FACULTY OF ENGINEERING, LTH

CENTRE FOR MATHEMATICAL SCIENCES

MASTER THESIS

Fingerprint Matching using Small Sensors

RIKARD DRUGGE
tpi11rdr@student.lu.se

Supervisor

MAGNUS OSKARSSON
magnuso@maths.lth.se

Examiner

ANDERS HEYDEN
anders.heyden@math.lth.se

April 2017

Abstract

Fingerprint software is widely used in applications such as smartphones. There has to be some overlap between two fingerprint samples in order for them to be considered as belonging to the same finger. Determining if this overlap exists is normally not a problem if a fingerprint sensor is big enough to capture a sample from the whole finger at once. However, if the sensor is small it might capture samples from different parts of the same finger such that there is no overlap. To find a predictor one needs a training set but constructing the set using small sensors lead to a highly noisy set. This project examines some methods to filter the noise. None of the filtering methods provided a conclusive improvement on all datasets compared to the already implemented method. The most promising methods, however, includes a substitution of the SVM algorithm with S^3VM and either to use no filtering, random downsampling of the majority class or a recursive filter.

Keywords. Fingerprint, small sensor, noise, filter, SVM, S^3VM, SMOTE, ADASYN.

Acknowledgements

I would like to thank the R&D team at Precise Biometrics for providing the foundation for an interesting project. Special thanks to Karl Netzell for exceptional guidance and insightful comments. Thanks also to Rutger Petersson for taking me on board and to Magnus Oskarsson for supervising the project.

Contents

1 Introduction **4**
- 1.1 Fingerprint systems . . . 5
- 1.2 Fingerprint characteristics . . . 7
- 1.3 Template matching . . . 7
- 1.4 System errors . . . 9
- 1.5 A brief introduction to machine learning . . . 10
- 1.6 Problem formulation . . . 10

2 Theory **12**
- 2.1 The matching module . . . 12
- 2.2 Matching module errors . . . 13
- 2.3 Machine learning . . . 15
  - 2.3.1 Predictor errors . . . 16
  - 2.3.2 Overfitting . . . 16
  - 2.3.3 Crossvalidation . . . 17
- 2.4 Algorithms . . . 18
  - 2.4.1 Linear predictors . . . 18
  - 2.4.2 Support Vector Machine (SVM) . . . 19
  - 2.4.3 Semi-Supervised Support Vector Machine (S^3VM) . . . 20
  - 2.4.4 Parameter selection . . . 22
- 2.5 Dataset characteristics . . . 23
  - 2.5.1 Noise . . . 23

3 Datasets and tools **24**
- 3.1 The dataset structure . . . 24
- 3.2 The datasets . . . 25
  - 3.2.1 Dataset $\mathbf{S_A}$ . . . 25
  - 3.2.2 Dataset $\mathbf{S_B}$ . . . 25
  - 3.2.3 Dataset $\mathbf{S_C}$ . . . 25
  - 3.2.4 Match information . . . 26
  - 3.2.5 Memory limitations . . . 26
- 3.3 Tools . . . 27
  - 3.3.1 Matlab/Octave . . . 27
  - 3.3.2 LIBLINEAR . . . 27
  - 3.3.3 SVMLIN . . . 28
  - 3.3.4 FLANN . . . 28

4 The original algorithm **29**
4.1 Introduction 29
4.2 The training algorithm $\mathbf{A_P}$ 30
4.2.1 The inner training algorithm $\widehat{\mathbf{A}}_\mathbf{P}$ 31

5 The new algorithm **33**
5.1 Introduction 33
5.2 The training algorithm $\mathbf{A_N}$ 33
5.2.1 The inner training algorithm $\widehat{\mathbf{A}}_\mathbf{N}$ 34
5.3 The selection function $\mu_\mathbf{N}$ 34
5.4 The filter functions ρ 34
5.4.1 ADASYN 35
5.4.2 SMOTE 37
5.4.3 Grey data 37
5.4.4 Mark below 37
5.4.5 Random downsampling 38
5.4.6 Recursive 38
5.4.7 Voting filter 38
5.5 Feature transformation 40

6 Results **42**
6.1 Evaluation procedure 42
6.1.1 The baseline 43
6.1.2 Parameters for the ρ and μ_N functions 43
6.1.3 Initial subset 44
6.2 Algorithmic performance using $\widehat{\beta}^\mathbf{0}_\mathbf{N} = \beta_\mathbf{P}$ 44
6.3 Algorithmic performance using $\widehat{\beta}^\mathbf{0}_\mathbf{N} = \beta_\mathbf{R}$ 46
6.3.1 Convergence 46
6.4 Overfitting 47
6.5 Feature transform - Centroid method 48

7 Conclusions **49**
7.1 Future work 49
7.1.1 Feature transformation 49
7.1.2 Robust SVM 50

8 Appendix **51**
8.1 The original training structure 51
8.1.1 Functions 51
8.2 The new training structure 53
8.2.1 Functions 53
8.2.2 An example 55

Symbols, terms and functions

Table 1 contains a summary of the symbols, terms and functions that are used in this report.

Symbol/Term/Function	**Meaning**
$\mathbb{R}$	The set of real numbers.
$\mathbb{R}^n$	The set of n-dimensional vectors over $\mathbb{R}$.
$\mathbb{N}$	The set of natural numbers.
$\mathbf{T}$	Enrollment template.
$\mathbf{I}$	Verification template.
$\mathbf{F}$	Feature vector.
$\mathcal{T}$	Multitemplate.
$\mathcal{I}$	Set of verification templates.
$\mathcal{F}$	Multifeature.
$\mathcal{X}$	Domain set.
$\mathcal{Y}$	Label set.
$L_{D,f}$	True error.
$A_L(S)$	Linear training algorithm SVM or S^3VM working on dataset S.
$A_P(S)$	Original training algorithm working on dataset S.
$A_N(S)$	New training algorithm working on dataset S.
λ_P	Predictor returned by the original training algorithm $A_P(S)$
λ_N	Predictor returned by the new training algorithm $A_P(S)$
α	Feature function.
β	Score function.
γ	Decision function.
FAR	False Accept Rate.
FRR	False Reject Rate.
Sample	Digital representation of a part or a full fingerprint.
Template	Compact representation of sample.
Multitemplate	Collection of templates.
Genuine match	Match between two templates from the same finger.
Impostor match	Match between two templates from two different fingers.

Table 1: Symbols, terms and functions.

Chapter 1

Introduction

Biometric recognition refers to the use of distinctive anatomical and behavioral characteristics, called biometric identifiers, for recognizing individuals [1, p.2]. Examples of anatomical characteristics are fingerprints, irises and facial features. Speech is an example of a behavioral characteristic. This method of recognition has garnered interest during the last century since biometric identifiers cannot be shared or misplaced. Fingerprints are among the more popular biometric identifiers [1, p.12]. In 1893, the Home Ministry Office, UK, accepted that no two individuals have the same fingerprints [1, p.2]. Since then, fingerprints have been used extensively in crime investigations. Early fingerprint analysis used a manual method of visual comparison which was tedious and slow. More recently, with the help of computers, the process has been able to become automated. Because of this, fingerprint technology has been integrated into devices such as smartphones and tablets to provide a method for easy system access.

Precise Biometrics AB provides fingerprint software and smart card readers for digital authentication of identity. Their fingerprint algorithm solution has been integrated into hundreds of millions of mobile phones and tablets worldwide. It is well suited for products with limited processing power and memory such as payment cards, wearables, access control systems, the Internet of Things and products with small sensors. Since the advent of devices such as smartphones where space is a valuable commodity, it has been of interest to shrink the size of the fingerprint sensors [1, p.89]. However, the smaller sensor size makes it harder to accurately perform fingerprint recognition. Precise Biometrics has developed high performing algorithm solutions specifically for use with small sensors. There is nevertheless room for improvement. This project evaluates different methods aimed at improving these algorithms.

This report is divided into eight chapters. Chapter 1 contains an overview of concepts of importance to fingerprint systems. Chapter 2 then provides a more in depth look at some of these concepts. An introduction to the datasets and tools used in this project is found in chapter 3. How the concepts, datasets and tools are connected in Precise Biometrics' current system is explored in chapter 4. Chapter 5 contains the details of the changes made to the system during this project. The effects of these changes and some comments are found in chapter 6 and 7. Chapter 8 provides information about the implementations of the functions in chapter 4 and 5.

1.1 Fingerprint systems

A fingerprint system may be a *verification system* or an *identification system* [1, p.3]. A verification system authenticates the claimed identity of a person by comparing a captured fingerprint with a previously captured template containing fingerprint data connected to the claimed identity. The output of a verification system is an *accept* or *reject* decision indicating whether the system believes that the person is who he/she claims to be.

With an identification system, the person makes no prior identity claim. Instead the entire template database is searched for a match. The output in this case is the identity connected to a template that matched sufficiently well. The output may be empty if no template matched well enough. An identification system can be implemented by letting a verification system perform a one-to-many comparison against an entire template database. The focus in this project is on the verification system.

Two processes are shown in figure 1.1: *enrollment* and *verification*. The enrollment process registers the fingerprint information from individuals and saves it in the system storage. The verification process is responsible for confirming the claim of identity of the subject. The claim can be made with the help of a user name, a PIN number or through information stored on a proximity card, depending on the type of system. As seen in figure 1.1, the enrollment and the verification processes can be broken down into the following modules:

- *Capture*: a digital representation of a fingerprint is captured using a sensor. The captured representation is known as a *sample*.
- *Template creation*: the sample is processed to extract the essential information contained in the fingerprint. This creates a more compact representation of the sample, known as a *template*. Using the template instead of the raw sample is often beneficial in terms of storage space and system speed.
- *Matching*: the matching module decides whether two templates should be considered as belonging to the same finger. This module can be broken down into three sub-modules. The first sub-module takes a *verification template* (obtained from a new sample) and an *enrollment template* (from the system storage) as inputs and produces a *feature set* containing information about the similarity of the two templates. This feature set is used as input to the second sub-module which calculates a *similarity score*. The similarity score is then compared to a predefined *system threshold* in the third sub-module. The templates are deemed to represent the same finger only if the similarity score is above the system threshold. If this is the case, the matching module makes a *match* (or *accept*) decision. If the score is below the threshold, the decision is *non-match* (or *reject*).

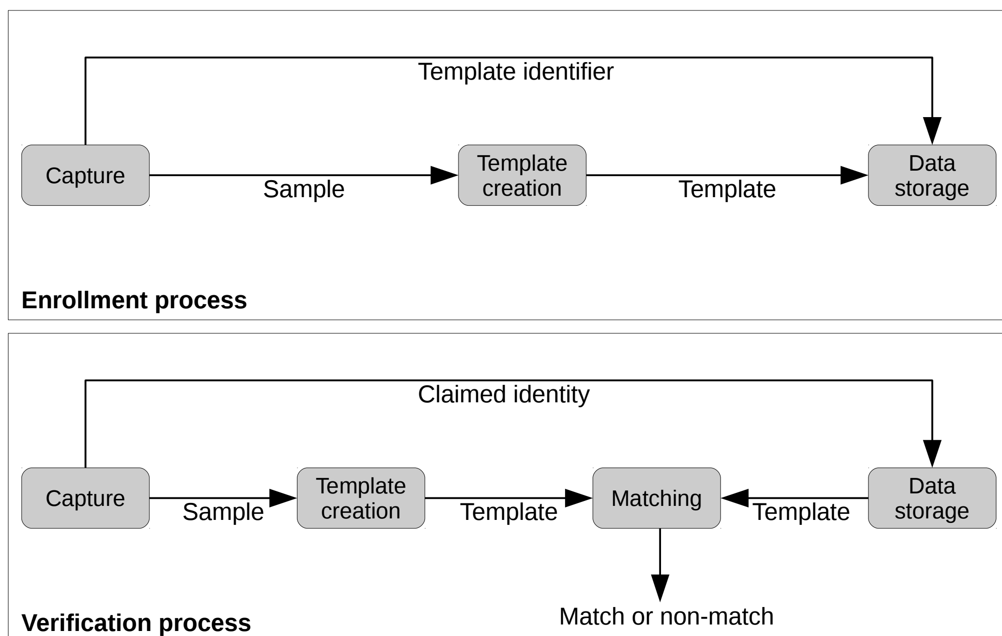

Figure 1.1: Overview of the enrollment and verification processes. The enrollment process is responsible for creating and storing the enrollment template that has been extracted from a captured sample. In the verification process, a new sample is captured and converted into a verification template. This template is then compared to a previously stored enrollment template and a match/accept or non-match/reject decision is made depending on the similarity of the two templates and the system threshold.

1.2 Fingerprint characteristics

The fingerprint system in section 1.1 and figure 1.1 utilizes a module called *template creation* which extracts the essential information from a captured sample. This section explores the kind of information that may be extracted during this stage.

A fingerprint is the reproduction of the exterior appearance of the fingertip epidermis (the outer of the two layers that make up the skin). Figure 1.2 shows some of the most characteristic features of a fingerprint, *ridges* and *valleys* [1, p.97]. The ridges are the dark lines and the valleys are the bright lines. A fingerprint consists of a series of interleaved ridges and valleys. It is common to describe ridge characteristics with three levels of detail. Level 1 is the overall global ridge flow pattern. Level 2 (local level) are the minutiae (small detail) points. Among Level 3 are things like pores and the local shape of ridge edges.

The global patterns in Level 1 can be classified into three categories, *loops*, *deltas* and *whorls* [1, p.98]. Examples of these can be seen in figure 1.3. Compared to the other areas of the fingerprint which consist of mainly parallel ridges, these areas contain higher curvature and frequent ridge terminations. The loop, delta and whorl types are typically characterized by $\cap$, Δ and O shapes respectively.

Minutiae are part of the local level (Level 2) characteristics [1, p.99]. Examples of minutiae are ridges that come to an end or divide into two ridges as can be seen in figure 1.4. These are called *ridge endings* and *bifurcations* respectively. Minutiae are the most commonly used characteristics in automatic fingerprint matching and it has been observed that minutiae do not change during an individual's lifetime.

Level 3 contains the very local characteristics [1, p.101]. These include the width, shape and edge contour of the ridges. The ridges are dotted with sweat pores which can also be used to identify a person assuming a sufficiently high resolution fingerprint sensor is used.

In order for the characteristics to be used in an automatic system they have to be quantified in some way. A ridge ending minutia may for example be quantified using three numbers; the two planar coordinates and the directional angle of the ridge ending tip.

1.3 Template matching

The matching module compares two templates and returns a decision on whether they match or not. The output of the first sub-module in the matching module is a feature set. This feature set is produced by comparing the two input templates. A template is simply put a collection of numbers (called *descriptors*) that are quantifications of the fingerprint characteristics as seen in section 1.2. A possible method for measuring the similarity of two templates is to measure the distance between their descriptors using some norm, e.g. the Euclidean or Hamming norm.

Figure 1.2: Example of a fingerprint sample. The dark lines are the ridges. The white space inbetween the ridges are the valleys.

