PREDICTING COCAINE RELATIONS USING MACHINE LEARNING TECHNIQUES

ANDREAS WINTHER LYKKE

20106183

Master’s Thesis
Department of Computer Science
Science & Technology
Aarhus University

Supervisor: Christian Nørgaard Storm Pedersen

June 2016
ABSTRACT

To determine whether cocaine samples relate to the same manufacturer, has shown to be a valuable knowledge that can be used in different aspects of crime investigation. It can, for instance, be used to determine whether a person might be connected to multiple seizures of cocaine or to get a map of how the cocaine market is distributed. Due to this there is a motivation for building a classifier that can help predicting this relationship.

In this thesis it is examined how to construct such classifiers by using various types of machine learning. First the thesis will focus on building traditional classifiers by using classical algorithms within supervised learning. Next it investigates whether metric learning can be used to create a better representation of the underlying problem, in order to improve the classification.

Aside from that, the thesis discusses how to convert a categorical classifier into a probabilistic classifier. The reason for examining this, is that the classifiers should be able to distinguish a strong prediction from a weak one, in order to be feasible in court. In extension, different calibration methods are investigated, in order to assure that the classifiers can provide a reasonable probability estimate.

Finally the thesis focus on how to take advantage of unlabeled samples by using a learning paradigm known as PU-learning. The reason for exploring this, is that it often can be difficult to obtain cocaine samples where the manufacturer is known. Also the data set that has formed the basis of this work, essentially contains a lot of samples that are unlabeled. Therefore there is a motivation for studying this paradigm.

The contribution of the thesis is an in-depth analysis of a problem that is rather unstudied in computer science.
ACKNOWLEDGMENTS

First of all thanks to Christian Storm for general discussions and for inspiring me with this rather alternative subject for my master thesis. Also, thanks to Palle Villesen for giving the necessary background knowledge and providing the data set that has formed the basis of this work. Last but not least thanks to Anders Høst Mikkelsen and Magnus Vinther for proofreading this thesis and providing valuable feedback.
6.3 Randomized searching ............................ 41

7 CLASSIFICATION EXPERIMENTS 43
7.1 Goal ................................................. 44
7.2 Expectations ....................................... 44
7.3 Results : alkaloids & solvents ................. 45
7.4 Discarded experiments ............................ 45
7.5 Results : alkaloids ................................. 46
   7.5.1 Cosine distance (R₁) ......................... 47
   7.5.2 Multiple distances (R₂) ...................... 48
   7.5.3 Similarity vector (R₃) ....................... 48
   7.5.4 Precision and recall ......................... 49
   7.5.5 Dot plot .................................. 50
   7.5.6 Class weighting .............................. 51
7.6 Validation score .................................. 52
   7.6.1 kNN ...................................... 52
   7.6.2 SVM RBF .................................. 53
   7.6.3 Logistic Regression ......................... 54

8 METRIC LEARNING 56
8.1 Metric learning in a nutshell .................... 57
8.2 Similarity transformation ......................... 57
8.3 Neighbourhood components analysis (NCA) ..... 58
   8.3.1 Stocastic neighbourhood assignment ...... 59
   8.3.2 Overview of NCA ............................ 60
   8.3.3 Strengths and weaknesses of NCA ....... 61
8.4 NCA implementation ............................... 61
   8.4.1 Initialization ................................ 62
   8.4.2 Numerical problems ......................... 62
   8.4.3 Performance ................................ 63
   8.4.4 Computational cost of the gradient ...... 64
   8.4.5 Verify NCA implementation ................. 64
8.5 Classification experiment with NCA ............. 66

9 PROBABILITY ESTIMATION 68
9.1 calibration ....................................... 68
9.2 Reliability diagram ................................ 70
9.3 Platt Scaling .................................... 70
9.4 Isotonic regression ................................ 71
9.5 Probability Experiment ........................... 72
9.6 results ............................................. 73

10 INLIERS AND OUTLIERS 76
10.1 Local outlier factor (LOF) ....................... 77
10.2 Applying LOF .................................... 78
10.3 Removing inliers .................................. 80

11 UNLABELED DATA 82
11.1 PU-learning ...................................... 83
   11.1.1 Selected completely at random assumption .... 83
   11.1.2 Estimating c ................................ 84
11.2 Implementation ............................................ 85
11.3 Classify with PU-learning .......................... 85

12 CONCLUSION ...................................................... 88
12.1 Future work .................................................. 90
12.1.1 Siamese neural network ......................... 90
12.1.2 Local based metric learning ...................... 91

A APPENDIX ............................................................ 93
A.1 Logistic regression ....................................... 93
A.2 PAV example ................................................ 94
A.3 Precision recall curves ............................... 95

BIBLIOGRAPHY .......................................................... 97
We were somewhere around Barstow on the edge of the desert when the drugs began to take hold. I remember saying something like ‘I feel a bit lightheaded; maybe you should drive...’ And suddenly there was a terrible roar all around us and the sky was full of what looked like huge bats, all swooping and screeching and diving around the car, which was going about a hundred miles an hour with the top down to Las Vegas. And a voice was screaming: ‘Holy Jesus! What are these goddamn animals?’

– Hunter S. Thompson
INTRODUCTION

Buy the ticket, take the ride.
— Raoul Duke, *Fear and Loathing in Las Vegas*

In 2014 a Ph.D. project was started in collaboration between Department of Forensic Medicine and Bioinformatics Research Center (BiRC) at Aarhus University. The goal of the project was to study how profile analysis can be used to determine whether two cocaine samples originate from the same manufacturer.

A profile is very comparable to a DNA sequence in that it constitutes a fingerprint of the drugs. By comparing these fingerprints it can be revealed if the drugs originate from the same manufacturer.

This insight has shown to play a valuable role in different aspects of police investigation. For instance, if two seizures of cocaine are found in the same neighborhood, and the cocaine originates from the same manufacturer, then it’s very likely that it’s the same person that is distributing both seizures. So, if a person is found to have a connection to one of the seizures, then that same person can be suspected to have a connection to the other cocaine seizure. In combination with techniques like wiretapping, etc., it is possible to convict the suspected person for both seizures. On the other hand, if the cocaine seizures originate from each manufacturer, then it is less likely that both types of cocaine are distributed by the same person.

Aside from that, the profiles can be used to determine

- how the cocaine market is connected.
- how the cocaine is distributed geographically.
- how the rate of turnover is for a particular type of cocaine.

So, in the long term it can be used to reveal the story of how the drug market is evolving. Due to this, there is an interest in methods that can compare these profiles, in order to stamp out the illegal drug market.

To determine whether two samples originate from the same manufacturer, different data science techniques have been applied at BiRC, to produce a model that statistically can determine their relationship. In this thesis we will try to extent this work and apply methods that have not been applied on this problem before. More concretely, we will be using different kinds of machine learning techniques.
1.1 THE COCAINE STRUCTURE

Cocaine is a drug that within chemistry is grouped as a weakly alkaloid, which means its a relatively natural chemical compound that comes from the leaves of the coca plant. In order to produce cocaine, the leaves are first gathered into a large barrel and the cocaine is extracted by using an aqueous solution such as petroleum or gasoline. Next, the solution is drained away and an intermediate product is left, which contains about $60 - 80\%$ cocaine and some other coca alkaloids [24]. After that, the cocaine is dissolved using solvents like acetone that increases the amount of cocaine. These solvents are thought of as chemical impurities that subsequently are removed by putting the cocaine under pressure. However, some of these solvents remain in the drug, and together with the alkaloids they form a strong fingerprint, which can be used to reveal the manufacturer of the cocaine.

1.2 EXTRACTING THE FINGERPRINT

During the Ph.D. study, a larger amount of drug samples have been collected from different police districts. Subsequently the samples have been exposed to things like sunlight, low temperature etc., in order to simulate different real life scenarios that the drugs undergo. All of these scenarios affect the drugs in some way and create more variability in the samples.

After that, the confiscated samples have been analyzed by using a method known as gas chromatography–mass spectrometry (GC-MS). The GC-MS is able to identify the different chemicals that exist within each sample. From this the amount of the different alkaloids and solvents have been extracted and encoded into a profile, as seen in Figure 1. Based on the distribution of the profiles it is possible to uniquely identify the drug samples.

![Figure 1: A profile illustrating the different alkaloids and solvents in a cocaine sample.](image)
It is profiles like the one in Figure 1 that have been constructed at Department of Forensic Medicine and captured into a data set. In the next section, we will see how this data set is structured.

1.3 THE DATA SET

The data set consists in total of 309 samples where each sample has 29 features. The first 10 features represent the alkaloids and the last 19 represent the solvents. Each sample has further been associated to one of the following groups: A, B, C, D, E and O. The first five groups consist each of 36 samples and their label represents the manufacturer that the samples originate from. Therefore, if two samples have a similar label, it indicates they are related to each other, and are therefore said to be linked. Conversely, when two samples are assigned with different labels, they have no relation to each other and are therefore said to be unlinked.

The last group, denoted as O, consists in total of 129 samples, and represents samples where the manufacturer essentially is unknown. In other words, it is not known whether they have a relation to other samples in the data set. In spite of this, it is highly believed that they are unlinked from every other samples, even though they could have a relation. The general motivation for including this group of samples is to put more stress on the classification. Thus if a classifier can distinguish linked from unlinked, when this group is included, it can be thought of as being very reliable.

From this, the aim is to determine the pair of samples that has a relation to each other. So what we are dealing with is a binary classification problem, where the goal is to predict the samples that are linked and the samples that are unlinked.

Throughout the thesis, we will refer to the data set above as $D_{\text{org}}$, since it represents the samples in their original form. The complete data set can be downloaded from 1 in CSV file format.

1.4 REQUIREMENT SPECIFICATIONS

First we will establish the different requirements that our classifier should fulfill in order to be feasible. The requirements that have been requested, can be captured into the following specifications:

CLASSIFY SIMILARITY

It is necessary that the classifier predicts linked and unlinked based on whether two profiles are similar distributed or not. The reason for this is due to the fact that every cocaine manufacturer is not known beforehand. Thus it should be able to pre-

1 http://cs.au.dk/~awl/thesis/data.zip
dict two A samples as linked, even though it never have been presented to any A samples before.

RELIABILITY MEASURE
Aside from predicting linked or unlinked, the classifier should be able to output a probability estimate of how reliable that the classification is. Having this enables us to differentiate a strong prediction from a weak prediction, which is necessary when the classifier is used for crime investigation. So if the probability estimate is below a certain threshold, it should not be thought of as being reliable, in order to prevent miscarriage of justice.

At BiRC they have achieved remarkable results by using both alkaloids and solvents and by constructing a model using Logistic regression (section 5.2). Unfortunately, the model becomes less precise, when only relying on the alkaloids. After all, the alkaloids are considered to be the most reliable substances by the injury. Therefore, there is a natural motivation for creating a model that is as accurate as possible by solely looking at these.

During this thesis we will try to build a feasible classifier by using different kinds of supervised machine learning techniques. The aim is to get as close as possible to the performance that they have got at BiRC, but only by using the alkaloids.

1.5 THESIS OUTLINE

The thesis is structured into the following chapters:

SUPERVISED MACHINE LEARNING
Introduces the concept of supervised machine learning, in order to make the basis of it clear.

DATA REPRESENTATIONS
Discusses different representations that can be used to reflect similarities between two cocaine profiles.

TRAINING AND EVALUATION
Explains concepts like stratified sampling and a variant of k-fold cross validation that has been used for training and testing the classifiers. In extension, the chapter introduces the $F_1$ score, which has been used as an evaluation metric for evaluating the classifiers.

LEARNING ALGORITHMS
Presents the learning algorithms that have been used to construct a feasible classifier. These includes Logistic regression, Support vector machines and k-nearest neighbor. The chapter explains the theory of how they work and discuss how they have been utilized through scikit-learn.
Tuning Hyperparameters
Discuss strategies for tuning the parameters of the learning algorithms.

Classification Experiments
Walks through the different classification experiments, where the main result can be found in Figure 17.

Metric Learning
Investigates whether the classification can be improved by enhancing the representation of linked and unlinked samples through a learning paradigm known as metric learning.

Probability Estimation
Explains how SVM and kNN classifiers can be turned into probabilistic classifiers in order to fulfill the second requirement in section 1.4. Additionally the chapter investigates different calibration techniques for improving the probability estimates.

Inliers and Outliers
Provides an cluster analysis of the samples that appears as respectively inliers and outliers. Based on this, we can get an idea about which samples that are affecting the overall classification.

Unlabeled Data
Investigates whether the classification can be improved by treating the unlinked samples as unlabeled. The reason for this, is that some of the samples from group O potentially could be linked although they are said to be unlinked. In other words, the underlying assumption about them could damage the classifiers.

1.6 Implementation
The software that has been implemented during this thesis can be found at \(^2\). It requires python 2.7 and depends on the following packages: scikit-learn, scipy, numpy and pandas.

\(^2\) http://cs.au.dk/~awl/thesis/src.zip
Before discussing how classification can be realized, we will shortly establish the concept of supervised learning, in order to make the basis of it clear. A lot of the notions that are used during this section are originally adapted from [33]. However, in [33] they establish the concept in a more rigorous manner, which we won’t be doing here. Instead, the reader is highly encouraged to consult [33] for a more detailed description.

In supervised learning we are given a data set \( D \) consisting of \( n \) labeled data samples on the form \( \{ (x_i, y_i) \}_{i=1}^{n} \), where \( x_i \in \mathbb{R}^d \) represents an actual data sample and \( y_i \) represents the associated label. So in our case, \( x_i \) will represent a pair of cocaine samples, and \( y_i \) will be a binary label that reflects whether the samples are linked or unlinked. Sometimes we will use \( X \) to denote the \( x_i \)’s encoded as rows in a matrix, and \( Y \) to denote the \( y_i \)’s encoded as a vector.

From this, the aim of a supervised learning algorithm is to find a function \( h \) that is good at predicting the label \( y_i \) for a given input \( x_i \). If we say that \( X \) represents the input space of our data samples and \( Y \) represents the output space of the output values, then we think there exists a true unknown function \( f : X \rightarrow Y \). The goal is therefore to find \( h \), such that it approximates \( f \) as much as possible.

The function \( h \) is generally referred to as a hypothesis or a model. However, since \( h \) in our case is specifically used for classification we will be referring to it as a classifier. The form that \( h \) take on can be anything that ranges from a linear function to a decision tree, but is essentially not relevant to us. What is relevant is to us, is that \( h \) can be used to make predictions on data that it has not seen before. In other words, we want it to generalize well to new data. So, how should we construct \( h \)?

The general procedure, within supervised learning is to feed a lot of labeled data samples to a learning algorithm. The learning algorithm then tries to construct \( h \) based on the relationship that exists between the features of the input samples and their associated labels. In other words, the learning algorithm is training \( h \)’s ability to make predictions from the labeled samples. For that reason, this phase is also what is known as the training session.

After the classifier has been constructed, we want to evaluate its performance, in order to see how it generalizes to new data. To do
this, we need to evaluate $h$ on samples that hasn’t been used during the training session. If we simply just reused the samples from the training session, then the learning algorithm could just construct $h$ such that it is very good at memorizing the data. However, that doesn’t reflect what we are aiming for, since the goal is to be able to make good predictions on unknown data.

To evaluate this, we will take our data set $D$, and split it into both a training set $D_{\text{train}}$ and a testing set $D_{\text{test}}$. The idea is then to first construct $h$ from $D_{\text{train}}$ and then evaluate its performance by making predictions on $D_{\text{test}}$. To measure how good the performance is, an error function $E$ is used, which simply summarizes how well the predictions agree with the labels of the samples. For instance, when working with classification, a common choice is usually

$$E(h) = \frac{1}{n} \sum_{i=1}^{n} I[h(x_i) = y_i], \quad (1)$$

where $I$ is an indicator function that returns 1 when $h(x_i)$ and $y_i$ agree and 0 when they disagree. In other words, it describes the average agreement between our predictions and the true labels. When taking $E$ on $D_{\text{test}}$, we achieve what is more specifically known as the test accuracy, which reflects how accurate our model is on the samples that we have in $D_{\text{test}}$. So, if we for instance have constructed two different classifiers, then we should pick the final classifier, based on the one that achieves the highest test accuracy. Sometimes, the error function is also applied on the $D_{\text{train}}$, yielding what is known as the training accuracy, which can be used to tell how well the learning algorithm benefits from the available training data.

Both the training phase and the evaluation phase is highly dependent on the number of samples, therefore we essentially want to have as many samples as possible in both sets. However, it is not well-defined how to do this split. But a usual heuristic is to put 80\% of the samples in $D_{\text{train}}$, and 20\% of our samples in $D_{\text{test}}$, since we want the learning algorithm to learn as much as possible.

When a final classifier have been evaluated on the $D_{\text{test}}$, we can train a new classifier on the whole data set, in order to let it benefit as much as possible from the samples. After that we should have a classifier that can generalize as much as possible, based on the available data we have. The steps described above, are some of the crucial parts of supervised learning and are usually performed independent of the learning algorithm we choose.

However, other learning paradigms exist, which probably can be used as well to determine the problem of predicting cocaine relations. For instance, in contrast to supervised learning, there also exists unsupervised learning, where the actual data labels are unknown to us. In such a paradigm the learning is often based on some sort of similarity metric that is used to learn some underlying pattern of the data.
Withal, since the labels of our data is known beforehand, and since we have a lot of samples available, supervised learning is an obvious choice.
Before the training session can be initiated, we need to assure that the representation of our input is reflecting the underlying problem well. If the representation is good, it should help the learning algorithm to benefit from the samples. To emphasize the importance of this, the following quote by Scott Locklin has been found [20]:

"Much of the success of machine learning is actually success in engineering features that a learner can understand."

So how should we construct a representation that reflects the underlying problem well? If we look at \( D_{org} \), we have 309 samples on the form \((x_i, y_i)\), where \( x_i \) represents the \( i \)'th cocaine sample and \( y_i \) represents the manufacturer that \( x_i \) originates from. Based on this we want to pair the samples together and construct a data set \( D \), where we easily can distinguish linked from unlinked samples.

Remember from section \( 1.4 \) that we should discriminate between linked and unlinked based on how the profiles are distributed. That is, if two profiles are similar distributed, then the drugs can be thought of as being very equal and should therefore be labeled as linked. In other words, we should not care about whether one drug has twice the amount of each chemical, but more about whether their relative frequency of chemicals are the same. Therefore we need to create a representation that are good at reflecting this similarity.

How to do this is unfortunately not a formal topic. Therefore we have devoted this chapter to discuss some different candidates that can be used. Based on this we will investigate empirically in chapter \( 7 \), the one that gives the best classification.

Some of the representations that we introduce in this chapter, are well establish representations for determine similarities between two inputs. For instance in [12], some of the representations have been used to determine object similarities in computer vision. However, before going into the different representations, we will first discuss a pre-processing step that has shown to be necessary.

