Abstract
This project investigates the development of a BCI system using a consumer grade EEG headset. This includes signal acquisition, preprocessing, feature extraction and classification and/or regression. Riemannian geometry is taken advantage of, because of the natural EEG signals can be directly classified in this space. The Riemannian methods investigated includes Minimum Distance to Riemannian Mean (MDRM) and Tangent Space LDA (TSLDA). These methods are tested and compared against the well known methods Common Spatial Pattern (CSP), combined with Linear Discriminant Analysis (LDA), which was investigated in our previous work. Furthermore it is investigated how it is possible to combine two predictor tasks, instead of one, e.g. classification. This is done by combining classification and regression simultaneously, which opens up new ways of how a BCI system can be used. This report documents the development of a combined two-predictor-task BCI system, and concludes the found results of said methods.

Developing a BCI system for multiple predictor tasks using a consumer grade EEG reader

Anders Jensen, Nicolai L. H. Kirk, Daniel Bøcker Sørensen

11/6 2015

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg Ø
Telefon 99 40 99 40
Fax 99 40 97 98
http://www.cs.aau.dk/

Title:

Developing a BCI system for multiple predictor tasks using a consumer grade EEG reader

Theme:

BCI research and development

Project period:
10th Semester

Project group:
mi101f15

Group members:
Anders Jensen
Nicolai L. H. Kirk
Daniel Bøcker Sørensen

Supervisor:
Bo Thiesson

Synopsis:

This report documents the process of researching and developing a BCI system. Advanced methods for EEG analysis is used to predict both the class and a continuous value of a signal. This process includes researching the theory of Riemannian geometry, to implement classification and regression methods applicable to the Riemannian manifold. To test the implemented methods, the Emotiv EPOC EEG Reader was used to record brain signals.

Circulation: 5

Number of pages: 60

Number of Appendices: 1

Finished: 11.6.15

Preface

This project is written on Aalborg university in the computer science department. This report concludes our master thesis project and documents the results and the means to reach them.
Parts of this report is based on our previous work that also involved brain computer interface systems.
The way literature is referred is in the following way: [?] where '?' is a number that can be found in the bibliography at the end of the report.
Finally we would like to thank our supervisor Bo Thiesson for his cooperation and help throughout this semester.

Anders Jensen *Daniel B. Sørensen*

Nicolai L. H. Kirk

CONTENTS

CHAPTER 1

INTRODUCTION

Traditionally, computer systems are controlled by using a mouse and keyboard to navigate and perform tasks. Today, a new way of controlling computer systems is evolving, namely brain computer interfaces (BCI). A brain computer interface is a system that use brain activity as input, instead of physically moving a mouse or pressing buttons on a keyboard. This means that no movement is required to operate a BCI system, and therefore removes the requirements that are normally associated with using computer systems. By removing the physical constraints, new opportunities arises for people with movement impairments, but also affects how people use computer systems in general.

The way a BCI system works, is by analyzing brain activity signals. These signals form different patterns based on what the user is doing and thinking. By analyzing the patterns of a user, a BCI system can learn the characteristics and be able to recognize known patterns. When a pattern is recognized it can be translated into a command, just like the input from a mouse or keyboard is translated into commands. In this way, if a BCI system is trained on a user, and that user starts imagining movement, e.g. moving the left hand and the system recognizes this pattern, then the command associated with left hand movement is carried out in the system, e.g. moving the mouse cursor left.

BCI research is a young field in computer science, which means there is still information to be discovered. Because of that, it is hard to develop a BCI system , that is reliable enough to be useful in real world scenarios. There are several factors that affect the performance of BCI systems, ranging from the user's ability to concentrate, to the techniques used for extracting relevant data from the brain signals. Obviously, the hardware used for recording the brain activity data is also an important factor. There are several methods to record brain activity data, among

them are electroencephalography and electrocorticography. Electroencephalography records data (EEG), with electrodes placed on the scalp. Electrocorticography records data (ECoG) with electrodes placed directly on the brain. To record ECoG data, a surgical incision into the skull is required to place the electrodes. For BCI systems EEG is widely used, because no surgery is required to use such systems. Both electroencephalography and electrocorticography are methods that performs well, when analyzing data in a temporal domain. On the flip side they are not the most accurate methods spatial-wise. There exist methods, such as Magnetic resonance imaging (MRI), which is often used to diagnose or analyze e.g. the brain. A MRI scanner takes a high spatial resolution picture of e.g. the brain, but only for a point in time. This means that the MRI scanner provides highly accurate spatial data, but only for a single time stamp. Therefore BCI systems often use either EEG or ECoG data, because of the high temporal resolution.
In this project a consumer grade EEG headset is used, because of the high temporal resolution. It is an easy convenient way of recording EEG data, as unlike ECoG data, no surgery is required to record data.
When working with BCI systems, they traditionally have one predictor task, either classification or regression. The difference between these predictor tasks, is the result from the analysis. If the predictor task is a classification task, the system's result predicts which class the input data belongs to. On the other hand, if the predictor task is regression, the result is a continuous response from the task such as the strength of a present signal, e.g. how fast is the left arm moved. Combining these, resulting in two predictor tasks allows for more advanced BCI systems, because the system will be able to recognize which class some brain data belongs to, but also output the strength of the signal. Combining the two predictor tasks in a BCI system is to our knowledge something that has never been done, or researched before.
This report investigates the development of BCI systems and advanced methods for feature extraction, classification and regression. The rest of this chapter introduces basic information about BCI systems. Chapter 2 introduces the problem statement and therefore also the goal of this project. Chapter 3 explains the theory behind important methods used in the development of a BCI system. Chapter 4 shows the implementation of the most interesting parts of the system. Chapter 5 tests the system and shows the results and reasoning behind them. Chapter 6 concludes the project based on the problem statement and the given results. Finally Chapter 7 discuss possibly improvements future work.

1.1 Brain Computer Interface Architecture

The execution of a BCI system can be divided into two phases, training time and runtime. As the brain is different from subject to subject, the system needs to be

trained to recognize the patterns of the brain for each subject. This is called the training time or calibration time, and is where the systems receives data, in which the corresponding result of the prediction task is known. With this labeled data, the system can extract the necessary information, to train the feature extraction and classification or regression model, which should be able to predict the result from new unlabeled data. The training time is illustrated in Figure 1.1 with its 3 stages. When a set of labeled data is received as input, the first stage involves applying preprocessing. The next stage trains the feature extraction to find the useful information. Lastly the classification/regression stage is trained with the extracted information. The methods can then learn the characteristics of the signal. After the training is done, the system can be used on runtime, tailored to a specific subject, or general use, depending on whether or not the system is trained on a single subject or a set of subjects.

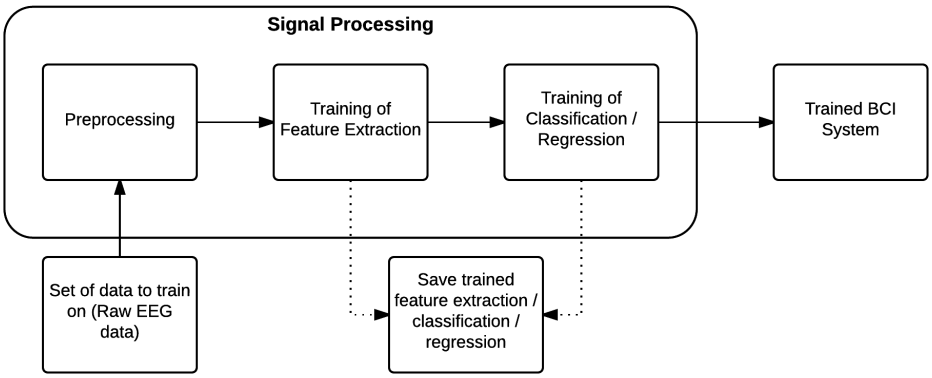

Figure 1.1: Classification/Regression BCI Architecture training time

On runtime when data is to be analyzed in a BCI system, it goes through five stages. The stages are Signal acquisition, Preprocessing, Feature Extraction, Classification/Regression and Application Interface, as seen in Figure 1.2. at runtime, both Feature Extraction and Classification/Regression have been trained to a user, so the signals can be recognized.

Preprocessing, Feature Extraction and Classification/Regression are commonly grouped together as one part called Signal Processing, as this is where the primary data analysis is executed. Below, each stage is briefly explained, for a more detailed explanation see [16].

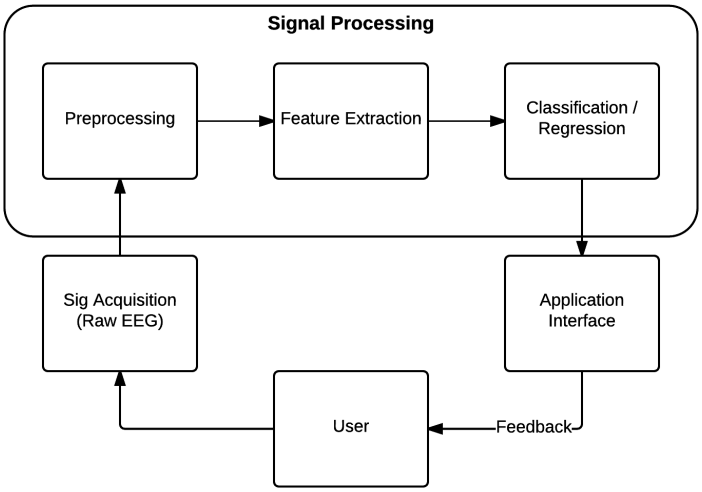

Figure 1.2: Classification/Regression BCI Architecture runtime

Signal Acquisition

Signal acquisition regards to how the data is read from the hardware, and formatted so it can be analyzed. At runtime, the system reads the data as a series of timestamp with each channel represented as the voltage difference from the base value. These are then analyzed and predicted by either classification or regression. When training these predictors, additional information is required. As such the signal acquisition keeps track of which additional value corresponds to the timestamps. For classification, the additional information is regarding which class is active at each timestamp, or when a class is activated or deactivated. For regression, the additional information is not regarding a class, as that is assumed to be the same for all timestamps. Instead a magnitude for the class is known for each timestamp. The hardware used for signal acquisition is often a headset with electrodes attached, which are placed on the scalp (EEG) or directly on the brain (ECoG).

Preprocessing

Preprocessing is the process of removing noise and irrelevant information from the signal, thus making the signal easier to analyze. The process involves smoothing the data by applying temporal filters, smoothing the channels by spatial filters and subsampling the data to remove specific timestamps. It can also involve selecting channels for processing, to focus the analysis to a specific area of the brain.

Feature Extraction

Feature extraction is, as the name implies, the process of selecting features of the signal. Commonly this means weighing or combining the channels, in ways that best reveal variations. The goal of the feature extraction stage is to select features, that are characteristic for the target classes.

Classification/Regression

The stage of classification/regression is to predict a class or a continuous value, based on brain signals. When using a classification method, e.g. Linear Discriminant Analysis (LDA), the system is trained using labeled data in order to predict new unlabeled data during runtime. When using regression, a model is trained, so it fits the labeled data. During runtime, this model can predict a continuous value, e.g. the strength of the signal, based on new data.

Application Interface

The last part, the application interface, is where the result of the classification is used. This is usually how the systems reacts to the signal, and how it processes feedback. The process of feedback can be separated into two types of systems, passive and active.
In a passive system, the user is not actively trying to control the application using the BCI system, instead the BCI system is used to enrich the user experience. An example of this could be an application, where the feedback is the selection of music that plays in the background based on the users emotional state.
In an active system, the user actively tries to control the application using the BCI system. An example could be an application, where the control of the mouse cursor is based on the imagined hand movement of the user.

1.2 Hardware Description

The hardware used in this project to read EEG data is the Emotiv EPOC EEG Reader [12]. The headset is equipped with 14 electrodes placed according to the international 10-20 system [1], which will be explained further in Section 3.1. The placement of the 14 electrodes of the headset can be seen in Figure 1.3, where the electrodes of the headset are marked with an orange outline.

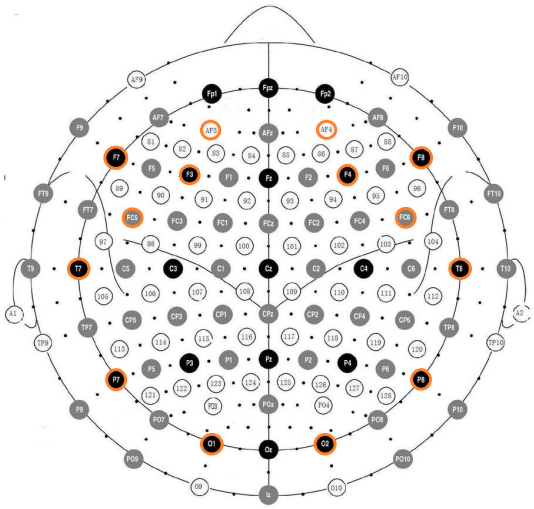

Figure 1.3: Emotiv EPOC EEG Reader electrode placement

The headset has a sampling rate of 128Hz for each electrode, and a bandwidth of 0.2 - 45Hz. To remove AC noise from the signal, the headset also comes with notch filters integrated at 50 and 60Hz, as they are the standards used depending on the user's geographical location, e.g. in Denmark the AC outputs 230V at 50Hz. For a list of the headsets full specifications see Appendix A.

CHAPTER 2

PROBLEM STATEMENT

Traditionally when developing Brain-Computer Interfaces (BCIs) one predictor task is used, either classification or regression. The difference between classification and regression is their output format. Classification predicts data to belong to a specific class among a set of two or more. Regression on the other hand is a continuous output.
It is clear that these two different predictor task are used for different purposes. A classification predictor task can be used in a scenario to determine whether left or right hand is moved. A regression predictor task can be used in a scenario where we are interested in the speed of hand movements instead.
Combining the aforementioned predictor tasks to open up for new ways to use them, could yield interesting opportunities. For instance, classifying whether the left or right hand is moved, followed by using regression to determine the speed of the hand movement.
To record brain activity we have a consumer grade Emotiv EPOC EEG Reader available, which means the recordings can be questionable. Due to low spatial resolution and low sampling rate in the recording hardware, effective software methods and implementations are important to get a usable result. In later research, methods employing Riemannian geometry has been shown to improve the prediction results. Based on the above observations it leads us to the following problem statement

Is it possible, using a consumer grade EEG reader, to combine two predictor tasks in order to simultaneously determine both class and magnitude of a signal?