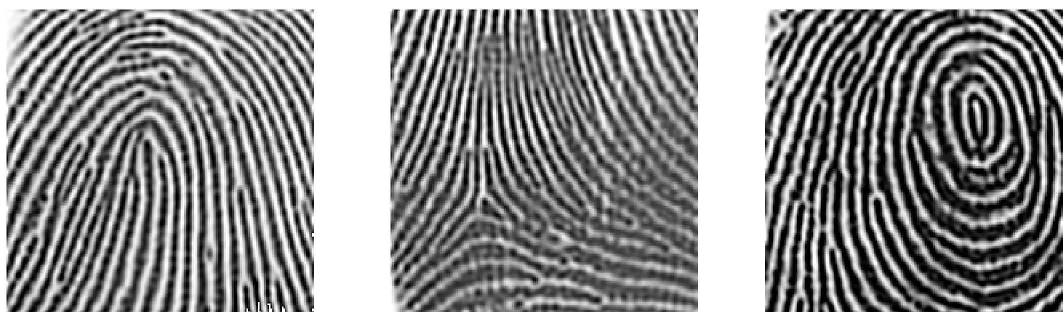

Figure 1.3: Examples of loops ($\cap$), deltas (Δ) and whorls and (O) in fingerprints.

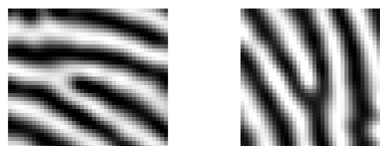

Figure 1.4: A ridge ending and a bifurcation.

Figure 1.5: The overlap between the enrollment template (red) and the verification template (green) is non-existent due to a small sensor.

1.4 System errors

There are many reasons why the fingerprint system could make the wrong decision and allow or deny access to the wrong person [1, p.12-14]. Problems with the sensor could lead to errors of the types: *Failure to Detect (FTD)* and *Failure to Capture (FTC)*, which respectively refer to the sensor failing to detect the presence of a finger and failure to capture a sample of sufficient quality. *Failure to Process (FTP)* occurs when the template extraction module is unable to extract a usable template from the sample.

An *information limitation* error occurs for example when a verification template and an enrollment template are from different parts of the same finger and there is very little overlap between the two [1, p.12]. This situation is common if the sensor size is small and unable to capture a full fingerprint in one sample. Figure 1.5 illustrates this problem. Even though both the enrollment and the verification template come from the same finger it is very likely that the owner of this fingerprint would be rejected due to the lack of similarity between the two templates. To increase the possibility of overlap between the enrollment and verification template, a method which is used by Precise Biometrics is to let users provide a set of enrollment templates, known as a *multitemplate* (figure 1.6), from each finger during the enrollment phase. With this method, a finger is accepted if the verification template is sufficiently similar to any of the templates in a multitemplate. Naturally, provided that the sensor is small enough and that the templates in the multitemplate are evenly distributed over the finger, a verification template is going to lack similarity to many, if not most, of the templates in the multitemplate. This is true regardless of whether or not the verification template and the multitemplate are created from the same finger.

Figure 1.6: A multitemplate is a collection of enrollment templates (red). The verification template (green) overlaps only some of the templates in the multitemplate.

1.5 A brief introduction to machine learning

Precise Biometrics utilizes machine learning in the second and third sub-module of the matching module to support its accept or reject decisions. In [2, p.vii, p.19] the subject of machine learning is introduced as a method of programming computers to *learn* using examples. The goal is to find meaningful, exploitable patterns among a set of examples. Machine learning algorithms may be divided into three different groups; *supervised*, *semi-supervised* and *unsupervised* [2, p.23]. The names of the three groups are connected to the type of example sets that the different types of algorithms expect as input when learning. Supervised learning algorithms expect the data in the training set to be paired with a label that indicates what class the example belongs to. Unsupervised learning does not expect the data to be labeled. Semi supervised learning is a mixture of the supervised and unsupervised learning; the dataset can consist of a mixture of labeled and unlabeled data.

1.6 Problem formulation

It is time to formulate the problem that is investigated in this project. The focus is on the matching module. The second and third sub-module in the matching stage (section 1.1) uses machine learning to learn when to make an accept or reject decision. Section 1.5 described machine learning as a method for learning from a set of examples. In this project, an example is a feature set (section 1.1). The example is connected to a label indicating whether the feature set is produced by two templates from the same finger or not. Section 1.4 however mentioned that, when a small sensor is involved, a verification template is often going to lack similarity to many of the templates in a multitemplate regardless of whether they come from the same finger or not. This means that the example

set is going to suffer from noise. The problem that is investigated in this project can be formulated as follows: *Is there a better method to remove noise from the example set compared to the one currently in use by Precise Biometrics?*

Chapter 2

Theory

This chapter explores some concepts of importance to the matching module which was introduced in section 1.1. However, the details of the actual implementation of this module is located in subsequent chapters. Section 2.1 introduces some notation and functions that are useful for further discussion of the matching module. Details on how to quantify the performance of the matching module is found in section 2.2. Section 2.3 contains a formal introduction to machine learning and some of the common concepts and issues encountered when dealing with this subject. A main concept is the *predictor*. Some training algorithms for linear predictors are explored in section 2.4. These algorithms require a training dataset. Section 2.5 presents some common problems encountered when constructing a training set.

2.1 The matching module

The matching module was introduced in section 1.1 and its place in the overall fingerprint system can be seen in figure 1.1. This section contains further details about this module.

A template can be regarded as a vector in $\mathbb{R}^m$ containing a quantification of the characteristics of a fingerprint sample. Examples of these characteristics were presented in section 1.2. The input to the matching module is an enrollment template, $\mathbf{T}$, and a verification template, $\mathbf{I}$. The output is a binary decision on whether these two templates should be considered as belonging to the same finger or not. It was further specified in section 1.1 that the matching module consists of three sub-modules. The first sub-module is the *feature function*, $\alpha : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^n$. A *feature vector* is acquired using $\alpha(\mathbf{T}, \mathbf{I})$. This feature vector contains information about the similarity of $\mathbf{T}$ and $\mathbf{I}$ and is the input to the second sub-module.

The *score function* is $\beta : \mathbb{R}^n \to \mathbb{R}$. A *similarity score* for a feature vector $\mathbf{F}$ can be calculated using $\beta(\mathbf{F})$. As the name implies, this function calculates a single number that measures the similarity of two templates (the dependency of β on the two templates

is implicit in $\mathbf{F}$). Finally, the *decision function* is $\gamma : \mathbb{R} \to \{0,1\}$. A *decision* is acquired from a similarity score $\mathbf{S}$ using $\gamma(\mathbf{S})$.

The function $s(\mathbf{T}, \mathbf{I})$ is used in section 2.2 when discussing error measurements. This function produces a similarity score from two templates and can be defined as a composite function, $s := \beta \circ \alpha : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$.

The complete matching module function, m, is acquired by also including the γ function after the similarity score calculation, $m := \gamma \circ \beta \circ \alpha : \mathbb{R}^m \times \mathbb{R}^m \to \{0,1\}$. This function takes two templates and makes an accept or reject decision.

Finally, a function that is going to be of importance during the remainder of this project and should be kept in mind especially when reading section 2.3 is the composition of the similarity function and the decision function, $\lambda := \gamma \circ \beta : \mathbb{R}^n \to \{0,1\}$. This function makes a decision given a feature vector containing similarity measurements of two templates.

2.2 Matching module errors

The similarity score that is used in the second sub-module in the matching module can without loss of generality be assumed to lie in the interval $[0,1]$. The closer a score is to 1, the more certain is the system that the verification template and the enrollment template comes from the same finger. A decision that the two templates do indeed come from the same finger is called a *match*. The opposite decision is called a *non-match*. Two types of errors can be committed at this stage, the *false match* and the *false non-match*. The false match error occurs when the system deems templates from two different fingers to be from the same finger. The false non-match error occurs when templates from one finger are mistaken for being from two different fingers. False match and false non-match are often referred to as *false acceptance* and *false rejection*. These terms are commonly used in the commercial sector and are also used in this report. It is also common to use the false acceptance rate (FAR) and false rejection rate (FRR), as described in detail below, when doing performance evaluation of a fingerprint system.

With $\mathbf{T}$ and $\mathbf{I}$ as in section 2.1, the null and alternate hypothesis are [1, p.16]:

H_0: $\mathbf{I} \neq \mathbf{T}$, verification template does not come from the same finger as the enrollment template.

H_1: $\mathbf{I} = \mathbf{T}$, verification template comes from the same finger as the enrollment template.

The associated decisions are as follows.

D_0 : non-acceptance.

D_1: acceptance.

Before the final decision in the matching module, the similarity score $s(\mathbf{T}, \mathbf{I})$ is produced. By collecting scores from many different template comparisons, both from comparisons from the same finger and from different fingers, one can estimate the score distribution

$p(s)$. The *genuine* distribution is a conditional distribution $p(s|H_1)$ of scores generated from template comparisons from the same finger. The *impostor* distribution $p(s|H_0)$ is the score distribution given comparisons from different fingers. If the similarity score is less than a *system threshold*, t, then D_0 is decided, else D_1. The FRR and FAR can now be declared as

$$FRR = P(D_0|H_1) = \int_0^t p(s|H_1)ds$$

$$FAR = P(D_1|H_0) = \int_t^1 p(s|H_0)ds$$

The FAR and FRR are functions of the system threshold t. By changing the threshold, the FAR and FRR also changes. From figure 2.1 it is apparent that to decrease the FAR one can simply increase the threshold. This however has the unfortunate effect of also increasing the FRR. If one wants to decrease the FRR however, the FAR increases. Depending on where a system is to be deployed, a low FAR level can be more important than a low FRR. In that case it would be beneficial to choose a high system threshold.

By varying the system threshold, new pairs of FAR and FRR can be acquired. Plotting each pair in a DET graph (Detection Error Tradeoff) is a convenient method of evaluating and comparing the performance of different systems. Figure 2.2 shows examples of DET curves for different systems. Since both the FAR and FRR ideally should be a small as possible for a good system, a curve that is as close as possible to the bottom is preferred.

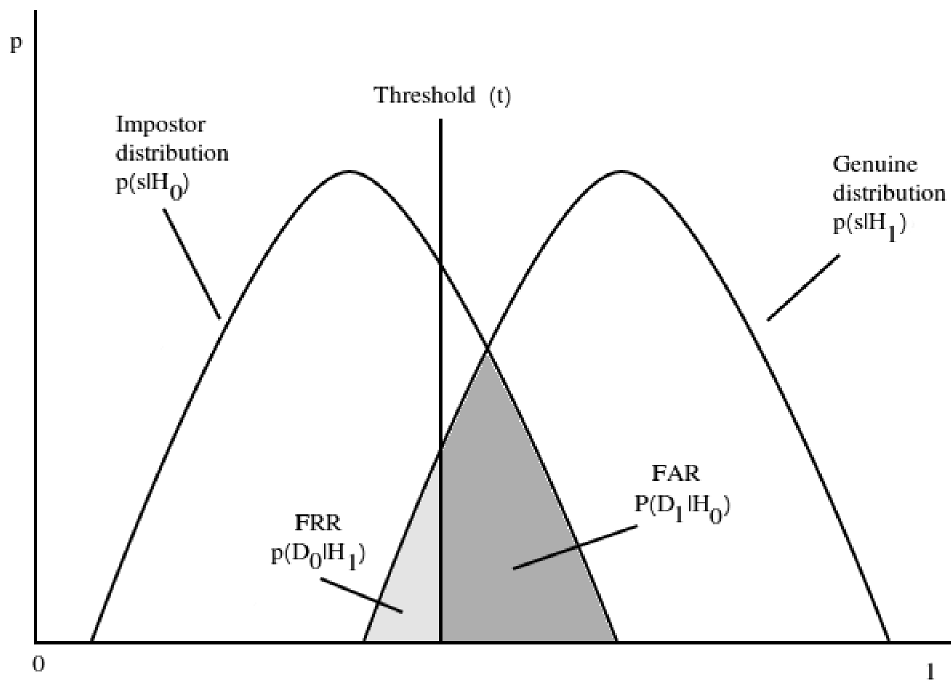

Figure 2.1: The FAR and FRR for a given threshold. By moving the threshold it is possible to decrease the FAR at the expense of increasing the FRR and vice versa.

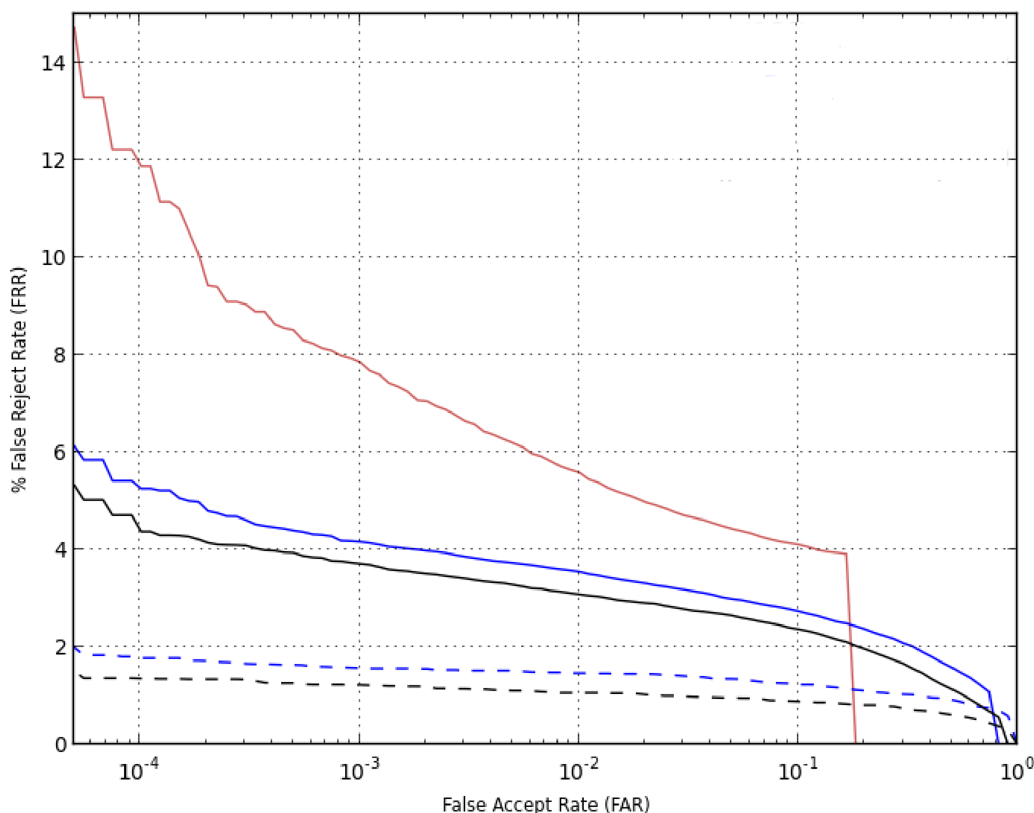

Figure 2.2: DET curves for different systems. A system with a DET curve close to the bottom is preferred. Here, the red curve indicates the worst performing system.

2.3 Machine learning

This section expands on section 1.5 with a formal introduction to machine learning.