### 3.1 Prepossessing

When solving a machine learning problem, it is always a good idea to visualize the available data set, in order to get an intuition of how the
samples are structured. A variety of techniques exists to do this. One strategy is to project the data into a lower dimensional subspace, by using a technique such as Principal component analysis (PCA). The overall idea behind PCA is to create a linear transformation that can transform the data into a space where the first axis captures the most variance, the second axis the second most and so on. By reducing the transformation into the two components that capture the most variance, we can get a two-dimensional plot as seen in Figure 2.

![Figure 2: PCA plots after applying the log-transform.](image)

To the right we have a PCA plot where the O samples have been excluded from the data set, and to the left we have a PCA plot where the O samples have been included. In both plots, we can clearly see that the samples are wildly spread out, so it becomes difficult to tell how they are related to each other. Therefore, we need a way of transforming the data, such that samples from the same group get squeezed together. Also if we look at Figure 2a, we can see that there exists an outlier in the lower left corner of the diagram, such that the scale of the plot is increased. This outlier could cause a bad influence on the learning algorithm, and should therefore be handled in some way.

One way to address this, is by performing a log-transform of the samples. For instance the logarithm is widely used in statistics where a skewed distribution needs to be transformed into a normal distribution. The motivation for applying this function is that it helps to stabilize the variance of the samples. In Figure 3a we can see a PCA plot after doing this.
Figure 3: PCA plots after applying the log-transform

From the plots we can clearly see that the log-transform reduces the variance, such that the relationship of the samples becomes more clear. Also, the outlier that was present before has now transformed into the same scale as the other samples. Therefore the logarithm removes the need for performing outlier detection as a pre-processing step.

In the step above we have applied the variant of the logarithm known as the log-plus-one transform, \( \logp_1(x) = \log(1 + x) \), which is an easy way to handle zero values in the data set. Additionally, the plus one should not have any practical effect on the samples, since the features either has a rather high value or is 0.

The log-transform is just one of many transformations that can be used to reduce the variance. For instance, the fourth root transform can often be used in many cases as well as the log-transform. Which one to choose is not a well-defined area, but previous work at BiRC has shown that the log transform imposes a good intermediate representation, which slightly improves the classification.

Of course the plots should always be taken with a grain of salt, as they only reflects a small side of the full feature space. Nonetheless, the PCA plots can be used as a guidance to investigate how the data set is structured and whether there is a need for transforming it. As we will see, applying the logarithm is a crucial step in order to create an intermediate representation that can be used to reflect similarity between the profiles. In the subsequent sections, we will look more at this.

3.2 Distance Based Representation

So how can we quickly determined whether two profiles are similar distributed? One way to do this is by representing the profiles as
vectors within a vector space, and then simply look at the direction that the vectors are pointing. The observation is: if the vectors coincide, it will indicate that their chemical distributions are very similar. In other words, the whole problem of drug similarity simply boils down to asking whether their vector representations are correlated or not. This question can easily be answered by utilizing some distance function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, that measures the degree of closeness between the two vectors. By doing so, we simply reduce the full feature representation to a single score that reflects similarity.

So which distance functions should we use? A lot of distance functions exist, that can be used to measure this. Some of the most commonly used are the following

- Cosine distance
- Euclidian distance
- Manhattan distance
- Pearson distance

For all the distance functions above, a small value will reflect that the samples are very similar and a large value, will reflect that the samples are dissimilar. Additionally the Euclidian and the Manhattan distance, requires the input vectors to be normalized before they are applied. If we do not normalize the vectors, they will not capture how far the chemical distributions are from each other. Besides that they all fulfill the mathematical conditions of being a metric, which especially means that

$$d(x, y) = d(y, x)$$
$$d(x, y) = 0 \iff x = y$$

Due to this, we end up with a total of $n(n - 1)/2$ unique distances.

In Figure 4, we can see a heat map that illustrates the cosine distances between the samples of our data set. Each cell in the map corresponds to a pair of samples, where the color reflects the distance between them. The darker the color, the larger is the distance.

Figure 4: Heat-map of the pairwise cosine distance.
From here we can clearly see that samples from the same group are related to each other, as the colors of the diagonal are very bright.

### 3.3 Transforming the Distance Space

To see how crucial the log transformation is as a pre-processing step, we will investigate the effect it has on the cosine representation. Below in Figure 5 we see a histogram of how the distances are distributed, when the log transform is omitted as a pre-processing step.

![Figure 5: Without log-transform.](image)

If we look at the unlinked samples, before the transformation is applied, we can see that the distances are widely spread out, over the whole distance space. As a consequence, a lot of the unlinked distances will coincide with a lot of the linked distances, and make it almost impossible to distinguish linked from unlinked.

![Figure 6: With log-transform.](image)
If we instead apply the log-transform before the distances are calculated, we get a situation as in Figure 6, where the distributions becomes more squeezed. The result is that linked and unlinked distances will become better clustered, such that we clearly can distinguish between the two classes of samples. Therefore the log-transform is a crucial step to perform, before we create the distance based representation.

3.4 **MULTI-DISTANCE REPRESENTATION**

The disadvantage of the distance based representation in section 3.2 is that we lose a lot of features, since we simply reduce the input to a single value. Alternatively, we could use a multi-distance representation, where we combine several distance measures together into a vector. The hope is to get a representation that contains more information that the learning algorithm can benefit from. Unfortunately, if the distances are highly correlated, then this representation would not contribute to more information, anymore than a single distance measure would. Also, for some learning algorithms, like Naive Bayes, it could in fact cause a negative effect, since it explicitly assumes features to be independent [9]. But since we will not be using any algorithm that makes this assumption, it should be reasonable to test.

Additionally, if the learning algorithm is highly dependent on the number of features, then this approach will reduce the training time. However, since we only are going to combine a relatively few number of distance measures, then this should not be an issue.

3.5 **ATTRIBUTE BASED REPRESENTATION**

Aside from the distance based representations, we will also construct a representation by just concatenating two samples into a single vector. Doing so, we should get a representation where each sample is a vector consisting of 20 features in total, since each sample consists of 10 alkaloids.

The idea is to let the learning algorithm learn the underlying pattern that is present when we do this. For instance, when a pair of samples are linked, there should be some sort of symmetry present that the learning algorithm should recognize. However, before concatenating the samples, we need to assure that their features are represented on the same level of scale. This can for instance be done by normalizing the input vectors, before they are concatenated. If this is not done, the representation could end up being very unstable, as the learning algorithm would need to learn similarities on different levels of scale.
The advantage of this representation is that it is highly informative, as it contains a lot features. Additionally, the number of samples will be equal to $n^2$, since concatenation is not a symmetric operation.

### 3.6 Similarity Vector

The last representation we will introduce is a representation where a similarity score is computed between the features of two samples. So, if we are given two samples, $x_i$ and $x_j$, then we will compare the $t$’th feature of $x_i$ against the $t$’th feature of $x_j$. The way we will do this is by taking two features, say $x_{it}$ and $x_{jt}$, and then compute

$$\frac{|x_{it} - x_{jt}|}{x_{it} + x_{jt}}, \text{ for } t = 1, \ldots, 10$$

If we normalize $x_i$ and $x_j$ before the calculation is applied, we will achieve a value that represents the percentage difference between the two features. Furthermore, if $x_{it} + x_{jt} = 0$, we will define the expression to be 0. So for very similar samples, this approach will produce a vector that contains a lot of values that are close to zero and is therefore characterized by having a rather small magnitude. The hope is to get a representation that is very robust at capturing similarities and at the same time very informative, as it contains more than just a single distance value. Since the operation above is both symmetric and gives the zero vector when two vectors are equal, we get a total of $n(n-1)/2$ unique elements, like with the distance based representations.
After having constructed the proper representation of our data set, we need to create the training set $D_{\text{train}}$ and the test set $D_{\text{test}}$. However, the way we construct these sets can have a huge impact on both training and testing, so we need to be aware of how we do this. Therefore we have devoted this chapter to discuss how to split $D$ properly.

To begin with we will first look at some well establish strategies, such as stratified sampling and cross validation. Next, we will investigate the problems with using the test accuracy (equation 1, chapter 2) as an evaluating metric for binary classification and try to discuss different alternatives.

### 4.1 Stratified Sampling

An easy way to construct respectively $D_{\text{train}}$ and $D_{\text{test}}$ is by randomly putting 80% of the samples in $D_{\text{train}}$ and let the last 20% go into $D_{\text{test}}$. Unfortunately, by chance, it could very well be that $D_{\text{train}}$ or $D_{\text{test}}$ won’t be representing the classes of $D$ properly. For instance linked and unlinked samples won’t be represented correctly, or if we are exceptionally unlucky, then one of the classes could be completely excluded from one of the sets, which of cause will have fatal consequences. Therefore we need to assure that both linked and unlinked samples are represented correctly. That is, the distribution of the linked and unlinked samples in $D_{\text{train}}$ and $D_{\text{test}}$ follow the same distribution as the full data set.

The above scenario is especially problematic in cases where the classes are very unbalanced, which is also known as skewed classes. In such cases, the chance that the classes won’t be correctly represented in $D_{\text{train}}$ and $D_{\text{test}}$ would be higher than if they where equally balanced.

Since we have a lot more unlinked than linked samples, we could indeed be affected by these problems. Therefore, there is a motivation for ensuring against this.

One way to do this, is by using techniques such as stratified sampling. In stratified sampling the idea is to divide the data set into groups, called stratas, where each strata represent samples of each class. So in our case we could have two stratas for respectively linked and unlinked samples. Next, we will construct $D_{\text{train}}$ and $D_{\text{test}}$ by
sampling from the two stratas in such a way, that the frequency of both linked and unlinked samples are preserved. Therefore, this strategy will ensure that both $D_{\text{train}}$ and $D_{\text{test}}$ will end up with the same distributions of the two classes of samples, such that we can prevent the problems above.

We will later in the thesis discuss different scenarios where this will be applicable. For instance will we use it to create a small representative set of $D_{\text{org}}$, where every group is correctly represented.

4.2 **K-Fold Cross Validation**

Usually we want the learning algorithm to benefit as much as possible from our samples, which can cause $D_{\text{test}}$ to be rather small. Consequently the testing error will be less precise. A common way of dealing with this, is by using techniques such as k-fold cross validation.

In k-fold cross validation the data set is divided into $k$ subsets, which are referred to as folds. The idea is then to repeatedly use one of the folds for testing and use the other $k-1$ folds for training. This procedure is then continued until every fold has been used as a test set. Next, the $k$ testing accuracies can be averaged together, in order to reduce the bias that is associated with the split.

Setting $k$ to a large value, means less bias towards the true expected error. So if the data set is very small, we could set $k = n$, which is also known as Leave-One-Out cross validation. However, for large data sets, this would produce a rather slow training time since training becomes $k$ times slower than the standard train-test-split approach. Consequently, $k$ needs to be set to a reasonable value, that fits the size of $D$. Additionally, in cases where the data set is highly skewed, the strategy should be mixed with stratified sampling, such that each fold gets the same class distributions.

K-fold cross validation is a rather common strategy that is often used for training and testing a classifier. However, we will not be using this strategy directly, but instead use a modified version, which we will explain in the next section.

4.3 **Leave-Group-Out Cross Validation**

In section 1.4 we presented the different requirements that should be met by our classifier. One of them was that the classifier should be able to classify samples from an unknown manufacturer. If the classifier fails this requirement, then it won’t be a feasible one that can be used in a real life scenario.

One way to ensure against this is by iteratively excluding a specific group of samples from $D_{\text{org}}$, and keep them entirely in the test set. If we repeat this for every known group ($A$, $B$, $C$, $D$, $E$) in $D_{\text{org}}$, then we
end up with a total of five train-test splits. Therefore this can be seen a variant of 5-fold cross validation. The last group in $D_{\text{org}}$, denoted as $O$, consist entirely of samples, which has been assumed to be unlinked to every other sample. Therefore there is no point in putting these samples in their own fold. However, instead we can distribute them equally over the five folds that we created, such that we get a scenario as in Figure 7.

![Figure 7: The overall structure of LGO cross validation.](image)

Since we iteratively exclude a specific group of samples, we will refer to this strategy as leave-group-out (LGO) cross validation.

As seen from the Figure 7, we will refer to the different evaluations as test 1, 2, · · · , 5. For each test we achieve two sets that we could call respectively $D'_{\text{train}}$ and $D'_{\text{test}}$. The samples in these sets are on the same form as the samples in $D_{\text{org}}$. Therefore we need to convert $D'_{\text{train}}$ and $D'_{\text{test}}$ into the appropriate similarity representation so we achieve $D_{\text{train}}$ and $D_{\text{test}}$. By doing so, we get a small pipeline as seen in Figure 8. After $D_{\text{train}}$ and $D_{\text{test}}$ has been constructed, they can be used for respectively training and testing.

![Figure 8: The LGO pipeline.](image)

### 4.4 Evaluation Metrics

We have earlier discussed how the test accuracy (see equation 1, chapter 2) can be used as a measure of how well a classifier is at making predictions on $D_{\text{test}}$. Unfortunately, this measure doesn’t say any-
thing about how a classifier is at discriminate linked from unlinked. It only says that the classifier happens to work well on the samples that were in $D_{\text{test}}$. To see why this is a problem, lets look at a concrete example.

Lets assume that $D_{\text{test}}$ only consists of 1% linked samples, and that we have constructed a classifier that achieves a test accuracy of 99%. Based on the test accuracy, we could quickly think that is does a very good job. Withal, it could also be that the classifier only predicted all the unlinked samples correctly, without predicting any of linked samples. In fact, this would for instance be the situation for a classifier that always predicts unlinked, i.e. $h(x) = \text{unlinked}$ for any sample $x$. However, such a classifier would generalize very badly and would not be feasible in practice.

The situation above is a classical problem within classification, which is also known as the accuracy paradox in literature [1]. It illustrates how we can end up with a classifier that has a very high test accuracy, but unfortunately is very bad at classifying linked samples. Instead it would be better to have a classifier with a lower accuracy score, but a higher ability to classify the different classes correctly. Therefore the test accuracy is not the best objective for measuring the quality of a classifier, when the data set is very skewed. Instead we should use a measure that reflects how the classifier is at discriminate linked from unlinked.

However, before establishing such a measure, we will introduce the notion of positive and negative samples, which are often used within binary classification. The positive samples are often the underrepresented class samples that we aim at getting correctly classified. So in this thesis they will refer to the linked samples. Consequently the negative samples will refer to the unlinked samples. Having this set, we can further classify a prediction at being either true positive (TP), true negative (TN), false positive (FP) or false negative (FN), where the true-false label indicates whether a prediction is correct or not. Based on these notions, we can define three classical measures that are often used in classification [14]:

**precision**

Defines a relation between the true positive samples and the false positive samples, and is formally given by

$$\text{precision} = \frac{TP}{TP + FP}$$

The expression describes the fraction of samples that actually are linked, given the set of samples predicted to be linked. Therefore, if precision is very high, it will indicate that the classifier is very accurate, when it predicts linked. In other words, the classifier will be very reliable. On the other hand, if precision is very low, then the classifier shouldn’t be trusted when it pre-
dict linked. Therefore a classifier with a high precession is often favorable from a crime investigating point of view.

**RECALL**

Is defined as the proportion of positive samples that is correctly classified and is formally given by

$$\text{recall} = \frac{TP}{TP + FN}$$

The expression describes the fraction of samples that classifier predicted as linked, given the set of samples that actually are linked. Therefore this measure can be used to explain, how good the classifier is at recognizing the linked samples.

**F$_1$-MEASURE**

The $F_1$ score is given as the harmonic mean between precision and recall. I.e.

$$F_1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

The $F_1$ score is simply a way of representing both the precision in a single number, where the best value is 1 and the worst value is 0. Using the harmonic mean instead of a arithmetic mean has the effect of punishing the values more. For instance if we have a situation where recall = 0 and precision = 1, then it will imply that $F_1 = 0$. If we instead had taken the arithmetic mean, then we would get a value of 0.5, which reflects that the classifier is capable of doing something, even though it isn’t.

We will use the measurements above in order to analyze the performance of the classifiers we create and to see how well they are at discriminating between linked and unlinked samples.

The above measurements can further be used to categorize a classifier as being either conservative or liberal [14]. For instance, a conservative classifier is often characterized by making positive predictions with very strong evidence, and can therefore be recognized by a very high precision. Unfortunately the high precision is often archived at the cost of a low recall. On the other hand, if a classifier is very liberal, it makes positive predictions with very weak evidence, and can therefore be recognized by having a high recall. Unfortunate, it often has this property at the cost of a low precision.

When we compare the performances of a classifier, we will mainly focus on the $F_1$ score. However, in case of a tie, it is more favorable to have a conservative classifier, as it is more reliable at predicting linked.
LEARNING ALGORITHMS

I wish we had one of them doomsday machines.
— General ‘Buck’, Dr. Strangelove

Until now we have focused on how to create a meaningful similarity representation, and established how we should evaluate the performance of a classifier. What remains is how to construct the actual classifier.

In this section we will walk through the different learning algorithms that have been used in order to do this. First we will introduce the concept of a separating hyperplane, which is a commonly used concept within classification. Then we will look at Logistic Regression and Support Vector Machines, which both builds upon the concept of hyperplanes. Lastly, we will introduce K-nearest neighbour, a rather simple, but yet powerful learning algorithm as we will see. For each algorithm we will briefly introduce the theory of how they work and motivate why we have chosen them.

In practice the algorithms have been exploited through the scikit-learn library¹, which is an open source machine learning library for the Python programming language. scikit-learn is widely used within the machine learning community and features a variety of machine learning algorithms and general techniques like evaluation metrics, etc. Sadly, Python is often considered to be a rather slow language when compared to other languages like C and C++. Therefore, it is often an undesired choice for computing intensive tasks. Fortunately, a lot of subroutines in scikit-learn is implemented in the Cython programming language, which is a superset of Python that gets compiled down to C. Additionally, it utilizes a lot of pre-compiled libraries, such as NumPy and SciPy for numerical calculations, which are also implemented in the Cython language. Aside from that, it offers bindings to LibLinear and LIBSVM, which are highly engineered C++ libraries for support vector machines. Lastly, it has a fairly clean API, which makes it rather convenient to work with. For these reasons, scikit-learn has been used as a basis for this work.

5.1 LINEAR CLASSIFICATION

In classification problems the label is characterized by some categorical value, which is usually encoded into a numerical value. E.g. in our case we can encode the label into a binary value of \{-1, 1\}, where

¹ http://scikit-learn.org/
+1 denotes a linked sample and −1 denotes an unlinked sample. The goal is then to construct a classifier $h$ that is capable of assigning an unknown input $x$ into one of these numerical values.

Different strategies exist to do this. A common one is by using a so-called decision boundary, which is nothing more than just a surface within $\mathcal{X}$ that separates the space in two. The goal is then to construct this decision boundary such that the linked samples exist on one side, and the unlinked samples exist on the other side. Doing so, the classification problem simply reduces to determine which side of the surface that a sample exists on.