CHAPTER 3

THEORY

This chapter describes the theory of the different algorithms and methods used in this project. Some of the algorithms have been explained in our previous work [16], and will therefore only be shortly introduced and then referenced.

3.1 Brain Basics

This section describes basic information about the brain, including brain areas, frequency ranges, and the international 10-20 system.

Brain Areas

To narrow the analysis of the signals, selecting specific channels to be analyzed can be useful, as this will narrow the brain areas analyzed and thus filter the noisy channels. To accomplish this task, knowledge of which areas of the brain represent which functions is useful. The cerebrum, which is the large part of the brain can be divided up into four sections, as shown in Figure 3.1.
The sections known as brain lobes are named:

- The frontal lobe (Blue area)
- The parietal lobe (Yellow area)
- The temporal lobe (Green area)
- The occipital Lobe (Red area)

Each of these lobes are associated with different functions, each of which will be given a brief description below [19].

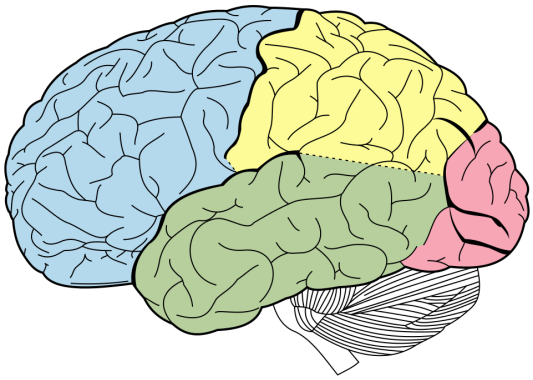

Figure 3.1: The cerebrum lobes

The frontal lobe

The frontal lobe in essence holds what makes up each individuals personality. The frontal lobe controls emotions, problem solving, memory, language, judgment as well as social and sexual behavior. It is also in the frontal lobe where the primary motor is located, as well as some of the key areas related to speech are stored.

The parietal lobe

The parietal lobe located just behind the frontal lobe near the center of the brain. This is the area in which sensory information gets interpreted such as hot and cold. The parietal lobe is also where spatial information gets interpreted thus allowing one to estimate distance, size etc.

The temporal lobe

The temporal lobe is responsible for interpreting auditory information as well as the interpretation of information gathered through smell. The Wernicke area that is part of the temporal lobe allows us to recognize speech and understand the meaning of the words.

The occipital Lobe

The occipital Lobe located at the back of the skull holds the visual cortex, this cortex receives information from the retina of the eye, it is here that the interpretation of color and important parts of the vision is handled.

Frequency Ranges

As with brain areas being affected by different functions, the frequencies of the signal is affected as well, and a reduction of the analyzed frequencies can thus narrow the analysis. According to [5], the frequencies extracted from a signal, can be grouped in five types of waves; Delta, Theta, Alpha, Beta and Gamma. These waves are comprised of different frequencies and different states of consciousness can make a wave more pronounced than the others.

- **Delta** waves are in the 0.5-4Hz range. They are primarily associated with deep sleep, and can be confused with muscle artifacts from neck and jaw.
- **Theta** waves lie in the range 4-7Hz. Theta waves increases with stress or frustration and have been seen with deep meditation as well.
- **Alpha** waves have frequencies in the 8-13Hz range and are associated with a mindless state. The waves are prominent with eyes closed.
- **Beta** waves are loosely defined as being comprised of frequencies between 13-30Hz. Beta waves are usually associated with active thinking.
- **Gamma** waves lie in the range 35Hz and up. The gamma waves are the most recent wave type to be discovered, and the functions associated with them are mostly unknown.

In addition there is **Mu** waves as well. These waves are in the same frequency band as alpha waves, namely 8-12Hz, but are recorded over the occipital cortex. They diminish with movement or the intention to move.

International 10-20 System

The international 10-20 system is a common method to describe positions of electrodes on the scalp during the recording of EEG data. The name 10-20 refers to the distance between the electrodes, by separating the electrodes with a distance of either 10 or 20 percent of the total distance from the back to the front of the skull, or from the left to the right side of the skull. Depending on the number of electrodes that are available, a fitting percentage is chosen. The reason why the international 10-20 system is used, is to ensure a standardized way to place electrodes on the scalp. Figure 3.2 shows an example of how the electrodes are placed according to the international 10-20 system. As can be seen, the electrode placements are labeled with a letter followed by a number. The letter represents which lobe it is placed on, where the lobes are frontal (F), temporal (T), central (C), parietal (P), and occipital (O) as described in Section 3.1. It should be noted that the central lobe does not

exist, the term is used purely for the identification of electrodes in the international 10-20 system. The numbers represents the location on the hemisphere, where even numbers are electrodes on the right hemisphere and odd number on the left. The numbers starts from the center and increment when moving away from the center of the skull.

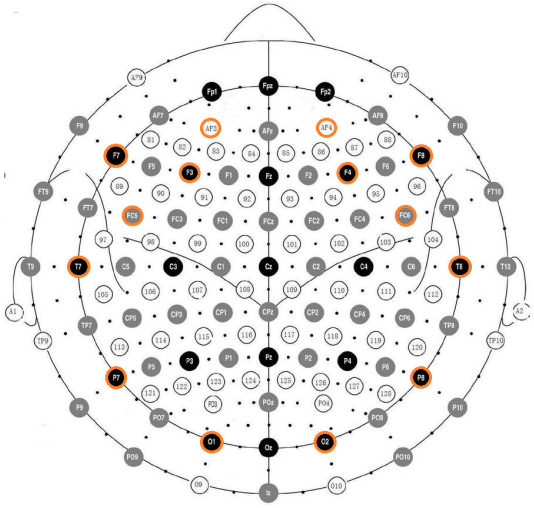

Figure 3.2: Example of electrodes placed according to the international 10-20 system.

3.2 Notation

This section outline the notation used throughout this report. EEG data is represented as a matrix X as seen in Equation 3.1, where $x_c(t)$ represents the EEG data for channel c at sample point t. X_c represents one row in X for channel $c \in 1, \ldots, C$, for all time points i.e. $X_c = (x_c(1), \ldots, x_c(T))$, where C is the total number of channels and T is the last time point in X. $X(t)$ refers to the elements across all electrodes, at a given time point $t \in 1, \ldots, T$, i.e., $X(t)$ refers to the vector elements in the t'th column of X, $X(t) = (x_1(t), x_2(t), \ldots, x_C(t))^\top$.

$$X = \begin{pmatrix} x_1(1) & x_1(2) & \cdots & x_1(T) \\ x_2(1) & x_2(2) & \cdots & x_2(T) \\ \vdots & \vdots & \ddots & \vdots \\ x_C(1) & x_C(2) & \cdots & x_C(T) \end{pmatrix} \tag{3.1}$$

Covariance Matrix

A covariance matrix (see e.g. [10]), often also called a variance-covariance matrix, is used for two different measures. The variance is the measure of how much a distribution varies from its mean, while the covariance is the measure of similarities between two variables.
Given the observations across time for two electrodes X_1 and X_2 the sample covariance is defined as:

$$Cov(X_1, X_2) = \frac{(X_1 - \mu_1)(X_2 - \mu_2)^\top}{T} \tag{3.2}$$

The variable μ_1 in Equation 3.2 refers to the mean for X_1 while μ_2 is the mean for X_2. The formula for calculating the mean can be seen in Equation 3.3.

$$\mu_c = \frac{\sum_{t=1}^{T} x_c(t)}{T} \tag{3.3}$$

In the case where $X_1 = X_2$ the covariance reduces to the variance shown in Equation 3.4.

$$Cov(X_1, X_1) = \frac{(X_1 - \mu_1)(X_1 - \mu_1)^\top}{T} \tag{3.4}$$

A covariance matrix provides the covariance between all variable pairs. Equation 3.5 shows the covariance matrix Σ, where the i, jth entry in the matrix is the covariance between X_i and X_j. In the cases where $i = j$, meaning the diagonal of the matrix, it is the variance of variable X_i.

$$\Sigma = \frac{(X - \mu)(X - \mu)^\top}{T} = \begin{pmatrix} \sigma_{1,1} & \sigma_{1,2} & \cdots & \sigma_{1,C} \\ \sigma_{2,1} & \sigma_{2,2} & \cdots & \sigma_{2,C} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{C,1} & \sigma_{C,2} & \cdots & \sigma_{C,C} \end{pmatrix} \tag{3.5}$$

Equation 3.5 can be simplified into the expression in Equation 3.6 if the matrix X is a mean corrected matrix.

$$\Sigma = \frac{(X)(X)^\top}{T} = \begin{pmatrix} \sigma_{1,1} & \sigma_{1,2} & \cdots & \sigma_{1,C} \\ \sigma_{2,1} & \sigma_{2,2} & \cdots & \sigma_{2,C} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{C,1} & \sigma_{C,2} & \cdots & \sigma_{C,C} \end{pmatrix} \tag{3.6}$$

Matrix definitions

This section describes some of the notation used in Riemannian Geometry [4]. The space of all real square matrices is defined as $M(n)$, where n denotes the dimension of a given matrix e.g. $M(4)$ denotes the space of real square matrices of dimension 4×4. The space of all symmetric matrices in $M(n)$ is defined as in Equation 3.7.

$$S(n) = \{S \in M(n), S^\top = S\} \tag{3.7}$$

The space of all symmetric positive-definite matrices in $S(n)$ also known as SPD matrices is given by $P(n)$, formally defined as seen in Equation 3.8.

$$P(n) = \{P \in S(n), u^T P u > 0, \forall u \in \mathbb{R}^n\} \tag{3.8}$$

A matrix is said to be positive-definite if it is diagonalizable with all real positive eigenvalues [2][4]. A matrix A is diagonalizable if there exists a matrix U of eigenvectors and a diagonal matrix Λ of eigenvalues so that Equation 3.9 is true.

$$A = U \Lambda U^{-1} \tag{3.9}$$

Finally the set of all invertible matrices in $M(n)$ is defined as $Gl(n)$.

Upper function

The *upper*() operation is used to convert an $n \times n$ symmetrical matrix into an m-dimensional vector, where $m = \frac{n(n+1)}{2}$ [23][4]. The *upper*() operation keeps the upper triangular part of a symmetrical matrix by vectorizing it. The vectorization is done by applying a unity weight to the diagonal part and a weight of $\sqrt{2}$ to the off diagonal part of the symmetrical matrix, as the values in the off diagonal appears twice in a symmetrical matrix.
Given the symmetrical matrix A seen in Equation 3.10 the the *upper*() operation gives the vector seen in Equation 3.11.

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} \tag{3.10}$$

$$upper(A) = [a_{1,1}, \sqrt{2}a_{1,2}, \cdots, \sqrt{2}a_{1,n}, a_{2,2}, \cdots, \sqrt{2}a_{2,n}, \cdots, a_{n,n}]^\top \tag{3.11}$$

Norms

For this project, two different norms will be used, the L2 norm $||a||_2$ and the Frobenius norm $||A||_F$, where a denotes a vector and A denotes a matrix

L2 Norm

The L2 norm is a vector norm defined as $||a||_2 = \sqrt{\sum_{i=1}^{n} |a_i|^2}$
The result of the L2 norm yields the length of the vector a.

Frobenius Norm

The Frobenius norm is a matrix norm defined as seen in Equation 3.12, where A is a matrix.

$$||A||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{i,j}|^2} \tag{3.12}$$

An example of the Frobenius norm calculated on the matrix A from Equation 3.10.

$$||A||_F = \sqrt{|a_{1,1}|^2 + |a_{1,2}|^2 + \cdots + |a_{1,n}|^2 + \cdots + |a_{n,1}|^2 + \cdots + |a_{n,n}|^2} \tag{3.13}$$

Exponential and Logarithmic matrix

For an SPD matrix the exponential matrix is calculated using the eigenvalue decomposition of the matrix as seen in Equation 3.14.

$$\exp(P) = U\text{diag}(\exp(\lambda_1), \cdots, \exp(\lambda_n))U^\top \tag{3.14}$$

While its inverse the logarithm of an SPD matrix is given by Equation 3.15.

$$\log(P) = U\text{diag}(\log(\lambda_1), \cdots, \log(\lambda_n))U^\top \tag{3.15}$$

Where U is the eigenvectors and $\lambda_1, \cdots, \lambda_n$ the eigenvalues of P. The operation 'diag' reinserts the eigenvalues $\lambda_1, \cdots, \lambda_n$ into a new diagonal matrix where the off diagonal contains 0's.

3.3 Preprocessing

Because the raw signal can contain noise, which can corrupt the data and influence the analysis negatively, steps will have to be taken to preprocess the data to reduce the noise. Preprocessing consist of several steps including temporal filters, frequency filters, and spatial filters.

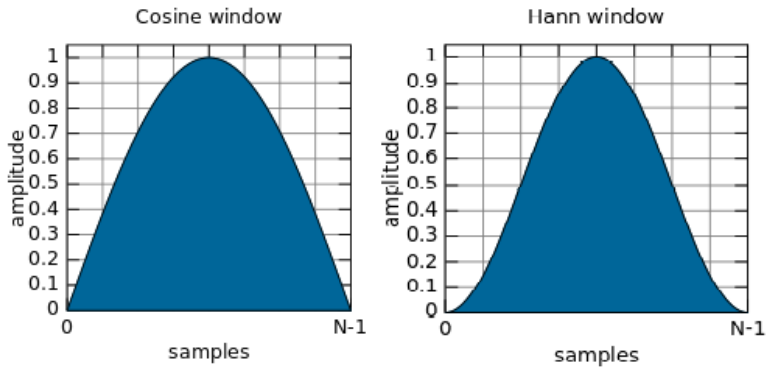

Figure 3.3: Window functions: Cosine & Hanning

Temporal Filters

When using temporal filters, the analysis is performed over a sequence of timestamps. The goal of using temporal filters is to focus on specific parts of a signal. To this end, a window function is applied, which emphasizes the signal variations at a specific point and deemphasizes the rest. The temporal filter smooths the signal using this window function as a weighted average at each timestamp, so that for each timestamp, the adjacent timestamps affects the magnitude. Depending on the goal of the filtering, different window functions can be uses, for example, as seen in Figure 3.3, either a Cosine window or a Hann window can be used. Many others exist with a couple described in [16], and are all interchangeable.