The output of the *learning* or *training* process is a *prediction rule* or *predictor* [2, p.34]. A predictor is used to assign a label to an object. The *domain set*, $\mathcal{X}$, is the set of objects that can be labeled. In this project $\mathcal{X} = \mathbb{R}^n$. The *label set*, $\mathcal{Y}$, is the set of labels that can be assigned to the objects in $\mathcal{X}$. For binary classification the label set is usually $\{0, 1\}$ or $\{-1, 1\}$. A predictor is a mapping from the domain set to the label set, $h : \mathcal{X} \rightarrow \mathcal{Y}$. The *training set*, $S = \{(x_1, y_1)...(x_N, y_N)\}$, is a finite set of pairs in $\mathcal{X} \times \mathcal{Y}$. $A(S)$ denotes the predictor that is returned by algorithm A using training set S. The *perfect predictor* $f : \mathcal{X} \rightarrow Y$ labels all points in the domain set correctly, i.e $y_i = f(x_i)$ for all i. It is assumed that objects in $\mathcal{X}$ are distributed according to some distribution $\mathcal{D}$ such that given a subset $C \subset \mathcal{X}$, the probability distribution $\mathcal{D}$ assigns a number $\mathcal{D}(C)$ which determines how likely it is to observe a point $x \in C$. A general dataset, S, can be thought of as being generated by sampling x_i according to the distribution $\mathcal{D}$ and assigning a label with $y_i = f(x_i)$.

The *true error* of a predictor, h, is $L_{\mathcal{D},f}(h) := \mathcal{D}(\{x : h(x) \neq f(x)\})$ i.e. the probability of randomly choosing an example x for which $h(x) \neq f(x)$. The distribution $\mathcal{D}$ and the perfect predictor, f, are unknown during the training process. These are what the

training process is trying to find.

2.3.1 Predictor errors

The goal of the training is to find a predictor $h : \mathcal{X} \to \mathcal{Y}$ that minimizes $L_{\mathcal{D},f}(h)$ [2, p.35]. However since $\mathcal{D}$ and f are unknown, an estimated error measurement instead has to be used. Assume a dataset S on the same form as in section 2.3. Then the *empirical error* is calculated as

$$L_S(h) = \frac{|i \in [m] : h(x_i) \neq y_i|}{m}, \quad [m] = \{1, ..., m\}. \tag{2.1}$$

Error measurements that are used extensively in this project were introduced in section 2.2. These are the *False Accept Rate (FAR)* and the *False Reject Rate (FRR)*. The following shows how to estimate these values for a predictor, $h : \mathcal{X} \to \mathbb{R}$, using a set of discrete pairs such as S.

Let $t \in \mathbb{R}$. Using [2, p.244] gives the definitions

$$\begin{aligned} True\ positives: &\quad a(h,t) = |i : y_i = +1 \wedge sgn(h(x_i) - t) = +1| \\ False\ positives: &\quad b(h,t) = |i : y_i = -1 \wedge sgn(h(x_i) - t) = +1| \\ False\ negatives: &\quad c(h,t) = |i : y_i = +1 \wedge sgn(h(x_i) - t) = -1| \\ True\ negatives: &\quad d(h,t) = |i : y_i = -1 \wedge sgn(h(x_i) - t) = -1| \end{aligned} \tag{2.2}$$

FRR and FAR are calculated according to

$$FRR = \frac{False\ negatives}{True\ positives + False\ negatives} \tag{2.3}$$

and

$$FAR = \frac{False\ positives}{True\ negatives + False\ positives}. \tag{2.4}$$

The FRR and FAR of a predictor h on a dataset S are thus

$$FRR_S(h,t) = \frac{c(h,t)}{a(h,t) + c(h,t)} \tag{2.5}$$

and

$$FAR_S(h,t) = \frac{b(h,t)}{d(h,t) + b(h,t)} \tag{2.6}$$

The variable t in 2.2, 2.5 and 2.6 corresponds, in this project, to the system threshold (sections 1.1 and 2.2).

2.3.2 Overfitting

When using an estimated error measurement to measure the fitness of a predictor it is possible to encounter a problem known as *overfitting* [2, p.36]. This is a phenomenon where there is a dissonance between the predictor's estimated error and the true error.

Assume that a predictor h, has been found using the training set S with empirical error $L_S(h) = 0$. If h has been overfitted it is possible that $L_{\mathcal{D},f}(h) > L_S(h)$. This implies that even though the predictor may seem to be perfect, it could still be making errors when encountering new data points if the empirical error is a poor approximation of the true error. A predictor is said to be overfitted if the difference between $L_{\mathcal{D},f}(h)$ and $L_S(h)$ is large [2, p.173].

2.3.3 Crossvalidation

In order to make sure a predictor has not overfitted, one should evaluate a trained predictor on data that the predictor has not previously encountered during the training session. A method for simulating new data encounters is to partition the available dataset into a training set and a test set. The training set is used to train the predictor. The predictor is then used on the test set where some error measure is applied to gauge the performance of the predictor.

A method that is commonly used to evaluate a predictor's performance is known as *k-Fold Crossvalidation* [2, p.150]. In k-fold crossvalidation the dataset of size m is partitioned into k subsets (folds) of size m/k. For each fold, the algorithm is trained on the union of the other folds and then the error of its output is estimated using the fold. Figure 2.3 shows an example of a 3-fold crossvalidation. The final performance measure of a predictor is some aggregate of the performance on each fold. Some common aggregation methods are the mean or the max method. The mean method calculates the mean of the evaluation results of all folds. The max method simply chooses the worst result among the folds.

To obtain reliable performance estimation or comparison, a large number of estimates are always preferred. In k-fold crossvalidation, only k estimates are obtained. A commonly used method to increase the number of estimates is to run k-fold crossvalidation multiple times [11]. The data is reshuffled and re-stratified before each round.

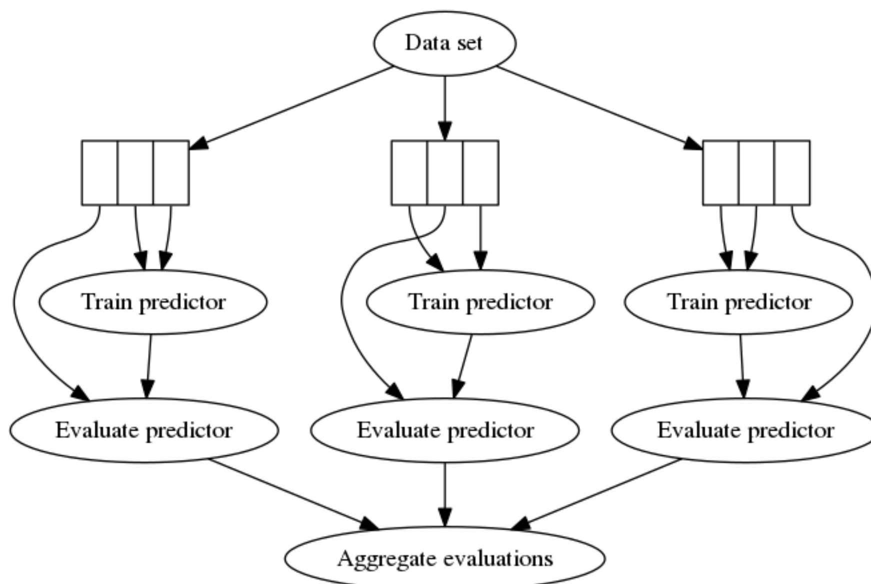

Figure 2.3: 3-fold crossvalidation. The full dataset is split into three equal parts. Two parts are used to train a predictor and the remaining part is used to evaluate the predictor. This process is repeated three times until each part has been used for evaluation.

2.4 Algorithms

After the introduction to predictors, $h : \mathcal{X} \to \mathcal{Y}$, in section 2.3 the following subsections contain the algorithms $A_L(S)$ that are utilized to find *linear* predictors in this project.

2.4.1 Linear predictors

Many learning algorithms that are being widely used in practice rely on linear predictors, partly because of the ability to train them efficiently in many cases. In addition, linear predictors are intuitive, easy to interpret, and fit the data reasonably well in many natural learning problems [2, p.117]. The decision boundary for binary linear predictors is a hyperplane. Assigning a label to a new data point is the same as observing on which side of the hyperplane the data point falls.

Let $\phi : \mathbb{R} \to \mathcal{Y}$. The family of linear predictors then have the form

$$h_{\mathbf{w},b}(\mathbf{x}) = \phi(\mathbf{w} \cdot \mathbf{x} + b) \tag{2.7}$$

where $\mathbf{w}, \mathbf{x} \in \mathbb{R}^n, b \in \mathbb{R}$ [2, p.117]. Often, ϕ is the sign function. In this case the prediction rule can be written more clearly as

$$h_{\mathbf{w},b}(\mathbf{x}) = \begin{cases} 1, & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.8}$$

It is sometimes convenient to introduce $\mathbf{w}' = (b, w_1, w_2, ..., w_n) \in \mathbb{R}^{n+1}$ and $\mathbf{x}' = (1, x_1, ..., x_n) \in \mathbb{R}^{n+1}$. The decision rule can now be written in a simpler form as

$$h_{\mathbf{w}'}(\mathbf{x}) = \phi(\mathbf{w}' \cdot \mathbf{x}') \tag{2.9}$$

or

$$h_{\mathbf{w}'}(\mathbf{x}') = \begin{cases} 1, & \text{if } \mathbf{w}' \cdot \mathbf{x}' > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.10}$$

2.4.2 Support Vector Machine (SVM)

The basic SVM algorithm finds a predictor that belongs to the family of linear predictors [2, p.202]. SVM searches for a separating hyperplane that not only makes sure that data points from different classes fall on different sides of the plane but also maximizes the margin between the plane and the closest points of each class. Larger margins should lead to better generalization and prevent overfitting in high-dimensional attribute spaces [10].

There are different types of SVM; Hard-SVM and Soft-SVM [2, p.202]. Hard-SVM assumes that the different classes are linearly separable and returns a hyperplane with maximum margin. However, in many scenarios the classes are not linearly separable. In this case Soft-SVM is a better choice. Soft-SVM is a generalization of the Hard-SVM and allows for some misclassified data points.

The procedure to find a separating hyperplane is similar in both the hard and the soft case. A training dataset, S, of the form established in 2.3 is needed. In this case $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \{1, -1\}$. Assume that S contains L data-label pairs. Figure 2.4 shows the Hard-SVM where the classes are linearly separable. The data points in the figure are in $\mathbb{R}^2$ but the following reasoning also applies for points in $\mathbb{R}^n$. As can be seen in the figure, $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$ holds for all data points in the set. A support vector fulfills $\mathbf{w} \cdot \mathbf{x} + b = \pm 1$. Let $\mathbf{x}_+$ and $\mathbf{x}_-$ be support vectors on either side of the separating plane i.e $\mathbf{w} \cdot \mathbf{x}_+ + b = 1$ and $\mathbf{w} \cdot \mathbf{x}_- + b = -1$. Now, the margin M can be calculated using

$$M = \frac{\mathbf{w}}{||\mathbf{w}||} \cdot (\mathbf{x}_+ - \mathbf{x}_-) = \frac{\mathbf{w} \cdot \mathbf{x}_+ - \mathbf{w} \cdot \mathbf{x}_-}{||\mathbf{w}||} = \frac{1 - b - (-1 - b)}{||\mathbf{w}||} = \frac{2}{||\mathbf{w}||} \tag{2.11}$$

So the margin is inversely proportional to $||\mathbf{w}||$. Hence, by minimizing $||\mathbf{w}||$ the margin is maximized. The Hard-SVM can thus be formulated as the optimization problem

$$\begin{aligned} &\min_{\mathbf{w}} \ ||\mathbf{w}||^2 \\ &\textit{subject to} \quad y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 \quad i = 1, ..., L \end{aligned} \tag{2.12}$$

As mentioned, the Soft-SVM is a generalization of the Hard-SVM. This generalization allows the SVM algorithm to be used even if the classes are not linearly separable. The extension is done by introducting slack terms, $\eta_i \in \mathbb{R}^+$ for every point such that if the

point is misclassified then $\eta_i \geq 1$. The Soft-SVM formulation is

$$\begin{aligned} \min_{\mathbf{w},\eta_i} \quad & \|\mathbf{w}\|^2 + C\sum_{i}^{L}\eta_i \\ \textit{subject to} \quad & y_i(\mathbf{w}\cdot\mathbf{x}_i + b) \geq 1 - \eta_i \qquad i = 1, ..., L \end{aligned} \tag{2.13}$$

The parameter C is a regularization parameter that indicates the relative cost between misclassified points and the width of the margin. With a small C, misclassified points can easily be ignored which allows for a larger margin. A large C implies that it is important to avoid misclassification even if this means that the margin will be smaller. If $C = \infty$, the Hard-SVM is acquired since no misclassifications are allowed.

With the help of a suitable penalty function $\Phi(\mathbf{w};\mathbf{x_i},y_i)$ it is possible to formulate equation 2.13 in another way,

$$\min_{\mathbf{w}} \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i}^{L}\Phi(\mathbf{w};\mathbf{x_i},y_i) \tag{2.14}$$

The penalty function could be either $\Phi_1(\mathbf{w};\mathbf{x_i},y_i) = \max(1-y_i\mathbf{w}\cdot\mathbf{x}_i, 0)$ or $\Phi_2(\mathbf{w};\mathbf{x_i},y_i) = \max(1-y_i\mathbf{w}\cdot\mathbf{x}_i, 0)^2$.