If the linked and unlinked samples are linearly separable, it is always possible to represent the decision surface as a hyperplane that perfectly separates the two classes. In such cases, the hyperplane is often described by a normal vector $w$ and an offset scalar $b$ that satisfies the expression $w^T x + b = 0$, for any point $x$ that lies on the hyperplane. The normal vector $w$ indicates how the hyperplane is orientated in $\mathcal{X}$, and the scalar $b$ can be interpreted as the offset to the origin.

One of the motivations for using this representation, is that it is very convenient to work with and very easy to evaluate, since it can be done by simply using the rule

$$h(x) = \text{sign}(w^T x + b)$$

The sign function helps to clamp the result to one of the two class labels, +1 or −1, which directly can be mapped to either linked or unlinked. In some cases the offset is part of the vector $w$ and $x$ is mapped into the form \begin{bmatrix} 1 & x_{(0)} & \cdots & x_{(d)} \end{bmatrix}^T$, where $x_{(i)}$ denotes the $i$’th component of $x$. Doing so allows us to write $w^T x$ instead of $w^T x + b$, which in some settings can be more convenient.

In the subsequent sections, we will see how algorithms like SVM and Logistic regression builds upon this idea of a hyperplane.

### 5.2 Logistic regression

In linear classification the model is also said to define a hard threshold, where the output it a categorical value \cite{33}. However, this hard threshold is a rather rigorous way of making prediction. So what could we do instead?

Another strategy would be to use Logistic Regression, where the output instead is a probability estimate that reflects how reliable a prediction is. To model this, we can think of our samples as been generated from some target function $p(y \mid x)$. Based on this, the goal is to construct a classifier $h$, which estimates $p(y = \text{linked} \mid x)$ - i.e. the probability of being linked given a sample $x$. So how should we do this?
One strategy could be to take a linear classifier \( h(x) = w^T x \), and transform its response into the range \([0, 1]\). A common way of doing this is by using a sigmoid function \( \sigma \), which for a real value \( z \) is given by

\[
\sigma(z) = \frac{1}{1 + e^{-z}}
\]

Putting \( z = w^T x \) we get that

\[
\sigma(w^T x) = \frac{1}{1 + e^{-w^T x}}
\]

The sigmoid function has the property of being a smooth function that is easy to take the derivative of \([22]\), which becomes handy later. Additionally, it has the property that it tends towards 1 as \( x \to \infty \), and tends towards 0 when \( x \to -\infty \). Therefore we are always guaranteed that the output will be in the range of \([0, 1]\). The shape of the sigmoid function is depicted below in Figure 9.

![Sigmoid function](image)

Figure 9: Sigmoid function.

The aim is then to construct \( h(x) = \sigma(w^T x) \), and use it as an estimator for \( p(y = \text{linked} | x) \). Based on this, we can classify a given sample \( x \) as linked, if the prediction \( h(x) \) is above a certain threshold. E.g. if \( h(x) > 0.5 \).

### 5.2.1 Estimate model parameters

Having established the model for logistic regression, the question that remains, is how to find the actual model parameters \( w \). If we look at our problem, then we are dealing with a binary classification problem where we will let \( y \in \{0, 1\} \) - i.e. 0 denotes unlinked and 1 denotes linked. If we model \( y \) as a binary random variable that is distributed according to a Bernoulli distribution, then we have that

\[
p(y|x) = \begin{cases} 
  p(y = 1|x) & \text{if } y = 1 \\
  1 - p(y = 1|x) & \text{if } y = 0 
\end{cases}
\]
Substituting $p(y = 1|x)$ with our model $h$, we get that

$$
p(y|x; w) = \begin{cases} 
h(x) & \text{if } y = 1 \\
1 - h(x) & \text{if } y = 0 \end{cases}
$$

I.e. the probability of $y$ given a sample $x$ and the parameters $w$, which can more compactly be written as

$$
p(y|x; w) = (h(x))^y(1-h(x))^{1-y}
$$

If we look at the expression $p(y|x; w)$, then it essentially describes a function of $x$ and $y$ for some fixed value of $w$. However, if we use the likelihood function $L$, we could view the expression as a function of $w$ for our fixed data set $D = (X, Y)$

$$
L(w) = L(w; X, Y) = p(Y|X; w)
$$

If we additionally assume that the $n$ samples in our data set $D$ are independently and identically distributed, then we can derive the following expression

$$
L(w) = p(Y|X; w, b)
$$

$$
= \prod_{i=1}^{n} p(y_i|x_i, w)
$$

$$
= \prod_{i=1}^{n} h(x_i)^{y_i}(1 - h(x_i))^{1-y_i}
$$

From this we can use the principle of maximum likelihood in order to determine $w$. The principle says that we shall choose $w$ such that the data we have seen is as likely as possible, which is formally written as

$$
w_{m1} = \text{argmax}_w L(w)
$$

Here $w_{m1}$ is known as the maximum likelihood parameter. However, instead of maximizing the likelihood function, we could equivalently minimize the negative log-likelihood function

$$
l(w) = -\log(L(w)),
$$

which is reasonable since the logarithm is a monotonically increasing function. Doing so will turn the product into a sum, and make it more convenient to work with. Therefore we end up with

$$
l(w) = - \sum_{i=1}^{n} y_i \log(h(x_i)) + (1 - y_i) \log(1 - h(x_i)),
$$

which in literature is also known as the cross entropy error [33].
A nice thing about \( l(w) \), is that it is a twice differentiable and convex \([28]\), so when \( w \) attains a minimum, we known its a global minimum. So in order to find \( w_{\text{ml}} \), we should find \( w \) where the gradient vanishes, i.e. where \( \nabla l(w) = 0 \). The gradient is more technically given as the the vector whose components are the partial derivatives of \( l \). So in order to derive the gradient, we will take the partial derivative of \( l(w) \) with respect to \( w_{(j)} \) - the \( j \)’th component of \( w \). Doing so we get that

\[
\frac{\partial}{\partial w_{(j)}} l(w) = \sum_{i=1}^{n} (h(x_i) - y_i)x_{i(j)}
\]

To see the complete derivation of this, the reader should consult appendix A.1.

If we design \( X \) as a matrix where the \( i \)’th row encodes the \( i \)’th sample \( x_i \), then we can write the gradient more compactly by using vector notation

\[
\nabla l(w) = (h(X) - Y)^T X = X^T (h(X) - Y)
\]

where \( h(X) \) indicates that \( h \) is applied on each row of \( X \). Lastly, we need to find the \( w \) that satisfies the condition

\[
\nabla l(w) = 0
\]

In other words we have a linear system of equations which we should solve with respect to \( w \). Unfortunately we cannot guarantee that the linear system is a well-posed system, so there might not be an exact analytical solution. Therefore, we cannot solve it by relying on simple matrix inversions. Instead, we can try and find a best fit solution by using gradient based optimization.

### 5.2.2 Gradient based optimization

A simple way of minimize the cost function in \( 15 \), is by using a numerical algorithm like gradient descent, where we iteratively take steps in the negative gradient \([22]\). At each step, the parameter \( w \) is updated according to the rule \( w = w - \alpha \nabla l(w) \). I.e. the algorithm takes a step in the steepest decrease of the cost function. This is continued until the improvement gets beneath a certain threshold. When this happens, we say the algorithm has converged, and the corresponding \( w \) can be returned as our best solution. The parameter \( \alpha \) can be seen as the learning rate that determines how fast the algorithm should converge to a near optimal solution. However, setting \( \alpha \) too high can result in a solution that diverges much from the near optimal. Contrary, if \( \alpha \) is very low, it will find a good solution, but at the cost of a slow convergence time. Especially if the objective function has a very
Algorithm 1 Gradient decent

1: \( w = \text{initialize randomly} \)
2: \( l_{\text{old}} = 0 \)
3: \( \text{for } i = 0,1,2, \ldots \text{ do} \)
4: \( w = w - \alpha \nabla l(w) \)
5: \( \text{if } | l_{\text{old}} - l(w) | \leq \text{tolerance} \text{ then} \)
6: \( \text{return } w \)
7: \( \text{end if} \)
8: \( l_{\text{old}} = l(w) \)
9: \( \text{end for} \)

skewed contour, it could cause the convergence time to be slow. A solution for this is therefore to normalize the data beforehand. Another problem with Gradient Decent is that it has a tendency to zig-zag in the same directions, which again can cause very bad performance. Therefore, more sophisticated approaches, like Conjugate Gradient, is usually more applicable in practice, since it does not have these tendencies. In scikit-learn, both Conjugate Gradient and a variety of other numerical algorithms are available, and can be used to perform Logistic Regression. Which algorithm to choose usually depends on the specific problem we are dealing with and whether or not we are using regularization (section 5.2.4), etc. We will not go into how these algorithms work, since that’s a field on its own. However, we have included gradient descent for completeness and since it’s a fairly simple example of how to solve Logistic regression.

5.2.3 Model fitting

The procedure described above will try to find the parameters \( w \), such that the cross entropy in equation 15 is minimized as much as possible on the data that we have in \( D_{\text{train}} \). Unfortunately, approaches like this could also produce a model that fits the data too well, and causes it to generalize poorly to unseen data. A situation like this is also what is known as overfitting [33], and is best described by looking at Figure 10a

![Figure 10: Instances of model fitting.](image)
From here we can clearly see that the data can be perfectly separated by using a complex decision boundary. Unfortunately, a decision boundary like this, might very well generalizing badly, since it tries to memorize the samples in $D_{\text{train}}$ too much.

The main reasons for why over-fitting occurs is usually due to

- $D_{\text{train}}$ is too small
- $h$ is too complex
- $D_{\text{train}}$ is noisy

In the first case, the learning algorithm will very likely produce a model that separates the samples very well. However, if $D_{\text{train}}$ is small, there is no guarantee that the solution will generalize well.

The second case usually occurs if we have too many features in our samples or if we have applied some sort of feature transformation. I.e. the features have been mapped into a complex representation like

$$
\begin{bmatrix}
  x_0 & x_1 & x_2 & \cdots & x_n^2
\end{bmatrix}^T
$$

Doing so, will force the learning algorithm to fit the parameters $w$ to a higher order polynomial, such that we get a very flexible model. Unfortunately the model could also become too flexible, as depicted in Figure 10a.

In the third case the learning algorithm will produce a model that fits the noise instead of the true data. In other words, the noise will prevent the algorithm from learning the underlying problem.

Since we in our case have a lot of samples (these group O samples) that potentially could have a noisy effect, there indeed is a motivation for ensuring against over fitting. The way this is done, is usually by constraining the model in some way.

However, we could also end up constraining the model too much such that we get a situation as in Figure 10b, which is also known as underfitting. Here the result is a model that is too simple to capture the overall structure of our samples. Therefore we need to balance these scenarios properly in order to get a situation as depicted in Figure 10c. A common way to do this is by using regularization.

### 5.2.4 Regularization

In regularization we put the variability of our parameters under control, by including an extra term like

$$
\lambda w^T w,
$$

in the objective, where $\lambda$ is a positive real value. Doing so, the objective in equation 15 becomes equal to

$$
I(w) + \lambda w^T w
$$
Fortunately, this does not change the convexity properties of the function \[28\]. Therefore, we can just derive the gradient as before, and use gradient based optimization (section 5.2.2) to find the optimal parameter \(w\).

The term above is also what is known as the penalty term, since it penalizes the objective when the magnitude of \(w\) tends to be very large. The hope is to get a less flexible model by reducing the magnitude of \(w\). In other words the regularization term shrinks the parameters towards zero.

Another way to interpret regularization is that it’s a way of controlling which model that our learning algorithm should prefer. By including this term in the objective, we can influence that more constrained models should be favored.

The \(\lambda\) parameter is a free parameter that needs to be chosen before the training session can be initiated, and is used to control the proper amount of regularization. If \(\lambda\) is very high, then a large magnitude of \(w\) will penalize the cost function a lot. Contrarily, if \(\lambda\) is small, then we will not pay a high price for a large magnitude of \(w\). In other words, the \(\lambda\) parameters can be seen a way of imposing which model that should be preferred. How to choose the proper value for \(\lambda\) will be discussed in chapter 6.

The regularization term above is also often called \(L_2\) regularization, since \(w^T w = \|w\|_2^2\), i.e. the squared Euclidian/\(L_2\)-norm. Another common choice is \(L_1\) regularization, which is often used when very sparse parameters are favored. This is especially useful in cases, such as DNA and text classification, where the samples has a lot of features. Thus \(L_1\) regularization can, in some sense, be seen as an alternative to PCA and other dimensionality reduction techniques.

However, since the samples in our case only has a few number of features, there is no motivation for choosing \(L_1\) regularization. Therefore we will primarily be using \(L_2\) regularization in conjunction with Logistic Regression.

### 5.3 Support Vector Machine

In this section we will briefly walk through the theory of Support Vector Machine (SVM) by Vapnik et al. [10], in order to see how they work and understand the motivation for using them. First will we look at the optimization problem that they try to solve in order to achieve what we could call a large margin hyper plane. Lastly, we will see how this decision bound can be converted into a non-linearly decision bound by introducing the concept of kernels.

The first subsections are mainly based on the theory from respectively [4], [7] and [21].
5.3.1 Hard margin SVM

When the linked and unlinked samples are linearly separable, we have seen they can be classified by a hyperplane that satisfies the expression \( w^T x + b = 0 \) for any point \( x \) on the plane (section 5.1). However, usually this hyperplane is not unique, so an obvious question is, which plane to choose?

Ideally we want to choose a decision plane that exists in the middle between linked and unlinked samples, since that imposes the largest separation of the two classes. In other words this plane could in some sense be seen as the optimal choice, as we believe it can generalize better so unseen samples. If we look at Figure 11, we can get an intuition of why this is a good idea.

![Figure 11: Optimal hyperplane vs. non optimal hyperplane.](image)

Finding this plane is in fact what a SVM is capable of and describes why its a reasonable choice for classification. So how does it do this?

To answer this, we will first consider two additional hyperplanes that are parallel to our decision plane \( w^T x + b = 0 \) and exist on each side of it. These hyperplanes can formally be written as

\[
\begin{align*}
    w \cdot x_i + b & \geq a & \text{if } y_i = 1 \\
    w \cdot x_i + b & \leq -a & \text{if } y_i = -1
\end{align*}
\]

for \( a > 0 \), or alternatively we can express them more compactly as

\[
y_i (w \cdot x_i + b) - a \geq 0 \quad \text{for all } i
\]

which will become useful later in this section. Additionally, we can force \( a = 1 \), simply by choosing the proper scaling of \( w \).

For convenience, we will refer to equation 3 as the positive hyperplane, since it exists on the positive side of the decision boundary, and equation 4 as the negative hyperplane. Together these planes forms what is known as the margin, which graphically is depicted in Figure 12.
The goal is then to find the parameters of the decision boundary, such that the margin is maximized as much as possible. Due to this, a SVM is also said to be a large margin classifier.

To maximize the margin, we will first derive an expression that describes its width, which we will denote $d_{\text{width}}$. Next we will form a constrained optimization problem that finds $w$ and $b$ such that $d_{\text{width}}$ is maximized.

To derive $d_{\text{width}}$ we will choose an arbitrarily point $x^+$ on the positive hyperplane, and an arbitrarily point $x^-$ on the negative hyperplane. In other words, we have that

$$w \cdot x^+ + b = 1 \Leftrightarrow w \cdot x^+ = 1 - b$$
$$w \cdot x^- + b = -1 \Leftrightarrow w \cdot x^- = -1 - b$$

These points are also what is known as support vectors and helps to constitute the margin of the SVM. If we take the vector $x^+ - x^-$, and projects it on to $\frac{w}{\|w\|}$, we get the Euclidean distance of the margin. This can also be geometrically explained by looking at Figure 13.

![Figure 12: The concept of a SVM.](image1)

![Figure 13: The margin explained from a geometrically point of view.](image2)
From this we can derive that
\[
\begin{align*}
  d_{\text{width}} &= (x_+ - x_-) \cdot \frac{w}{\|w\|} \\
  &= ((1 - b) - (-1 - b)) \cdot \frac{1}{\|w\|} \\
  &= \frac{2}{\|w\|}
\end{align*}
\]
which states that the width of the margin is inverse proportional to the length of the normal vector \(w\).

As pointed out earlier, our goal is to maximize this expression. However, a notable thing is that maximizing \(\frac{1}{\|w\|}\) is equal to minimizing \(\|w\|^2\), which we will choose to do instead. The reason for this is that maximizing \(\frac{1}{\|w\|}\) leads to an inconvenient optimization problem that is non-convex, and therefore difficult to optimize with respect to \(w\) and \(b\) [21]. Putting all these things together, we can write the whole problem as a constraint maximization problem where we maximize \(\|w\|\) subject to the constraint in 5

\[
\begin{align*}
  \text{minimize} & \quad \frac{1}{2}\|w\|^2 \\
  \text{subject to} & \quad y_i (w \cdot x_i + b) - 1 \geq 0 \quad \forall i
\end{align*}
\]

The constraint in expression 5 assures that the margin separates the linked and unlinked samples properly. Aside from that, the constant \(\frac{1}{2}\) is added to the objective, in order to make it more convenient to work with. For instance, in situations where the objective needs to be differentiated, then the constant helps to simplify the derivative. Together, the whole problem defines a quadratic function with linear constraints, which is also known as a quadratic program (QP), and can be solved by using a QP solver. From this, we should be able to obtain the parameters \(w\) and \(b\), which we can use to construct our classifier 
\[
h(x) = \text{sign}(w^T x + b)
\]

5.3.2 Soft margin SVM

In many cases we cannot provide any guarantee that our data set is linearly separable, which means that we cannot be guaranteed to find a feasible solution to the optimization problem above. One way to get around this is by allowing some of the samples to be misclassified. Also, even though we actually can separate our data set perfectly, it can sometimes be better to choose a decision plane that introduces some errors. For instance, if we look at Figure 14b, we see that there indeed exists a hyperplane that separates the data, but choosing the plane in Figure 14a will properly generalize better to future samples.
So how can we allow this kind of misclassification to happen? One solution is to introduce what is known as slack variables $\xi_i$, which can be used to relax the optimization problem. By including the slack variables as below, we allow some of the samples to fall within the margin or even on the wrong side of the decision plane.

$$\begin{align*}
\text{minimize} & \quad \frac{1}{2}||w||^2 + C \sum_{i=1}^{n} \xi_i \\
\text{subject to} & \quad y_i(w \cdot x_i + b) \geq 1 - \xi_i \quad \forall i \\
& \quad \xi_i \geq 0 \quad \forall i
\end{align*}$$

If a sample $x_i$ is between its margin and the hyperplane, then $0 \leq \xi_i \leq 1$. On the other hand, if the sample exists on the wrong side of the decision plane, then $\xi_i \geq 1$, and will therefore penalize the objective more. The $C$ parameter is a regularization term that is added to control how sensitive the SVM should be to misclassification. The higher $C$ value we choose the more we penalize errors.