Fourier Transform

The brain emits several different wave lengths each associated with different mechanics or mental activities [24]. It can be useful to discard or deemphasize some of these wavelengths depending on what brain functions are being analyzed. The brain waves can be divided into frequency bands as described in Section 3.1. Using a Fourier transform, the frequencies can be extrapolated from the signal, in other words the Fourier transform can take the signal from the temporal domain, `P(t)` in Figure 3.4, and convert it to the frequency domain, `P(v)` in Figure 3.4.

When the operations, which should be applied on the frequency domain, have been applied, the data needs to be transformed back. This is called an inverse Fourier transform, which takes the extrapolated frequencies and combines them into the now filtered signal, i.e. from `P(v)` to `P(t)` in Figure 3.4.

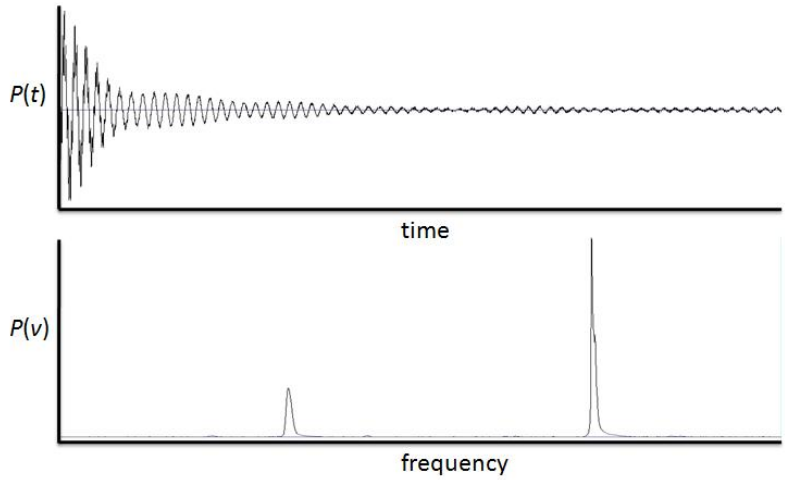

Figure 3.4: Example of frequency range, taken from [18]

Subsampling

Subsampling can be used to counter act the reaction time of the subject, as when a test is performed, the subject is shown an indication of what to do, but as the subject first have to interpret the indication and then execute the task, there can be some delay, which can be removed by using sumsampling to remove the delay.

Frequency Filters

After having converted the signals to the frequency domain, specific frequencies can now be discarded with the help of frequency filters. There exists several different types of frequency filters, though only High-Pass and Low-Pass filters will be used in this project. These, as the name implies, lets either high or low frequencies through the filter, and the rest are deemphasized, as seen in Figure 3.5. We will use the Butterworth implementation, as explained in [21]. According to the hardware specifications for the Emotiv EPOC EEG Reader as explained in Section 1.2, the bandwidth for the reported signal is between 0.2Hz and 45Hz. Furthermore as explained in Section 3.1, certain ranges within the bandwidth are more important than others, when specific brain functions are active. The Low-Pass filter is applied with 30Hz as the restriction, and the High-Pass with 8Hz as the restriction. The brainwaves covered by the frequencies within the range of 8 and 30Hz are the Alpha, Beta and Mu waves, which as described in Section 3.1 are the active and mindless states of brain activity.
The signal seen in Figure 3.6 have a lot of small frequencies. When frequency filters are applied with the above settings, the resulting signal can be seen in Figure 3.7,

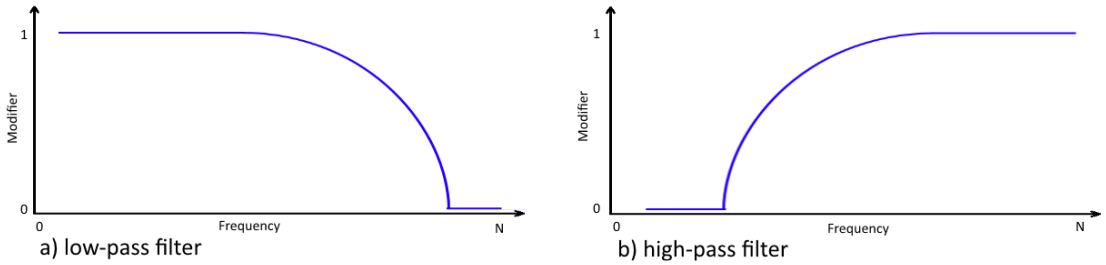

Figure 3.5: Illustration of High-Pass filter & Low-Pass filter

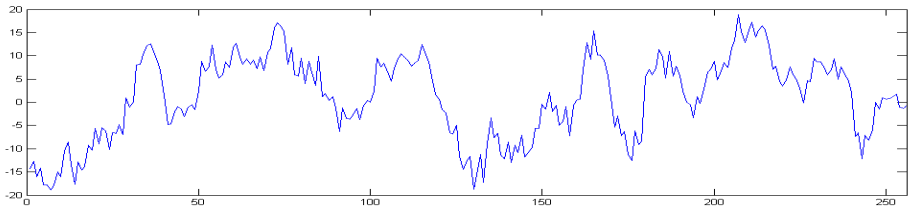

Figure 3.6: Illustration of signal before frequency filters

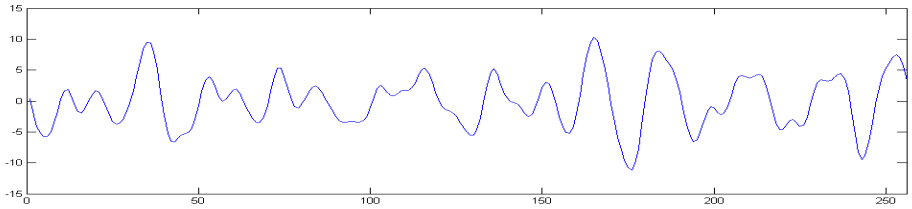

Figure 3.7: Illustration of signal after frequency filters

with the signal significantly smoothed.

3.4 Feature Extraction

Feature extraction is the process of reducing the number of channels to improve the analysis using the information from Section 3.1. As such the process of feature extraction includes selecting the channels with the highest variance and transforming the data to feature vectors for the classification. The process of selecting channels is not done explicitly though, but implicitly by feature extraction methods, which will be explained in this section.

Common Spatial Pattern

Common Spatial Pattern (CSP) [15][8], is a spatial filter method and one of the most used methods in BCI systems. CSP is used to improve the distinction between two classes, such that the classification can better distinguish between these two in a new signal.
For the increase in distinction, CSP needs to be trained, which is done using two sets of trials, representing the two classes. Using features created from the data, the training creates a set of weights, which can be sorted by how well each of the features distinguish between the two classes. The training can be done in Matlab using the `eig` command to solve the eigenvalue problem in Equation 3.16, which implements simultaneous diagonalization of the covariance matrices Σ_1 and Σ_2. The simultaneous diagonalization is visualized in Figure 3.8 and Figure 3.9, where the red points being class 1 and the blue class 2. The first image in Figure 3.8 displays the variance of the two classes before the `eig` command and Figure 3.9 displays the points after the `eig` command where the simultaneous diagonalization have been performed, which aligns and rotates the coordinate system such that large variances are axis-aligned.

$$[VD] = \text{eig}(\Sigma_1, \Sigma_1 + \Sigma_2) \tag{3.16}$$

The resulting matrices V and D are the eigenvectors and corresponding eigenvalues respectively. The eigenvectors in V chosen to construct the weight matrix, are those with the highest and lowest variance found in matrix D. Normally one choose $2m$ eigenvectors for the weight matrix, where m is the number of eigenvectors chosen from each end, with m typically being between 1 and 3.

$$Y = WX \tag{3.17}$$

When this weight is used on a new set of data using Equation 3.17, where W is the weight and X the signal data, the variance for one class will be maximized, while minimizing it for the other. CSP can then select the features or channels that best distinguish the classes, and extract them.

3.5 Euclidean Classification

Classification is the task of classifying observations to specific classes e.g. is the signal a left or right hand movement signal. In our previous work [16] two linear classification methods were studied, namely linear discriminant analysis (LDA) and support vector machines (SVM). The performance of the two classifiers yielded results that were similar, and therefore in this project we will limit our attention to LDA. This section gives a brief introduction to LDA and the theory.

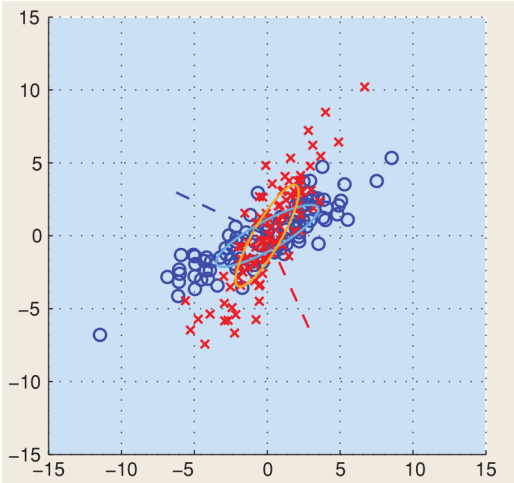

Figure 3.8: Before CSP [8]

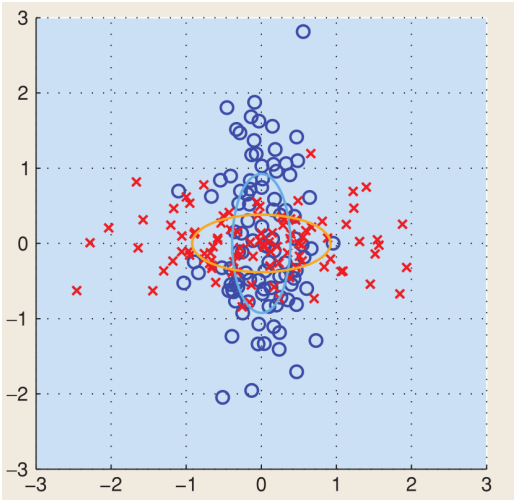

Figure 3.9: After CSP [8]

Linear Discriminant Analysis

Linear discriminant analysis [11] is, as the name suggests, a linear classifier. A linear classifier is a classifier, which divide the data, by separating it with a linear hyper-plane. Figure 3.10 illustrates an example of a binary classification problem. The blue and green dots represent the training data for each class, and the black bar the hyperplane seperating the data.
The process of evaluating data according to a hyper-plane, is done using a number of equations, one for each class. These equations evaluates how a new data point's location compares to the trained classes. An equation will result in a higher value, if the point is closer to the center of the trained classes points. Equation 3.18 is the function used, with μ_i as the mean for class i, $\Sigma\Sigma^{-1}$ as the inverse pooled covariance matrix, p_i as the influence of the class and Y as the data to be classified.
The class function which evaluates to the greater value, is the predicted class for the new data point.

$$f(i) = \mu_i \times \Sigma\Sigma^{-1} \times Y^\top - 0.5 \times \mu_i \times \Sigma\Sigma^{-1} \times \mu_i^\top + \ln(p_i) \tag{3.18}$$

3.6 Euclidean Regression

In our previous work [16], classification theory have been studied and used to develop a BCI system for simple binary classification tasks. In this section we will extend the study further, by looking at another way to output results from a BCI system, namely regression.

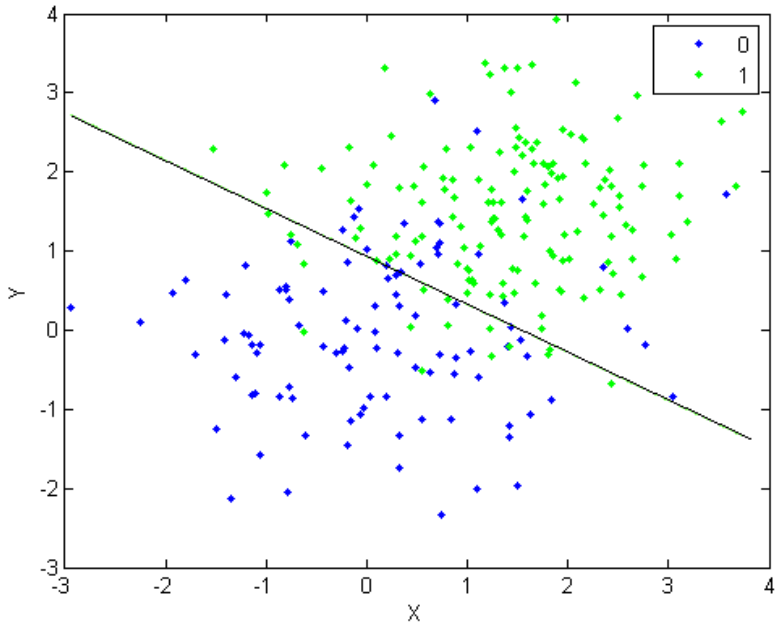

Figure 3.10: Example of binary separation

The difference between classification and regression is the predicted result. A classification method returns a prediction of which class the data belongs to. Regression, on the other hand, is used to predict a continuous value such as the magnitude of some data, with the class already being known, e.g. how fast the right hand is moving.
It is clear that these two different ways of evaluating a signal are used for two different scenarios. Given the problem statement in Chapter 2, we are interested in knowing how fast a user moves his arm, which means regression is a relevant topic to investigate.
Just as classification, regression is used to estimate the relationship between variables, to predict data.

