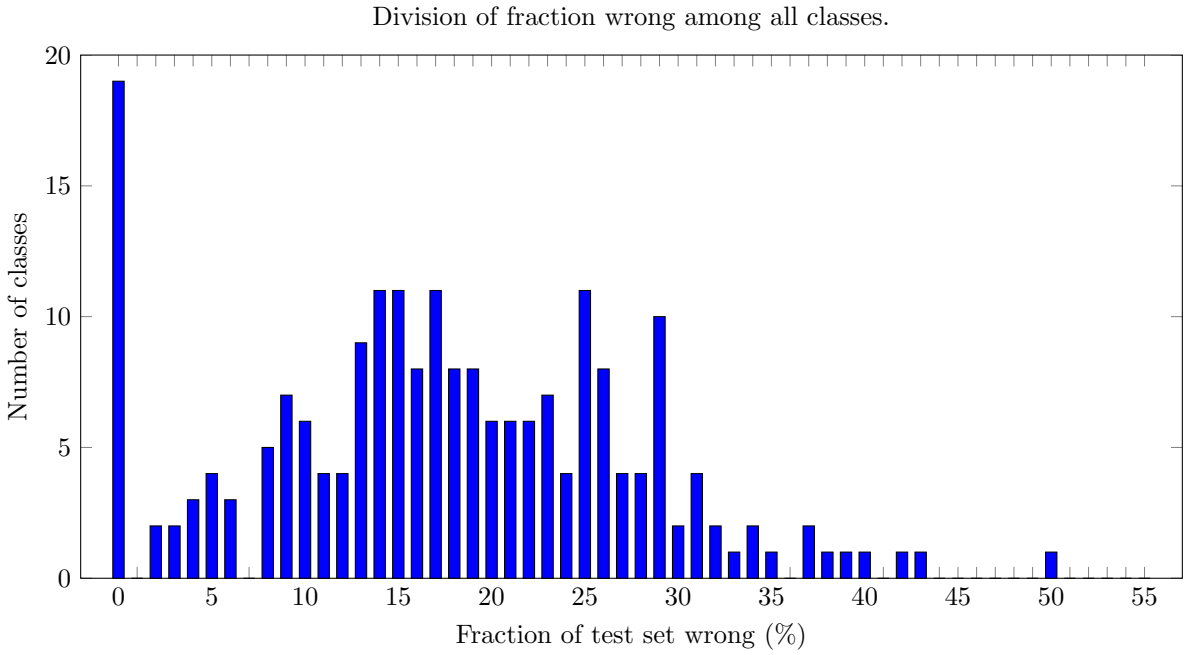


Figure 21: The distribution of errors over the 201 tested classes. 192 out of 201 classes created 1 or more errors.

The final boosted classifier was trained with a training set containing 97.836 items. Training the model took 525.98 seconds on a 64-bit Linux machine with a 3.1 GHz quad-core processor and 16 GB of RAM. The prediction time for 24.576 unknown samples was 71.08 seconds and the accuracy on this test set was 85.04%. Figure 22 shows the distribution of the error across all 447 classes. The error is distributed around the 15% mark. This suggests that the larger classes are equally distributed around the 15% mark, but slightly to the lower end, and that the smaller classes are at the higher end of the distribution. The distribution is similar to figure 21 but more extreme. This combined with a similar accuracy as the classifier in section 6.1 shows that class size has more effect on the performance of the boosted classifier than the number of classes.