For very large value of $C$ we will pay a very high price for points that violate the margin constraint. When $C$ goes towards infinity, we will get the same effect as with the hard margin SVM and be very sensitive to outliers. On the other hand, if $C$ has a low value the SVM will be very prone to errors, but this can be preferable in the case that the data set contains noise.

5.3.3 Skewed classes

When the data set is skewed, as in our case, it is not always appropriate to penalize both classes equally. A solution to this will therefore be to weight the slack differently for the two classes. One way do this is by factoring the slack expression into two terms

$$C \sum_{i=1}^{n} \xi_i \Rightarrow C_{\text{linked}} \sum_{i \in \text{linked}} \xi_i + C_{\text{unlinked}} \sum_{i \in \text{unlinked}} \xi_i$$

Where $C_{\text{linked}}$ is the slack weight associated to linked samples, and $C_{\text{unlinked}}$ is the slack weight associated to unlinked samples. If we set $C_{\text{linked}}$ to a larger value than $C_{\text{unlinked}}$, it indicates that a linked
sample will be penalized more than unlinked samples. Therefore, the SVM will put more attention on getting the linked samples on the right side of the margin, which is exactly what we want. The idea of weighting the class samples differently can be utilized in the LIBSVM library by setting the parameter `class_labels` to `balanced`. Thereby the soft margin parameters get weighted according to $n/(n_c \cdot n_i)^2$, where $n$ is the total number of samples, $n_c$ is the total number of classes and $n_i$ is the number of samples that belongs to class $i$. In other words, the classes get weighted proportionally to the inverse of their frequency.

### 5.3.4 Dual representation

The formulation of the optimization problem above is usually rewritten into its dual representation, which is given by

$$\begin{align*}
\text{maximize} & \quad \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j x_i^T x_j \\
\text{subject to} & \quad \sum_{i=1}^{n} y_i \alpha_i = 0 \quad \forall i \\
& \quad C \geq \alpha_i \geq 0 \quad \forall i
\end{align*}$$

and can be obtained by using the theory of Lagrangian Duality. The introduced variables $\alpha_i$ and $\alpha_j$, are also referred to as Lagrangian multipliers and are commonly used within mathematical optimization. A notable thing about this representation, is that it depends on an inner product $x_i^T x_j$ of our samples. As we will see, this has some useful properties that can be exploited, in order to create a non-linear decision boundary. Another reason for writing it into the dual representation is, that it can be solved efficiently by a method known as the Sequential Minimal Optimization (SMO) algorithm [26]. This algorithm serves as basis for many real life implementations, such as LIBSVM for instance, which we will be utilizing through the scikit-learn API. Deriving this dual is rather technical and will not contribute to the understanding of how SVMs work in a practical context. Instead the reader should consult [21] to see how it is derived step by step.

### 5.3.5 The kernel trick

When our data set cannot be separated linearly, we have earlier introduced the concept of feature mapping, which allows us to produce a non-linear decision bound. Sadly, this is usually a time consuming procedure. So instead of computing the mapping directly, we can take advantage of what is known as a kernel, which is a function that expresses an inner product in another feature space, which could be of

\[http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html\]
a higher dimension. If $x$ and $x'$ are two input samples in our feature space $\mathcal{X}$, and $\phi$ is a function $\phi : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$, then the kernel $k$ is given by $k(x, x') = \langle \phi(x), \phi(x') \rangle$. By using a kernel, we can implicitly define our samples in a higher dimensional space without computing the mapping directly.

So how do we connect this to our SVM? To see this, let's turn our attention to the dual representation, which was introduced above. As pointed out, the notable thing about this representation, is that it depends on an inner product $x_i^T x_j$ of our samples. Therefore, we can simply just replace this inner product by a kernel of $x_i$ and $x_j$, in order to turn our SVM into a non-linearly classifier. Doing so will allow us to learn a very complex decision boundary that can separate a lot of data.

An infinite number of kernels exists, which can be used for different purposes. But some of the more common ones includes the following:

**Polynomial Kernel**

The polynomial kernel is simply just a $d$-degree polynomial that is given on the form

$$K(x, x') = (\gamma x^T x')^d$$

where $d$ is the most important parameter, as it effects the degree of the polynomial, and $\gamma$ is just a coefficient that can be used to adjust the scale of the inner product. When $d = 1$ then this is referred to as the linear kernel.

**RBF Kernel**

The radial basis kernel (RBF), or sometimes also called the Gaussian kernel, is a kernel that has a lot of similarities to a Gaussian distribution. Formally it is given by

$$K(x, x') = e^{-\|x' - x\|^2 / 2\sigma^2}$$

or usually simplified to

$$K(x, x') = e^{-\gamma \|x' - x\|^2}$$

by setting $\gamma = \frac{1}{2\sigma^2}$. The value of the RBF kernel decreases exponentially in every direction, as the distance between $x$ and $x'$ increases. Therefore the RBF kernel can be interpreted as a similarity measure, which attains its maximum value of 1 when $x$ and $x'$ are similar, and decreases towards 0 as the distance becomes larger than $\frac{1}{\sqrt{\gamma}}$. For instance, if we fix $x$, then the expression can be seen as defining a region of interest, with $\gamma$ as the radius. Increasing $\gamma$ will have the inverse effect of increasing the $\sigma^2$ in a Gaussian distribution. So for very small values of $\gamma$,
then the region around each sample becomes very large. The result will be a model that is very constrained, since every other sample in our feature space will have an influential effect. On the other hand, if $\gamma$ is very large, the region around the samples will be very narrow, and could in worst case cause over-fitting. Therefore $\gamma$ needs to be set properly. If we insert the RBF kernel in the dual representation, we get an effect that best can be understood as a sum of Gaussian distribution, which is centered on each data sample. Thus, the RBF kernel allows us to learn very complex decision boundaries.

As mentioned earlier, the main advantage of kernel functions comes from their fast evaluation time, since we just need to evaluate an inner product of two input samples. In other words, the time complexity will be linear in the number of features. If we instead expanded the features explicitly, then this would yield a time complexity of $O(n^d)$ for a polynomial of degree $d$ and $n$ features. Furthermore, expanding the RBF kernel we would yield an infinitely sum of features [30], and therefore be impractical. Therefore, kernels are essential, in order to learn a non-linear decision boundary efficiently.

We will mainly focus on the nonlinear RBF kernel and the linear kernel, in order to compare how the SVM performs against Logistic regression.

5.3.6 Feature scaling

When working with SVMs it is highly recommended to perform feature scaling as a pre-processing step, as it makes each feature equally important. Additionally, feature scaling can also have an effect on reducing the training time, which is noted in [29]. Therefore it should be a mandatory step to perform, when using SVMs.

The most common way of performing feature scaling is by using either: standardization, normalization or traditional scaling. In standardization the features are scaled to have 0 mean and a variance of 1 by applying the following rule

$$X = \frac{X - \text{mean}(X)}{\sqrt{\text{var}(X)}}$$

where $\text{mean}(X)$ and $\text{var}(X)$ denotes respectively the mean and the variance of the features. Unfortunately, if our data set is very sparse, then this destroy the sparsity [4], which in some cases is not preferable. As an alternative one can apply normalization instead, where each sample is divided by its norm to make it a unit vector. This is especially useful in cases where the SVM is using a kernel. The reason for this is that normalization can scale the features into a range that makes some of the kernel computations more numerically stable,
which is noted in [7]. Particularly for kernels that rely on an inner product, it is advisable to use normalization [7].

As an alternative, one could also choose to scale the features to a specific range such as $[0, 1]$ or $[-1, 1]$. Since most of our representations will be within this range, it will not be a crucial step to perform. But as it rarely hurts to apply normalization, we will be including this in our pre-processing step.

5.4 K-NEAREST NEIGHBOR

The K-nearest neighbor (kNN) algorithm is a rather simple learning algorithm, where classification is based on the neighborhood that exists in our feature space $\mathcal{X}$. The observation is that the samples that belong to the same class often are closely related in $\mathcal{X}$. In spite of this, it is reasonable to base our classification by simply looking at the neighborhood. This is done by first collecting the $k$ neighbors according to some distance metric $d : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$. Next the label is determined by performing majority voting, where the most representative label is returned as answer. In other words, we base our prediction on the similarity that exists among the $k$ closest neighbors. In case of a tie, the behavior is not well-defined, but can be solved in a variety of ways. For instance, one suggestion is to always choose $k$ to be an odd number (but this is not guaranteed to work for multi-classification). Another suggestion is to reduce $k$ by a value of 1, in order to break the tie. In scikit-learn utilizes a third approach, which is simply just based on the ordering of the data.

Conceptually, kNN is referred to as a lazy learning algorithm, which means that the training in some sense is deferred until a prediction is made. Thus instead of performing the whole training session ahead of time, the data is just placed in a data structure $\mathcal{DS}$, which we perform our query on. Thus, kNN creates a non-parametric model, which makes its predictions based on the available data, the free parameter $k$ and the chosen distance function $d$. This is opposed to algorithms like Logistic regression and SVM, where an unknown target function is approximated before a prediction is made.

For a given sample $x$, the overall algorithm can roughly be summarized by the following steps

- compute $d(x_i, x)$ for each $x_i \in \mathcal{DS}$
- select $k$ closest neighbors
- perform majority voting

4 [https://en.wikipedia.org/wiki/Lazy_learning](https://en.wikipedia.org/wiki/Lazy_learning)
In the naive case, the data structure DS is simply just a list that we iterate through. Therefore the prediction time will have a time complexity of $O(k \cdot n \cdot d)$, since we need to perform $O(n)$ comparisons for $k$ samples, and for each comparison we need to evaluate $d$ features. In cases where we have a lot of samples, this could create a rather slow prediction time. As an alternative, we could use spatial data structures such as kd-trees or ball-trees, which have a query time of $O(\log(n))$, when the trees are well-balanced. Both these data structures are available through scikit-learn’s kNN API \footnote{http://scikit-learn.org/stable/modules/neighbors.html#nearest-neighbor-algorithms}, but since we only have at most $309^2$ samples, it is not expected to be relevant in practice.

### 5.4.1 Choosing $k$

Before kNN can be used, we need to carefully determine the neighborhood constant $k$, as it have an impact on the quality of the classification. If $k$ is too small, then kNN will create a rather complicated decision boundary that fit the data very well, such that over-fitting can occur. For instance, if we choose $k = 1$, the algorithm can be seen as subdividing the feature space into a Voronoi diagram, by creating a Voronoi cell $V_i$ for each sample $x_i$, which formally can be written as

$$V_i = \{x \in \mathbb{R}^d : \text{d}(x, x_i) \leq \text{d}(x, x_j), \forall i \neq j\}$$

The decision boundary will in this scenario be a piecewise linear surface that follows the boundary of the Voronoi diagram.

If we increase $k$ it will have the effect of producing a smooth decision boundary, since we average over a larger neighborhood. This is especially relevant in situations where noise and outliers are present in the data set, since it will reduce the effect of anomalies. Unfortunately, this will also make the boundaries between the classes less distinct. Therefore there is a trade-off to be made when selecting $k$.

### 5.4.2 The distance function

Aside from choosing the proper value of $k$, the proper distance metric also has a huge impact on the prediction, since it defines which samples that are similar. However, the most common choice is the Euclidean distance, as it treats all the features equally. On the other hand, if one of the features is numerically larger than the others, it will have a huge impact on the distance calculation and could cause a negative effect. Therefore, it is necessary to perform feature scaling before using kNN (section 5.3.6).
Another thing to be aware of is how the dimensionality of the feature space can have an effect on the performance. For instance if the feature space is highly dimensional, it can cause a situation where the notion of distance becomes meaningless. This is also sometimes referred to as the curse of dimensionality\(^6\), and is a general problem that relates to most distance based machine learning algorithms. Informally speaking, the distance between the closest and the farthest sample becomes less clear such that it becomes harder to determine the class label. A way to handle this is therefore by applying dimensionality reduction techniques such as PCA. However, since our samples has a relatively low number of features, we should not expect this to be a problem.

5.4.3 skew\(\)ed classes

Lastly the algorithm can be rather sensitive to skewed classes, since the majority labels can have a dominant impact on the classification. One way to handle this, would be to weight their importance differently. For instance, in scikit-learn this can be handled by weighting the samples by the inverse of their distance, such that closer neighbors will get more importance than neighbors that are further away\(^7\). Since we generally have a lot more unlinked samples than linked, we will be exploiting this functionality.

5.5 choice of algorithms

The three algorithms that we have looked at are rather well established algorithms within supervised learning. Other alternatives could have been chosen as well, so in this section it will try to argue why these algorithms have been chosen.

The reason for why Logistic regression has been included is mainly because earlier studies at BiRC is based on this. In other words, it serves as a basis for making comparisons. One of the advantages with Logistic regression is that it outputs a value that can directly be interpreted as a probability estimate. Therefore it automatically produces a feasible model. The drawback with Logistic regression is that it produces rather simple models. Therefore it might not separate linked from unlinked properly if there exists a complex structure in the samples.

The second algorithm we have looked at is SVM. The reason for this choice, is that SVM can be seen as an optimized version of Logistic regression, due to its large margin principle that helps it to generalize well. Additionally, when SVM is combined with non-linear kernels, it

---


is often capable of producing a very complicated decision boundary that can fit many problems well. The disadvantage of SVM is that its training time can in some cases be huge. The scikit-learn documentation states ⁸

The QP solver used by this LIBSVM-based implementation, scales between \( O(n_{\text{features}} \cdot n_{\text{samples}}^2) \) and \( O(n_{\text{features}} \cdot n_{\text{samples}}^3) \) depending on how efficiently the LIBSVM cache is used.

Thus, it could be rather time consuming to train, when the number of samples is large. Aside from that, it is rather sensitive to how we choose the kernel parameters and the slack regularization.

The last algorithm we have focused on is kNN, which is a rather simple algorithm. However, although kNN is a straightforward and intuitive its performance is often comparable to more advanced techniques. For instance in [16] it is shown that kNN’s performance very similar to SVM in the context of text classification. The reason for this comes from its ability to create a rather complex decision boundary. Therefore there might not be a need for sophisticated techniques like SVMs.

The drawbacks with both SVM and kNN is that they does not directly provide the probability estimate as required. Therefore there is some extra work to done, in order to turn kNN and SVM classifiers into probabilistic classifiers. We will look more at this in chapter 9.

---

For all the learning algorithms that we have seen, there exists a special set of parameters, which needs to be adjusted, before the training session can be initiated. For Logistic Regression this includes the regularization parameter $\lambda$, for kNN it includes the neighborhood constant $k$ and for SVMs it covers the amount of slackness $C$ and the kernel parameter $\gamma$. Together these make up what we call hyperparameters [34]. Although the hyperparameters are not part of the actual model parameters, they can have a huge impact on the final classifier that is produced. Therefore there is motivation for finding the hyperparameters that produce the best classifier, which is often done through what is known as parameter tuning.

In this chapter we will first discuss how the evaluation of hyperparameters should be carried out in order to be unbiased. Next we will look at two common strategies for performing parameter tuning known as respectively grid- and randomized-searching.

### 6.1 Validation Set

The most naive way to carry out the evaluation, would simply be to choose the hyperparameters that produces the classifier with the highest classification score on $D_{test}$. However, if we do this, we could end up fitting the hyperparameters too well to this data set. I.e. our classifier will very likely do better on $D_{test}$ than on future samples. Thus the score would not be a fair estimate of how the classifier generalizes. Consequently, it is crucial to evaluate the hyperparameters on a separated set of samples, which we will denote as our validation set $D_{val}$. As in previous cases, the more samples we put in $D_{val}$, the more reliable the score will be.

When constructing $D_{val}$, we do this by using stratified sampling (section 4.1) on $D_{org}$, so it represents the different groups properly, i.e. A, B, · · · , O.

### 6.2 Grid Searching

The first tuning strategy that we will introduce, is a rather naive one, known as grid searching. In grid searching the user defines a sequence of values for each parameter, and then train a classifier for
each combination of the values. The values that give the best validation score will be chosen for our learning algorithm. Unfortunately, the number of combinations grow exponentially with the number of parameters, so the strategy quickly becomes a time consuming procedure.

In order to improve performance, a coarse-to-fine searching can be used. Here we first identify a region with good parameters, by using a coarse search, and then we locate the best parameter within this region, by using a finer grid search. Additionally, the searching can be done in a variety of ways. For instance, in [7] they suggest an exponentially growing sequence, like $2^k$ for $k \in 2^k$, as it often has a good exploring effect in practice.

Another strategy for leveraging the searching is to only use a subset of the training data. So instead of performing the grid search on the full training set we can subsample it into smaller one, by using stratified sampling. Doing so, we ensure that the small data set will have the same distribution of labels as the original one. Unfortunately, reducing the training set will also make the procedure more prone to overfitting and produce a larger variance.

One of the advantages of grid searching is that it is an embarrassingly parallel strategy. That is, it can easily be parallelized without any effort in separating the problem, since each classifier can be evaluated independently from each other. Withal, even though grid searching can be accelerated using different kinds of techniques, it is still a time consuming procedure, due to its exhaustive nature.

### 6.3 RANDOMIZED SEARCHING

One of the problems with using grid searching, is when one of the parameters tend to have no effect on the model building process. In this scenario the procedure quickly ends up training the same classifier multiple times, which will be waste of resources. Instead, it would be better to use random searching where we just try some random combination of values. Thus instead of specifying a list of values for each parameter, one specifies a distribution, e.g. a uniform or a Gaussian, from which the actual values are sampled. The motivation is illustrated in Figure 15, where we can get an intuition of why random searching is often better at exploring the parameter space.
Randomized searching is of course not guaranteed to find the optimal parameters faster than grid searching, but the changes that it does, are often good. This is in fact shown in [17], to have both theoretically and empirically evidence. For instance, if some of the parameters are unimportant, then random searching can at best provide an exponential speed up over grid searching. Therefore it is often preferable in such situations. Lastly, this method is also embarrassingly parallel, for the same reasons as grid searching is.
CLASSIFICATION EXPERIMENTS

The whole trial seemed surreal.
― Bernhard Goetz

Different experiments have been conducted in order to measure the performance of the different classifiers. In this section we will walk through the experiments and discuss the results. The general steps that have been performed, can be summarized as follows:

1. Parse the data and transform it into log space.
2. Construct $D_{val}$ by using stratified sampling.
3. Perform parameter tuning.
4. Train and evaluate the classifier by using LGO cross validation.

When performing parameter tuning, we have mainly been using grid searching, due to its deterministic nature.

However, in order to speed up the performance for SVMs, we use a coarse-to-fine approach, where we first search on a large scale of values and then on more fine grained interval. For instance, when using SVM with the RBF kernel, we use the following exponential sequence of values

$$C : \{10^{-2}, 10^{-1}, \ldots, 10^8\}$$

$$\gamma : \{10^{-6}, 10^{-5}, \ldots, 10^3\}$$

and obtain what we could call $C_{\text{coarse}}$ and $\gamma_{\text{coarse}}$. Next, we perform a linear searching starting 10% below $(C_{\text{coarse}}, \gamma_{\text{coarse}})$ and ending 10% above these values.