Linear Regression

In linear regression the goal is to find a linear equation of the form seen in Equation 3.19 which models the relationship between the dependent variable and a vector of explanatory variables.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_P x_P \tag{3.19}$$

In Equation 3.19 the dependent variable is y, which is the output and the p explana-

tory variables are $x_1, x_2, \ldots, x_P$, which is the input. The remaining variables β_0 is the intercept and $\beta_1, \beta_2, \ldots, \beta_P$ are the regression coefficients.

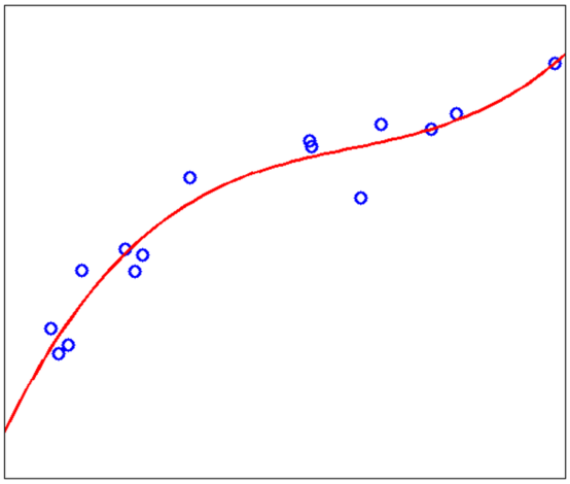

Figure 3.11: Linear Regression

An example of a linear regression model can be seen in Figure 3.11. The blue dots in the figure indicate the actual points of the data and the red line is the linear equation found to estimate the data.
A common method for fitting a regression line is least-squares.

Least-Squares regression

Least-squares is a method for estimating a solution. The idea is to minimize the sum of squared errors in the results of every equation.
The sum of squared errors is the measure of discrepancy there exists between a value estimated by the model and the observed value in the data.
The goal is to minimize the sum of squared error, denoted SSE. This can be achieved by solving the minimization problem in Equation 3.20, where there are P explanatory variables.

$$SSE = \arg \min_{\beta_0,\ldots,\beta_P} \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_{1i} - \cdots - \beta_P x_{Pi})^2 \tag{3.20}$$

Basis Function

One thing to note is that the linear regression does not have to be linear in the explanatory variables, but instead must be it in the parameters β defined as ($\beta =$

$(\beta_0, ..., \beta_P)$). This means that a basis function $\phi()$ can be employed on the explanatory variables in order to constrain complex and powerful regression models which are still linear. This gives the linear equation in Equation 3.21, where M denotes the number of parameters.

$$y = \beta_0 + \sum_{m=1}^{M-1} \beta_m \phi_m(x) \tag{3.21}$$

The basis function $\phi_m()$, where m denotes the index of the parameter, where x is being transformed, e.g. given the 3 parameters$(\beta_1, \beta_2, \beta_3)$ the following basis functions can be used.

- $\phi_1(x) = x$
- $\phi_2(x) = x^2$
- $\phi_3(x) = x^3$

Using these feature representations gives the linear equation in Equation 3.22.

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 \beta_3 x^3 \tag{3.22}$$

For convenience it can be practical to define the basis function $\phi_0(x) = 1$ to simplify Equation 3.21 into Equation 3.23.

$$y = \sum_{m=0}^{M-1} \beta_m \phi_m(x) \tag{3.23}$$

Over- and under-fitting

Even though the goal is to find a line fitting the training data, the training data can also be fitted too well causing overfitting of the data. The example in Figure 3.12 shows how the linear regression line goes through all the points of the training example, depicted by the dark blue dots. However, even though the estimated regression line fits the data perfectly, it has large errors on new trials, as depicted by the light blue dots.
The opposite of overfitting is underfitting. Underfitting happens when a model is too simple. Thus the goal of finding a regression line for a set of data, should not be to just have a small sum of squared error on the training data, but also to have a small sum of squared error on the unseen data.
To help ensure a regression line is not overfitted given a complex linear function, cross-validation is used. Cross-validation also ensures that the regression line found using the training data, also yield the best predictive result on new data. If a

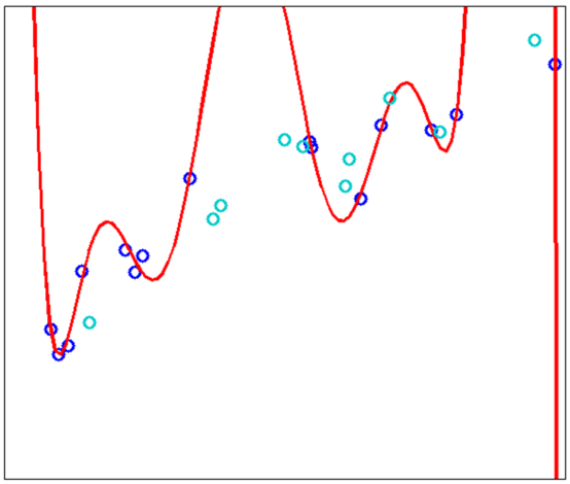

Figure 3.12: Overfitted linear regression

regression line instead is underfitted, it can be beneficial to add more complexity to the model e.g. by using the basis functions described previously.

Regularization

Regularization is another method that can be used to help prevent overfitting. The most commonly used regularization methods are known as L_1 and L_2 regularization. L_1 regularization also called Lasso, is a method that try to sparsify the solution by pushing parameters to 0, which can also effectively be seen as feature selection and is useful in situations where many features are used but only a few are relevant. In L_1 regularization each of the parameters β_p in the calculation of the SSE in Equation 3.20 where $p > 0$ is subject to the constraint in Equation 3.24, where $\sum_{p=1}^{P} |\beta_p|$ is the L_1 norm of the vector $[\beta_1, \cdots, \beta_P]^\top$, where $t \geq 0$ is a tuning parameter. The learning algorithm for L_1 regularization will equal Equation 3.25, where the $\hat{\beta} = (\beta_1, \cdots, \beta_P)$, and $\lambda \geq 0$ [22].

$$\sum_{p=1}^{P} |\beta_p| \leq t \tag{3.24}$$

$$SSE = \arg \min_{\beta_0,\ldots,\beta_p} \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_{1i} - \cdots - \beta_P x_{Pi})^2 + \lambda ||\hat{\beta}||_1 \tag{3.25}$$

L_2 regularization does not sparsify the solution like L_1 regularization and is thus more useful in situations where most features are considered relevant. The definition of L_2 regularization adds a penalty term like L_1 regularization, but does not add

any constraints, meaning none of the features or variables will be pushed to 0. The L_2 regularization method simply adds the L_2 norm as described in Section 3.2 to the learning algorithm in Equation 3.20, giving the new minimization problem in Equation 3.26, where $\hat{\beta} = (\beta_1, \cdots, \beta_P)$, and $\lambda \geq 0$.

$$SSE = \arg\min_{\beta_0,\ldots,\beta_p} \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_{1i} - \cdots - \beta_P x_{Pi})^2 + \lambda ||\hat{\beta}||_2^2 \tag{3.26}$$

3.7 Riemannian Classification

This section describes multi-class classification utilizing Riemannian geometry. In [4] it is explained, how Riemannian geometry can be used in a multi-class BCI system. The paper introduces two different methods to achieve this classification. One is the Minimum Distance to Riemannian Mean (MDRM) method and the other is the Tangent Space LDA (TSLDA), both of which will be described in this section.
The most common way to use a BCI system, is to have the user record EEG data for the calibration phase. The EEG data is then used to construct spatial filters using e.g. CSP. Finally the spatial filters are used to construct feature vectors that are given to a classifier e.g. LDA. The idea of Riemannian classification is to merge the spatial filtering and classification into one step by exploiting the covariance structure of the data.

Riemannian Geometry

In contrast to Euclidean geometry, the Riemannian geometry studies curved spaces. The spatial brain signals are represented as points on the manifold, where each point is a covariance matrix, which also means it is a SPD matrix.
where each covariance matrix is a SPD matrix. The space of SPD matrices is a differentiable Riemannian manifold, which is different but still similar enough to a linear space to be able to do calculus. The derivatives at a matrix on the manifold lies in a vector space, which is the tangent space at that point [4]. For each point on the manifold a corresponding tangent space exists. All the tangent spaces of a manifold have the same dimension, which is equal to the dimension of the manifold. The Riemannian manifold and tangent space are $m = \frac{n(n+1)}{2}$ dimensional, where n represents the number of electrodes, because the covariance matrix is mirrored on the diagonal, which means the duplicated data can be removed.
In Figure 3.13, a manifold $\mathcal{M}$ and point $P's$ tangent space T_P is illustrated. As can be seen, the vector S_i, is a vector corresponding to the distance from point P to S_i, where S_i is the point P_i on $\mathcal{M}$, mapped onto the tangent space. Because the tangent space is a standard Euclidean space, Euclidean distances can be calculated.

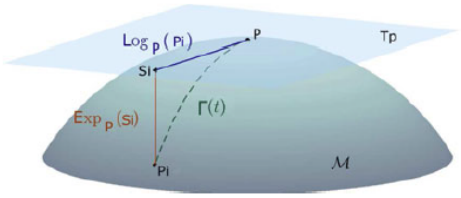

Figure 3.13: Tangent space at point P, S_i a tangent vector at P and $\Gamma_i(t)$ the geodesic between p and P_i [4]

As such a standard classification method can be used as well. To be able to go from the manifold to the tangent space, or vice versa, mapping functions are used, which is explained later.

Riemannian Geodesic Distance

As seen in Figure 3.13, the path between two point on the Riemannian manifold, is not a straight line, and the distance can therefore not be calculated as such. Instead, the path between two points P_1 and P_2 is defined as $\Gamma(t) : [0, 1] \rightarrow P(n)$ where $\Gamma(0) = P_1$ and $\Gamma(1) = P_2$. The length of the path $\Gamma(t)$ is then defined as Equation 3.27.

$$L(\Gamma(t)) = \int_0^1 ||\dot{\Gamma}(t)||_{\Gamma(t)} \mathrm{d}t. \tag{3.27}$$

The Riemannian geodesic distance between two points, is defined as the minimum length curve that connects the two points on the manifold, with the simplified length calculation of the curve defined as Equation 3.28.

$$\delta_R(P_1, P_2) = ||\log(P_1^{-1}P_2)||_F = \left(\sum_{i=1}^{I} \log^2 \lambda_i\right)^{1/2} \tag{3.28}$$

With the real eigenvalues of the term $P_i^{-1}P_2$ being λ_i where $i = 1...n$.
It should be noted, that Equation 3.28 avoids the necessity of find the path Γ and thus simplifies Equation 3.27.
More information on Riemannian geodesic distance can be found in [4][17].

Exponential Map

A tangent map can be defined for each point $P \in P(n)$ as the set of tangent vectors S. A tangent vector S_i can be seen as the derivative at $t = 0$ of $\Gamma_i(t)$ between the point P and the exponential mapping of P_i as defined in Equation 3.29.

$$\exp_P(S_i) = P_i = P^{\frac{1}{2}} \exp(P^{-\frac{1}{2}} S_i P^{-\frac{1}{2}}) P^{\frac{1}{2}} \tag{3.29}$$

The exponential function is used to map a point S_1 from the tangent space into a point on the Riemannian manifold, using the point P as the center of the tangent space. The pseudocode in Algorithm 1 shows an implementation of the Riemannian exponential mapping function [13].

Algorithm 1 Riemannian exponential mapping

Input: Initial point $P \in P(n)$
Input: Tangent point $S_1 \in S(n)$
Output: $\exp_P(S_1)$
1: Let $P = U_1 \Lambda_1 U_1^\top$, where U_1 and Λ_1 is eigenvectors and eigenvalues of P
2: $G = U_1 \sqrt{\Lambda_1}$
3: $Y = G^{-1} S_1 (G^{-1})^\top$
4: Let $Y = U_2 \Lambda_2 U_2^\top$, where U_2 and Λ_2 is eigenvectors and eigenvalues of Y
5: $\exp_P(S_1) = (GU_2) \exp(V_2)(GU_2)^\top$

The logarithmic mapping defined in Equation 3.30 is the inverse of the exponential mapping. The logarithmic function is used to map the point P_1 from the Riemannian manifold into a point in the tangent space, using the point P as the center of the tangent space. The pseudocode in Algorithm 2 shows the implementation [13].

$$\log_P(P_i) = S_i = P^{\frac{1}{2}} \log(P^{-\frac{1}{2}} P_i P^{-\frac{1}{2}}) P^{\frac{1}{2}} \tag{3.30}$$

Algorithm 2 Riemannian logarithmic mapping

Input: Initial point $P \in P(n)$
Input: End point $P_1 \in P(n)$
Output: $\log_p(P_1)$
1: Let $P = U_1 \Lambda_1 U_1^\top$, where U_1 and Λ_1 is eigenvectors and eigenvalues of P
2: $G = U_1 \sqrt{\Lambda_1}$
3: $Y = G^{-1} P_1 (G^{-1})^\top$
4: Let $Y = U_2 \Lambda_2 U_2^\top$, where U_2 and $V\Lambda 2$ is eigenvectors and eigenvalues of Y
5: $\log_P(P_1) = (GU_2) \log(V_2)(GU_2)^\top$

As it is not possible to take square root of a matrix, the operation have to be done on the eigenvalues, similar to the 'exp' and 'log' operation. Thus Algorithm 1 and Algorithm 2 is used to describe the actual calculation of Equation 3.29 and Equation 3.30.
A visual illustration of the mapping can be seen in Figure 3.13, a red and blue indicator is used for '$\text{Exp}_p(S_i)$' and '$\text{Log}_p(P_i)$' respectively.