2.4.3 Semi-Supervised Support Vector Machine (S³VM)

While the Soft-SVM extended the Hard-SVM, an algorithm known as S³VM extends the Soft-SVM [10]. The ordinary SVM algorithms assume that each data point that is used for training is labeled. S³VM only needs some of the data points to be labeled. When working with S³VM the set of labeled data is known as the *training set* and the unlabeled data is the *working set*. If the working set is empty the method becomes the standard SVM algorithm. If the training set is empty, the method becomes a form of unsupervised learning. *Semi-supervised* learning occurs when both the working and the test set is non-empty. In S³VM, two constraints are added for each unlabeled point in the training set. One constraint calculates the misclassification error as if the point belongs to class 1 and the other as if the point belongs to class -1. The objective function then calculates the minimum of the two possible misclassification errors. The S^3VM problem formulation is

$$\begin{aligned} \min_{\mathbf{w},b,\eta,\xi,z} \quad & \|\mathbf{w}\| + C\left[\sum_{i=1}^{L} n_i + \sum_{j=L+1}^{L+K} min(\xi_j, z_j)\right] \\ \textit{subject to} \quad & y_i(\mathbf{w}\cdot\mathbf{x}_i + b) + \eta_i \geq 1 \quad \eta \geq 0 \quad i = 1, ..., L \\ & \mathbf{w}\cdot\mathbf{x}_i - b + \xi_j \geq 1 \quad \xi_j \geq 0 \quad j = L+1, ..., L+K \\ & -(\mathbf{w}\cdot\mathbf{x}_i - b) + z_j \geq 1 \quad z_j \geq 0 \end{aligned} \tag{2.15}$$

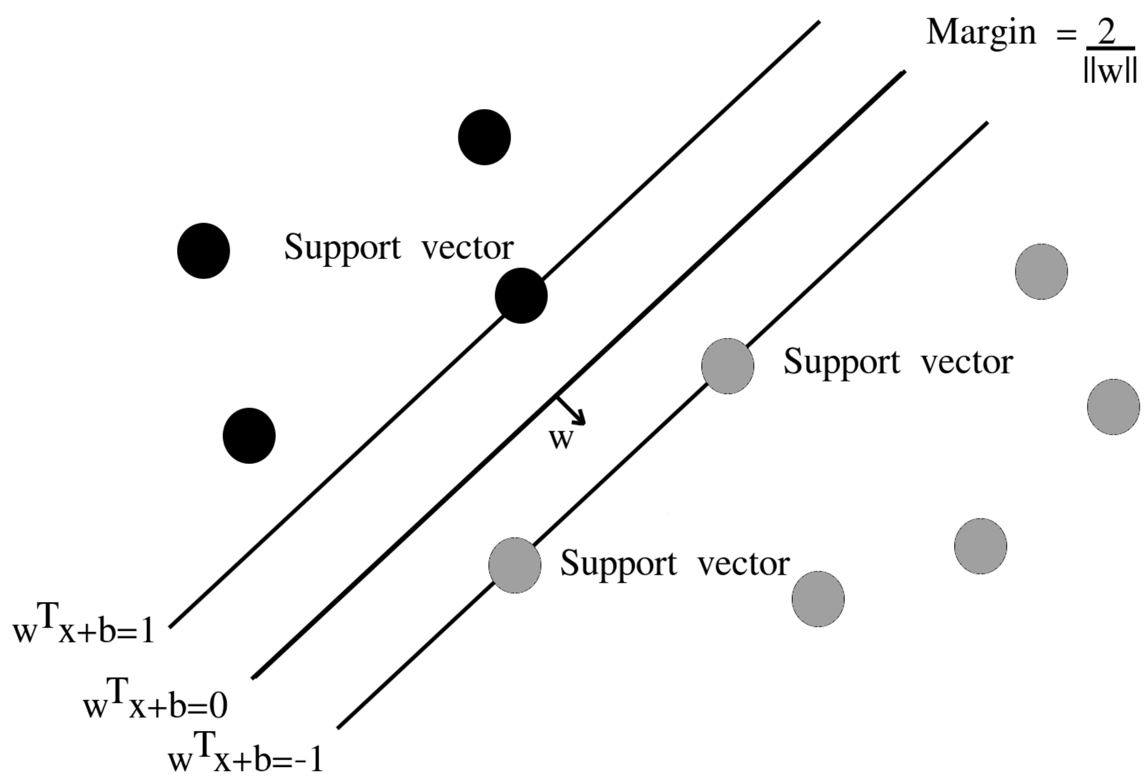

Figure 2.4: SVM with linearly separable classes.

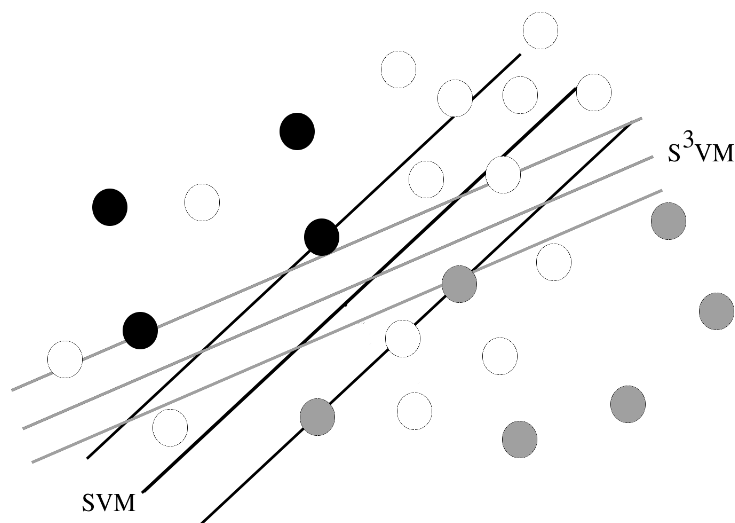

Figure 2.5: Example of the difference in the solutions found by SVM and the S^3VM algorithms. The black and the grey data points belong to a positive and a negative class respectively. The ordinary SVM algorithm finds a separating boundary that maximizes the margin to the two classes. However, by including unlabeled data (white points), the boundary is adjusted to also maximize the margin with the respect to the unlabeled data points.

2.4.4 Parameter selection

In addition to a training set and a label set, the previously introduced algorithms depend on the parameter C. It is common to let this parameter be C for one class and $C \cdot C_p$ for the other class. These parameters affect the final weight vector and have to chosen before the training session. In case the training set is very imbalanced, a careful choice of parameters can be the difference between finding a good or a bad predictor. A common method of finding suitable parameters is to perform a grid search with crossvalidation [7]. A grid search is performed by systematically testing various pairs of C and C_p and then simply choose the pair that yielded the best predictor. However, to avoid the risk of finding a predictor that performs well only on the available dataset but poorly on other datasets (i.e an overfitted predictor), crossvalidation is recommended [7]. Crossvalidation ensures that the predictor is found using one dataset but tested on another. This should be a better indicator of the expected performance of predictor than if the predictor was trained and tested on the same dataset.

It is also recommended that the initial grid search is performed on a coarse grid [7]. A finer grid search should then be performed around the point(s) of best performance in the coarse grid. Finer and finer searches can then be added if it is deemed beneficial.

A practical method for finding good parameters is to start with a coarse grid with exponentially growing sequences of C and C_p [7]. For example $C = (2^{-5}, 2^{-3}, ..., 2^{15})$ and $C_p = (2^{-15}, 2^{-13}, ..., 2^{3})$.

2.5 Dataset characteristics

The training set S is a window through which the learner gets partial information about the distribution $\mathcal{D}$ and the labeling function, f. The larger the set is, the more likely it is to reflect more accurately the distribution and labeling used to generate it [2, p.38]. It is not realistic to expect that with full certainty S will suffice to direct the learner toward a good classifier (from the point of view of $\mathcal{D}$), as there is always some probability that the sampled training data happens to be very nonrepresentative of the underlying $\mathcal{D}$.

2.5.1 Noise

Since the training of a predictor depends on a training set S it is important that the training set is constructed well. A fundamental requirement to be able to train a confident predictor is that there is some statistical correlation between objects in the domain set and the labels in the label set. Assume that the training set S was acquired by randomly assigning labels to a series of objects from the domain set. Since there is no pattern between the object and the label, a predictor that has been trained using this training set could be expected to do no better than random label assignment.

While random labels are an extreme form of label noise, some degree of label noise is common in certain problem areas where some objects in the domain set are mislabeled due to various reasons [6, p.2]. Label noise can occur for reasons such as subjectivity, data-entry error or inadequate information. Subjectivity may arise when observations need to be ranked in some way, such as disease severity, or when the information used to label an object is different from the information to which the learning algorithm will have access. Further, it may not be possible to perform the tests necessary to guarantee that a diagnosis is 100% accurate. The latter is a case of label noise due to information inadequacy.

If the training set is known or suspected to suffer from noise it is common to try to filter the training data before using it to train a predictor. The effect of the filter is to remove or relabel the objects that are suspected to be due to noise. Pre-filtering often improves the predictive accuracy [6, p.2]. However, the improvement is not a general case, it depends on the type of noise. Assume a binary class problem with a positive and a negative class and an object that has negative class characteristics but a positive label. If the object resembles a negative class object due to inherent noise in the problem area, this noise will be present in both training and test cases. Such a seemingly incorrectly labeled object should then not be removed as it would increase the predictive accuracy. However, if the object is actually mislabeled, the predictive accuracy increases if the object is removed.

A danger in automatically removing instances that cannot be correctly classified is that they might be exceptions to the general rule. A key question in improving data quality is how to distinguish exceptions from noise.

Chapter 3

Datasets and tools

The theoretical training dataset S that is needed to find a predictor was introduced in section 2.3. Section 3.1 describes the general structure of the realizations of S that are used in this project. Detailed information about the three datasets that are used as a basis for the evaluations in chapter 6 is found in section 3.2. In section 3.3 some information about the development environment and external packages used in the implementation of the new training algorithm is presented.

3.1 The dataset structure

A typical fingerprint *sample database* at Precise Biometrics is constructed using a set of volunteers who provide multiple samples from multiple fingers. They provide both enrollment and verification samples. In case a sensor is not large enough to cover the whole fingerprint, the volunteers are instructed to move the finger between each sample capture to make sure at least one sample from each part of the fingerprint has been captured at some point.

By creating a template from each of the enrollment samples collected from a single finger, a set of templates known as a *multitemplate* is formed, denoted $\mathcal{T} = \{\mathbf{T}_1, ..., \mathbf{T}_k\}$. A set of verification templates is denoted $\mathcal{I} = \{\mathbf{I}_1, ..., \mathbf{I}_l\}$. A fingerprint *template database* then consists of a collection of pairs ($\mathcal{T}$,$\mathcal{I}$), collected from each finger from each person. A feature vector is acquired by using the α function introduced in section 2.1 on two templates. The set of feature vectors acquired by matching a verification template with all the templates in a multitemplate is called a *multifeature* and is denoted $\mathcal{F} = \alpha(\mathcal{T}, \mathbf{I})$.

A training set S can now be created by repeatedly selecting a verification template and a multitemplate from two of the fingers in the database and creating a multifeature from these using the α function. If the verification template and the multitemplate comes from the same finger, the feature vectors in the resulting multifeature will all be considered genuines, otherwise they will be considered impostors. In this project, $y_i = 1$ is the label for genuines and impostors are $y_i = -1$. The dataset S is a set of labeled multifeatures $S = \{(\mathcal{F}_1, y_1), ..., (\mathcal{F}_k, y_k)\}$. Let $\mathbf{F}$ denote a single feature vector. Since a multifeature is

just a set of feature vectors where all feature vectors have the same label, it is also possible to view S as a collection of labeled single feature vectors $S = \{(\mathbf{F}_1, y_1), ..., (\mathbf{F}_k, y_k)\}$.

If one views S as a set of labeled feature vectors, S is in many cases going to contain a high degree of noise. This is because of the situation described in section 1.4; due to the small sensor size, each genuine multifeature is going to contain feature vectors that will resemble impostor feature vectors but they will nonetheless be labeled genuine (i.e $y_i = 1$) in S. The level of feature noise in S depends on how evenly distributed the templates in the multitemplates are over the fingers. Other factors that contribute are the size of the sensor as well as the number of templates in the multitemplates.

3.2 The datasets

The results of the changes made to the original training algorithm in this project are evaluated on three different datasets of different sizes. These datasets are referred to as S_A, S_B and S_C. The following subsections present some specifics about the datasets.

3.2.1 Dataset $\mathbf{S_A}$

Dataset S_A consists of feature vectors from $N_p = 50$ people. Each person contributed with samples from $N_f = 6$ fingers and $N_e = 35$ samples per finger during the enrollment phase. They also provided $N_v = 10$ verification samples per finger. This gives a total of $N_G = N_p \cdot N_f \cdot N_e \cdot N_v = 105000$ genuines. When creating the impostors, no feature vectors from different fingers from the same person were included and only the first verification template from each finger was used. Thus the number of impostors were $N_I = N_p \cdot (N_p - 1) \cdot N_f^2 \cdot N_e = 3087000$. The number of multifeatures are 91200; 3000 genuine and 88200 impostor multifeatures.

3.2.2 Dataset $\mathbf{S_B}$

The next dataset consists of feature vectors from 94 people and so called *super impostors*. These are impostors that have very similar features compared to genuines so they are easily mistaken. However, the number of samples provided per finger is variable so a concise expression for the number of genuines and impostors in this set is not as easily provided. The number of genuines are 223872 and impostors 11784960 spread out on 6996 genuine and 368280 impostor multifeatures.

3.2.3 Dataset $\mathbf{S_C}$

Dataset S_C contains feature vectors from 98 people. The number of genuines are 784000 and impostors 4792200. These are spread out on 39200 genuine and 239610 impostor multifeatures.

3.2.4 Match information

Each feature vector in the datasets has a 7-dimensional numerical vector that contains information about the two templates that were used when creating the feature vector. Table 3.1 describes the information in the vector. The match information can be used to partition the dataset into training and test sets in such a way that no fingers from the same person ends up in both the training and the test set. The match information is also used to construct the label set $\mathcal{Y}$.

Column	Description
1	ID of the person who is the owner of the enrollment template.
2	ID of the finger of the person who is the owner of the enrollment template.
3	Index of the enrollment template.
4	ID of the person who is the owner of the verification template.
5	ID of the finger of the person who is the owner of the verification template.
6	Index of the verification template.
7	The match score given by a score function.

Table 3.1: Match information.

3.2.5 Memory limitations

When working on large datasets, it is necessary to consider the memory limitations of the computer architecture in use. In Matlab/Octave the datasets are represented using double precision floating point format, i.e 8 bytes. With datasets containing several millions of elements, it can be impossible to load the whole dataset into memory. Table 3.2 shows the sizes of the datasets. Both the original training algorithm in chapter 4 and the new algorithm in chapter 5 work around this issue. The evaluations in chapter 6 are performed on a computer with 12GB RAM.

Dataset	Rows	Columns	Elements	Approx size in RAM (GB)
S_A	3192000	114	363888000	3
S_B	12008832	128	1537130496	12
S_C	5576200	120	669144000	5

Table 3.2: Dataset sizes.

3.3 Tools

The following are the important tools used in this project.

3.3.1 Matlab/Octave

MATLAB (**mat**rix **lab**oratory) is a multi-paradigm numerical computing environment and fourth-generation programming language. A proprietary programming language developed by MathWorks, MATLAB allows matrix manipulations, plotting of functions and data, implementation of algorithms, creation of user interfaces, and interfacing with programs written in other languages, including C, C++, C#, Java, Fortran and Python.[1]

GNU Octave is software featuring a high-level programming language, primarily intended for numerical computations. Octave helps in solving linear and nonlinear problems numerically, and for performing other numerical experiments using a language that is mostly compatible with MATLAB. It may also be used as a batch-oriented language. Since it is part of the GNU Project, it is free software under the terms of the GNU General Public License.[2]

3.3.2 LIBLINEAR

LIBLINEAR is a library for large linear classification developed by the machine learning group at National Taiwan University [3]. It is designed to efficiently handle data with millions of instances and features and can be integrated into MATLAB/Octave.

LIBLINEAR supports two popular binary linear classifiers: logistic regression (LR) and linear support vector machine (SVM). Given a set of instance-label pairs $(x_i, y_i), i = 1, ..., l, x_i \in \mathbb{R}^n, y_i \in \{-1, 1\}$, both methods solve the following unconstrained optimization problem with different penalty functions $\Phi(w; x_i, y_i)$,

$$\min_{w \in \mathbb{R}^n} \frac{1}{2}||w||_p + C \sum_{k=1} \Phi(w; x_i, y_i) \tag{3.1}$$

where $C > 0$ is a penalty parameter. For SVM the two common penalty functions are $\max(1 - y_i w \cdot x_i, 0)$ and $\max(1 - y_i w \cdot x_i, 0)^2$. These two types are known as L1-SVM and L2-SVM respectively. For LR the loss function is $\log(1 + e^{-y_i w \cdot x_i})$.

[1] https://en.wikipedia.org/wiki/MATLAB
[2] https://en.wikipedia.org/wiki/GNU_Octave

3.3.3 SVMLIN

SVMLIN is a software package for linear SVMs created by Vikas Sindhwani at the Department of Computer Science at the University of Chicago [12]. SVMLIN can also utilize unlabeled data and thus implements the S^3VM algorithm introduced in section 2.4.3. The usage is very similar to LIBLINEAR as SVMLIN also has a MEX-wrapper which makes it available to MATLAB users.