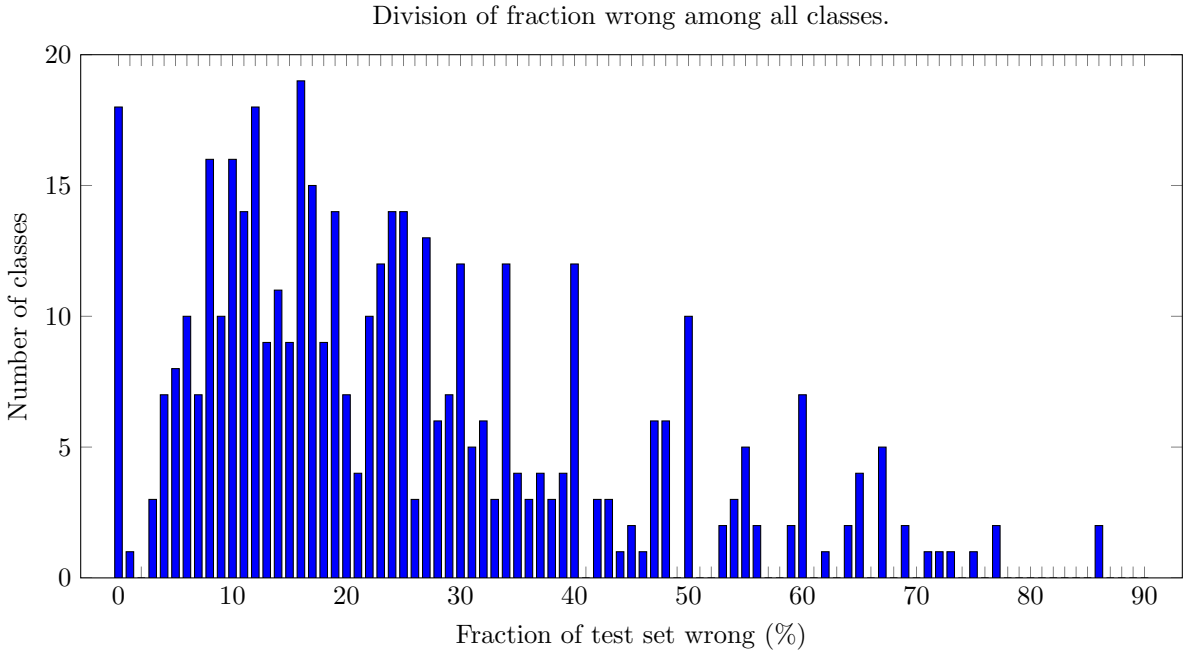


Figure 22: The distribution of errors over the 447 tested classes. 429 out of 447 classes created 1 or more errors.

7 Discussion and conclusion

In this thesis several methods for building a variant-base name model were evaluated. The model selection phase showed that boosted decision trees are superior to kNN and SVMs on pure accuracy and speed. As a result of this boosted decision trees were selected to build the final model. The final model is a boosted decision tree model with a learning rate of 1.0 and 200 decision trees with a maximum depth of 17 levels. The final model has a validation error of 84.58%. The accuracy on a test set with 24.576 samples divided in 447 classes is 85.04% and the classification speed is more than 300 samples per second.

This study shows that the main problem is the availability of data. When the training set contained 150-200 samples per class the validation accuracy was 94.21%. When the training set contained 150 samples per class the validation accuracy was 84.72%. When the available samples for training were lowered to a maximum of 150 samples per class and a minimum of 50 samples per class the validation accuracy was 75.84%. This clearly shows that performance is limited by the number of samples available for training. Having more data available is crucial for the performance of this model. This can be done by finding extra data or by creating dummy data.

This study also shows that AdaBoost is good at handling multiclass classification problems with class imbalance. Increasing the number of classes from 151 to 447, while increasing the class imbalance from 150 - 250 training samples per class to 50 - 3520 samples per class, has no significant influence on the overall accuracy of the classifier. The only measurable effect is that the number of classes that have more than 25% of the test set incorrect increased dramatically. However this is mainly caused by the addition of many small classes with insufficient samples to learn all variation within the class.

In future work the most important part is to find or, reliably, generate more data. Extra data is required to improve the accuracy of the model and might get the accuracy to 95%. More accuracy can also be achieved by studying the errors made by the classifier more closely. Are errors mainly made on very rare names or are the errors mainly made on very common names. What kind of mistakes are hard, insertion errors, deletion errors or substitutions. What is the effect of manually creating those mistakes to the training material.

Improving kNN with Levensthein distance might also be an improvement of the current model. The main reason boosted decision trees are preferred over kNN with Levenshtein distance are the computational demands for kNN with Levenshtein distance. One of the main issues with kNN is that the time complexity depends on the number of samples and the time required for calculating the edit distance. The time complexity of kNN can be reduced by using optimised data structures and by optimising the edit distance algorithm. If the computational demands of kNN with Levenshtein edit distance can be lowered to a reasonable speed for record linkage than kNN might be a fierce competitor.