Riemannian Mean

When working in the Riemannian space it is not as straightforward as in the Euclidean space, to find the mean of a set of points. The problem occurs because the Riemannian space is curved, which complicates the computation.
According to [4] the mean is given by Equation 3.31.

$$\mathfrak{G}(P_1, ..., P_l) = \min_{\mu \in P(n)} \sum_{i=1}^{I} \delta_R^2(\mu, P_i) \tag{3.31}$$

which uses the geodesic distance explained in the subsection Riemannian Geodesic Distance. The equation finds the $\mu \in P(n)$ which results in the smallest sum of the distances between each SPD matrix given to μ. As μ can be any SPD matrix in the space of SPD matrices, an optimization algorithm must be employed to calculate the mean.
Such an algorithm is defined in [13] and given as Algorithm 3.

Algorithm 3 Iterative calculation of Riemannian mean

Input: $p_1, \ldots, p_N \in P(n)$
Output: $\mu \in P(n)$
1: $\mu_0 = I$
2: **repeat**
3: $\quad X_i = \frac{1}{N} \sum_{k=1}^{N} Log_{\mu_i}(P_k)$
4: $\quad \mu_{i+1} = Exp_{\mu_i}(X_i)$
5: **until** $||X_i|| <= \epsilon$

The algorithm takes a set of SPD matrices as input, and returns an SPD matrix of same dimensions, which is the estimated mean SPD matrix of the input set in the algorithm. An initial mean is created, namely an identity matrix. The algorithm then calculates an arithmetic mean, in line 3, using the logarithmic mapping function, which can be seen in Equation 3.30. The mapping function uses the currently calculated Riemannian mean μ_i as center for a tangent space. The newly calculated arithmetic mean is then used, using the same tangent space, to find a new Riemannian mean, as seen in line 4. The process of creating new arithmetic means and moving them to the Riemannian manifold, continues until the norm of the matrix X_i is below ϵ, as seen in line 5. This condition is a comparison of the largest singular value of the distance matrix X_i, and the preset ϵ. When this condition is met, the loop exits and the algorithm returns the current Riemannian mean.
The idea of the process of the algorithm can be illustrated as seen in Figure 3.14. Here the X axis is the norm of the distance matrix and the Y axis is a multi dimen-

sional axis of all the available features. The goal is then to get $||X_i||$ below the line of epsilon.

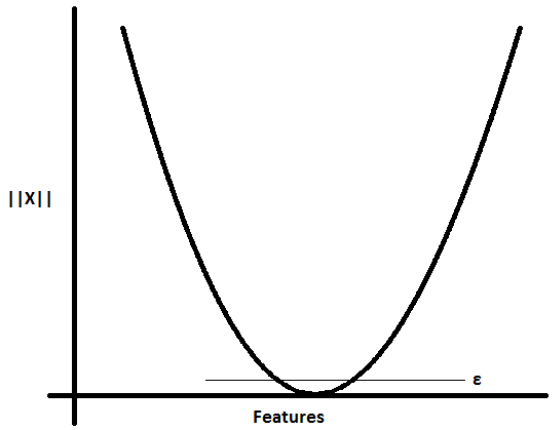

Figure 3.14

The value ϵ is what affects the runtime and the precision of the algorithm. The closer towards 0 the value is, the closer to the true mean of the matrices the estimated mean is, but this will result in extended amounts of iterations of the loop as well. Therefore ϵ should be tested, to find an optimal value depending on the precision desired and the time available for the training session.

Classification on the Riemannian Manifold

One simple way to classify on the Riemannian manifold is by using the method MDRM [4], which can be seen in Algorithm 4. This method works directly on the Riemannian manifold, and yields results with an accuracy comparable to a CSP LDA combination [4].
The MDRM method works by comparing the Riemannian geodesic distance, from the data in question, to the trained means of the available classes. The pseudo code for this method can be seen in Algorithm 4.
The algorithm works by computing the Riemannian mean, as seen on line 3-5, of the set of sample covariance matrices (SCM), which is calculated on line 1, of each class that were given for training.
The computed means are then used on runtime, as a comparison to the SCM of new data, which is calculated on line 2, in order to predict the class it belongs to. This prediction is based of the distance from the SCM to the different class means. The label returned as the prediction of the new data, belongs to the class of the mean with the minimum distance to the SCM being classified, as per line 6 and line 7.

Algorithm 4 Minimum Distance to Riemannian Mean

Input: a set of trials X_i of K different known classes
Input: X an EEG trial of unknown class
Input: $\mathfrak{J}^{(k)}$ the set of indices of the trials corresponding to the k-th condition
Output: $\hat{k}$ the estimated class of test trial X

1: Compute SCMs of X_i to obtain P_i (see Equation 3.6)
2: Compute SCM of X to obtain P
3: **for** $k = 1$ to K **do**
4: $\quad P_{\mathfrak{G}}^{(k)} = \mathfrak{G}(P_i, i \in \mathfrak{J}^{(k)})$ (see Equation 3.31)
5: **end for**
6: $\hat{k} = \text{argmin}_k \delta_R(P, P_{\mathfrak{G}}^k)$
7: **return** $\hat{k}$

Classification in the Riemannian Tangent Space

An alternative to MDRM, is to use recognized classification methods such as LDA. The problem is that these methods requires the vectors used, to be in Euclidean space and therefore does not work with the current points, which are in Riemannian space. As explained above, the logarithmic mapping, is a method to create a Euclidean plane from a point in Riemannian space. Using this method, the points in Riemannian space can get transfered to a shared Euclidean plane, and thus used with the Euclidean methods.

Algorithm 5 is the pseudo code for creating tangent vectors for a set of points in the Riemannian space, which works by first computing the Riemannian mean $P_{\mathfrak{G}}$ on line 1.

The mean is used to get the best possible plane for the classification methods. Because the points needs to be on the same plane, i.e. a tangent space created with the same center point, using the mean will result in the least distance from the points to the plane.

On line 3 the logarithmic mapping is used, with the mean as center, to create a tangent vector for each point given as input.

The vectors returned from the algorithm are then used to train the classifier, e.g. LDA. The mean created is saved, and used to create new vectors of a new data set, as it represents the point of which the tangent space is created.

When new data should be classified, a SPD matrix is first created. This matrix is used in collaboration with the previously calculated Riemannian mean, to create a new vector, using the same calculation as in line 3 in Algorithm 5. This vector is then classified as usual, using the trained LDA classifier.

Algorithm 5 Labeled vectors creation

Input: a set of I SPD matrices $P_i \in P(n)$
Output: a set of I vectors S_i
1: $P_{\mathfrak{G}} = \mathfrak{G}(P_i, i = 1 \dots I)$
2: **for** $i = 1$ to I **do**
3: $\quad s_i = upper(P_{\mathfrak{G}}^{-\frac{1}{2}} Log_{P_{\mathfrak{G}}}(P_i) P_{\mathfrak{G}}^{-\frac{1}{2}})$
4: **end for**
5: **return** S_i

3.8 Preparing the data for LDA

As explained in Section 3.2, the dimension of the vector, created from Algorithm 5 in Section 3.7, is $m = \frac{n(n+1)}{2}$. Because not all the dimensions contribute, or have varying degrees of contribution to the analysis, reducing the dimensionality, improves the analysis.
This reduction is done, using a method called principal component analysis (PCA) [9]. As the name implies, the method finds the principal components, i.e. the dimensions which best separates the data.
To compare the vectors returned from Algorithm 5, they are combined into a matrix $S = [s_1, \cdots, s_I]$, where I denotes the number of s_i vectors created in Algorithm 5. Similar to CSP as explained in Section 3.4, the variance of a square data matrix can be found using eigen decomposition. The matrix S is not guaranteed to be square though, so instead a method called singular value decomposition (SVD) is used.
The formula for SVD can be seen in Equation 3.32, where U is a $m \times m$ matrix of left singular-vectors, V a $I \times I$ matrix of right singular-vectors and Λ a $m \times I$ diagonal matrix containing, the singular values of the matrix S.

$$S = U \Lambda V \tag{3.32}$$

With the matrix U, found using Equation 3.32, the singular value decomposition can be used to create an orthogonalized tangent space, where the variables are uncorrelated using Equation 3.33.

$$S_0 = U^\top S \tag{3.33}$$

With the transformed variables in the orthogonal space S_0, a one-way analysis of variance (one-way ANOVA) [20], can be applied. This analysis will calculate a p-value for each variable. The p-value is a measurement of the statistical significance of a given variable, which can be used to determine, whether or not the variable should be used during the classification stage.

A false discovery rate [7] threshold is applied on the p-values for the variables, called a q-value, and will in this way reduce the number of variables. The q-value can be adjusted, to avoid having too many or too few variables in the end.

3.9 Riemannian Regression

Like Euclidean regression, explained in Section 3.6, the goal of Riemannian regression is to find a correlation between the EEG readings and a continuous value, such that a new reading can be translated to the corresponding value. As with the Euclidean classification, explained in Section 3.5, the Euclidean regression does not work in Riemannian space. Therefore other means must be employed.

Regression in the Riemannian Tangent Space

To enable the usage of Euclidean regression methods, it is necessary to transform the points from the Riemannian manifold into the tangent space similarly to how it was done in Section 3.7 where the goal was to use Euclidean classifications methods. Using the same method as TSLDA, a mean point on the Riemannian manifold is used as the center for a tangent space. Vectors can now be created using Algorithm 5 in Section 3.7. One part is different though. In TSLDA the mean point used, was for the entire training set. The reason being, the classification should cover all classes possible on the same tangent space. When using regression, the assumption is that the class is predetermined, therefore a mean point for a specific class, is used instead. Instead of using only one mean point as in TSLDA, one for each class will be used, similar to MDRM, which in turn means a tangent space for each class will be created. Inspired by TSLDA, this method will henceforth be designated as tangent space regression (TSREG). With TSLDA, various methods can be employed to reduce the number of dimensions of the vectors, as the base size is $\frac{n(n+1)}{2}$, as explained in Section 3.8. For regression, the same idea can be used, of reducing the number of dimensions, but instead of PCA, regression can use regularization, which is explained in Section 3.6.

3.10 Combined Classification & Regression

As explained in Chapter 2, the goal is to combine both predictor tasks, classification and regression.
Classification is used to predict the class of the signal, while regression is used to predict the magnitude. As both predictor tasks will be used on the same signal, the

training data will need to be labeled with both the corresponding class, as well as the signal's magnitude.
The choice of classification method does not change the structure of the system. The method used will be found through testing, to ensure the best result. For classification a single trained classifier is sufficient. For regression two models (see Section 3.6) are needed, one for each of the possible classes e.g. a regression model for the speed of left hand movement, and another for the speed of right hand movement. A model is created for each of the possible classes to be able to predict the magnitude of a new signal, no matter the classification result.
When using the trained system, the analysis will consist of several steps. First the preprocessing, as explained in Section 3.3 prepares the data. Then the classification method is used to predict the class, and finally the data is evaluated, using the corresponding class' regression model. This process is illustrated in Figure 3.15.

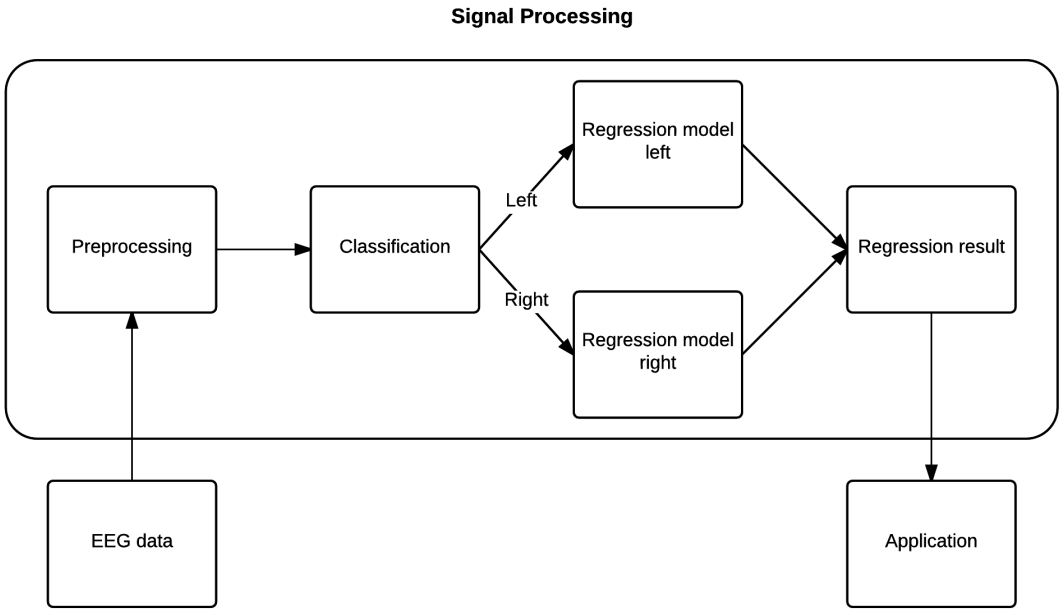

Figure 3.15

CHAPTER 4

IMPLEMENTATION

In this chapter, the implemented algorithms and functions are explained. The algorithms are implemented according to the theory in Chapter 3. Because some of the algorithms have been explained in our previous work [16], they will not be explained here. Some exceptions are made, for implementations that have changed, which will be explained in their respective sections.

4.1 Matlab

The API matlabcontrol [3] is used to connect to Matlab, as many of the functions are implemented in Matlab. In Listing 4.1 it is shown how to connect, which makes it possible to send commands from Java to Matlab.

Listing 4.1: Connecting to matlab

```
1 MatlabProxyFactory factory = new MatlabProxyFactory();
2 MatlabProxy proxy = factory.getProxy();
```

Through the created proxy it is now possible to send commands to Matlab using the `eval()` method, an example of this can be seen in Listing 4.2.

Listing 4.2: Example of using eval

```
1 proxy.eval("a = 1 + 2");
```

The example in Listing 4.2 shows the command '`a = 1 + 2`' being send via the proxy to Matlab to be executed.
In order to do calculations on a matrix from Java in Matlab, it needs to be converted to a type that Matlab can work with. This is achieved using `MatlabTypeConverter` as shown in Listing 4.3, where a jagged array in Java representing a matrix is saved

as a variable named `matrix` in Matlab so it can be manipulated in the Matlab environment.