3.3.4 FLANN

FLANN is a library developed by Marius Muja and David G. Lowe at the computer science department at University of British Columbia [9]. The library performs fast approximate nearest neighbor searches in high dimensional spaces. This library provides about one order of magnitude improvement in query time over the best previously available software.

Chapter 4

The original algorithm

This chapter introduces Precise Biometrics' current training algorithm, $A_P(S)$, for the matching module. Section 4.1 expands on the theory established in chapter 2. Section 4.2 looks at the specifics of the original training algorithm.

4.1 Introduction

The focus in this project lies on the matching module; more specifically, on the second and third sub-modules as introduced in section 1.1. In section 2.1, these sub-modules were formalized using the $\lambda(\mathbf{F})$ function that makes a binary decision given a feature vector $\mathbf{F}$ that contains the similarity information of two templates. The λ function should of course be such that the correct decision is given with high probability.

Section 2.1 describes the λ function as a composite of the functions β and γ. The β function produces a similarity score given a feature vector, while the γ function makes a binary decision given a similarity score. These functions have previously only been introduced in terms of domains and co-domains but will now be explored further.

The β and γ functions used by Precise Biometrics are on the forms $\beta(\mathbf{F};\mathbf{w}) = \mathbf{F}\cdot\mathbf{w}$ and $\gamma(s;t) = sgn(s-t)$. Thus, $\lambda(\mathbf{F};\mathbf{w},t) = \gamma(\beta(\mathbf{F};\mathbf{w})-t) = sgn(\mathbf{F}\cdot\mathbf{w}-t)$ which is recognized as a linear predictor, due to section 2.4.1. Section 2.3 introduced the concept of training data S and a training algorithm $A(S)$ as objects needed to find a predictor. The details of the training data that is used by Precise Biometrics was specified in section 3.1. This chapter explores Precise Biometrics' training algorithm, denoted A_P. The details of the functions that are used in the implementation of A_P can be found in section 8.1. These details are best understood after reading this chapter.

4.2 The training algorithm $\mathbf{A_P}$

Since λ is a linear predictor it would be natural to use one of the linear training algorithms, A_L, presented in section 2.4 on a dataset S to find a predictor. However, one immediately encounters a few issues when using this approach. First, the datasets used in this project and by Precise Biometrics in general are, as seen in section 3.2.5, on the scale of millions of feature vectors. The biggest dataset that is considered in this project takes up 12 GB when loaded into memory. With the computer setup used in this project it is unfeasible to work directly on datasets of this magnitude.

Secondly, the predictor should generalize between fingers, meaning that it should not work just on one specific kind of finger but on many. Therefore the performance of the predictor on the individual feature vectors in S should not hold the same importance as how well it does on the multifeatures. Also, as explained previously, the genuine feature vectors in S can be expected to contain a high degree of noise due to lack of overlap between verification templates and most templates in multitemplates.

Third, the A_L algorithms both need two hyperparameters C and C_P and there is no universal rule on how to choose these parameters.

To somewhat relieve the first two issues, the first step in A_P is to extract a subset $U \subset S$. This is done by using the function

$$\mu_P(\mathcal{F};\beta) = arg\max_{\mathbf{F}\in\mathcal{F}} \beta(\mathbf{F}) \tag{4.1}$$

on every multifeature, $\mathcal{F}_k$, in S. This function selects the highest scoring feature vector in a multifeature where the score is calculated according to a score function β. Hence, U contains one vector from each multifeature. Depending on the number of multifeatures in S, the subset U can be considerably smaller. In section 3.2 the difference in size between S and U for the datasets used in this project can be seen. In addition to reducing the size of the dataset, μ_P also acts as a filter for removing some noise depending on the β function used.

U is then used as input to an inner training algorithm, $\widehat{A}_P(U)$, the details of which can be found in section 4.2.1. This algorithm returns the linear predictor $\widehat{\lambda}_P$ and the corresponding score function $\widehat{\beta}_P$. It is thus possible to select a new subset from S using $\widehat{\beta}_P$.

The A_P algorithm consists of a few iterations like this were a subset is used to train a predictor that is used to select a subset. An initial score function $\widehat{\beta}_P^0$ produces the subset/predictor progression

$$U^n = \mu_P(S;\widehat{\beta}_P^{n-1}) \tag{4.2}$$

$$\widehat{\lambda}_P^n = \widehat{A}_P(U^n). \tag{4.3}$$

Let the sequence of score functions after K iterations be $B = (\widehat{\beta}_P^1, ..., \widehat{\beta}_P^K)$. The best performing score function is then chosen as the final score function using

$$\beta_P(B) = arg\min_{\beta\in B} FRR_{\mu_P(S,\beta)}(\beta, t) \tag{4.4}$$

The predictor returned by $A_P(S)$ is thus the predictor λ_P that corresponds to β_P. Figure 4.1 illustrates the training process.

4.2.1 The inner training algorithm $\widehat{\mathbf{A}}_{\mathbf{P}}$

As described in the previous section, the input to the inner training algorithm, $\widehat{A}_P$ is the subset $U \subset S$ where S is the full dataset. The core of $\widehat{A}_P$ are calls to the algorithm $A_{SVM}(E)$ for $E \subset U$. Along with a dataset, A_{SVM} expects two hyperparameters, C and C_p. Suitable parameters are found using the parameter selection method described in section 2.4.4, i.e. a crossvalidated grid search. The crossvalidation is a version of 10-fold where the subset E that is used as input to A_{SVM} is 1/10:th the size of U. However, instead of using $U \backslash E$ for performance evaluation of $A_{SVM}(E)$ as would be expected for k-fold crossvalidation, U is used.

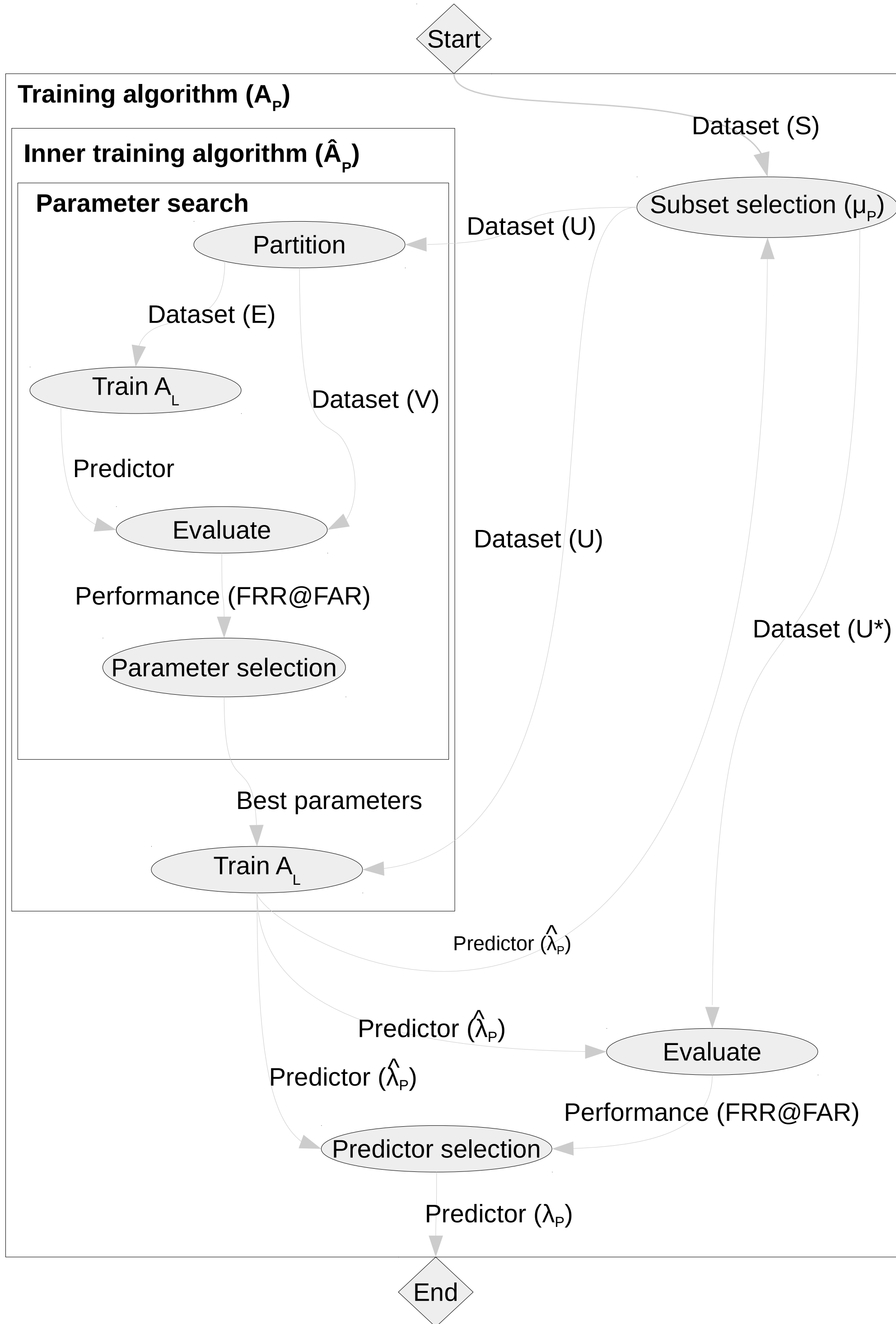

Figure 4.1: The algorithm A_P. A subset U is selected from a dataset S. Then, U is used in the inner algorithm $\widehat{A}_P$ to train a predictor $\widehat{\lambda}_P$. The predictor $\widehat{\lambda}_P$ is found by performing a crossvalidated parameter search and then training a predictor on U with the parameters that were found. A new subset can be selected using $\widehat{\lambda}_P$. The predictor $\widehat{\lambda}_P$ is evaluated on the new subset. The new subset can also be used in $\widehat{A}_P$ to perform additional training iterations.

Chapter 5

The new algorithm

This chapter introduces the new training algorithm $A_N(S)$. Section 5.1 describes the overall strategy motivating the changes. Section 5.2 contains the details of the new training algorithm. In section 5.3 and 5.4, the new data selection functions are presented. Section 5.5 describes a feature manipulation method that is outside the scope of this project but could nevertheless be interesting to consider.

5.1 Introduction

The objective of this project is to investigate if and how the original training algorithm A_P can be improved. The main strategy in the quest for improving A_P is to generalize the data selection function μ_P. Since μ_P selects only one feature vector per multifeature, there is a possibility that valuable information is lost. The functions used in the implementation of the new training algorithm, A_N, are listed in section 8.2 along with some design motivations.

5.2 The training algorithm $\mathbf{A_N}$

Similar to the original training algorithm A_P, the input to the new training algorithm A_N is a set S of of the form seen in section 3.1. The subset selection in the original training algorithm was handled by the function μ_P which selects a subset U from the full dataset S. In A_N, this subset selection is handled using the new function $\mu_N(S)$ (details in section 5.3). Aside from μ_N and some changes in the implementation of A_N in code, A_N and A_P is similar at this stage. The major changes are in the inner training algorithm $\widehat{A}_N$. Figure 5.4 shows the structure of A_N.

5.2.1 The inner training algorithm $\widehat{\mathbf{A}}_\mathbf{N}$

As with the original inner training algorithm, $\widehat{A}_P$, the input to the new training algorithm, $\widehat{A}_N$, is a subset U of the full dataset S. In A_N this subset is selected using μ_N. The algorithm returns the predictor $\widehat{\lambda}_N = \widehat{A}_N(U)$. Similar to $\widehat{A}_P$, $\widehat{A}_N$ also employs a crossvalidated grid search method to find the hyperparameters C and C_p. However, the subset $E \subset U$ that was used as input to the A_L algorithm in $\widehat{A}_P$ is first processed by a preprocessing function $\rho(E)$ (section 5.4). Different ρ functions need a different set of parameters. In the implementation of $\widehat{A}_N$, the original grid search has been extended to also include a parameter search for the selected ρ function.

A new A_L algorithm is introduced in the new training structure, the S^3VM algorithm (section 2.4.3). This method can handle a mixture of unlabeled and labeled data.

To achieve a good separation of training and test sets, the cross validation function has been changed to employ a 50/50-split instead of the previous 90/100-split. This means that U is partitioned into two sets, E and V, of roughly equal size. Dataset E is used to find the predictor $A_L(E)$ and V is used in the FRR calculation.

5.3 The selection function $\mu_\mathbf{N}$

The new training data selection method μ_N is an extension of the original μ_P function. While μ_P selects the highest scoring feature vector in a multifeature according to a score function, μ_N allows selection of multiple of the highest scoring feature vectors from each multifeature. How many can also depend on whether the the multifeature is a genuine or an impostor. Three parameters are required for μ_N; the number of genuines, N_G, to include in E, the number of impostors, N_I, to include in E and the number of genuines and impostors, N, to include in V.

5.4 The filter functions ρ

This section introduces the new inner training data filter methods $\rho(E)$ for the dataset E. These methods are classified as either additive or subtractive depending on whether they add or or remove data points from E.

The genuines are the minority class and should therefore be upsampled in some way. It is possible to use both the additive and the subtractive methods to upsample the genuines, if the ρ methods are combined with a suitable μ_N method. By choosing a μ_N method that selects many feature vectors from each genuine multifeature, for example the five highest scoring feature vectors instead of only the single highest, a subtractive ρ method could then be used to remove some elements from E. When using the additive ρ functions, E should however probably contain the best genuine candidates. By using the μ_N that chooses only the highest scoring feature vector in each multifeature, one acquires a small

but confident dataset. By allowing more feature vectors per multifeature, a bigger but less confident dataset is acquired.

In the implementation of ρ (section 8.2), the subtractive methods do not actually remove elements from E. They merely relabel feature vectors from ± 1 to 0 or 2 depending on the method of choice. The implementation of the SVM algorithm ignores feature vectors that are labeled 0 or 2. However, S^3VM uses the 0-labeled vectors as unlabeled but ignores the 2-labeled vectors. The following subsections provide a description of the different ρ methods investigated during this project. Table 5.1 contains a brief summary of the methods.

Preprocessing method ρ	**Type**
ADASYN	Additive
SMOTE	Additive
Grey data	Subtractive
Mark below	Subtractive
Random downsampling	Subtractive
Recursive	Subtractive
Vote	Subtractive

Table 5.1: Data preprocessing methods used in this project.

5.4.1 ADASYN

Adaptive Synthetic Sampling Approach for Imbalanced Learning (ADASYN) generates more data for the minority class examples that are difficult to classify using a weighted distribution. In [5] the ADASYN algorithm is described as follows:

ADASYN

Input
Training dataset D_{tr} with m samples $\{\mathbf{x}_i, y_i\}$, $i = 1, ..., m$, where $\mathbf{x}_i$ is an instance in the n dimensional feature space X and $y_i \in Y = \{-1, 1\}$ is the class identity label associated with $\mathbf{x}_i$. Define m_s and m_l as the number of minority class examples respectively. Therefore, $m_s \leq m_l$ and $m_s + m_l = m$.