Regarding kNN, we perform a linear search in the interval from 1 to 40. However, in case of a tie, we choose the greatest value of $k$, since we believe that this is more robust against over-fitting.

For Logistic regression, it has been discovered that the sequence

$$\{1, 10, \ldots, 100, 200, \ldots, 1000\}$$

generally have worked well in most situations.

In all cases, the parameters are chosen based on the values that produce the highest $F_1$ score, since that is what we aim at maximizing. When the best parameters have been found, we initiate the actual training.

As pointed out in chapter 5, the algorithms have been utilized through the scikit-learn API. To see how this is done, the reader
should consult \(^1\), where the behavior of the algorithms have been wrapped into their own abstraction. Next the different experiments have been defined in \(^2\).

### 7.1 Goal

First we will replicate the experiment performed at BiRC, to see how good the classification can get when both the alkaloids and the solvents are used. After that, the aim is to get as close as possible to this, by only using the alkaloids.

We will primarily focus on the average \(F_1\) value and the standard deviation (SD) of the \(F_1\) scores. A low SD will indicate that the \(F_1\) scores are very similar and show that the classifier predicts in a rather consistent way. Therefore the aim is to achieve a classifier where the average \(F_1\) score is as high as possible and the SD is as low as possible.

We will not go through every single experiment that has been conducted, but only focus on the most interesting ones. The results will be presented through different plots, which have been created using the Seaborn library - a statistical visualization library for Python \(^3\).

### 7.2 Expectations

The performance of the different classifiers is generally expected to be dependent on the representation used. For instance, when using a pure distance based representations, it is not expected that SVM should have an advantage over Logistic regression. The reason for this, comes from the fact that the classifier more of less reduces to a simple threshold. Therefore it would be rather strange to see any advantages.

In contrast, when the representation contains a lot of features, it is generally expected that the SVM with a RBF kernel should have an advantage over Logistic regression. The reason for this comes from its ability to produce a very complicated model that can capture very complex structures in the data.

Regarding linear SVM, it is generally expected to have the same performance as Logistic regression, since both algorithms create very simple models. Therefore, it would be utmost specious if the difference is huge. On the other hand, if the classes are very close at being linearly separable, then linear SVM could have an advantage over Logistic regression, due to its large margin principle.

For kNN, it is difficult to say how well it performs against the other algorithms, since it works in a rather different way. But it generally could have an advantage when the representation is informative (i.e.

---

\(^1\) classify\_wrapper.py

\(^2\) experiment\_classification.py

\(^3\) [https://stanford.edu/~mwaskom/software/seaborn/](https://stanford.edu/~mwaskom/software/seaborn/)
contains many features) and when k is chosen correctly. The reason for this, comes from its ability to produce a rather complex decision boundary.

7.3 Results: Alkaloids & Solvents

The model produced at BiRC has been constructed by using Logistic regression and the cosine distance as input representation. As seen from table 1, the results look almost perfect. Thus the solvents help to create an incredibly stable representation, which makes the relationship between the drugs very clear. This is also in some sense reflected by looking at the PCA plot in Figure 16 below.

<table>
<thead>
<tr>
<th>TEST</th>
<th>FP</th>
<th>FN</th>
<th>TP</th>
<th>TN</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>406</td>
<td>816</td>
<td>0.993</td>
<td>1.0</td>
<td>0.996</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>406</td>
<td>818</td>
<td>0.998</td>
<td>1.0</td>
<td>0.999</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>406</td>
<td>816</td>
<td>0.993</td>
<td>1.0</td>
<td>0.996</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>406</td>
<td>770</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>406</td>
<td>770</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 1: When both alkaloids and solvents are used.

We will use the result in table 1 as a reference for what we should aim for.

![PCA plot without solvents](image1.png)

![PCA plot with solvents](image2.png)

(a) Without solvents. (b) With solvents included.

Figure 16: Clearly the solvents help to impose a better relationship between the samples.

7.4 Discarded Experiments

Not every experiment turned out to work as intended. For instance, the attribute based representation from section 3.5, did not work in practice, as it could not generalize to groups that were excluded from the training session.
The reason for this is that the representation does not reflect similarity in general. For instance when concatenating two samples from e.g. group A, then we are only reflecting how linked samples from this group looks like. Therefore we can only use this representation in cases where we know each group beforehand. However, this does not fit our requirements.

What it should respect instead is that similar samples are clustered together in one part of the feature space and that dissimilar samples are clustered into another part of the space. Fortunately, the other representations respect this.

7.5 Results: Alkaloids

As pointed out earlier, we will not go through every single experiment that have been conducted. For instance, regarding the distance based representations, they turned out to work very similarly. Due to this, we will only focus on the cosine distance, since it showed a slight improvement in some cases.

Additionally it has been tried to train a SVM in conjunction with a polynomial kernel. However, it was not better than using a linear kernel and for some of the representations it turned out to be worse, which could indicate that it was too prone to over-fitting. Furthermore, the training time was tremendously long, which reveals that the SVM had a hard time fitting the polynomial to the data. In spite of this, the experiment has been omitted.

Having said this, the overall results can be seen in Figure 17, where they are summarized by using a box-plot. Every box in the box-plot represent a single experiment where each horizontal line represent one of the five tests that was performed through LGO cross validation.

The reason why a box plot has been used is due to the fact that it is very easy to examine whether outliers are present in an experiment. If there is an outlier, it will be characterized by a box that is very stretched out. On the other hand, if a box is relatively squeezed together, it will illustrate that the five tests are rather consistent and consequently have a low SD. As indicated below, we will be referring to the different representations by using the shorthand notation $R_i$ for $i \in \{1, 2, 3\}$. 
7.5 Results: Alkaloids

For details about the average $F_1$, the SD and the best parameters found, the reader should consult table 2 below.

<table>
<thead>
<tr>
<th>$R_i$</th>
<th>Algorithm</th>
<th>Mean($F_1$)</th>
<th>SD($F_1$)</th>
<th>Best Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1$</td>
<td>Log. reg.</td>
<td>0.959</td>
<td>0.017</td>
<td>$\lambda = 600$</td>
</tr>
<tr>
<td></td>
<td>SVM linear</td>
<td>0.947</td>
<td>0.0224</td>
<td>$c = 50.000$</td>
</tr>
<tr>
<td></td>
<td>SVM RBF</td>
<td>0.954</td>
<td>0.019</td>
<td>$c = 50.000, \gamma = 1000$</td>
</tr>
<tr>
<td></td>
<td>kNN</td>
<td>0.883</td>
<td>0.048</td>
<td>$k = 17$</td>
</tr>
<tr>
<td>$R_2$</td>
<td>Log. reg.</td>
<td>0.953</td>
<td>0.02</td>
<td>$\lambda = 400$</td>
</tr>
<tr>
<td></td>
<td>SVM linear</td>
<td>0.946</td>
<td>0.024</td>
<td>$C = 133333$</td>
</tr>
<tr>
<td></td>
<td>SVM RBF</td>
<td>0.937</td>
<td>0.038</td>
<td>$C = 8, \gamma = 1000$</td>
</tr>
<tr>
<td></td>
<td>kNN</td>
<td>0.936</td>
<td>0.058</td>
<td>21</td>
</tr>
<tr>
<td>$R_3$</td>
<td>Log. reg.</td>
<td>0.964</td>
<td>0.018</td>
<td>$\lambda = 1000$</td>
</tr>
<tr>
<td></td>
<td>SVM linear</td>
<td>0.961</td>
<td>0.017</td>
<td>$C = 5000$</td>
</tr>
<tr>
<td></td>
<td>SVM RBF</td>
<td>0.981</td>
<td>0.015</td>
<td>$C = 0.5, \gamma = 133$</td>
</tr>
<tr>
<td></td>
<td>kNN</td>
<td>0.982</td>
<td>0.017</td>
<td>$k = 4$</td>
</tr>
</tbody>
</table>

Table 2: An overview of the average $F_1$, the SD and model parameters.

In the next subsections, we will go through each of the representations.

7.5.1 Cosine distance ($R_1$)

If we look at $R_1$, we can see that both Logistic regression, Linear SVM and SVM RBF perform equally well. However, this does not come as a surprise, since they all produce a model that more or less reduces to a simple threshold. Therefore there is no advantage in using SVM over Logistic regression, when this representation is used.
If we instead look at kNN we can see that the performance is much worse than the other algorithms. The reason for this is that the data set contains these O samples, which have been assumed to be unlinked although some of them could be linked. Consequently there will be a lot of unlinked samples that coincide with a lot of linked samples, as seen in Figure 18.

![Figure 18: Histogram of both linked and unlinked distances.](image)

This especially hurts algorithms like kNN, since it predicts linked and unlinked based on a local neighborhood (i.e. k closest neighbors).

The other three models work more in a global manner by using a fixed decision threshold. Therefore it gives them an advantage over kNN when using this representation.

7.5.2 Multiple distances ($R_2$)

When using $R_2$, we can see that the performance gets slightly worse, and for both SVM RBF and kNN, they seem to have problems with one of the tests. Unfortunately, it has not been possible to find any cause for this. Nonetheless, we can conclude that multiple distances do not create a representation that is better than just a single distance function.

7.5.3 Similarity vector ($R_3$)

From Figure 17 we can clearly see that $R_3$ gives the best classification, which also corresponds to our expectations. The reason for this, can in some sense be understood from the PCA plot in Figure 19.
As we can see the representation creates a rather large dispersal of the samples, which seems to improve the separation between linked and unlinked samples. This especially benefits both SVM RBF and kNN, since they both are capable of constructing a very complex decision boundary. However, although SVM is far more sophisticated than kNN, they have very similar performance. For instance, if we look at table 2, then we can see that

- SVM RBF achieves a mean F₁ of 0.991 and a SD of 0.015.
- kNN achieves a mean F₁ of 0.982 and a SD of 0.017.

In other words, there is no notable difference between them. Therefore we can say that it is a tie between SVM RBF and kNN.

### 7.5.4 Precision and recall

In Figure 20 and 21, we have included some additional box-plots that show the precision and the recall, of the respective models.
When using $R_1$, we can see that the models produced by Logistic regression, SVM linear and SVM RBF are very liberal, which indicates they are good at recognizing all the linked samples. However, they do so, at the cost of a low precision, which is not a favorable property, since it can lead to miscarriage of justice. The reason for this behavior, is due to the global threshold they produce, which will cause a lot of unlinked samples to be predicted as linked. If we instead look at kNN we can see that this algorithm creates a model that is fairly conservative when using $R_1$. I.e. when it predicts linked, it can be thought of as being very reliable, which is a more favorable behavior. Sadly it does so at the price of a very low recall, which indicates that it has some difficulties with recognizing the linked samples.

If we look at $R_3$ we can see that both Logistic regressing and Linear SVM has a rather liberal behavior, which indicates that the decision boundary they produce are too simple to properly capture the linked samples. If we instead look at respectively kNN and SVM RBF, we can see their precision have improved a lot, which indicates they do not have these problems. However, they also have a much higher model complexity.

To see the results for different threshold values, the reader should consult appendix A.3.

7.5.5 Dot plot

In the next plot we can see how the different tests are ranked relative to each other. If a test systematically performs worse than the others, then it could indicate that it contains a group of samples that are difficult to classify correctly.
7.5 RESULTS: ALKALOIDS

Figure 22: Dot-plot of F₁ score.

From the plot we can see that the tests for R₁ and R₂ are generally ranked the same way when using Logistic regression and SVM. However, this also makes good sense, since they produce very similar models when using this representation.

For R₃ the tendency is that test 1 and test 4 generally achieve the best performance, which is interesting since test 4 had a bad performance when using both R₁ and R₂. In other words, the way that the tests are ranked, varies between the representations.

7.5.6 Class weighting

For some of the algorithms, it has shown to be important to weight the linked and unlinked samples differently. For instance, when using SVM RBF with R₃ and we weight the slack equally for both linked and unlinked samples, then we get an average F₁ of 0.965 and a SD of 0.043. Contrary, if the slack is weighted as discussed in section 5.3.2, we achieve the result in table 2, where the average F₁ is 0.981 and the SD is 0.015. In other words, SVM RBF is highly affected by the skewed nature of the data set.

Regarding kNN, it is less important to weight the samples differently. For instance, if we weight the samples equally, then we achieve a F₁ of 0.981 and a SD of 0.016, for R₃. This is almost the same as if we weight the samples by the inverse of their distance, since that gives a F₁ of 0.982 and SD a of 0.017. Thus the difference is so small that it could almost be neglected. In conclusion, we could say that kNN is less sensitive to the imbalance between linked and unlinked samples.
7.6 Validation Score

In this subsection we will investigate the effect that the hyperparameters have on the learning algorithms. We will mainly restrict our attention to $R_1$ and $R_3$, and see how sensitive these representations are to changes in hyperparameters. Some of the results will diverge a bit from the optimal scores above. The reason for this is that we evaluate the performance on $D_{val}$ and not on $D_{test}$.

7.6.1 kNN

For kNN the effect of increasing $k$ from 1 to 40 have been plotted and can be seen in in Figure 23. Here the continuous curve represents the average $F_1$ and the dotted curves represent the SD from running the five tests on $D_{val}$.

![Figure 23: The effect of increasing k in kNN.](image)

Although $R_1$ does not produce as good results as $R_3$ does, it shows to be more stable against changes in $k$. This is thought to be due to the fact that the dimensionality of $R_1$ is lower than $R_3$.

In low dimensions, things are generally more clustered together and give a more representative neighborhood. I.e. for a sample $x$ the $k$th nearest neighbors should be very close to $x$. Additionally, when the neighborhood is very representative, it should be less affected by changes in $k$. Therefore, we see a rather constant curve in 23a. In higher dimensions the samples are less clustered together, so the neighborhood become less representative. Thus, when we increase $k$ we get more prone to errors, which is also what we see in 23b.

The situations above are also reflected by looking at the SD that is associated with the curves. For instance, in $R_1$ we get a fairly constant SD, regardless of how much we increase $k$. Conversely, for $R_3$ the SD increases proportionally with the value of $k$ and gives a more instable result.

However, if we are in higher dimensions and we at the same time choose $k$ carefully, then we generally achieve a good performance. I.e. we achieve a decision boundary that separates the samples very well.
7.6.2 SVM RBF

For SVM, a heat map is used to illustrate how the value of $F_1$ changes with respect to both the $\gamma$ and the $C$ parameter. Each cell in the heatmap correspond to a pair of values, and the color reflects the corresponding $F_1$ score. The brighter the color, the higher the $F_1$ is. Below in figure 24, we can see the effect of using an exponential search.

In both figures we see that there exists some sort of a ridge where the parameters seem to fall within. However, as noted in [7], this is in fact not an unusual phenomenon when using SVM with a RBF kernel. The reason behind this is, that the more we decrease $\gamma$, the wider the RBF kernel becomes. I.e. we make it less flexible, and therefore more prone to misclassification.

To compensate for this, we can increase $C$, such that we penalize misclassification harder. Doing so will increase the number of support vectors and shape the decision surface, such that we get the misclassified samples on the right side of the surface. Thus, the $C$ and the $\gamma$ parameter are in some sense complementary, which is also what is reflected by this ‘ridge’.

Figure 24: Heat map of the parameter space when using SVM.

If we start inspecting 24a, we see a rather flat parameter space, where changes in both $C$ and $\gamma$ contribute very little. Thus it indicates that the cosine representation is rather robust against variations in the parameters of the RBF kernel. Intuitively, this also makes very good sense, since the representation only consists of a single dimension, where the linked samples have a rather low value and the unlinked samples have a rather high value. Due to this simple structure, it is hard to imagine that $C$ and $\gamma$ should contribute to much variability. However, when $C$ and $\gamma$ attain extreme values they end up producing an infeasible model, which is also what we see by this dark area.

If we change our attention to 24b, we see a different landscape, where the first impression is that there exists a lot of variability. Especially $\gamma$ seems to play a dominant role. If $\gamma$ is very low, it gives a
very poor $F_1$, which could indicate that the model becomes too con-
strained. I.e. the model cannot capture the overall shape of the data,
since the region around each sample becomes too large.

When $\gamma$ ranges in the interval from 0.01 to 100, it decreases the
region around each sample and gives rise to a more flexible classifier.
The increase in flexibility has the effect that the structure of the data
can be captured properly, and thus produces a better $F_1$. Also we see
that the $C$ values begin to have an influence. For instance:

- When $\gamma$ is small we have a large region around each sample, so
  we need to increase the complexity of the decision surface by
  increasing $C$.

- For larger values of $\gamma$ we get a smaller region around each sam-
  ple, so the complexity of the decision surface should be lowered
  by reducing the $C$ value.

Additionally when $(\gamma, C) \in \{(0.1, 10^7), (0.1, 10^6), (10^4, 1), (10, 100),
(100, 1)\}$, we see regions that produce a very high $F_1$. I.e. we have a
lot of maxima, which indicate that the parameter space has a non-
convex shape. It has further been discovered that the number of sam-
pies in $D_{val}$ have an impact on which of these regions that give the
best score. For instance, if $D_{val}$ consists of 20% of the overall samples,
then $(\gamma, C) = (10, 100)$ are chosen as the best parameters. Contrary, if
$D_{val}$ consists of 30% of the samples, then $(\gamma, C) = (100, 1.0)$ are cho-
sen instead.

When the value of $\gamma$ approaches 1000, the performance starts to
drop, which could indicate that the region around each sample be-
comes too small. In the extreme case the region will become so small
that it only contains a single sample. In other words, overfitting oc-
curs.

As we have seen above, the range of values that we choose to search
within, have a large impact on the performance of the model, and
should therefore be chosen carefully. Sadly, this often relies more on
intuition than on exact science. Therefore it is crucial to have a tool
for inspecting how the parameter space changes. For instance, if we
just inspected the $F_1$ scores one by one, then it would be rather cumber-
some and tedious to find this range. However, by using the heat
map, we get an image that can be understood in a blink of an eye.
Thus it serves as a guidance for determining the range to search in.

7.6.3 Logistic Regression

Finally, we have investigated the effect of changing the regularization
parameter $\lambda$ when using Logistic Regression. The results can be seen
in Figure 25
Figure 25: The effect of increasing the regularization parameter.

As we can see, when $\lambda$ is increased it improves the quality of the classification. And when $\lambda$ exceeds a value of 100, it achieves a maximal classification, for both representations. The exact reason for this is hard to pinpoint, but what we can say is that the regularization helps the learning algorithm to select a classifier, that a better can classify the linked samples correctly.

Although we see that regularization has an effect on the performance, it does not play as crucial a role, as the hyperparameters do for the other learning algorithms. Thus Logistic regression is less sensitive to changes in the hyperparameters when compared to the SVM and kNN.
The worst form of inequality is to try to make unequal things equal.