Listing 4.3: Example of using the MatlabTypeConverter

```
1 double[][] matrixJava = new double[][] { {1, 2}, {2, 1} };
2
3 MatlabTypeConverter conv = new MatlabTypeConverter(proxy);
4 conv.setNumericArray("matrix", new MatlabNumericArray(matrixJava, null));
5
6 proxy.eval("result = inv(matrix)")
```

Finally to extract the result from the previous example one again needs to make use of the `MatlabTypeConverter`, as shown in Listing 4.4. The result is stored in a type known as a `MatlabNumericArray`, which can be used in java.

Listing 4.4: Getting the result

```
1 MatlabNumericArray result = conv.getNumericArray("result");
```

4.2 Riemannian basic function

Several functions are required for both MDRM and TSLDA, and therefore does not belong to either, but to a more general understanding of Riemannian geometry. This includes the functions for moving between Euclidean space and Riemannian space and method to calculate the mean in Riemannian space.

Exponentials

The exponential mapping function is used to find a point on the Riemannian manifold given a point in the tangent space, and is covered in Section 3.7.
The implementation for the exponential function, can be seen in Listing 4.5, takes two inputs `S1` and `P`, where `S1` is a point in tangent space and `P` is the center point of the tangent space.

Listing 4.5: Exponential mapping

```
1         conv.setNumericArray("S1", S1);
2         conv.setNumericArray("P", P);
3
4         proxy.eval("[U1 Lambda1] = eig(P);");
5         proxy.eval("G = U1*sqrt(Lambda1);");
6         proxy.eval("Y = inv(G)*S1*inv(G)';");
7         proxy.eval("[U2 Lambda2] = eig(Y);");
8         proxy.eval("P1 = (G*U2)*diag(exp(diag(Lambda2)))*(G*U2)';");
9
10        return conv.getNumericArray("P1");
```

Logarithms

The logarithmic mapping function is the inverse of the exponential function. Given a point on the Riemannian manifold it maps it to a point in the tangent space.
The implementation for the logarithmic function, can be seen in Listing 4.6, takes as the exponential function two inputs `P1` and `P`, where `P1` is a point on the Riemannian manifold and `P` is the center of the tangent space.

Listing 4.6: Logarithmic mapping

```
1         conv.setNumericArray("P1", P1);
2         conv.setNumericArray("P", P);
3
4         proxy.eval("[U1 Lambda1] = eig(P);");
5         proxy.eval("G = U1*sqrt(Lambda1);");
6         proxy.eval("Y = inv(G)*P1*inv(G)';");
7         proxy.eval("[U2 Lambda2] = eig(Y);");
8         proxy.eval("S1 = (G*U2)*diag(log(diag(Lambda2)))*(G*U2)';");
9
10        return conv.getNumericArray("S1");
```

Riemannian mean

As explained in Section 3.7 the task of calculating a mean in Riemannian space, is not as easy as in Euclidean space.
The variable `ClassMatrices` is the collection of SPD matrices given as parameters which are used to calculate the mean. This collection is first checked if there is only one matrix or less, which then just return the first elements in the list, either `null` if empty or the single matrix, which is also the mean.
A base mean is calculated, just an identity matrix to start from, and a base distance for all SPD matrices in `ClassMatrices` to this mean.
As the base mean is just a starting point, a new mean is calculated, using the function `ExponentialMappingOfPoint`, and the `while` loop is entered on line 27 in Listing 4.7. In the `while` loop, the current mean is set to the previously calculated mean on line 28, and using this the next distance is calculated. At this point the mean for the next iteration is calculated. The check for the `while` loop is then performed, which is a comparison between the norm of the distance matrix, calculated from the SPD matrices and the current mean, and the constant epsilon value. If the norm is greater than epsilon, the loop is ended and the mean used for the distance calculation is returned.

Listing 4.7: Riemannian mean

```
1         // Retrieve matlab connections
2         MatlabProxy proxy = Matlab.getProxy();
3         MatlabTypeConverter conv = Matlab.getConverter();
4
5         //Exit if wrong input
```

```
6           if (ClassMatrices.size() <= 1) {
7                   return ClassMatrices.get(0);
8           }
9
10          // Variable to compare wiht norm
11          double epsilon = 0.005d;
12          // Amount of channelse
13          channels = ClassMatrices.get(0).getLengths()[0];
14
15          // Create identity matrix
16          proxy.eval("CurrentMean = eye(" + channels + ");");
17
18          MatlabNumericArray currentMean = conv.getNumericArray("CurrentMean");
19          MatlabNumericArray nextMean;
20
21          // Calculate first distance between points and currently used mean
22          MatlabNumericArray currentDistance = CalculateDistance2Mean(ClassMatrices,
                currentMean);
23
24          // Calculate next possible mean
25          nextMean = ExponentialMappingOfPoint(currentMean, currentDistance);
26
27          do {
28                  currentMean = nextMean;
29                  currentDistance = CalculateDistance2Mean(ClassMatrices,
                        currentMean);
30
31                  nextMean = ExponentialMappingOfPoint(currentMean, currentDistance);
32          } while (MatrixOperations.Norm(currentDistance) > epsilon);
33
34          return currentMean;
```

4.3 Upper Operation

This section covers the implementation of the upper operation described in Section 3.7 which is used to convert the upper triangular part of a symmetrical matrix into a vector. The implementation of upper can be seen in Listing 4.8.

Listing 4.8: Upper operation

```
1  int dimension = (matrix.length * (matrix[0].length + 1)) / 2;
2
3  double[] sVector = new double[dimension];
4
5  int mIndex = 0;
6
7  for (int i = 0; i < matrix.length; i++) {
8          sVector[mIndex] = matrix[i][i];
9          for (int j = i + 1; j < matrix[0].length; j++) {
10                 mIndex++;
11                 sVector[mIndex] = Math.sqrt(2) * matrix[i][j];
12         }
13         mIndex++;
14 }
15
16 return sVector;
```

On line 1 in Listing 4.8 the length of the vector is calculated using the formula from Section 3.7 that says the length of the vector is m calculated as $\frac{n(n+1)}{2}$ where n is the electrode count.

The final result after having run through the matrix in the `for` loop is a vector where the diagonal gets a unity weight applied, while the off diagonal gets a weight of $\sqrt{(2)}$ applied.

4.4 Temporal function

The temporal smoothing function used in this implementation is a Hanning Window [16]. The length of the window is the same as the length of the signal. In line 5 the algorithm iterates each timestamp of the signal sample. For each of these timestamps, a weight or modifier is calculated. This weight is calculated by taking half the length of the window, to each side of the current point, as seen in line 8. As the window is moved over each timestamp, it is not guaranteed that the window does not exceed the range of the data set. Because of that, we check in line 9, the current modifier. If the condition is true, the modifier value is calculated and saved in line 10 and applied in 13, as well as added to a joint modifier value in line 12. This is done such that the correct divider can be used in line 16, to calculate the new value. And lastly, in line 17 the modified value is saved in the data set.

```
1  double currentValue = 0;
2  double currentModifier = 0;
3  double modifierDivider = 0;
4
5  for (int i = 0; i < data.size(); i++) {
6          currentValue = 0;
7          modifierDivider = 0;
8          for (int N = -1 * (data.size() / 2) + 1; N < data.size() / 2; N++) {
9                  if (i + N >= 0 && i + N < data.size()) {
10                 currentModifier = 0.5 * (1 - Math.cos((2 * Math.PI * (N +
                       (data.size() / 2))) / (data.size())));
11
12                 modifierDivider += currentModifier;
13                 currentValue += data.get(i + N) * currentModifier;
14                 }
15         }
16         currentValue /= modifierDivider;
17         data.set(i, currentValue);
18 }
```

This implementation, and the implementation in [16], have been tested with several different data set, but was found to result in worse classification each time. Therefore it will not be used in the final tests.

4.5 Common Spatial Pattern Implementation

This section covers the implementation of the Common Spatial Pattern method, described in Section 3.4, and follows the implementation described in our previous work [16] closely. The code consists of two primary functions `trainCSP()` and `applyCSP()`. the function `trainCSP()` is used during the calibration of the BCI system and creates a weight matrix to be used during the feedback phase. The function `applyCSP()` is used to apply the weight data to some new data, to create a feature vector to be classified. The creation of the feature vector is shown in Listing 4.9 and follows Equation 4.1.

Listing 4.9: Extraction of feature vector

```
1  for (int i = 0; i < Z.length; i++) {
2          nominator = calcVariance(Z[i]);
3          double denominator = 0;
4
5          for (int j = 0; j < Z.length; j++) {
6                  denominator += calcVariance(Z[j]);
7          }
8
9          featureVector[i] = Math.log10(nominator / denominator);
10 }
```

$$f_p = \log\left(\frac{var(Z_p)}{\sum_{i=1}^{2m} var(Z_i)}\right) \tag{4.1}$$

In Equation 4.1 the term Z_p refers to the pth row of the matrix Z of CSP filters, with p varying from 1 to $2m$ and the operation $var(.)$ gives the variance of the input.

4.6 Linear Discriminant Analysis

In our previous work [16] Matlab have been used to train and also classify data. To train the classifier the following Matlab method is called `fitcdiscr(data, labels)` where `data` is a matrix consisting of rows corresponding to feature vectors and columns trials. The `labels` are the class the data belongs to, where each label corresponds to one trial in the `data`. The result of the function call is a classification object, containing the information of how data patterns look like for each class. When classifying, this object is used to predict the class data belongs to with the following function `predict(LDAClassifier, dataToBeClassified)`. The `LDAClassifier` is the object computed from the test data and `dataToBeClassified` is the data we want to classify.
The implementation of LDA are split into two methods `LDA.train()` and `LDA.classify()` which are shown in Listing 4.10 and Listing 4.11 respectively.

Listing 4.10: Linear Discriminant Analysis Training

```
1  public static void train(ArrayList<double[]> data, int[] labels) throws
       MatlabInvocationException {
2
3  .
4  .
5  .
6
7  // Stores the variables in the Matlab environment
8  conv.setNumericArray("data", dataNumericArray);
9  conv.setNumericArray("labels", labelsNumericArray);
10
11 // Creates the classifier with the stored variables in Matlab
12 proxy.eval("LDAClassifier = fitcdiscr(data, labels);");
13 }
```

The method `train` shown in Listing 4.10 trains the LDA classifier. The methods takes an `ArrayList` of vectors and an array of same dimension containing labels corresponding to each entry in the `ArrayList` of vectors. After some type conversions the variables are stored in a Matlab environment on line 8-9. Next the classifier is trained using Matlab on line 12. After this step a trained LDA classifier is stored in Matlab and is ready to be used to predict new data. The method `classify` is shown in Listing 4.11 and is used to classify data after the classifer is trained. The method takes as input an `ArrayList` of vectors containing data and returns an integer array of the same size as the input `ArrayList`, containing a classification result (e.g. 0 or 1) for each entry in the data being classified. On line 8 the Matlab method `predict` is called, using the `LDAClassifier` object, created when training the classifier and the data to be predicted. On line 14-18 the result is extracted and returned.

Listing 4.11: Linear Discriminant Analysis Classify

```
1  public static int[] classify(ArrayList<double[]> data) throws
       MatlabInvocationException {
2
3  .
4  .
5  .
6
7  // Classify using the classifier object in Matlab
8  proxy.eval("LDAClassificationResult = predict(LDAClassifier,
       dataToBeClassified);");
9
10 // Retrieve the result
11 double[] tmpLDAClassificationResult = (double[])
       proxy.getVariable("LDAClassificationResult");
12
13 // Convert to integer array
14 int[] LDAClassificationResult = new int[tmpLDAClassificationResult.length];
15 for (int i = 0; i < tmpLDAClassificationResult.length; i++) {
16 LDAClassificationResult[i] = (int) tmpLDAClassificationResult[i];
17 }
18 return LDAClassificationResult;
19 }
```

4.7 Minimum distance to Riemannian mean

The implementation of the MDRM method follows the theory as described in Section 3.7 under 'Classification in the Riemannian Manifold'.
The method is divided into two parts, a calibration phase and a runtime phase. In the calibration phase the Riemanninan mean is found for the desired classes, while in the runtime phase, the new data is checked up against the Riemannian means to find the class to which the distance is shortest.

Calibration phase

The code in Listing 4.12 shows how for each class the corresponding Riemannian mean is found. The list `classSCMs` contains the covariance matrices for each class, such that the method `classSCMs.get(i)` returns all the covariance matrices for class `i`. Finally the mean calculated for a class is stored in the list `classMeans`.

Listing 4.12: Calculate Riemannianm ean for each class

```
1 classSCMs.size();
2 for (int i = 0; i < classCount; i++) {
3         MatlabNumericArray meanPoint =
              IntrinsicMeanOfDiffusionTensors(classSCMs.get(i));
4         classMeans.add(meanPoint);
5 }
```

Runtime phase

During the runtime phase new data is given as input, to be classified. The data's distance to the different class means are compared in order to determine the class it belongs to, as shown in Listing 4.13. The data is given to the method `GeodesicDistance()` as a parameter, together with the class mean, if the distance to the mean is shorter than the current mean, the class is stored. When all classes have been tested, the class with the shortest distance will be returned as the prediction.

Listing 4.13: Classify new data

```
1 // find minimum distance to determine class of data
2 for (int i = 0; i < classCount; i++) {
3         double distance = GeodesicDistance(classSCM, classMeans.get(i));
4
5         if (distance < currMinDistance) {
6                 currMinDistance = distance;
7                 classLabel = i;
8         }
9 }
```

4.8 Tanget space mapping

This section covers the implementation of the mapping from the Riemannian space to the tangent space, the reason for this, is to enable the use of methods that rely on the Euclidean space e.g. LDA and linear regression.
The method for converting the points in the Riemannian manifold represented as matrices to an equivalent vector in tangent space can be seen in Listing 4.14.