References

[1] Yaser Abu-Mostafa, Malik Magdon-Ismail and Hsuan-Tien Lin. *Learning from data*. Amlbook.com, 2012. Chap. 6.

[2] Yaser Abu-Mostafa, Malik Magdon-Ismail and Hsuan-Tien Lin. *Learning from data*. Amlbook.com, 2012. Chap. 8.

[3] Erin L Allwein, Robert E Schapire and Yora1m Singer. "Reducing multiclass to binary: A unifying approach for margin classifiers". In: *Journal of machine learning research* 1.Dec (2000), pp. 113–141.

[4] Mohamed Aly. "Survey on multiclass classification methods". In: *Neural Netw* (2005), pp. 1–9.

[5] Gerrit Bloothooft. *Persoonsnamen in de 19e eeuw*. 2015.

[6] Gerrit Bloothooft and Marijn Schraagen. "Learning Name Variants from Inexact High-Confidence Matches". In: *Population Reconstruction*. Springer, 2015, pp. 61–83.

[7] Sungmoon Cheong, Sang Hoon Oh and Soo-Young Lee. "Support vector machines with binary tree architecture for multi-class classification". In: *Neural Information Processing-Letters and Reviews* 2.3 (2004), pp. 47–51.

[8] Vladimir Cherkassky and Nikolaos Vassilas. "Back-propagation networks for spelling correction". In: *Neural Net* 1.3 (1989), pp. 166–173.

[9] Vladimir Cherkassky and Nikolaos Vassilas. "Performance of back propagation networks for associative database retrieval". In: *Neural Networks, 1989. IJCNN., International Joint Conference on*. IEEE. 1989, pp. 77–84.

[10] Koby Crammer and Yoram Singer. "On the algorithmic implementation of multiclass kernel-based vector machines". In: *Journal of machine learning research* 2.Dec (2001), pp. 265–292.

[11] Gao Daqi, Li Chunxia and Yang Yunfan. "Task decomposition and modular single-hidden-layer perceptron classifiers for multi-class learning problems". In: *Pattern Recognition* 40.8 (2007), pp. 2226–2236.

[12] Thomas G. Dietterich and Ghulum Bakiri. "Solving multiclass learning problems via error-correcting output codes". In: *Journal of artificial intelligence research* 2 (1995), pp. 263–286.

[13] Yoav Freund and Robert E Schapire. "A decision-theoretic generalization of on-line learning and an appliction to boosting". In: *Journal of Computer and System Sciences* 55.1 (1997), pp. 119–139.

[14] Guang-Bin Huang et al. "Extreme learning machine for regression and multiclass classification". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 42.2 (2012), pp. 513–529.

[15] Daniel Jurafsky and James H Martin. *Speech and language processing*. 2000. Chap. 3.

[16] Sotiris B Kotsiantis, I Zaharakis and P Pintelas. *Supervised machine learning: A review of classification techniques*. 2007.

[17] Karen Kukich. "Techniques for automatically correcting words in text". In: *ACM Computing Surveys* 24.4 (1992), pp. 406–410.

[18] Mark Lewellen. "Neural Network Recognition of Spelling Errors". In: *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics - Volume 2*. Association for Computational Linguistics. 1998, pp. 1490–1492.

[19] Guobin Ou and Yi Lu Murphey. "Multi-class pattern classification using neural networks". In: *Pattern Recognition* 40.1 (2007), pp. 4–18.

[20] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[21] Robert E Schapire. "A brief introduction to boosting". In: *Ijcai*. Vol. 99. 1999, pp. 1401–1406.

[22] Marijn Schraagen. "Aspects of Record Linkage". PhD thesis. Leiden: Universiteit Leiden, 2014. Chap. 2, pp. 22–26.

[23] Yanmin Sun et al. "Cost-sensitive boosting for classification of imbalanced data". In: *Pattern Recognition* 40.12 (2007), pp. 3358–3378.