— Aristotle

As we have seen in section 7.5 the quality of the classification is very dependent on the way the samples are represented. In spite of this, it is a natural thing to ask whether we can improve the representation.

One idea could be to increase the separation of the linked and unlinked samples. For instance, if we look at Figure 26, we can see an area where a lot of linked and unlinked samples are clustered together. Thus, if we can separate the samples in this area, it should be easier to distinguish linked from unlinked, which will hopefully improve the classification.

![Figure 26: Shows how the linked and unlinked samples are clustered together.](image)

The way that we will try to attempt this is by applying a learning paradigm known as metric learning - a rather specialized subfield of machine learning.

First we will introduce the concept behind metric learning, and see how it can be utilized in pure distance based algorithms such as kNN. Next we will see how metric learning can be generalized to work with non-distance based algorithms such as SVM and Logistic regression. Lastly, we will look at a concrete metric learning algorithm known as Neighborhood components Analysis (NCA) by Goldberger et al. [15], and investigate whether this algorithm can help improve the classification experiment (section 7.5).

The theory in the first two sections of this chapter is mainly based on a metric learning survey by Kulis [18].
8.1 Metric Learning in a Nutshell

When using a learning algorithm like kNN, the classification is very depended on the distance metric that we choose to use. For instance, if we look at Figure 27, we can see that it would be better to choose a metric that gives more weight to the vertical direction than the horizontal-direction. However, this metric is not known a priori, but could maybe be inferred from the samples. In fact this is the whole idea behind metric learning.

![Figure 27: Euclidian distance (left) and the ‘optimal’ distance (right).](image)

The most common way to do this is by learning the so-called Mahalanobis distance metric, which can be thought of as a parameterized version of the Euclidian distance. Formally this metric is given by

\[ d_M = \sqrt{(x - x')^T M (x - x')} \]

where \( M \succeq 0 \). That is, \( M \) is a positive semidefinite (PSD) matrix (i.e. all its eigenvalues is non-negative), and captures the covariance of the data. As we will see, the PSD property of \( M \) will become handy later. If we set \( M \) equal to the identity then the metric simply reduces to the Euclidian distance. Thus the Mahalanobis distance can be seen as a generalized version of the Euclidean distance.

The aim is then to learn \( M \) such that the samples from the same class achieve a small distance. A problem like this can be formulated as an optimization problem in a variety of ways. For instance, in NCA it is formulated as an unconstrained maximization problem, where \( M \) is inferred, such that the Mahalanobis distance is maximizing the kNN accuracy. However, before going into the details of NCA, we will first look at how metric learning can be utilized for non-distance based algorithms like Logistic regression and SVM.

8.2 Similarity Transformation

Learning the Mahalanobis metric for algorithms like kNN, seems to be a reasonable idea. However, a better idea would be to just transform the feature space \( \mathcal{X} \), such that samples with the same class label get more contracted. Doing so would not only benefit distance based
algorithms like kNN, but would be useful to any classification algorithm, where it is valuable to have a good class separation.

Interestingly, this can actually be done by exploiting the fact that $M$ is a PSD matrix. For instance, one feature that the PSD property ensures, is that $M$ can be factorized into the following matrix product $M = A^T A$ (an Cholesky decomposition), where $A \in \mathbb{R}^{d \times d}$. The proof of this factorization will not be explained, but only assume its existence. As a consequence, we can derive the following expression

$$d_M(x, x') = \sqrt{(x - x')^T M (x - x')}$$

$$= \sqrt{(x - x')^T A^T A (x - x')}$$

$$= \sqrt{(A(x - x'))^T A (x - x')}$$

$$= \sqrt{(Ax - Ax')^T (Ax - Ax')}$$

which states that the Mahalanobis distance is equivalent to computing the euclidean distance after applying some linear transformation $A$ on the samples.

Thus if we could infer $A$, we could achieve a representation, where the linked and unlinked samples are better separated, as seen in Figure 28.

![Figure 28: The Mahalanobis distance in the original space (left) and the Euclidean distance in the transformed space (right).](image)

In the following, we will see how the linear transformation $A$ is inferred in NCA.

### 8.3 Neighbourhood Components Analysis (NCA)

NCA is based on the idea of finding the transformation $A$ such that the leave-one-out (LOO) error of kNN is minimized as much as possible. That is, project the training samples by $A$, and iteratively take a sample out and classify it using kNN. If we average the results we achieve the LOO error.

Unfortunately the LOO error is a piecewise function of $A$, that has a rather discontinuous shape. I.e. an infinitesimal change in $A$ can cause a finite change in the neighborhood. Due to this the LOO error
is hard to optimize, since we cannot rely on gradient based optimization (section 5.2.2). Alternatively a surrogate function of the LOO error could be used instead.

8.3.1 stochastic neighbourhood assignment

The suggestion in [15] is to use a function that is based on a stochastic neighborhood assignment in the transformed space. That is, instead of inspecting the k nearest neighbors in the transformed space, we inspect a set of transformed stochastic neighbors.

The notion of stochastic neighbors is based on a quantity $p_{ij}$, which describes the probability that sample $x_i$ assigns sample $x_j$ as its neighbor. Technically $p_{ij}$ is defined as the $softmax^1$ over the Euclidean distances in the space transformed by $A$, which can formally be written as

$$p_{ij} = \frac{e^{-d_{ij}^2}}{\sum_{k \neq i} e^{-d_{ik}^2}}$$  \hspace{1cm} (6)

Here $d_{ij} = \|Ax_i - Ax_j\|$ denotes the Euclidian distance after applying $A$ on the samples. Thus the probability $p_{ij}$ decreases exponentially with respect to the distance $d_{ij}$. Additionally $p_{ij}$ is defined to be 0 when $i = j$, which indicates that the probability that $x_i$ chooses itself as neighbor is 0. However, this is reasonable, since $x_i$ does not know anything about its label.

Based on this, the expected value for which $x_i$ is correctly classified, can be defined as

$$p_i = \sum_{j \in C_i} p_{ij}$$

where $C_i = \{j \mid y_i = y_j\}$. Thus, if $x_i$ and $x_j$ have the same label, then $p_{ij}$ will contribute to the expected probability of being correctly classified.

Intuitively the $p_i$s should be maximized as much as possible. Therefore the objective $f$ can be defined as a sum over all the $p_i$s

$$f(A) = \sum_{i} p_i$$  \hspace{1cm} (7)

which can be seen as the expected number of correctly classified samples. So maximizing $f(A)$ with respect to $A$ should be equivalent to minimizing the LOO error. Since each $p_{ij}$ is smooth and differentiable, the whole expression in 7 is differentiable. In spite of this, $f$ can be optimized by using gradient based optimization techniques.

---

1 A generalized logistic function, which is smooth and differentiable
(see section 5.2.2). To do this, we need to derive an analytically expression of the gradient, by taking the derivative of $f$ with respect to $A$. According to [?] this should be equal to

$$\frac{\partial \phi}{\partial A} = 2A \sum_i \left( p_i \sum_k p_{lk} - x_{lk}x_{lk}^T \right) - \sum_{j \in C_i} p_{ij}x_{ij}^T$$

(8)

where $x_{lk} = x_l - x_j$.

8.3.2 Overview of NCA

We now have all the ingredients settled in order to perform NCA. The following summarizes the overall steps of the algorithm

1. Initialize $A$ to some temporary solution
2. Calculate the distances $d_{ij}$ for all $i \neq j$
3. Calculate the ‘soft neighborhood’ $p_{ij}$
4. Calculate $p_i$ - i.e. the probability that sample $x_i$ will be classified correctly
5. Define $f(A)$ and $\frac{\partial}{\partial f(A)}$
6. Optimize $f(A)$ by using $\frac{\partial}{\partial f(A)}$

When $A$ is achieved, it can be applied on respectively $D_{train}$ and $D_{test}$, in order to obtain the effect in Figure 28. Alternatively, the Mahalanobis distance metric could be constructed by setting $M = A^TA$. However that would only be useful for pure distance based algorithms.

The whole learning setup can be seen in Figure 29, and illustrates how NCA is applied as a pre-processing step in order to achieve a better representation. Here $D_{metric}$ denotes a subset of the complete training set $D_{train}$.

Figure 29: The learning pipeline when NCA is used as a pre-processing step.
8.3.3 Strengths and weaknesses of NCA

The nice thing about NCA is that it assumes nothing about how the class labels are distributed. Also, it can work with any kind of gradient based optimization method, just like Logistic regression. Additionally, we can force it to perform dimensionality reduction by restricting $A$ to a non-square $d' \times d$ matrix, where $d' < d$. As a consequence the learned metric will have a low rank, and the transformed samples will exist in $\mathbb{R}^{d'}$. This is especially applicable when we are dealing data of very high dimensions. However, in our case we will not be exploiting this, since we at most have ten features.

The disadvantage of NCA is that the cost function in equation 7 is a non-convex function, which means that we cannot be guaranteed to find a global optimum, when the gradient vanishes. Aside from that, the algorithm can be quite time consuming, since the computation of the gradient (equation 8) is quadratic in the input - i.e. $O(n^2)$ where $n$ is the number of samples used. Thus it will be very impractical for large data sets. Alternatively, we should construct $A$ from a small representative training set.

8.4 NCA implementation

NCA has been implemented as an object that can be used by invoking a method called fit on a given training set. Doing so will initiate the optimizing of $f$ w.r.t. $A$. When the algorithm encounters a maximum, it terminates and stores the corresponding $A$, which can be applied on future data by invoking a method called transform.

To make the convergence time as fast as possible, the Scipy optimization module for Python have been utilized. The module is fairly simple to use, since it can be executed in only one line of code (see listening 8.4).

```python
result=opt.minimize(fun=self.cost, 
x0=self.A, 
jac=True, 
method='CG', 
options={'maxiter': 50} )
```

Listing 1: Scipy optimization module

The following list summarizes the parameters that are given to the module

- **fun** takes a function that returns $(f(A), \frac{\partial}{\partial f(A)})$ - or at least $f(A)$. Since the optimizer performs minimization, we provide the negative cost and the negative gradient, in order to turn it into a maximization problem.

- **x0** takes an initial solution for $A$. 
• \textit{jac=True} indicates that the gradient (aka. the Jacobian matrix) is served to the \textit{fun} parameter. Alternatively we would need to provide the gradient by a second call, which would slow down the performance, since the gradient and the cost function shares some computations.

• \textit{method} indicates the underlying optimization algorithm that is used - in this case the Conjugate gradient algorithm.

For the complete specification of how to use the module, the reader should consult \footnote{http://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html}. Furthermore the NCA implementation can be found in \footnote{NCAimpl.py}.

8.4.1 Initialization

Since the objective \( f \) is a non-convex function, the quality of the solution will highly depend on how \( A \) is initialized. For instance, if \( A \) is initialized badly, it could cause the optimization module to get stuck in a bad local maximum, and produce an unacceptable solution. To leverage the changes of this, multiple solutions are constructed by using different random seeds. The transformation that achieves the highest score is stored as the final solution \( A \).

8.4.2 Numerical problems

It has been discovered that NCA has some numerical issues related to the quantity \( p_{ij} \). For instance, if \( d_{ij} \) is very large it will cause \( p_{ij} \) to be rather small - remember \( p_{ij} \) drops exponentially with respect to the distance \( d_{ij} \). Therefore we could easily end up with a lot \( p_{ij} \)s that are zero, which means that they will not influence the solution. So how do we handle this?

Recall that \( d_{ij} = d_A(Ax_i, Ax_j) = \|Ax_i - Ax_j\| \), so a simple solution involves initializing \( A \) to have small numerical values. Consequently the \( p_{ij} \)s will increase.

Alternatively, we could scale the training samples to have zero mean and unit variance (standardization), since that will make the samples better clustered. This can be done by performing the following computation

\[
X = \frac{X - \text{mean}(X)}{\sqrt{\text{var}(X)}}
\]

as covered in section 5.3.6.

When we do this, we need to store \( \text{mean}(X) \) and \( \sqrt{\text{var}(X)} \) for future samples. Thus, when the method \textit{transform} is invoked on future samples, then they are first standardize by using \( \text{mean}(X) \) and
\(\sqrt{\text{var}[X]}\) and subsequently transformed by \(A\). Doing so, we ensure they have the same representation as the samples that \(A\) were constructed from.

It has further been discovered that standardization can help fix some scaling issues related to \(A\). The problem we were confronted with was that \(A\) had a tendency to scale one of the dimensions to an extremely low interval. However, by performing standardization we managed to remove artifact.

8.4.3 Performance

To see the performance of the algorithm, the actual running time has been plotted as a function of the number of samples used. The result can be seen in Figure 30, and illustrates that the performance of the algorithm is rather slow. Consequently, it is necessary to train the algorithm on a small representative training set.

![Figure 30: Performance of NCA.](image)

Additionally, it has been discovered that the implementation suffers from some memory issues, when the number of samples exceeds 16,000. One of the reasons for this is that the pairwise distances, \(d_{ij}\), are computed at once, and are stored into a two-dimensional array. Thus, if each entry in \(d_{ij}\) takes up 4 bytes of memory, then the complete distance matrix will consume

\[
16,000^2 \cdot 4 \approx 1\text{GB}.
\]

Along with the other data that NCA consumes, it seems to be lot of memory for an interpreted language like Python. However, since we only will use NCA for a small amount of samples, it has been chosen to ignore this problem.
8.4.4 Computational cost of the gradient

Next, it has been investigated whether the computational cost of the gradient respects the theoretical time complexity of $O(n^2)$, where $n$ denotes the number of samples used.

To do this, we take the actual CPU time $M(n)$, and divide it by the theoretical time complexity $T(n) = n^2$. If the expression converges towards a constant factor, then it indicates that the theoretical time complexity is respected. In Figure 31 we can see the plot of $M(n)/T(n)$, which shows that the expression converges nicely until it reaches 12,000 samples, then it begins to increase. The reason for this, is believed to be due to the large memory overhead, that is associated with the implementation.

![Figure 31: Actual running time divided by the theoretical running time, when computing the gradient in NCA.](image)

8.4.5 Verify NCA implementation

To verify the implementation, we have created a small test set consisting of eight samples, with three different labels: blue, red and orange, as seen in Figure 32. From the figure we can observe that the blue samples get squeezed together after A is applied. Thus we get exactly the effect we are aiming for, in this small scenario.

![Figure 32: Before (left) and after applying NCA on test samples (right).](image)
Next the effect that NCA has on the samples in $D$, has been visualized by using a PCA plot. The effect can be inspected in Figure 33, where the similarity vectors have been used as input representation.

![Figure 33: Before (left) and after (right) applying NCA on data set.](image)

As we can see, NCA indeed affects the samples in some way. However, the scale of the transformed samples has increased a lot, which makes it difficult to tell whether the clustering has improved.

As an alternative we can measure its effect by computing a score known as the silhouette coefficient. Roughly speaking, the silhouette coefficient is a measure between $-1$ and $1$ that reflects how well a given sample is assigned to its cluster. The higher the value is, the better the assignment is. For a detailed description of the measure, the reader should consult 4.

In Figure 34 and 35, we can see two plots that visualize the different silhouette coefficients before and after NCA is applied.

![Figure 34: The silhouette coefficients before applying NCA. The average silhouette coefficient is in this case $-0.021$.](image)

The dotted line illustrates the average silhouette coefficient, and gives an idea of how well the overall clustering is. Based on this, we can conclude that NCA actually provides the effect that we are aiming for. So the question that remains is whether it improves the classification.

8.5 Classification experiment with NCA

The result of the classification experiment after NCA has been applied can be seen in Figure 36. Compared to the results in Figure 17 from section 7.5, we can see that the classification actually has decreased, which was not intended. Especially the performance of kNN has decreased a lot, which is ironic, since NCA is constructed to minimize the LOO error of kNN.
Another problem could be that the algorithm gets stuck in a bad local maximum, as discussed in section 8.4.1 (Initialization). Using global search techniques like Simulated Annealing etc. is a common way to prevent this. On the other hand, techniques like this will just make the performance even worse.

Global Transformation

Lastly, it could be the case that the transformation $A$ is bad at separating local differences in our data set. For instance, if we have an unlinked sample $x$ that is closely related to a cluster of linked samples $C$, then applying $A$ could push $x$ even further into $C$, as depicted in Figure 37. A situation like this will of course have a negative effect on the classification, as it will make it even harder to distinguish linked from unlinked samples. Since the samples from group $O$ often are closely located to other samples in our feature space (see Figure 3 in section 3.1) this could actually be the case.

An alternative would be to use a transformation that is better at separating the samples on a local localities basis. Unfortunately such transformations are often non-linear and more difficult to obtain. We will discuss this in more detail in section 12.1. Additionally, we will look more at the $O$ samples in Chapter 10 and 11, and try to analyze their influence to see whether we can take advantage of them.

Figure 37: The side effect of using a linear transformation.
PROBABILITY ESTIMATION

We can’t stop here, this is bat country.
— Raoul Duke, Fear and Loathing in Las Vegas

Until now we have mainly been focused on how to solve the classification problem. Withal, having a good classification is not enough. We should also be able to provide a probability score that reflects how certain our classification is. For instance, the posterior probability \( p(y = \text{linked} \mid x) \).

As we have seen, Logistic regression directly provides an estimate of this. However, kNN and SVM only provide a categorical output that reflects whether a given sample is linked or unlinked. So instead they should be modified to provide a score that reflects how reliable their classification is.

For instance, regarding kNN, we could choose the following estimate

\[
p(y = \text{linked} \mid x) \approx \frac{|\text{kNN}(x)_{\text{linked}}|}{k}, \tag{9}
\]

as suggested in [3]. Here \( |\text{kNN}(x)_{\text{linked}}| \) describes the number of linked samples among the k-nearest neighbors of \( x \).

For SVM, we could try to base our measure on the distance from a sample \( x \) to the decision surface. For instance, the greater the distance is, the more reliable the classification is. Unfortunately, this distance measure is a relative quantity that can be arbitrarily large. Therefore we need to apply some sort of normalization, in order to convert the distances into a probabilistic range of \([0, 1]\). This could for instance be done by performing

\[
p(y = \text{linked} \mid x) \approx \frac{d(x) - d_{\text{min}}}{d_{\text{max}} - d_{\text{min}}}, \tag{10}
\]

where \( d(x) \) denotes the distance from the hyperplane to sample \( x \), and \( d_{\text{min}} \) and \( d_{\text{max}} \) denotes respectively the smallest and the largest distance to a sample. Doing so should return a score between 0 and 1 that reflects how reliable that the classification is.

So although kNN and SVM do not directly provides class probabilities, we can easily provide a score that approximates \( p(y = \text{linked} \mid x) \).

9.1 Calibration

Be that as it may, it does not mean these scores are good estimates for \( p(y = \text{linked} \mid x) \). Even for Logistic regression, we cannot guarantee
the output to be a good probability estimate. So what do we mean by that?