Listing 4.14: Tangent Space Mapping

```
1  conv.setNumericArray("SPDMeanPointModified", meanPoint);
2  proxy.eval("[U Delta] = eig(SPDMeanPointModified);");
3  proxy.eval("G = U*sqrt(Delta);");
4  proxy.eval("SPDMeanPointModified = inv(G)");
5
6  // List to store tangent vectors in
7  ArrayList<double[]> tangentVectors = new ArrayList<double[]>();
8
9  // Create tangent vectors from SPDs
10 for (int i = 0; i < SPDs.size(); i++) {
11 MatlabNumericArray tmp = LogarithmicMappingOfPoint(meanPoint, SPDs.get(i));
12 conv.setNumericArray("LogResult", tmp);
13 proxy.eval("InnerUpperResult = SPDMeanPointModified * LogResult *
       SPDMeanPointModified");
14 MatlabNumericArray InnerUpperResult = conv.getNumericArray("InnerUpperResult");
15 tangentVectors.add(upperOperation(InnerUpperResult.getRealArray2D()));
16 }
17
18 // Return the tangent vectors
19 return tangentVectors;
```

The code as shown in Listing 4.14 follows the pseudocode of Algorithm 5 in Section 3.7 and goes through each covariance matrix that is stored in the `ArrayList` named `SPDs` and applies the algorithm. The resulting list `tangentVectors` is then returned containing a tangent space vector corresponding to each of the points in the Riemannian manifold.

4.9 Tangent Space Linear Discriminant Analysis

The implementation of Tangent Space Linear Discrimiant Analysis combines the mapping of points into tangent space (see Section 4.8) and LDA (see Section 4.6). The implementation of the TSLDA method can be split into a calibration phase and a runtime phase just as the Minimum Distance to Riemannian Mean method in Section 4.7.

Calibration phase

During the calibration phase the labeled data is gathered in a list and converted into covariance matrices. Based on these covariance matrices a mean point is found

to be used in the mapping of points into tangent space. The implementation in Listing 4.15 shows how this is achieved, where first the SCM's for all classes stored in `classSCMs` are passed to the function `calculateMeanForEntireSet()` for a mean point to be calculated, as can be seen on line 2 in Listing 4.15.

Listing 4.15: Calculate mean and tangent space mapping

```
1 // Calculate riemannian mean for whole data set
2 Riemannian.calculateMeanForEntireSet(classSCMs);
3
4 // Calculate the tangent vectors
5 ArrayList<MatlabNumericArray> leftTangentVectors =
      Riemannian.tangentSpaceMappingV2(classSCMs.get(0));
6 ArrayList<MatlabNumericArray> rightTangentVectors =
      Riemannian.tangentSpaceMappingV2(classSCMs.get(1));
```

After the mean is calculated the covariance matrices are mapped into the tangent space, as can be seen in Listing 4.15 on line 5-6. This is done by calling the method `tangentSpaceMappingV2()`, where the passed parameter `classSCMs.get(0)` contains the covariance matrices for class 1, while `classSCMs.get(1)` contains the covariance matrices for class 2. The tangent vectors are now as described in Section 3.8 going to be joined together to create a new matrix as seen in Listing 4.16.

Listing 4.16: Classifier training

```
1 conv.setNumericArray("tangentVector", vectors.get(0));
2 proxy.eval("tangentVectorMatrix = tangentVector';");
3
4 for (int i = 1; i < vectors.size(); i++) {
5 conv.setNumericArray("tangentVector", vectors.get(i));
6 proxy.eval("tangentMatrix = horzcat(tangentVectorMatrix, tangentVector');");
7 }
8
9 return conv.getNumericArray("tangentSpaceMatrix");
```

The first thing that happens in Listing 4.16 on line 1-2 is that the first tangent vector gets inserted into the variable `tangentVector` and afterwards it is transposed and inserted into the variable `tangentVectorMatrix` as the first vector. The lines 4-7 is a `for`-loop, where each of the remaining vectors are concatenated into the matrix `tangentVectorMatrix`, that is then returned.

Listing 4.17: Classifier training

```
1 MatlabNumericArray principalComponents =
      MatrixOperations.principalComponentAnalysis(tangentVectorMatrix);
2 MatlabNumericArray pValues = MatrixOperations.oneWayAnova(principalComponents,
      group);
3 MatlabNumericArray fdrValues = MatrixOperations.findFalseDiscoveryRate(pValues);
```

Listing 4.17 shows how the returned matrix of tangent vectors `tangentVectorMatrix` is run through the remaining operations covered in Section 3.8. On line 1 in Listing 4.17 a principal component analysis is performed on the matrix, which is done using the '`svd`' operation in matlab, the output is then used to perform the principal

component analysis as shown in Equation 3.33 from Section 3.8. One-way ANOVA is then performed using the matlab method '`anova`' for each of the variables returning a vector of p-values. Finally the FDR method will be used on the p-values returning a vector of 0's and 1's where 1 indicate that the variable is selected.

Listing 4.18: Classifier training

```
function [ out ] = selectVariables( vector, fdr )
        count = sum(fdr);
        tmpVector = zeros(1,count);
        vectorLength = length(vector);
        index = 1;
        for i=1:vectorLength
                if index > count
                        break;
                elseif fdr(i) == 1
                        tmpVector(index) = vector(i);
                        index = index + 1;
                end
        end
        out = tmpVector;
end
```

For the list of tangent vectors belonging to each class the variables selected by the previous operations are now extracted into a new vector as seen in Listing 4.18. These vectors are now ready to be used to train an LDA classifier.

Listing 4.19: Classifier training

```
1  // Create class labels and merge left/right
2  int[] classLabels = new int[leftTangentVectors.size()
3  + rightTangentVectors.size()];
4  ArrayList<MatlabNumericArray> mergedTangentVectors = new
       ArrayList<MatlabNumericArray>();
5
6  for (int i = 0; i < leftTangentVectors.size(); i++) {
7  classLabels[i] = 0;
8  mergedTangentVectors.add(leftTangentVectors.get(i));
9  }
10 for (int i = leftTangentVectors.size(); i < leftTangentVectors.size() +
       rightTangentVectors.size(); i++) {
11 classLabels[i] = 1;
12 mergedTangentVectors.add(rightTangentVectors.get(i - leftTangentVectors.size()));
13 }
```

On line 6-9 in Listing 4.19 shows how the data is labeled for the left hand trials, while line 10-13 shows the labeling of the right hand trials while the tangent vectors are merged into a list to be given together with the labels to train the classifier.

Runtime phase

During the feedback phase new data is read and these points are mapped into the tangent space, using the same method as described in the calibration phase, this is shown on line 1-4 in Listing 4.20.

Listing 4.20: Feedback phase

```
1 // Calculate the tangent vectors for left class
2 leftTangentVectors = Riemannian.tangentSpaceMappingV2(classSCMs.get(0));
3
4 // Calculate the tangent vectors for right class
5 rightTangentVectors = Riemannian.tangentSpaceMappingV2(classSCMs.get(1));
6
7 leftTangentVectors = Riemannian.selectVariables(leftTangentVectors, fdrValues);
8 rightTangentVectors = Riemannian.selectVariables(rightTangentVectors, fdrValues);
```

On line 7-8 in Listing 4.20 the selected variables are extracted from the tangent vectors, using the previously calculated FDR values. This is done by calling the method shown in Listing 4.18. The tangent vectors now containing only the selected variables are once again merged into a list and given to the LDA method to be classified as shown in Listing 4.21.

Listing 4.21: Feedback phase

```
1  mergedTangentVectors.clear();
2
3  for (int i = 0; i < leftTangentVectors.size(); i++) {
4  mergedTangentVectors.add(leftTangentVectors.get(i));
5  }
6  for (int i = leftTangentVectors.size(); i < leftTangentVectors.size() +
       rightTangentVectors.size(); i++) {
7  mergedTangentVectors.add(rightTangentVectors.get(i - leftTangentVectors.size()));
8  }
9
10 // Convert MatlabNumericArray to double[][]
11
12 int[] classificationResults = Classification.classifyMatlab(1,
       mergedTangentVectors);
```

4.10 MDRM combined with regression

This section describes the combination of the classification method MDRM (see Section 3.7) and a linear regression method (see Section 3.6). The implementation of the MDRM method is described in Section 4.7. This section describes the regression aspect of the implementation and how it is combined with the MDRM method.
The idea as described in Section 3.10 is to use the classification method to predict the class the signal belongs to whereafter the magnitude is predicted, using linear regression.

Listing 4.22: Fitting the regression model

```
1 ArrayList<MatlabNumericArray> leftTangentVectors =
      Riemannian.tangentSpaceMappingV2(classData.get(0),
      Riemannian.classMeans.get(0));
2 ArrayList<MatlabNumericArray> rightTangentVectors =
      Riemannian.tangentSpaceMappingV2(classData.get(1),
      Riemannian.classMeans.get(1));
3
```

```
4  MatlabNumericArray leftTangentMatrix =
       MatrixOperations.createMatrixFromVectors2(leftTangentVectors);
5  MatlabNumericArray rightTangentMatrix =
       MatrixOperations.createMatrixFromVectors2(rightTangentVectors);
6
7  Regression.trainRegressionLeft(leftTangentMatrix, timestampsLeft);
8  Regression.trainRegressionRight(rightTangentMatrix, timestampsRight);
```

Each of the class covariance matrices are converted into the tangent space using their own class mean point as the center on line 1-2 in Listing 4.22. The resulting tangent vectors are then gathered together to create a matrix for each of the classes on line 4-5, where each row in the matrix is a tangent vector. The linear regression model is then trained on line 7-8 for each of the classes, where the variables `timestampsLeft` and `timestampsRight` are vectors containing the time it took to perform the 90 degree hand movement as described in Section 5.7. The linear regression model is trained using the matlab command 'fitlm' as shown in Listing 4.23.

Listing 4.23: Regression method call

```
1  MatlabProxy proxy = Matlab.getProxy();
2  MatlabTypeConverter conv = Matlab.getConverter();
3
4  conv.setNumericArray("DataForRegression", data);
5  conv.setNumericArray("LabeledDataForRegression", labeledData);
6
7  proxy.eval("LinearModelRegressionLeft = fitlm(DataForRegression,
       LabeledDataForRegression, 'linear');");
```

After the MDRM method have classified a new signal to a given class, the linear regression model fitted for that class is then used to predict the magnitude of the signal using the method in Listing 4.24.

Listing 4.24: Regression method prediction

```
1  MatlabProxy proxy = Matlab.getProxy();
2  MatlabTypeConverter conv = Matlab.getConverter();
3
4  conv.setNumericArray("DataToPredict", dataToPredict);
5
6  proxy.eval("RegressionResult = predict(LinearModelRegressionLeft,
       DataToPredict);");
7
8  // Retrieve the result
9  double[] regressionResult = (double[]) proxy.getVariable("RegressionResult");
10
11 return regressionResult;
```

Line 6 in Listing 4.24 shows the call to the matlab method 'predict' that takes as input a linear model, and the new data in order to give a prediction. The prediction is then converted to a `double` and returned.

CHAPTER 5

TESTING

To evaluate the different methods used in this project, several tests will be performed. Because the project investigates two different predictor tasks, namely classification and regression, the two will be combined and tested. A test of CSP, MDRM, TSLDA and finally a combination of MDRM/TSREG is conducted. The goal of this chapter is to compare the various methods with one predictor task and observe how well it works when combining two predictor tasks. In the end a conclusion is given based on the observed results for each method and possible complications.

5.1 Data

To confirm the validity of the used and implemented algorithms, the data used for the classification test is professional recorded data. Berlin BCI, abbreviated as BBCI have previously staged competitions in EEG analysis [6], with the last competition being in November-December 2008. The data recorded for the competition and the corresponding labels have been made publicly available. The data was recorded, using an EEG reader with 22 electrodes, placed as illustrated in Figure 5.1. The competition contains five data sets from four different institutes. The data set is sampled at 250 Hz and bandpass filtered between 0.5 - 100 Hz. The set contains recordings from nine test subjects and all will be tested by the classification prediction methods CSP, MDRM and TSLDA. There is no existing competition data for regression tasks, which have forced us to record our own data for the test that involves a regression prediction task, namely TSLDA combined with TSREG. The data we have recorded and used to test the regression predictor task is explained later in this chapter.

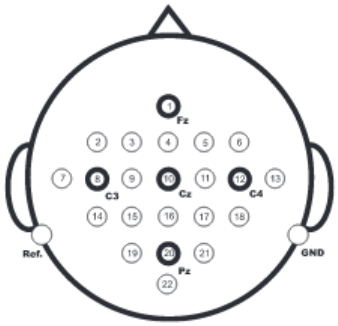

Figure 5.1: Placement of the 22 electrodes of the EEG reader used for competition IV set 2a

5.2 Cross Validation

Throughout all tests, cross validation is performed to tune various parameters, e.g. the value of ϵ used for Riemannian mean calculation. The way it is done, is by dividing the data set into two parts, a training set and a test set. The training is then split into five sets, as we use 5-fold cross validation. One set of the five are chosen as temporary test data, and the remainder four sets are used to train the system. This procedure is done for each set. This set of tests are performed for a variety of different parameter values, and the parameter value, which results in the best mean for the five tests, is then used in the final test. After the system is trained with the tuned parameters and the complete training set, the original test set is evaluated and the result of the accuracy is given, which is the result that is listed in the tables shown later on in this chapter. An illustration of this process is shown in Figure 5.2

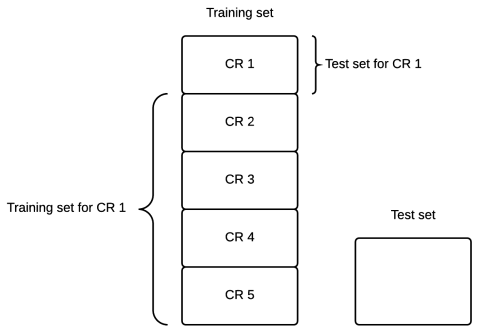

Figure 5.2: 5-fold cross validation

5.3 Subsampling

Subsampling is used exclusively for cutting out reaction time. When the competition data is used, as explained in Section 5.1, subsampling is enabled. To address the reaction time the first 0.5 second is cut from the start and only the next 500 samples (2 seconds) are considered as a part of the signal. The reason we limit the signal length to 500 samples is because it yielded the best results, found by cross validation. It was observed that if a longer time frame of the signal was considered, the accuracy decreased. A possibly explanation of this, can be the primary characteristics of moving the left or right hand, at the early stage of the movement (and the thought to move). The further of the initial thought or the actual movement of the hand, the signal gets less focused.