Procedure
(1) Calculate the degree of class imbalance

$$d = m_s/m_l \tag{5.1}$$

where $d \in (0, 1]$.
(2) If $d < d_{th}$ then (d_{th} is a preset threshold for the maximum tolerated degree of class imbalance ratio):

(a) Calculate the number of synthetic data examples that need to be generated for the minority class:

$$G = (m_l - m_s) \times \theta \tag{5.2}$$

Where $\theta \in [0, 1]$ is a parameter used to specify the desired balance level after generation of the synthetic data. $\theta = 1$ means a fully balanced dataset is created after the generalization process.

(b) For each example $\mathbf{x}_i \in minorityclass$, find K nearest neighbors based on the Euclidean distance in n dimensional space, and calculate the ratio r_i defined as

$$r_i = \Delta_i / K, \quad i = 1, ..., m_s \tag{5.3}$$

where Δ_i is the number of examples in the K nearest neighbors of $\mathbf{x}_i$ that belong to the majority class, therefore $r_i \in [0, 1]$;

(c) Normalize r_i according to $\hat{r}_i = r_i / \sum_{i=1}^{m_s}$ so that $\hat{r}_i$ is a density distribution ($\sum_i \hat{r}_i = 1$).

(d) Calculate the number of synthetic data examples that need to be generated for each minority example $\mathbf{x}_i$,

$$g_i = \hat{r} \times G \tag{5.4}$$

where G is the total number of synthetic data examples that need to be generated for the minority class as defined in equation 5.2. (e) For each minority class data example $\mathbf{x}_i$, generate g_i synthetic data examples according to the following steps:

– Do the **Loop** from 1 to g_i

(i) Randomly choose one minority data example, $\mathbf{x}_{zi}$ from the K nearest neighbors for data $\mathbf{x}_i$.

(ii) Generate the synthetic data example

$$\mathbf{s}_i = \mathbf{x}_i + (\mathbf{x}_{zi} - \mathbf{x}_i) \times \lambda \tag{5.5}$$

where $(\mathbf{x}_{zi} - \mathbf{x}_i)$ is the difference vector in n dimensional space, and λ is a random number: $\lambda \in [0, 1]$.

– End **Loop**

The key idea of the ADASYN algorithm is to use a density distribution $\hat{r}_i$ as a criterion to automatically decide the number of synthetic samples that need to be generated for each minority data example. $\hat{r}_i$ is a measurement of the distribution of weights for different minority class examples according to their level of difficulty in learning. The resulting dataset post ADASYN will not only provide a balanced representation of the data distribution (according to the desired balance level defined by the θ coefficient), but it will also force the learning algorithm to focus on those difficult to learn examples. Two parameters are needed as input, θ and K.

5.4.2 SMOTE

Synthetic Minority Oversampling Technique (SMOTE) is a technique in which the minority class (in this project, the genuines) is oversampled [4]. Instead of creating exact copies of existing minority class samples like in a simple upsampling approach, SMOTE interpolates between samples to create synthetic samples. The synthetic samples are created by introducing a new sample along each of the lines connecting to the k nearest neighbors of each minority class sample. One parameter, $a \in \mathbb{N}$ is needed for this algorithm. Let the number of genuines in the training set be N_G. After applying this method, the new number of genuines will be $\hat{N}_G = a \cdot N_G$.

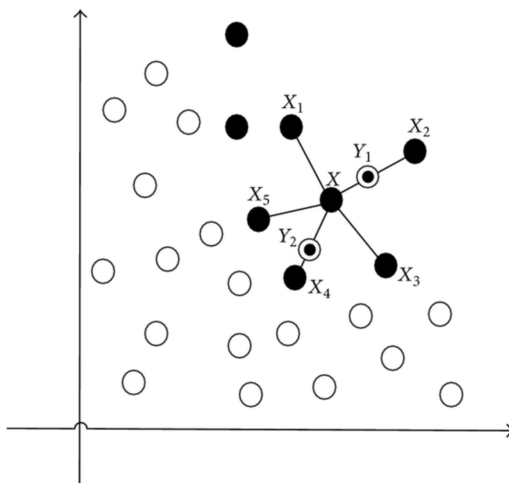

Figure 5.1: The SMOTE algorithm. The black class is being upsampled. The five nearest neighbors to the point X are X_1 through X_5. A new data point is being introduced along the lines connecting to the neighbors.

5.4.3 Grey data

This function assigns the label 0 to all but the highest scoring vector in each multifeature. If E contains only the highest scoring features from all multifeatures, the functions does nothing. Since the original SVM algorithm ignores vectors labeled 0, this function should be used together with S^3VM.

5.4.4 Mark below

Mark below calculates the average score, s_{avg} of the highest scoring 10% of genuines in E. Given a variable $a \in \mathbb{R}$, the method then relabels data points whose score s is less than $s < a \cdot s_{avg}$. The new label is 0.

5.4.5 Random downsampling

The random down-sampling approach is among the simpler approaches to class balancing. This method randomly chooses data points to remove or reclassify. Unlike the up-sampling methods which are applied to the minority class, this method is applied to the majority class. There are pros and cons with this method. A predictor's confidence and training time is generally positively correlated with the size of the dataset; a bigger dataset produces a more confident predictor but it takes longer to find it. The random down-sampling method removes data points randomly from the training set. Two parameters are required as it is possible to remove points both from the genuines and the impostors. Assume that the number of genuines and impostors in the training set is N_G and N_I respectively. Given parameters $a, b \in (0, 1]$ the new number of genuines and impostors are $\hat{N}_G = a \cdot N_G$ and $\hat{N}_I = b \cdot N_I$.

5.4.6 Recursive

In [6] it is suggested that the same algorithm can be used to train both the predictor and a filter. This method thus finds a predictor λ_L using $A_L(E)$. Then, λ_L (or rather its score function β_L) is applied to the same set E to remove the genuines with the worst score. The name of this method is due to it being implemented recursively, meaning that the method allows for a filter to be found using $A_L(\rho(E))$ when ρ is the recursive method. The parameter needed for this function is the number of recursive calls that should be made. This method is illustrated in figure 5.2.

5.4.7 Voting filter

The K-nearest neighbors algorithm has, since 1972, been used extensively as a filter method [6]. The voting filter relabels a data point according to a majority vote among its nearest nearest neighbors. If a majority of the neighbors surrounding a genuine are impostors, the genuine is relabeled to 0. Two parameters are needed for this method; the number of neighbors to use and how many of those neighbors that need to be impostors in order for the data point to be relabeled.

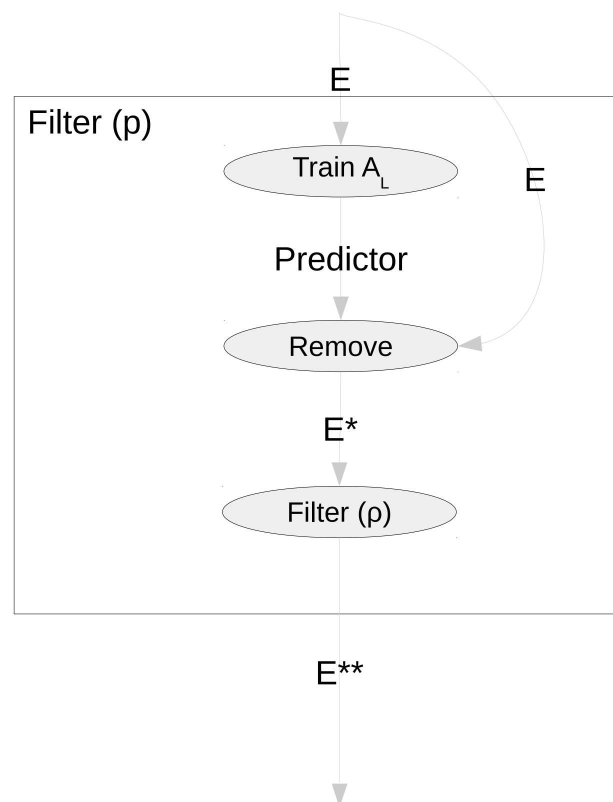

Figure 5.2: The recursive filter. Dataset E is used to train a predictor. The predictor is then used to relabel or remove the points from E that were not classified correctly, thus creating dataset E^*. Depending on recursive depth, E^* can be used as input to the same filter.

5.5 Feature transformation

Note: the theory suggested in this section can not currently be easily implemented due to Precise Biometrics' system architecture. These changes are nevertheless interesting to consider.

None of the algorithmic changes so far have made any transformations to the feature vectors in S. Selecting the feature vector with the highest score from each multifeature is done under the assumption that the most characteristic information contained in the multifeature can be fairly well represented by this single vector. This is often true if the enrollment and the verification templates are from roughly the same parts of the same finger so that there is a good amount of overlap. However, there is a risk that the verification template does not overlap one single enrollment template well. In this case the verification template may instead overlap several enrollment templates, but only partially. Selecting only the highest scoring feature vector to represent the whole multifeature could then be a bad representation of the multifeature characteristics.

A simple method of capturing some extra information about the multifeature that was implemented during this project is to concatenate each feature vector with its corresponding multifeature centroid.

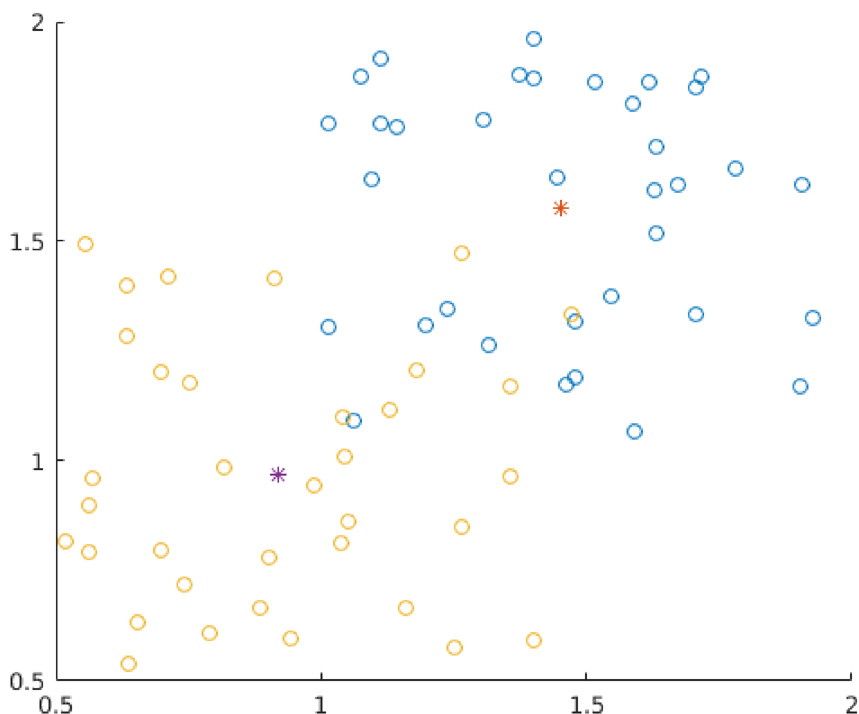

Figure 5.3: The centroid method. The stars are centroids of two multifeatures. In this figure, the features are in $\mathbb{R}^2$. Concatenating each feature with the centroid extends the features to $\mathbb{R}^4$.

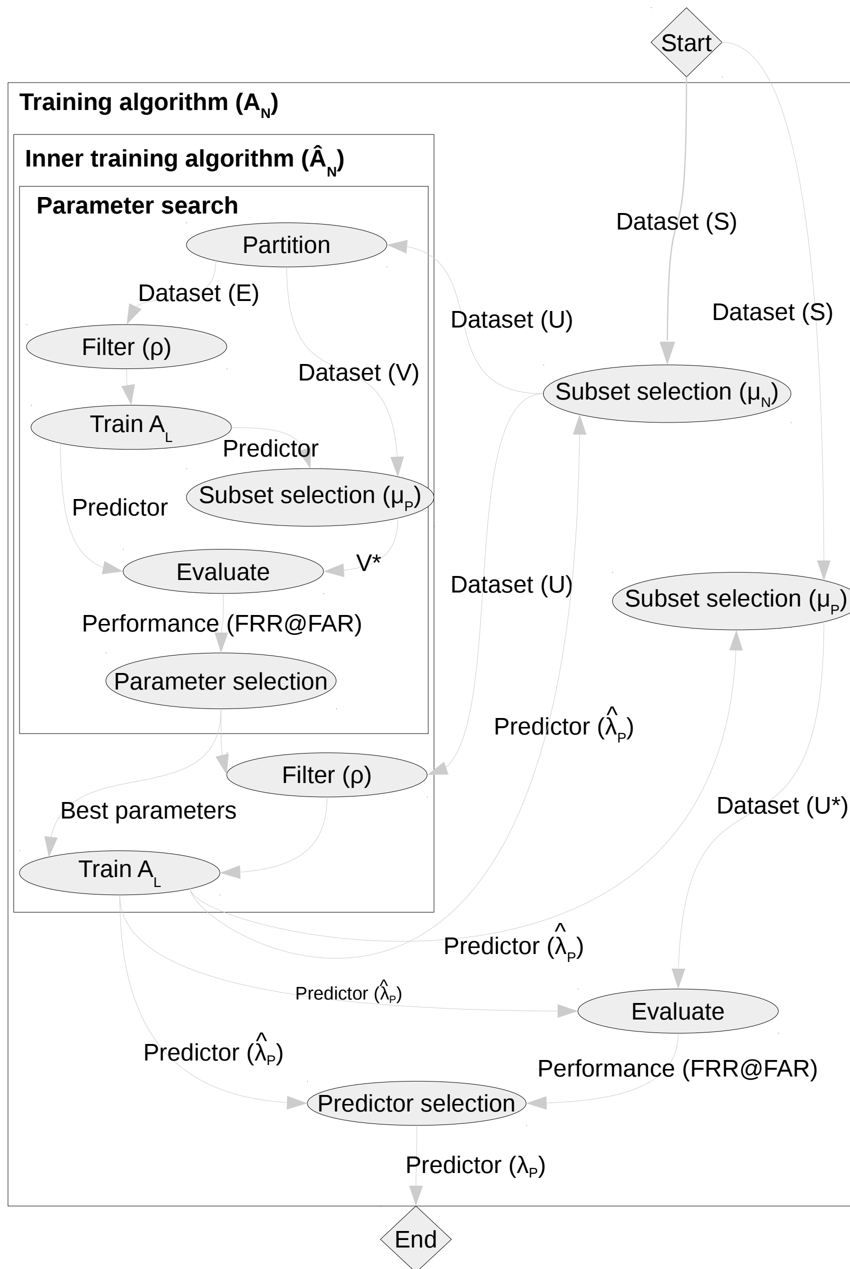

Figure 5.4: The new training algorithm A_N. The input is the full dataset S. The selection function μ_N is used to select a subset U which is used as input to the inner training algorithm $\widehat{A}_N$. During the parameter search, U is partitioned into the datasets E and V. Using a filter function, ρ, E is filtered before training using A_L. The predictor returned from $A_L(\rho(E))$ is evaluated on V. Once the best parameters have been found, the predictor that is returned from $\widehat{A}_N$ is found using a filtered version of U. The predictor can be used to select a new subset from S and once again train a predictor using $\widehat{A}_N$. Each predictor returned by $\widehat{A}_N$ is evaluated on a subset of S that is selected using the predictor and the original selection function μ_P. The best performing predictor is chosen as the final predictor.