Assume that we have a subset $D' \subseteq D$ where our classifier $h$ predicts 0.3 for each sample $x \in D$, i.e. $D' = \{ x \in D : h(x) = 0.3 \}$. If the output of our classifier is reliable, then we would expect that 30% of the samples in $D'$ actually are linked (this is a reasonable assumption if our data set is large). If not, then our classifier outputs a bad probability value that needs to be rectified by performing some sort of mapping.

This process of mapping ‘bad’ probability estimates into ‘good’ probabilities is also what is known as calibration [23], and is performed as a post-processing step after the classifier has been constructed. The input to the calibration algorithm is a calibration set $D_{\text{cal}}$ on the form $\{(y_i, f_i)\}_{i=1}^n$, where $h(x_i) = f_i$ is a bad probability estimate. Based on this, the calibration algorithm outputs a model $g$, that better reflects the true fraction of linked samples. Thus, if we have a bunch of samples where $g$ outputs a value of 0.3, then we should expect to see that 30% of these samples actually are linked.

Usually the data set for training and the data set for calibration are two distinct sets, in order to avoid over-fitting. However, in our case we can just reuse the samples in $D_{\text{val}}$. Therefore there is no need for creating this additional calibration set in our situation.

In Figure 38 we see what the learning pipeline looks like when calibration is applied as a post-processing step.

In general the calibration algorithm works by performing some sort of regression over these ‘bad’ probability estimates (the $f_i$’s). During this project we have studied two specific methods known as respectively Platt Scaling and Isotonic regression. Both of them will be used since they can directly be accessed through the scikit-learn API. However, before going into how they work, we will first look at how to evaluate the probability estimates properly.

In the following sections we will use the shorthand notation $f_i$ to denote the uncalibrated score and $p_i$ to the calibrated output for a sample $x_i$. 

Figure 38: The learning pipeline when using calibration.
9.2 RELIABILITY DIAGRAM

A common way to evaluate the probability estimates, is by using what is known as a reliability diagram \cite{K"unz17}. In a reliability diagram the probability space is first discretized into \( k \) bins. E.g. if \( k = 10 \) then the space will be discretized into the intervals

\[
\{[0.0, 0.1], [0.1, 0.2], \ldots, [0.9, 1.0]\}
\]

Next, every single sample \( x \) is assigned to one of the bins, based on their probability estimate \( p(y = \text{linked} \mid x) \). So if an sample \( x \) achieves a probability between 0.0 and 0.1, it is assigned to the first bin, and so on. Subsequently, the mean probability of every single bin is calculated, by averaging the probabilities.

After that the mean probability is plotted against the true fraction of linked samples that exists in the corresponding bin. Doing so, we achieve a picture as in Figure 39, which reflects how reliable that the probability estimates are (hence the name). If the classifier is well-calibrated, then the mean probabilities should be equal to the true fraction of the linked samples. As a consequence the points will coincide with the diagonal of the diagram.

9.3 PLATT SCALING

Platt scaling builds on the idea of trying to fit a Logistic regression model to the score of a classifier. The method was originally applied to SVMs by Platt \cite{Platt99}, but has proved to work well for many other classifiers. The idea is to take an uncalibrated score \( f(x) \), and then pass it through a sigmoid function on the form

\[
P(y = 1 \mid x) \approx P_{A,B}(f(x)) = \frac{1}{1 + \exp(A \cdot f(x) + B)}
\]

The values \( A \) and \( B \) are model parameters that need to be estimated a priori. As we have seen in section 5.2, we can derive a cost function for \( A \) and \( B \) by performing maximum likelihood estimation. Doing so leads to the following expressing

\[
\arg\min_{A,B} - \sum_{i=1}^{n} (y_i \log(p_i) + (1 - y_i) \log(1 - p_i))
\]

where \( p_i = P_{A,B}(f(x_i)) \). By deriving an analytical expression of the gradient, we can estimate \( A \) and \( B \) by using gradient based optimization (section 5.2.2).
9.4 ISOTONIC REGRESSION

As an alternative to Platt scaling, we can perform isotonic regression, where the training set is fitted to an isotonic function. That is, a step-wise function that is monotonically increasing. Usually this is done by finding the function $z$ over the following optimization problem [23]

$$z = \arg\min_p \sum_{i=1}^{n} (y_i - p(f_i))^2$$

subject to $f_1 \leq f_2 \leq \cdots \leq f_n$

That is, the isotonic function that minimizes the mean squared difference between the true class labels and the calibrated probabilities. In literature, this cost is also denoted as the Brier cost function $^1$.

A simple way to find $z$ is by using the pool adjacent violators (PAV) algorithm [31]. The PAV algorithm works by first sorting the training samples $(f_i, y_i)$ with respect to the $f_i$’s. Next the algorithm assigns a probability value of 1 to every positive sample and a probability value of 0 to every negative sample. That is, if $y_i = 1$ we set $p_i = 1$ and if $y_i = 0$ we will set $p_i = 0$. Aside from that, it assigns each individual sample to its own group.

Afterwards the algorithm works by iteratively inspecting whether the monotonicity constraint is violated (i.e. a pair of groups where the probability values are locally decreasing). If so, the groups are pooled into a single group, and their probabilities a averaged together. However, averaging the probabilities could itself introduce a violation of the monotonicity constraint. Therefore the algorithm needs to backtrack and search for places where this must be corrected by pooling and averaging. It then continues doing this until a sequence of monotonically increasing probabilities have been reached. To see a concrete example of this, the reader should consult Appendix A.2.

After the $p_i$’s has been found, they can be used as a basis to from a isotonic function $z$. A suggestion in [19], is to use the following expression

$$z(f_i) = \begin{cases} 
p_1 & \text{if } f_i \leq p_1 \\
p_i & \text{if } f_i = p_i \\
p_i + p_{i+1}/2 & \text{if } p_i < f_i < p_{i+1} \\
p_n & \text{if } p_n \leq f_i 
\end{cases}$$

The PAV algorithm has a time complexity of $O(n^2)$, due to its need for backtracking. As an alternative, there exists a linear time algorithm, known as the Active set algorithm (the one that is used in

$^1$ https://en.wikipedia.org/wiki/Brier_score
scikit-learn\(^2\)), which works on the dual representation of the optimization problem. However, this thesis will not discuss the details of how it works, since it is fairly technical. Instead we will focus on the practical effect of performing isotonic regression and try to compare it against Platt Scaling.

### 9.5 Probability Experiment

In order to calibrate the classifiers, the calibration module in scikit-learn has been utilized. Essentially the module is just a simple wrapper that can be used around a given classifier. The wrapper takes the following parameters:

- an base estimator (i.e. SVM, kNN, etc.)
- an validation strategy (explained below)
- a parameter that determines the calibration technique (i.e. isotonic or sigmoid)

Next the classifier is calibrated by invoking a procedure called \texttt{fit}, which takes a calibration set as input. Listing 9.5 shows how to setup the calibration module in scikit-learn.

```python
from sklearn.calibration import CalibratedClassifierCV

X_cal, y_cal, X_train, y_train, X_test, y_test = cal_train_test_split(X, y)
...
% perfrom grid searching on X.cal
% train classifier on X.train
...

isotonic = CalibratedClassifierCV(est, cv='prefit', method='isotonic'
)
isotonic.fit(X.cal, y.cal.ravel())

sigmoid = CalibratedClassifierCV(est, cv='prefit', method='sigmoid')
sigmoid.fit(X.cal, y.cal.ravel())

classifiers = [isotonic, sigmoid]
for clf in classifiers:
    y_pred = clf.predict(X.test)
    prob_pos = clf.predict_proba(X.test)[:, 1]
    ...
% evaluate
```

Listing 2: Setting up the calibration wrapper in scikit-learn

The validation strategy can either be set to \texttt{prefit} or to an integer. If \texttt{prefit} is selected then the calibration module will use the entire data

\(^2\) \url{http://scikit-learn.org/stable/modules/generated/sklearn.isotonic.IsotonicRegression.html} - see references
set to perform calibration on. As a consequence, the wrapper expects the base estimator to be trained beforehand.

Conversely, if an integer \( k \) is given, then the calibration module will perform \( k \)-fold cross-validation, in order to both train and calibrate the classifier. I.e. it will use the training folds for training, and the test folds for calibration. Next the probability estimates are averaged for predictions.

As Listing 9.5 indicates, we have chosen to use the prefit approach, since we want to reuse the same samples in \( D_{val} \) (corresponds to \( X_{cal}, y_{cal} \) in Listing 9.5). After the calibration has been performed, we use \( D_{test} \) to predict probabilities on. The estimated probabilities can be used to create a reliability diagram, in order to investigate the quality of the calibration.

9.6 RESULTS

In order to reduce the number of results, we have restricted our attention to only Logistic regression, SVM with RBF kernel and kNN. Additionally we have only used the similarity vectors (section 3.6) as input representation. Besides that, it has been necessary to remove some of the samples from group O, in order to achieve reasonable results. From this, the goal is to investigate whether Platt scaling and Isotonic regression can make the probability estimates more reliable.

The reliability diagrams of the three models can be seen in respectively Figure 40, 41 and 42. For each curve, we have associated the corresponding Brier score, as it reflects how far the predicted probability estimates are from the true labels. Since we have a lot of unlinked samples with a probability estimate around 0, the scores are generally somewhat low.

If we look at Figure 40 we can see the reliability diagram of the model produced by SVM. The diagram illustrates, that the model benefits a lot from both Platt scaling and Isotonic regression, since their calibration curves are squeezed a lot towards the diagonal. Conversely, when the model is uncalibrated, we see that the curve almost has a sigmoidal shape, which indicates that the linear scaling in equation 10 gives very unreliable probability estimates. In fact, this tendency is not unusual for large margin classifiers like SVM, as noted in [23]. Fortunately, by performing calibration the probability estimates get pushed towards the extreme values, and gives a well-calibrated result.
In Figure 41 we can see the reliability diagram of the model produced by Logistic regression. The figure shows that the model already gives well-calibrated probability estimates, and does not benefit from either Isotonic regression nor Platt scaling. The reason for this is that Logistic regression directly approximates $p(y = \text{linked} | x)$ by optimizing the cross entropy error (equation 15, section 5.2). Therefore it generally has a tendency to provide reliable probability estimates.

In the last figure, Figure 42, we can see the reliability diagram of the model produced by kNN. Like Logistic regression, this model also provides very reliable probability estimates, which indicates that the expression in equation 9 is reasonable estimate for $p(\text{linked} | x)$. Therefore the only model that really benefits from calibration, is the model produced by SVM.
As pointed out earlier, it has been difficult to achieve reasonable results when all the samples from group O are included in the data set. Consequently it have been necessary to remove some of them before initiating the calibration module. On the other hand, if we completely omitted all these O samples, then we would automatically achieve a perfect calibration, and could therefore not demonstrate it’s effect. Therefore, the results should be considered more as a proof of concept.

Figure 42: Calibration of kNN.
I have no idea where this will lead us.
— Dale Cooper, Twin Peaks

If we look at $D_{org}$ it contains this extra group of samples with the label $O$, which originally has been included in order to stress the classification. However, the problem with these samples is that their manufacturer is essentially unknown. So although they are said to be unlinked from everything else, they could potentially be linked.

As we have seen, when two samples are linked, it is often reflected by the similarity that exists in their features. As a consequence, they have a tendency to be very clustered in our feature space $X$. Thus, if $x$ denotes a sample from group $O$ that is closely clustered to other samples in $X$, then it is very likely that $x$ is linked to these samples. In this case $x$ is also said to be an inlier, which is a notion that is often used in clustering methods.

The problem with having these inliers from group $O$, is that they will very likely cause a bad influence on both the training and the evaluation, since they are likely to be linked. In other words, they could be providing false information. On the other hand they are necessary to include, in order to show that the solvents can contribute to a better classification, and in order to give the classifier a conservative behavior.

In this chapter we will try to determine the inliers of $D_{org}$, in order to identify which samples that are influencing the classification. The way this will be done is by computing a score known as the local outlier factor (LOF) by Breunig et al. [5]. Originally the LOF score was intended for outlier detection, in order to detect anomalies that exist in the data set (hence the name). In spite of this, it can be used to distinguish inliers from outliers.
10.1 Local Outlier Factor (LOF)

The LOF is based on the idea of assigning a density score to each sample in the data set. From this the following is inspected: if the density of \( x \) is slightly lower than the density of its neighbors, then \( x \) should be thought of as being an outlier. Conversely, if its density is smaller than or equal to its neighbor’s, it should be thought of as an inlier.

As we will see the notion of density is based on a measure known as the reachability distance \( r_{d_k} \), which is also used in density based clustering algorithms such as DBSCAN \(^{[13]} \), OPTICS \(^{[2]} \), etc. Formally \( r_{d_k} \) is given by

\[
r_{d_k}(x, x') = \max\{kdist(x'), d_{eu}(x, x')\}
\]

where \( kdist \) defines the distance to the \( k \)th nearest neighbor of \( x \) and \( d_{eu}(x, x') \) defines the euclidean distance between \( x \) and \( x' \). Thus, if \( x \) and \( x' \) are close to each other, then the \( k \)-distance of \( x' \) will be used instead of the direct distance. The reason for incorporating the \( k \)-distance is to smooth out the densities in the neighborhood and to get a more stable result. Essentially the amount of smoothing is controlled by the \( k \) parameter, which reflects the size of the neighborhood that we want to inspect.

The intuition behind the reachability distance is visualized in Figure 43, where \( k = 3 \). Here we see that both \( p, s \) and \( t \) has a reachability distance to \( x \) that is equal to the \( k \)-distance, since they are all part of the \( k \) nearest neighbors of \( x \). Conversely, the reachability distance from \( q \) to \( x \) is not equal to the \( k \)-distance, since it is not included in the neighborhood.

Although we mention \( r_{d_k} \) as a distance function, it should be noted that it is not a distance in the mathematical sense, since it does not satisfy the property of being symmetric. Having said this, the LOF score can be computed by using the following steps:

**Step 1 - Identify the Neighbourhood**

First the neighborhood is identified by performing kNN for each sample in the data set. The identified neighbours are stored into a two-dimensional array, such that they efficiently can be reclaimed.
**Step 2 - Density Estimation**

For each sample \(x\), we calculate the local reachability density \(lrd_k\), given by

\[
lrd_k(x) = \left( \sum_{x' \in kNN(x)} rd_k(x, x') \right)^{-1}
\]

That is, the inverse of the average reachability distance of the \(k\)-nearest neighbors of a sample \(x\). If \(x\) is fairly close to its neighbors then \(lrd_k\) will be large, and therefore indicate a rather dense neighborhood. Hence the density measure reflects how reachable the neighboring samples are.

**Step 3 - Density Comparison**

Based on the measures above, we can compute the LOF score for each sample in the data set. This is formally achieved by using the formula

\[
lof_k(x) = \frac{\sum_{x' \in kNN(x)} lrd_k(x')}{|kNN(x)|}
\]

That is, the average \(lrd_k\) of the neighbors relatively to the \(lrd_k\) of \(x\). The higher the \(lrd\) is around the neighbors of \(x\), the larger the LOF factor will get for \(x\).

If the LOF is around 1, it will indicate that \(x\)’s density measure is very similar to its neighbor’s. If \(\text{LOF}(x)_{k} \leq 1\) it should be reasonable to mark \(x\) as an inlier. Conversely, if \(\text{LOF}(x)_{k}\) is slightly above 1, then \(x\) should be assumed to be an outlier.

The algorithm above can easily be implemented on top of scikit-learn’s kNN algorithm, which was the approach chosen for this thesis.

### 10.2 Applying LOF

In this section we will apply the LOF procedure on the samples in \(D_{org}\) and investigate the influence that the neighborhood parameter \(k\) has on the LOF scores. In Figure 44 we can see the result when only the alkaloids are used and in Figure 45 we can see the effect when using both the alkaloids and the solvents. The black stick on each bar represent a 0.95 confidence interval of the LOF scores.

If \(k\) is between 20 and 35, when only the alkaloids are used, then the LOF scores for group A, B, C, D and E are rather low and consistent. In other words, the samples in these groups appear as inliers. However, when \(k\) exceeds a value of 40, it becomes difficult to distinguish an inlier from an outlier, due to the fact that the inspected neighborhood is too large. On the other hand, if \(k\) is below 10 then
the LOF scores begin to fluctuate. This fluctuation is in fact a known problem that is mentioned in [5], and appears in some cases when $k$ is too low.

When both the alkaloids and solvents are used, we can see that the samples in group A, B, C, D and E appear as inliers for all values of $k$. Additionally, when $k$ ranges from 6 to 35, we can see that the average LOF score for group O is generally higher than in Figure 44. In other words, the solvents help to make the samples in group A, B, C, D and E better clustered, and the samples in group O more separated.
10.3 REMOVING INLIERS

In this section, we will use the LOF scores to remove the inliers from group O, and investigate the effect it has on the classification. To do this, we need to fix the neighborhood parameter $k$ and set an appropriate threshold $\tau$.

Ideally $k$ should be chosen such that all the samples in group $A, B, C, D$ and $E$ appear as inliers and the samples in group $O$ appear as outliers. Based on the plot in Figure 44, we will set $k = 10$, since it gives a fairly high average LOF score for group $O$, and a rather low average LOF score for group $A, B, C, D$ and $E$.

Next we need to define the threshold $\tau$ that will be used to distinguish an inlier from an outlier. To do this, a histogram in Figure 46 that shows how the various LOF scores are distributed in group $O$ for $k = 10$, has been created.

![Figure 46: The LOF scores for the samples in group O when $k = 10$.](image)

Based the histogram we can tell that there exists around 70 samples with a LOF score between 1.0 and 1.5. Therefore we should expect to see an improvement, if we set $\tau$ within this interval. The effect of increasing $\tau$ from 1.1 to 1.5 is visualized in Figure 47.

![Figure 47: PCA plot after applying LOF on the original data set.](image)

(a) $\tau$ set to 1.1.

(b) $\tau$ set to 1.5.
where the inliers (I) are the samples that appear in black. From this, we can clearly see how the densely clustered O samples are identified as the threshold is increased. Also when $\tau$ is around 1.5, it seems that most of the inliers from group O are captured. Due to this, we will fix $\tau = 1.5$ and remove every sample in group O that has a LOF score below this threshold. The result from doing this, can be seen in table 3, where a SVM with a RBF kernel and the similarity vector as input representation have been used.

<table>
<thead>
<tr>
<th>TEST</th>
<th>FP</th>
<th>FN</th>
<th>TP</th>
<th>TN</th>
<th>PRECESSION</th>
<th>RECALL</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>299</td>
<td>330</td>
<td>1.0</td>
<td>0.996</td>
<td>0.998</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>300</td>
<td>295</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>300</td>
<td>295</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>5</td>
<td>295</td>
<td>295</td>
<td>1.0</td>
<td>0.983</td>
<td>0.991</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>298</td>
<td>294</td>
<td>0.996</td>
<td>0.993</td>
<td>0.994</td>
</tr>
</tbody>
</table>

Table 3: SVM RBF after applying LOF.