Chapter 6

Results

This chapter presents the effects of the changes made to the algorithm A_P during this project. Section 6.1 describes how these effects are measured. Section 6.2 - 6.5 presents the results.

6.1 Evaluation procedure

When creating the algorithm A_N, the old algorithm A_P was used as a base. The changes mainly consist of the new selection function μ_N (section 5.3), the preprocessing functions ρ (section 5.4) and the additional A_L algorithm S^3VM. By varying the parameters of the μ_N function along with different combinations of ρ functions and A_L algorithms, different versions of the A_N algorithm are acquired. This chapter is dedicated to investigating the performance of some versions of A_N.

The predictors (or rather the associated score functions) returned by the different versions are evaluated with the FRR@FAR measure (section 2.3.1). To calculate the FRR@FAR for a predictor, a few things are required. Beyond the predictor itself one also needs to specify the desired FAR level and provide a dataset to test the predictor on. A good predictor should give low FRR on a low FAR level but due to the size of the datasets in this project it is not possible to calculate a confident FRR on very low FAR levels. The predictors in this project are thus evaluated using FRR@1/10000. Table 6.1 shows how many falsely accepted impostors this actually implies for the different datasets S_A, S_B and S_C used in this project. The most confident results are therefore going to be the ones acquired from dataset S_B and the least from S_A.

Dataset	**Impostor multifeatures**	**False accepts at FAR=1/10000**
S_A	88200	8
S_B	368280	36
S_C	239610	23

Table 6.1: False accepts for the different datasets.

The dataset that is used when calculating FRR@FAR for a predictor should be selected with care. It is important that a predictor performs well on data points it has never encountered before since this is the situation that the predictor will experience in the real world. Crossvalidation is often used in order to simulate new data. However, this technique requires multiple training sessions. Depending on the dataset S and the settings used for A_N, the search for a predictor $A_N(S)$ can take a long time, sometimes days. The FRR@FAR values in section 6.2 are therefore calculated using the same dataset as used for training. This method is also the preferred method by Precise Biometrics when measuring the performance of a predictor and should be at least indicative of the performance that would be acquired by crossvalidation. An estimate of the difference between the performance reported by a non-crossvalidated versus a crossvalidated method is provided in section 6.4.

6.1.1 The baseline

To investigate whether or not the changes made to A_P during this project have any kind of effect it is important to have a baseline to compare against. The goal of this project is to examine if some dataset processing could be included in the training algorithm so as to reliably achieve a better performing predictor. In A_P, some dataset processing is already done through μ_P. The ρ functions and the S^3VM algorithm are however new to this project. The baseline algorithm hence utilizes the μ_P function but no ρ function. Furthermore, the A_L algorithm used in the original training algorithm is SVM. These settings are considered the baseline.

6.1.2 Parameters for the ρ and μ_N functions

Some of the ρ functions require one or a set of parameters to specify how to process a dataset. Through the new parameter search method (section 5.2.1) these parameters can be included in the crossvalidated grid search in $\widehat{A}_N$, along with the hyperparameters required for the A_L algorithms.

While the μ_N parameters also can be included in the new parameter search method, the range of suitable parameters is small; since the subset $U \subset S$ that is selected from S is loaded into RAM, U should not be too big. The inner training algorithm $\widehat{A}_N$ includes a crossvalidation step when searching for hyperparameters. However, this ensures that the predictor returned by the inner training algorithm $\widehat{A}_N(U)$ does not overfit on dataset U. Depending on how well U represents the characteristics of S this could also imply that the predictor does not overfit on S. To hopefully increase the chance that the important characteristics in S are also in U one can include more feature vectors per multifeature in U using the μ_N function.

The three parameters needed for the μ_N function are in this chapter denoted N_G, N_I and N as introduced in section 5.3.

6.1.3 Initial subset

The initial subset $U^1 \subset S$ in A_N depends on an initial score function $\widehat{\beta}_N^0$. Each feature vector in the datasets used in this project is connected to a score that has been assigned by a previous score function, β_P. This is the score function given by a few iterations in the original A_P algorithm. A possible initial subset could then be selected using the scores in the datasets. In this case, A_N could be considered as a continuation of A_P. However, if A_N is to be considered as a replacement for A_P it is informative to also examine to what extent A_N:s performance depends on the initial score function. This is evaluated by also performing tests using a random initial score function β_R.

6.2 Algorithmic performance using $\widehat{\beta}_{\mathbf{N}}^{\mathbf{0}} = \beta_{\mathbf{P}}$

By using $\widehat{\beta}_N^0 = \beta_P$ to select U^1 using the μ_N function it can be expected that the initial selection is already relatively good so only a few iterations in A_N should be needed. The number of iterations are set to 10. The tests are also performed using a parameter search for the ρ functions that need it. This parameter search is, however, not extensive in order to keep the training time down. Instead, each ρ function is supplied with a small set of parameter candidates among which the parameter search method selects the best suited. The results of this case are presented in table 6.2. The baseline algorithm is highlighted with blue. The best settings found on each dataset is highlighted with green. Since tests using dataset S_C can take days to complete for some settings, some tests are not performed on this dataset. The tests on S_C were mostly performed after observing a good result for some settings on S_A or S_B.

As seen in table 6.2, there are settings that perform better than the baseline settings. There is however no setting that manages to considerably outperform the baseline for all three datasets. The setting that seems to have the highest chance of providing a universal improvement is to switch from A_{SVM} to A_{S^3VM}, use a recursive filter and performing the parameter search on the five highest scoring feature vectors in each multifeature. This setting manages to outperform the baseline on both S_A and S_C while the result on S_B is somewhat improved.

Increasing N (and thereby increasing the resemblance between subset U and dataset S when performing evaluations in $\widehat{A}_N$) seems to provide a slight increase in performance, at least when using A_{SVM}. As could be expected, increasing N_G from 1 to 2 or 5 without also including some ρ function decreases the performance since this introduces noise that is not filtered out.

N_G	N_I	N	A_L	ρ	$\mathbf{FRR_A}$	$\mathbf{FRR_B}$	$\mathbf{FRR_C}$
1	1	1	SVM	None	1.43	2.59	10.71
1	1	1	SVM	ADASYN	1.43	2.94	-
1	1	1	SVM	Random	1.37	2.62	10.59
1	1	1	SVM	Recursive	1.47	2.76	10.87
1	1	1	SVM	SMOTE	1.37	2.62	10.79
1	1	5	SVM	None	1.37	2.59	10.65
1	1	5	SVM	ADASYN	1.47	2.70	-
1	1	5	SVM	Random	1.53	2.64	10.82
1	1	5	SVM	Recursive	1.37	2.60	11.07
1	1	5	SVM	SMOTE	1.27	2.66	10.75
2	1	5	SVM	None	1.50	2.76	10.58
5	1	5	SVM	None	1.43	3.00	10.95
5	1	5	SVM	Recursive	1.33	2.86	10.79
5	1	5	SVM	Mark below	1.50	2.96	-
5	1	5	SVM	Vote	1.40	3.03	10.70
1	1	1	S^3VM	None	1.00	2.59	10.41
1	1	1	S^3VM	ADASYN	1.27	2.68	11.57
1	1	1	S^3VM	Recursive	1.17	2.54	9.83
1	1	1	S^3VM	Random	1.17	2.53	10.47
1	1	1	S^3VM	SMOTE	1.13	2.66	10.42
1	1	5	S^3VM	None	1.13	2.57	10.38
1	1	5	S^3VM	ADASYN	1.40	2.97	-
1	1	5	S^3VM	Random	1.47	2.49	10.50
1	1	5	S^3VM	Recursive	1.17	2.52	9.74
1	1	5	S^3VM	SMOTE	1.17	2.59	10.53
2	1	5	S^3VM	None	1.37	2.74	10.58
2	1	5	S^3VM	Grey data	1.13	2.57	10.54
3	1	5	S^3VM	Grey data	1.23	2.57	10.54
4	1	5	S^3VM	Grey data	1.23	2.57	10.54
5	1	5	S^3VM	None	1.47	3.16	10.83
5	1	5	S^3VM	Grey data	1.13	2.57	10.54
5	1	5	S^3VM	Mark below	1.37	2.99	10.78
5	1	5	S^3VM	Recursive	1.13	2.76	10.30
5	1	5	S^3VM	Vote	1.40	2.79	10.21

Table 6.2: FRR@1/10000 for datasets S_A, S_B and S_C. The N_G highest scoring genuines and N_I highest scoring impostors are used when training with algorithm A_L. The N highest genuines and impostors were used when evaluating the predictor returned by A_L. Initial score function $\widehat{\beta}_N^0 = \beta_P$. 10 iterations in A_N.

6.3 Algorithmic performance using $\widehat{\beta}^0_N = \beta_R$

If one instead selects U^1 using a random initial score function, i.e $\widehat{\beta}^0_N = \beta_R$, the results in table 6.3 are acquired. Comparing the FRR of each setting in table 6.3 with the corresponding FRR in table 6.2 suggest that A_N might be able to manage without an initial score provided by A_P since the acquired FRR values are similar to those acquired when using $\widehat{\beta}^0_N = \beta_P$. The new training algorithm A_N should therefore be able to function as replacement for A_P and not just a continuation.

Table 6.3 also reports the training times when using A_N on the different datasets. For the big and complex dataset S_C, this time is several hours. During some test runs it was observed that for S_C most of this time was spent in A_L. Another test using the random downsampling ρ function was therefore performed in an effort to bring down the training time. As table 6.3 shows, the training time is indeed reduced when using random downsampling for all datasets while still maintaining similar FRR values compared to the no ρ function case. If one accepts a slight FRR difference, the training time should thus be able to be reduced.

N_G	N_I	N	A_L	ρ	$\mathbf{FRR_A}$	$\mathbf{FRR_B}$	$\mathbf{FRR_C}$	$\mathbf{T_A}$	$\mathbf{T_B}$	$\mathbf{T_C}$
1	1	1	SVM	None	1.53	2.64	10.70	17	133	735
1	1	1	SVM	Random	1.40	2.70	10.69	11	90	92
1	1	1	S^3VM	None	1.07	2.62	10.46	43	237	1045
1	1	1	S^3VM	Random	1.23	2.53	10.49	17	73	118

Table 6.3: FRR@1/10000 for datasets S_A, S_B and S_C. The training times T_A, T_B and T_C are in minutes. The N_G highest scoring genuines and N_I highest scoring impostors are used when training with algorithm A_L. The N highest genuines and impostors were used when evaluating the predictor returned by A_L. Initial score function $\widehat{\beta}^0_N = \beta_R$. 10 iterations in A_N.

6.3.1 Convergence

The interior of algorithms A_P and A_N both produce a sequence of score functions $B = (\beta^1, ..., \beta^n)$ as seen in section 4.2. By measuring the performance of these score functions on the full dataset S, a sequence of FRR@FAR values is acquired. This sequence can be used to evaluate the convergence of A_N for specific settings. The sequence for dataset S_B is shown in figure 6.1. The solid line represents the FRR evolution given the initial score function $\widehat{\beta}^0_N = \beta_P$. The dashed line instead uses $\widehat{\beta}^0_N = \beta_R$. After four iterations the FRR values are roughly the same regardless of the initial score function used. The settings that are used are $N_G = N_I = N = 1$ and A_{SVM} and no ρ function.

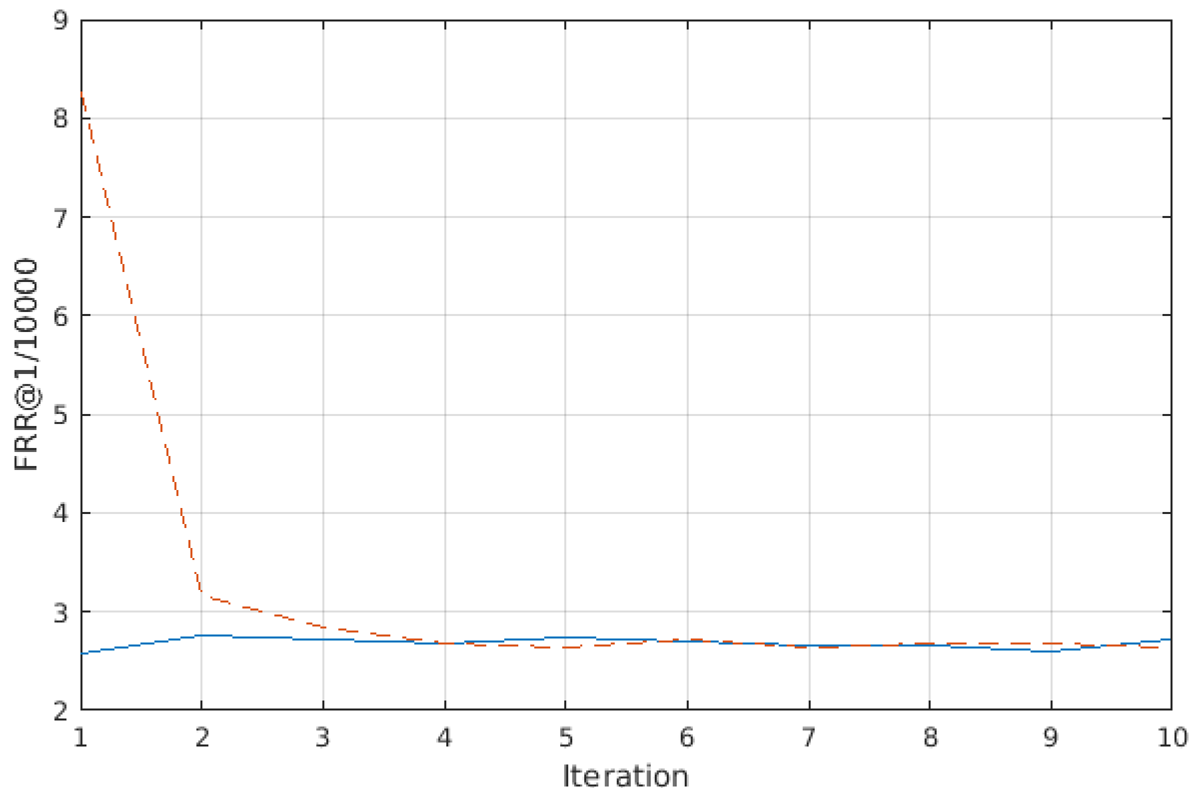

Figure 6.1: Convergence for dataset S_B with $N_G = N_I = N = 1$, A_{SVM} and no ρ function. Solid line is training using the initial score function $\widehat{\beta}_N^0 = \beta_P$ while dashed uses $\widehat{\beta}_N^0 = \beta_R$. 10 training iterations in A_N.

6.4 Overfitting

The implementation of the new training algorithm includes an evaluation method that measures the degree of overfitting experienced in $A_N(S)$. This method uses a 5×2-crossvalidation as seen in section 2.3.3. Algorithm A_N is thus evaluated ten times on different datasets. The subset $S_1 \subset S$ is used to find the predictor λ_N returned by $A_N(S_1)$. Another subset $S_2 \subset S$ is then used to evaulate the performance of λ_N. Since the crossvalidation method is 5×2 the subsets S_1 and S_2 are (roughly) equal in size and disjoint. The settings used for A_N are the same as in section 6.3.1. Datasets S_B and S_C were used for this evaluation.

For S_B, the mean difference of the performance measured on S_2 compared to S_1 is 1.60% using $\widehat{\beta}_N^0 = \beta_P$ and 1.07% using $\widehat{\beta}_N^0 = \beta_R$. For S_B, these numbers are 0.54% and 0.61% respectively. Thus, A_N does seem to experience some overfitting using these settings, more so on dataset S_B compared to S_C.

6.5 Feature transform - Centroid method

Section 5.5 introduced a feature transformation method which consisted of appending the multifeature centroid to the feature vectors in the dataset. As noted, this theory can not be easily implemented due to the surrounding architecture of Precise Biometrics' system. Hence, no extensive testing has been done with this method. However, the method was tested on dataset S_A with A_{SVM}, the original selection function μ_P and no ρ and 10 training iterations. The centroid method achieved a FRR@1/10000 of 0.33%. This is considerably better than the results presented for S_A in section 6.2. The centroid dataset was generated from S_A through a function that could easily handle a set of S_A:s size. To generate similar datasets from S_B and S_C would require a modification of this function but since the centroid method was deemed unimplementable due to the surrounding system architecture, similar tests for S_B and S_C were not performed.

Chapter 7

Conclusions

None of the dataset processing techniques explored in this project is able to considerably improve on the originally used technique on all three datasets. The $\rho_{recursive}$ function for example, produces the best results among the tested ρ functions on dataset S_C. The same good performance can, however, not be seen on the other datasets using this function. It might be the case that a more careful parameter search could make a technique perform well on a general dataset. However, the reason for not doing an extensive ρ function parameter search in this project was simply due to the considerable time it would take. It could therefore be beneficial to explore methods of reducing the training times. One such method that was explored in this project was to downsample the impostors randomly. This reduced the training time while still maintaining a similar performance compared to the baseline.

The downsampling method reduces the time spent in the A_L algorithms. Another major time sink observed when implementing A_N was the loading of large amounts of data from disk to memory. A time efficient algorithm implementation should thus avoid unnecessary loadings. For precise memory control, Matlab/Octave might not be the best choice of language.

The performance of A_N does not seem to be heavily influenced by the initial subset selection when using the baseline settings so it should be able to function as a replacement for A_P.

7.1 Future work

7.1.1 Feature transformation

The centroid method presented in section 5.5 showed promising results on dataset A. Even though it is not possible at the moment to implement this technique due to Precise Biometrics' surrounding system architecture it could be interesting to investigate this technique on bigger datasets in case it proves to be a successful technique. The centroid technique suggest that there might be other, better ways of capturing the characteristics of

a multifeature compared to simply choosing one of the feature vectors in the multifeature according to some criteria as is done today.

7.1.2 Robust SVM

During the final stages of this project an interesting paper named *Support Vector Machines Under Adversarial Label Noise* was found [8]. This paper presented a robust SVM, meaning an SVM that was built under the assumption that the labels contain noise. The robust SVM is formulated using a modification of the ordinary SVM dual problem with only a relatively simple modification of the kernel. It is reported to not add much complexity beyond the original SVM.

Chapter 8

Appendix

The appendix contains descriptions of functions and function dependencies from the original and the new training structure. Provided is also a usage example for the new training structure.

8.1 The original training structure

This section contains the implementation details of the original training algorithm A_P. This algorithm is implemented using a mixture of Matlab/Octave and C functions. The subset selection μ_P is implemented in C while the inner training algorithm $\widehat{A}_P$ uses Matlab/Octave functions with the exception of A_L which uses the LIBLINEAR package (i.e C/C++). Algorithm $\widehat{A}_P$ corresponds to the function **pb_ml_train** in the list of functions in section 8.1.1.

8.1.1 Functions

The following are short descriptions of the Matlab/Octave functions in the implementation of $\widehat{A}_P$. These can also be found in figure 8.1 where arrows indicate function dependencies.

pb_ml_train
The main function for the inner algorithm $\widehat{A}$. Trains a predictor given a dataset.

pb_ml_filter_features
Removes feature dimensions that have the same value for all instances in the training set.

pb_ml_load_features
Loads the training set from .txt files.

pb_ml_normalize_features
Normalizes the features using the method of choice.

pb_ml_print_model
Prints weight vector and the scale parameters of the trained predictor to a C-file for use in the next step in the training loop.

pb_ml_train_model
Control function for the hyper-parameter search and crossvalidation.

pb_ml_crossvalidate
Performs a crossvalidation over a grid of hyper-parameters given a set of hyper-parameters.

pef_plot_roc_simple
Plots a ROC curve for a predictor.

pb_ml_liblinear_predict Predicts labels for a dataset given a linear predictor.

pef_compute_frr_at_far
Computes FRR given a predictor, dataset and a FAR value.

pb_ml_liblinear_train Wrapper function for the LIBLINEAR package. Trains a predictor given a training set.

8.2 The new training structure

Nearly every function that forms the implementation of the original training algorithm A_P has been changed when designing and implementing the new training algorithm, A_N although the general structure is the same. The modifications and additions to A_P were made with two criteria in mind; speed and versatility. When working with datasets of the magnitude seen in this project, poorly designed functions could slow down the training time considerably. Hence, whenever possible, functions were constructed using a time saving design. To achieve a better versatility, A_P, was modified according to a *plug and play* philosophy so that ρ and A_L functions could easily be switched. This would also make it easy to investigate new such functions after this project has ended.

A major new feature in the implementation of A_N is that all the code is gathered under one main function. In the original training structure, the function **pb_ml_train** implements $\widehat{A}_P$. In the new training structure **pb_ml_train** corresponds to A_N while the inner training algorithm, $\widehat{A}_N$ is implemented using **pb_ml_train_model**.

Collecting the entire training process under one main function makes it easier for users to handle to training process.

8.2.1 Functions

The following are short descriptions of the important functions in the new training structure. These can also be found in figure 8.2 where arrows indicate function dependencies.

pb_ml_train
The main function for the training algorithm A_N. Trains a predictor given a dataset.

calculate_scores
Assigns a score to each point in a dataset.

pb_ml_predict
Calculates a score for each point in a dataset.

pb_ml_predict_linear_score
MEX-wrapper for the C function pb_ml_predict_linear_score

add_dataset_info
Extracts information such as the number of people, fingers, enrollment templates and verification templates.

pb_ml_train_model
The main function for the inner training algorithm $\widehat{A}_N$.

new_original_search
Initialization method for the grid search for the parameters required in the A_L algorithms.

grid_search
Searches for best parameters among a given set.

pb_ml_crossvalidate
Crossvalidation function.

cont_eval
Calculates FRR from a set of scores.

partition_50_50
Partitions a dataset into two (roughly) equally sized parts.

select_subset
Selects a training and test set according to the given N_G, N_I and N parameters (section 5.3).

select_training_data
Filters the training data according to the chosen ρ function.

recursive
The recursive ρ function.

flann_select
The majority vote ρ function.

SMOTE
The SMOTE ρ function.

ADASYN
The ADASYN ρ function.

random_sampling
The random downsampling ρ function.

mark_below
The mark below ρ function.

get_model
Trains using the selected linear training algorithm A_L.

compute_rank
Ranks the features in each multifeature with respect to a score function.

8.2.2 An example

Algorithm	**Alias**
Support Vector Machine	linear svm
Semi-Supervised Support Vector Machine	linear svm semi supervised
Logistic Regression	linear lr

Table 8.1: Available algorithms and corresponding code alias.

The following table contains the pre-processing methods that have been implemented during this project. Vectors in this table are row vectors. Further explanations of these methods are referred to section 5.4.

Method	**Alias**	**Parameter format**	**Parameter range**
SMOTE	smote	a	$a \in \mathbb{N}^+$
ADASYN	adasyn	[a b]	$a \in \mathbb{N}^+, b \in$
Grey data	delabel	-	-
Random downsampling	random	[a b]	$a, b \in (0, 1]$
Recursive	recursive	a	$a \in \mathbb{N}^+$
Mark below	mark below	a	$a \in \mathbb{R}$
Vote	flann	[a b]	$a, b \in \mathbb{N}^+, a \geq b \in (0, 1]$

Table 8.2: Pre-processing methods.

The following is a MATLAB script showing an example of how to specify options and what type of options that can be specified in the new training environment.

```
% Folder with genuines and impostors files.
options.folder='./Features/dataset_a';
% Number of highest genuines and impostors to test
% on in each crossvalidation iteration.
options.test_on_max=1;
% Number of highest genuines to train
% on in each crossvalidation iteration.
options.training_set_genuines=1;
% Number of highest impostors to train
% on in each crossvalidation iteration.
options.training_set_impostors=1;
% Model to use during training.
options.modeltypes='linear svm';
% Training data selection method.
options.data_selectors='flann';
% Parameters to use for training data selection.
options.training_ops=[5 2];
% Number of crossvalidation iterations to use.
options.crossval_folds=10;
% Number of training iterations to run.
options.training_iterations=10;
% Number of evaluation iterations to run.
options.evaluation_iterations=0;
% FAR level to evaluate the results on.
options.fars=[1/10000];
% Level of console outputs during training.
options.verbosity=2;
% Train using the square of each feature.
options.use_square=1;
% Use the standard deviation method to find
% scaling parameters.
options.use_std=1;
% Block size to read into memory (approximate).
options.block_size_bytes=1e9;
% Use random initial subset selection.
options.initial_zero_score=0;
% Run test
[frrs,ref_frr,eval_frr,time,w,scale,discarded_cols]=pb_ml_train(options);
```

Figure 8.1: Structure of the implementation of $\widehat{A}_P$. The arrows show function dependencies.

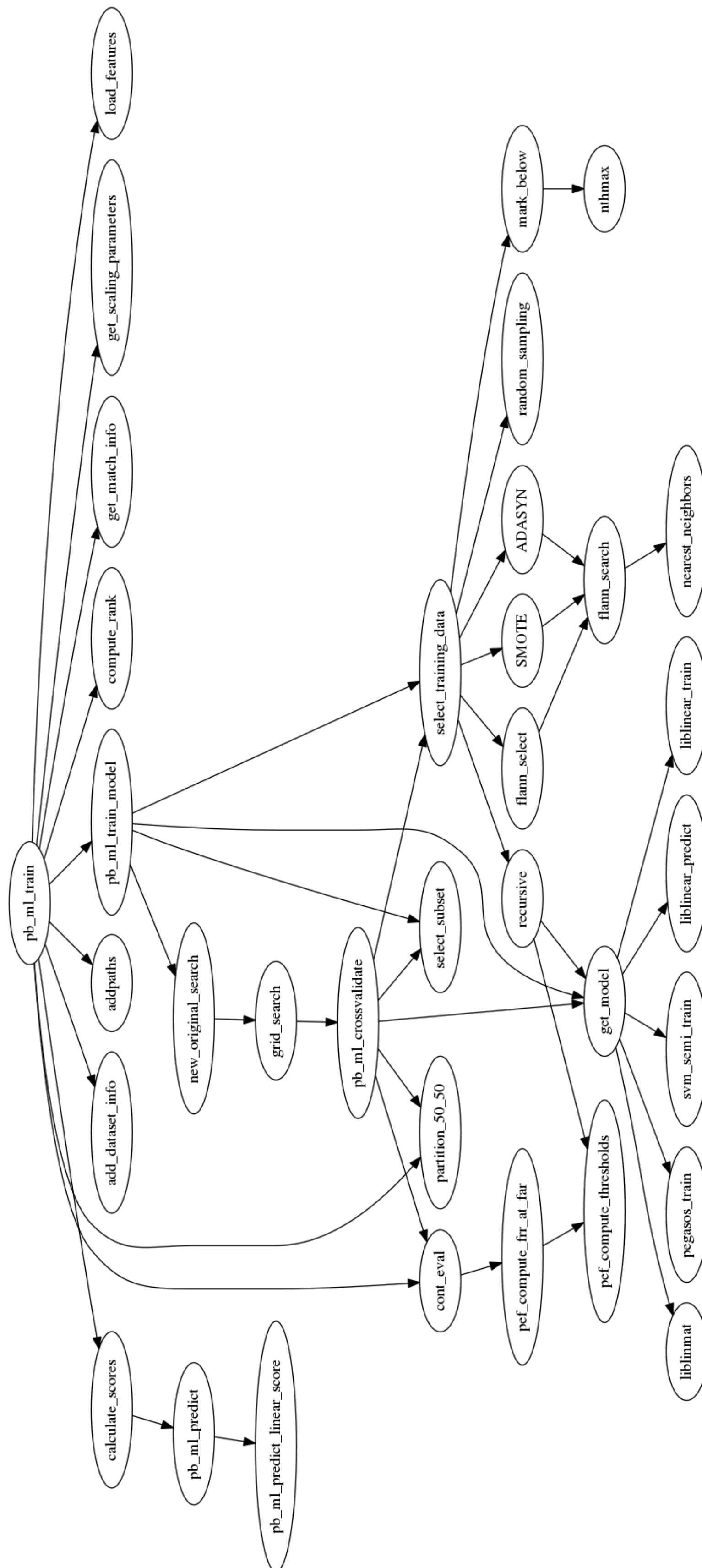

Figure 8.2: Structure of the implementation of A_N. The arrows show function dependencies.

Bibliography

[1] Maltoni, Davide, et al. Handbook of fingerprint recognition. Springer Science & Business Media, 2009.

[2] Shalev-Shwartz, Shai, and Shai Ben-David. Understanding machine learning: From theory to algorithms. Cambridge university press, 2014.

[3] Fan, Rong-En, et al. "LIBLINEAR: A library for large linear classification." Journal of machine learning research 9.Aug (2008): 1871-1874.

[4] Chawla, Nitesh V., et al. "SMOTE: synthetic minority over-sampling technique." Journal of artificial intelligence research 16 (2002): 321-357.

[5] He, Haibo, et al. "ADASYN: Adaptive synthetic sampling approach for imbalanced learning." Neural Networks, 2008. IJCNN 2008.(IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on. IEEE, 2008.

[6] Brodley, Carla E., and Mark A. Friedl. "Identifying mislabeled training data." Journal of artificial intelligence research 11 (1999): 131-167.

[7] Hsu, Chih-Wei, Chih-Chung Chang, and Chih-Jen Lin. "A practical guide to support vector classification." (2003): 1-16.

[8] Biggio, Battista, Blaine Nelson, and Pavel Laskov. "Support Vector Machines Under Adversarial Label Noise." ACML 20 (2011): 97-112.

[9] Muja, Marius, and David G. Lowe. "Fast approximate nearest neighbors with automatic algorithm configuration." VISAPP (1) 2.331-340 (2009): 2.

[10] Bennett, Kristin, and Ayhan Demiriz. "Semi-supervised support vector machines." NIPS. Vol. 11. 1998.

[11] Refaeilzadeh, Payam, Lei Tang, and Huan Liu. "Cross-validation." Encyclopedia of database systems. Springer US, 2009. 532-538.

[12] Sindhwani, Vikas, and S. Sathiya Keerthi. "Large scale semi-supervised linear SVMs." Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval. ACM, 2006.