As we can see, the result is almost perfect, and is very compatible with the result in table 1, where the solvents have been used. In conclusion we can say that the quality of the classification indeed depends on the amount of inliers that exist in group O.
This is not a good town for psychedelic drugs. Reality itself is too twisted.

— Raoul Duke, *Fear and Loathing in Las Vegas*

As we have seen, the assumption that the samples are unlinked is simply too harsh, since they potentially could have a relation to other samples. Due to this, it would be more reasonable to change the underlying assumption about them. For instance, marking them as unlabeled samples would make more sense, since the production that they originate from, essentially is unknown to us. Based on this, we could try and adapt a machine learning paradigm that can take advantage of both labeled and unlabeled data. Such a paradigm is also what is known as semi-supervised learning, and overlaps with both supervised and unsupervised learning.

A variety of techniques exist in this field, but an interesting one is PU-learning (positive unlabeled learning), where a binary classifier is learned from only positive an unlabeled samples (hence the name). PU-learning is intriguing, because it only makes distributional assumptions about the positive samples in our data set. Additionally it can easily be wrapped around our existing classifiers, as we will see.

The technique has especially showed promising results in areas of bioinformatics that focus on identifying disease genes from the human genome [25]. Here the input is a set of genes with a known disease relation (i.e. the positive samples) and a set of genes where it is unknown whether they have a disease relation or not (i.e. the unlabeled samples).

Since our basis is essentially the same, it could be interesting to adapt this approach, and investigate whether it can contribute to a better classification. More concretely, we will try to investigate questions like:

- Can PU-learning be used to improve the classification.

- Can the underlying problem be learned from only a small amount of positive samples and a large amount of unlabeled samples. If so, then it could reduce the effort of getting labeled data.
11.1 PU-LEARNING

PU-learning has been discussed in a variety of papers. During this thesis we have looked at the work by Elkan et al. [11], where the goal is to infer a probabilistic classifier \( f = p(y = 1 \mid x) \) from both positive and unlabeled samples. As we have seen, \( f \) can easily be learned by using both positive and negative samples on the form \((x, y)\). In [11] this concept is slightly changed as each sample should be seen as a triple on the form \((x, y, s)\), which has been drawn from a probability distribution \( p(x, y, s) \). However, although a sample consists of three components, it is only the information about \( x \) and \( s \) that has been recorded.

The introduced variable \( s \) is a binary variable that denotes whether a sample is either positive or unlabeled. That is, if \( s = 1 \) then we know for sure that \( y = 1 \). Conversely, when \( s = 0 \), we do not know whether \( y = 1 \) or \( y = 0 \). Based on the value of \( s \), we can partition \( D \) into the following sets

- \( \mathcal{P} \): A set of positively labeled samples (i.e. samples where \( s = 1 \)).
- \( \mathcal{U} \): A set of unlabeled samples (i.e. samples where \( s = 0 \)).

In our case the set \( \mathcal{U} \) consists of a mixture of both linked and unlinked samples.

The idea is then to first to construct a classifier \( g \) that approximates \( p(s = 1 \mid x) \), i.e. the probability for being labeled. This classifier is also what in [11] is noted as a nontraditional classifier. Next the goal is to obtain our traditional classifier \( f \) from this nontraditional classifier \( g \).

At first this seems like an impossible task. However, by making some underlying assumption about our samples, it can be realized.

11.1.1 Selected completely at random assumption

The assumption that is made in [11] about the samples is the following: the probability that a positive sample is labeled is the same for all positive samples. Formally this can be expressed as

\[
p(s = 1 \mid y = 1, x) = p(s = 1 \mid y = 1) = c,
\]

where \( c \) is the constant probability for being labeled, which is associated to every positive sample. Thus the probability for being labeled will be independent of \( x \) and only depend on \( y = 1 \).

This can also be viewed as the positive samples being chosen completely at random. Due to this, the assumption is also what is denoted as the “selected completely at random” (SCR) assumption in [11].
Having established this, we can formally derive that

\[
p(s = 1|x) = p(y = 1 \land s = 1 \mid x) \\
= p(y = 1 \mid x)p(s = 1 \mid y = 1, x) \\
= p(y = 1 \mid x)c
\]

The first line follows from the fact that only positive samples are labeled. The second line follows from the chain rule of probabilities. And the third line is obtained by applying the SCR assumption. If we further divide by the constant \(c\) we get that

\[
p(y = 1 \mid x) = \frac{p(s = 1 | x)}{c}
\]  
(13)

That is \(p(y = 1 \mid x)\) and \(p(s = 1 | x)\) only differ by a constant factor. The proof above is rather simple and follows almost naturally from the assumption. Despite this, it has first been formalized in [11].

The consequence of this is that our traditional classifier \(f\) and can be approximated if both \(g\) and \(c\) are known. As we have discussed, \(g\) is just the nontraditional classifier that can be obtained from training positive samples against unlabeled samples. Therefore we just need some way that we can determine the constant \(c\).

In the next subsection we will see how \(c\) can be estimated, in order to obtain \(f\).

11.1.2 Estimating \(c\)

The suggestion in [11] is to utilize the function \(g\) and a validation set \(V\) in order to estimate \(c\). Based on this, we can construct an estimator \(e\) by using the formula

\[
e = \frac{1}{|P_V|} \sum_{x \in P_V} g(x)
\]  
(14)

where \(P_V\) denotes the set of positive samples in \(V\).

To see why this is reasonable, we will let \(x\) denote an arbitrarily positive sample, and then apply \(g\) on \(x\). If we do so, we get the following result

\[
g(x) = p(s = 1 \mid x) \\
= p(s \mid x, y = 1) \\
= c
\]

The first line follows by the definition of \(g\). The second line follows from the fact that \(x\) is positive. The third line is obtained by applying the SCR assumption.

Thus, if \(g(x) = p(s = 1 | x)\) holds for all values of \(x\), then the estimator in 14 should be precise. Of course this is never true in practice,
since $g$ is learned from both a finite and noisy training set. But by averaging the result, we can reduce the variance that is associated with the estimator.

After $e$ has been constructed, we can use it to approximate $f$ by

$$f(x) \approx \frac{g(x)}{e}$$

### 11.2 Implementation

PU-learning have been implemented as a simple wrapper around a traditional scikit-learn algorithm, very much like the wrapper used for calibration (section 9.5). The way it works can be summarized by the following steps

1. First a basis algorithm (e.g. SVM, kNN, etc.) is given as input to the wrapper, which is responsible for determining $g = p(s = 1 \mid x)$.

2. To construct $g$ and $c$, a method called `fit` is invoked on the training set. The procedure first extracts a small fraction of linked samples in order to create the validation set for $c$. Next the basis algorithm is initiated to construct $g$. When $g$ is constructed we can use it to create an estimate of $c$ by using equation 14.

3. Lastly, we can obtain our probabilistic classifier $f$. To use it for classification, we need to set a threshold (e.g. 0.5) to distinguish a linked from a unlinked sample.

The implementation is straightforward and can be found in 1. In the following section, we will investigate its effect in practice.

### 11.3 Classify with PU-learning

First we applied PU-learning on our classification experiment from chapter 7. Sadly it did not show any improvement at all. Instead a small experiment was made, to verify that the implementation works as intended. The way this is done is by iteratively marking a group of linked samples as unlinked, and subsequently record the performance.

$D_{\text{train}}$ consists of 20018 unlinked samples and 1421 linked samples. At each iteration we sacrifice 100 linked samples in $D_{\text{train}}$. I.e. we mark them as unlinked. The result from doing so, can be seen in Figure 48. Note that a traditional train-test split approach has been used during this experiment.

---

1 `pu_wrapper.py`
Figure 48: PU-learning vs. supervised learning.
The effect is clearly observed. The performance of the regular classifiers begins to drop after a single iteration, whereas the PU-classifiers are more stable against linked samples that are marked as unlinked. Especially the model produced by Logistic regression is very stable when PU-learning is applied.

In conclusion we could say that PU-learning could potentially have an effect on the classification, and that we indeed can take advantage of unlabeled samples. However, the experiment shows that it only gives an advantage in situations where the number of linked samples is rather small and the number of unlinked samples is large.
CONCLUSION

Cocaine is a hell of a drug.
— Rick James

The main focus of the thesis has been to create a classifier that is capable of determining whether two cocaine samples originate from the same manufacturer. The problem was first approached using classical algorithms within supervised machine learning. Later the traditional leaning pipeline was extended, by using metric learning as a pre-processing step and calibration techniques as a post-processing step. Additionally, it was investigated how the classification potentially could be improved by using semi-supervised learning techniques such as PU-learning.

However, as we have seen, some of the techniques have worked better than others. Nonetheless we have tried to clarify and motivate the reason for using them.

The main points and achievements of the thesis have been summarized below:

REPRESENTATIONS
Initially different representations for reflecting whether two cocaine samples are linked or unlinked, was investigated. As discussed in chapter 3, this boils down to determining whether two cocaine profiles have a similar distribution or not. The first representation we looked at was a pure distance based solution for reflecting this similarity. The results from these distance functions were good, and gave an average $F_1$ score around 0.95 (section 7.5). However, by using similarity vectors (section 3.6) the performance slightly improved. Especially when using both kNN and SVM with a RBF kernel we saw that it was possible to achieve an average $F_1$ score around 0.98.

EVALUATION
It was justified how a classifier should be evaluated using a variant of k-fold cross validation that was noted as LGO cross validation (section 4.3). The motivation for using this variant is to ensure that the classifier can work on samples that originate from an unknown manufacturer. Additionally, it was covered why the $F_1$ measure is a well-suited evaluation metric for this problem.

LEARNING ALGORITHMS
It was shown that the most advanced learning algorithm does
not necessarily provide the best classification. In fact, when using similarity vectors as input representation, the performance of kNN and SVM with a RBF kernel was the same. Intuitively, this also makes sense, since the problem we are dealing with is a rather low dimensional problem. Therefore there is no motivation for using sophisticated techniques like non-linear SVMs, as long as we have a representation that reflects the underlying problem well. This result is also very consistent with Scott Locklin’s point, which we cited in the beginning of chapter 3.

**Metric Learning**

Initially metric learning seemed to be a reasonable idea for improving the classification. The aim of it was to achieve a linear transformation $A$ that could better separate linked from unlinked samples. As we saw in section 8.4.5, the clustering seemed to improve, but it did not increase the quality of the classification. In fact, it got worse.

The reason for this is believed to be due to the fact that some of the samples from group O is located too close to the other samples in the data set. So by using a linear transformation they could be pushed even further into these samples (see section 8.5). Instead it would be better to learn a nonlinear transformation that does not work in a global manner like a linear transformation does.

**Calibration**

In this chapter it was discussed how categorical classifiers produced by SVM and kNN can be turned into probabilistic classifiers and provide an estimate of $p(y = \text{linked} \mid x)$. Additionally different calibration techniques, such as Platt scaling and Isotonic regression was covered, in order to map ‘bad’ probability estimates into ‘good’ ones. Sadly, the calibration showed to have very little effect in practice. Also, it was necessary to remove some of the O samples from the data set, in order to show its impact.

**PU Learning**

PU learning had initially no effect on the classification. Nonetheless, it was shown that PU-classifiers are far more stable than regular classifiers, in scenarios where we have a low number of linked samples and a large number of unlinked samples that essentially are linked. Especially when using Logistic regression in conjunction with PU learning, the classifiers became remarkably stable against these ‘hidden’ linked samples. Therefore, there indeed is a motivation for using this learning paradigm in a scenario like this, where we potentially have a lot of unlinked samples that are linked.
PERFECT CLASSIFICATION - A ROPE OF SAND

The aim of the project was to achieve the same result as was achieved at BiRC, but only by using the alkaloids. As was shown in section 7.3 the solvents help to make the relationship between the drugs very clear, such that it becomes much easier to distinguish linked from unlinked. By using Logistic regression and the cosine distance as representation, it was possible to get a near perfect classification, when both alkaloids and solvents are included in the samples.

However, as experienced with the data set, it seems to be utmost suspicious that this is possible to achieve by solely using the alkaloids. When using the alkaloids each sample only consists of 10 features, which make the samples more clustered in our feature space. Consequently, a lot of the unlinked samples from group O will coincide with the other samples in the data set and put much more stress on the classification. Therefore, it seems to be a rather difficult goal to achieve, without using the solvents.

12.1 FUTURE WORK

Based on the experience of this project it could be interesting to explore some other techniques for creating a good similarity representation. Following is a list containing some suggestions that have shown promising results.

12.1.1 Siamese neural network

Siamese neural network was originally proposed in 1994 by LeCun et al. [6] as a solution to signature verification, and later reintroduced in 2005, where it was improved for face verification [8].

As depicted in Figure 49, the architecture consists of two feed forward neural networks. These networks take respectively $X_1$ and $X_2$ as input and map them into a representation that is better at capturing similarities. Technically the aim of the network is to create a function $G$, parameterized by the weights $W$, which map $X_1$ and $X_2$ in order to achieve a

- small distance, when they belong to the same category.
- large distance, when they belong to different categories.

![Figure 49: Siamese architecture - conceptual drawing from [8].](image-url)
This was achieved by formulating what the authors note as a contrastive loss function $E$, which can be found in [8]. Additionally, the weights are kept synchronized between the two neural networks, by first performing standard forward/backward propagation on each network, and then averaging the weights before calculating the gradients. Doing so ensures that the function $G$ becomes a symmetric function.

In contrast to methods like metric learning, this approach is capable of learning a nonlinear mapping, that is far more general than a linear transformation. Thus it could potentially have an improvement on the problem we are dealing with. The only thing that speaks against this approach, is that it originally was intended for high dimensional data like images etc. Therefore it is hard to say if it actually is applicable to our problem.

### 12.1.2 Local based metric learning

As we have discussed, the problem with metric learning is that there is no guarantee that the transformed samples will be better separated locally. The reason for this is that the linear transformation is learned from the complete data set, which gives it a rather global behavior. Instead it would be better if we learned a linear transformation on a more local basis.

This idea has in fact been exploited in papers like [32] by Kilian et al. Here the space is first partition into multiple regions by using a clustering algorithm like k-means. Next a linear transformation is learned for each region in the space. Finally they transform the samples based on the region they are assigned to. The result is that the samples will be better separated locally. In spite of this, it could be applicable in our case, and help to improve the classification.
A.1 LOGISTIC REGRESSION

In this section we will derive the partial derivative of the cross entropy error from section 15, which is defined as

\[ l(w) = - \sum_{i=1}^{n} y_i \log(h(x_i)) + (1 - y_i) \log(1 - h(x_i)) \]  

(15)

The first term in equation 15 can be rewritten into

\[
\log(h(x_i)) = \log(\sigma(w^T x_i)) \\
= \log\left(\frac{1}{1 + e^{-w^T x_i}}\right) \\
= -\log(1 + e^{-w^T x})
\]

whereas the second term can be written as

\[
\log(1 - h(x_i)) = \log(1 - \sigma(w^T x_i)) \\
= \log\left(1 - \frac{1}{1 + e^{w^T x_i}}\right) \\
= \log\left(\frac{1 + e^{-w^T x_i}}{1 + e^{-w^T x_i}} - 1\right) \\
= \log\left(\frac{e^{-w^T x_i}}{1 + e^{-w^T x_i}}\right) \\
= \log(e^{-w^T x_i}) - \log(1 + e^{-w^T x_i}) \\
= -w^T x_i - \log(1 + e^{-w^T x_i})
\]

Due to this, we get that

\[ l(w) = - \sum_{i=1}^{n} -y \cdot \log(1 + e^{-w^T x_i}) + (1 - y)(-w^T x_i - \log(1 + e^{-w^T x_i})) \]

\[ = - \sum_{i=1}^{n} y \cdot w^T x_i + w^T x_i - \log(1 + e^{-w^T x_i}) \]

Furthermore we have that

\[ -w^T x - \log(1 + e^{-w^T x_i}) = -(w^T x_i + \log(1 + e^{-w^T x_i})) \]

\[ = -(\log(e^{-w^T x_i}) + \log(1 + e^{-w^T x_i})) \]

\[ = -\log(1 + e^{-w^T x_i}) \]
where the third line follows from the fact that
\[
\log(a) + \log(b) = \log(a \cdot b)
\]

Therefore we get
\[
l(w) = -\sum_{i=1}^{n} y_i \cdot w^T x_i - w^T x_i - \log(1 + e^{-w^T x_i})
\]
\[
= -\sum_{i=1}^{n} y_i \cdot w^T x_i - \log(1 + e^{-w^T x_i})
\]

Next we can compute the partial derivates
\[
\frac{\partial}{\partial w_j} y_i w^T x_i = y_i x_i(j)
\]
\[
\frac{\partial}{\partial w_j} \log(1 + e^{w^T x_i}) = \frac{x_i(j) e^{w^T x_i}}{1 + e^{w^T x_i}} = x_i(j) h(x_i)
\]

where \(x_i(j)\) comes from the fact that
\[
\frac{\partial}{\partial w_j} w^T x_i = \frac{\partial}{\partial w_j} (w(0) + w(1)x_i(1) + \cdots + w(d)x_i(d))
\]
\[
= x_i(j)
\]

Finally we end up with
\[
\frac{\partial}{\partial w_j} l(w) = -\sum_{i=1}^{n} y_i x_i(j) - x_i(j) h(x_i)
\]
\[
= \sum_{i=1}^{n} (h(x_i) - y_i) x_i(j)
\]

A.2 PAV EXAMPLE

To understand the PAV algorithm properly, we have included the following example, where it is applied on 14 test samples (see Table 4). In the initial step, denoted as \(S_0\), the algorithm assigns a probability of 1 to the positive samples and a probability of 0 to the negative samples. Next, the algorithm starts iterating until a violation is met, which happens at sample 13 and 12. As a consequence, the groups are pooled together and assigned the average probability of 1/2. The algorithm continues doing this until the end is reached. However, in \(S_4\) we can see that the average introduces a new violation, between sample 8 and the group formed by sample 7 and 6. Due to this, the algorithm needs to back track, in order to fix this violation.
### A.3 Precision Recall Curves

The following shows the precision and the recall (PR) when varying the threshold of the different classifiers. NB: during this experiment a standard stratified 5-fold cross validation strategy have been used.

![Figure 50: PR-curves for Logistic regression.](image)

(a) Cosine distance. 

(b) Similarity vector.
Figure 51: PR-curves for SVM RBF.

Figure 52: PR-curves for kNN.
BIBLIOGRAPHY


[25] Jian-Ping Mei1-Chee-Keong Kwoh1 Peng Yang1, Xiao-Li Li2 and See-Kiong Ng2. Bioinformatics.


[34] Alice Zheng. How to evaluate machine learning models, 2015. "[Online; accessed 15-march-2016]".