5.4 Result format

The rest of this section shows the results for the aforementioned methods. The format is denoted in the following way 75/80, where the first number is the percentage of correct classifications for left hand movement and the second number indicates the percentage of correct classification for the right hand.

5.5 Euclidean Classification

CSP is tested, mainly to have a comparison to the Riemannian methods. According to [4], CSP and MDRM have similar accuracy when classifying. Below are the results for the classification accuracy when using CSP on the data described in Section 5.1 and using cross validation as described in Section 5.2.

CSP					
Subject 1	95/72	Subject 4	44/73	Subject 7	59/78
Subject 2	98/1	Subject 5	15/90	Subject 8	96/95
Subject 3	83/100	Subject 6	79/40	Subject 9	86/96

The first notable discovery by examining the results, is that they differ a lot from test subject to test subject. Considering test subject 2 and 8, it is clear that the quality of the data gathered from the two are very different. It seems like the data from test subject 2 looks very similar, and therefore nearly all trials are classified as left hand movement. On the other hand test subject 8 provides very promising results with a nearly perfect accuracy rate on both hands. This pattern of varying results from subject to subject is something that is present in all the tests performed.

5.6 Riemannian Classification

For Riemannian classification two methods are tested, namely MDRM and TSLDA, which are two very different methods. MDRM takes advantage of the structure of the covariance matrices and therefore stays in the Riemannian space. TSLDA on the other hand converts the Riemannian space into an Euclidean space.

Epsilon

As explained in Section 3.7 the ϵ value determines the precision of the mean SPD matrix of a set. When performing tests that use Riemannian mean the ϵ influences the results. For our Riemannian classification methods we set ϵ to 0.5 for all tests, as this value gave the best result while using cross validation, while still terminating within a reasonable time frame. Furthermore this is also the value that is used when testing Riemannian classification and regression combined, which is shown in the next section.

MDRM					
Subject 1	92/87	Subject 4	55/84	Subject 7	45/89
Subject 2	92/2	Subject 5	54/58	Subject 8	96/97
Subject 3	79/100	Subject 6	71/69	Subject 9	87/96

As mentioned in [4], it was observed that the results from CSP and MDRM are very similar. As seen in the MDRM table, the results are very similar compared to the results in CSP and thus confirms this observation. It should be noted that even though CSP and MDRM performs with very similarly results, MDRM still shows considerably improvements over CSP runtime wise. The runtime of CSP takes 14 min, compared to the runtime of MDRM that takes 7 min, with the same input. In other words, it takes twice as long and achieves similar results.

TSLDA					
Subject 1	80/63	Subject 4	64/56	Subject 7	65/42
Subject 2	87/14	Subject 5	1/98	Subject 8	71/97
Subject 3	79/77	Subject 6	77/22	Subject 9	60/91

Furthermore it is observed in [4], that the TSLDA method yields better results in their test. That is not the case in our tests, as can be seen in the TSLDA table. This is of course worrying, but there is a possible explanation. When running the TSLDA method it was observed that the number of variables selected by FDR (see Section 3.8) varied greatly, from all of them to none. In an attempt to overcome this, FDR was modified to dynamically change the q-value, which determines the

threshold, to ensure that the number of found variables were within a given range. This could possibly be improved further by manually selecting a q-value for each subject or by using cross-validation to find the q-values.

5.7 Riemannian Classification Regression Test

To test our Riemannian regression method, it is needed to generate test data, as it has not been possible to locate any publicly available data with regression labels. This section covers how the data is gathered in a test setup, the test application, methods used in the test, and finally the results of the Riemannian regression.

Test Setup

The subject is placed on a chair in a relaxed position. At the side of the test subject there is a table in a height, such that the elbow can rest comfortably on it. Two buttons are placed so the subject have the hand placed on one button when the arm is relaxed, we call this the lower button. The other button is placed such that, when the subject raises the arm, the button will be pushed when the arm reaches 90 degrees, we call this the upper button. The buttons are attached to a LEGO NXT 2.0 computer, which in turn is attached to a PC. The subject is instructed to raise his/her arm to reach the button at different speeds (e.g. 0.5, 1, 1.5, 2 seconds), to ensure a separation of the data points. By doing this test, data is gathered with the EEG headset, which starts recording when the test subject lift his/her arm and continues until the button at 90 degrees is pressed. Usually, when recording EEG data, the movements are not actually performed, but imagined instead, with actual movement indicating a possible noise source. As imagining moving a hand at a specific speed, is difficult, for the tests performed in this section, the movements are actual. The goal of the test setup is to record data at different speeds, such that a regression model can be calculated and used on runtime to estimate the speed of arm movements. The over all design of the test setup can be seen in Figure 5.3.

Test Application

A test application is developed to be able to record EEG data in our test environment. The main screen is illustrated in Figure 5.4, which is the first and only screen presented to the user. The screen consists of a start button and two radio buttons. The start button starts the test, which should be pressed when the user is in position and is resting his/her hand on the table on top of the start button. The radio buttons indicates whether or not the lower or upper button is pressed, this is for observational use only. Furthermore a timer is positioned in the middle of the

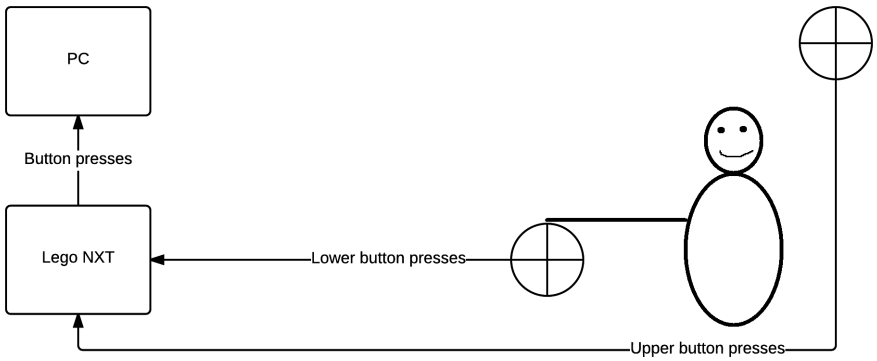

Figure 5.3: Test setup design

screen, showing the user the time given to raise his/her arm 90 degrees. The way the application works is that it sends a signal to a PC running the actual recording software, which starts recording when the user raises his/her hand from the start button and records until the stop button is pressed. This is called a trial and is stored in a text file, with the exact time it took to do the arm movement, alongside with the actual time it took. The reason why we record the exact time is because it is often hard to perform the arm movement in the exact time frame given, so by storing the exact time the regression training will be more accurate. It should be noted by using this test setup that no reaction time needs to be accounted for, as the EEG reader only records between when the start button is released and until the end button is pressed.

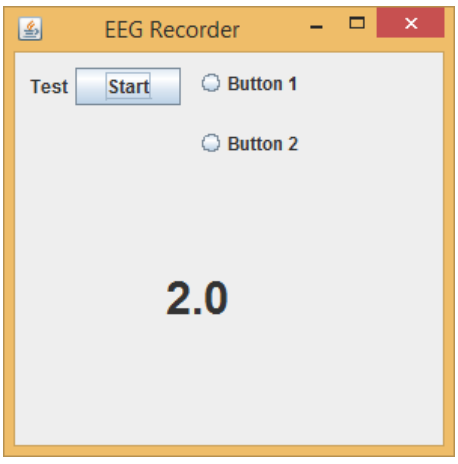

Figure 5.4: Test - Main Screen

Test Recording

The EEG recording was performed for two separate subjects. Each of these subjects performed the recording once for each hand, recording 25 trials at each of the designated speeds. Therefore the test were performed using 100 trials for each hand for the classification.

Test Methods

After having gathered the data for the two predictor tasks, namely class (left/right hand movement) and the speed of the movement, the two predictor task tasks can be tested. First the data needs to be classified, this can be done with either CSP, MDRM or TSLDA. Based on the test results in Section 5.5 and Section 5.6 MDRM is chosen for the classification task. The MDRM method use the same epsilon value as used when tested alone (see Section 5.6). The test is then performed, with the class predicted first. Using the result of the classification, a regression model is selected, and the test for regression is performed, thus resulting in both a class prediction and a speed prediction.

Test Results

In this section, the results of our combined predictor task method is shown. The test is performed on two test subjects. The tables shows the actual class and the predicted class denoted as Class and P Class respectively, and the same way for predicted speed.

MDRMTSREG - Subject 1							
Class	P Class	Speed	P Speed	Class	P Class	Speed	P Speed
Left	Left	676	2040	Right	Right	1059	1107
Left	Left	1006	416	Right	Right	1917	7423
Left	Left	1966	-2330	Right	Right	1150	5283
Left	Left	1301	2314	Right	Right	605	-2703
Left	Left	616	5465	Right	Right	1043	228
Left	Left	1058	2776	Right	Right	559	295
Left	Left	1899	532	Right	Right	1539	3396
Left	Left	1322	1253	Right	Right	1883	-3346
Left	Left	934	4093	Right	Right	1014	-2553
Left	Left	1916	-11857	Right	Right	515	1260
Left	Left	609	3974	Right	Right	1688	2606
Left	Left	604	-1661	Right	Right	2503	4358
Left	Left	1575	112	Right	Right	1157	4270
Left	Left	1376	27	Right	Right	1424	5276
Left	Left	1242	2760	Right	Right	549	-798
Left	Left	641	-522	Right	Right	2211	2769
Left	Left	1758	2419	Right	Right	1731	1117
Left	Left	1419	446	Right	Right	1126	-1165
Left	Left	1923	2759	Right	Right	2035	1113

The first noteworthy observation is the classification results. The classification performs well, with an accuracy of 100%. It should be noted though, that during cross validation this was not the case, but after choosing the best settings after cross validation, the accuracy was perfect. The regression prediction, on the other hand, is inaccurate. As can be seen in the result table for subject 1, the actual speed and the predicted speed, are rarely close to each other.

MDRMTSREG - Subject 2							
Class	P Class	Speed	P Speed	Class	P Class	Speed	P Speed
Left	Left	1675	968	Right	Right	567	-5184
Left	Left	1879	2962	Right	Right	2625	7821
Left	Left	1204	418	Right	Right	1074	5555
Left	Left	740	1127	Right	Right	1655	-5736
Left	Left	2454	3022	Right	Right	1922	1220
Left	Left	1717	261	Right	Right	1197	-4374
Left	Left	1382	478	Right	Right	592	3273
Left	Left	1094	292	Right	Right	1709	-662
Left	Left	710	2911	Right	Right	1137	10615
Left	Left	955	1624	Right	Right	2025	10550
Left	Left	2691	684	Right	Right	1528	-3316
Left	Left	1649	552	Right	Right	673	22
Left	Left	1595	2426	Right	Right	1261	3327
Left	Left	2000	1670	Right	Right	1799	4756
Left	Left	654	653	Right	Right	1126	10199
Left	Left	1550	947	Right	Right	613	5219
Left	Left	1043	1665	Right	Right	2241	-4242
Left	Left	2023	1324	Right	Right	617	-1302
Left	Left	690	1396	Right	Right	1325	7199

The patterns observed for test subject 1 is similar to test subject 2. The classification results are good, but the regression results are still very inaccurate.

Based on the above results, of the two predictor tasks combined method, it is clear that the regression results are not acceptable. It is hard to conclude anything based on the results from the regression task. It could be, that it simply is not possible to fit a linear model to determine the speed of the subjects hand movement. Another possibility is that the model might be overfitted, as no regularization was applied to the model on account of erroneous implementation. An indication of overfitting as a cause, can be found in the table for first test subject, where the predicted values generally follow the increase and decrease of the actual time recorded during the recording. As further prove, a subsequent test was performed, with the evaluation data used for calibration as well. As this test showed near perfect results, the possibility of overfitting has been further proved.

CHAPTER 6

CONCLUSION

Throughout this project, a BCI system was developed, which uses Riemannian classification methods and Euclidean regression on a tangent space, based on the Riemannian manifold. Based on the inplementation from our previous work in [16] the preprocessing was implemented, as well as a classification method using CSP combined with LDA. This implementation was used to compare the prediction performance of CSP combined with LDA against the two Riemannian methods, namely MDRM and TSLDA.
As shown in Section 5.5 and Section 5.6, the results of the classification predictions have a high degree of variance, depending on the test subject. Despite the variance, the methods CSP and MDRM performs similarly, and TSLDA slightly worse, and therefore the method MDRM was chosen to be combined with regression. In the test, the combined method classified well, but the regression was inaccurate, as seen in Section 5.7.
The question posed in the problem statement, whether it is possible using a consumer grade EEG reader to combine two predictor tasks to determine both class and magnitude of a signal, can not be answered as of yet. Though the classification was successful, the inaccuracy of the regression, means that new tests and improvements would need to be made, to be able to give a well-founded answer to the problem. One known improvement would be regularization, as the inaccuracy of the regression result, is most likely caused by overfitting.

CHAPTER 7

FUTURE WORK

This chapter outlines our thoughts of possible aspects of our product that can be further improved. This includes methods, implementation and test.

7.1 Methods and implementation

Using the assumption, that a continuous value, i.e. regression, can be predicted from BCI data, would mean that the methods used here, are insufficient. During the research for this project, another regression method using Riemannian manifolds was discovered, though it was not implemented on account of the time restraint. The paper titled Geodesic Regression on Riemannian Manifolds [14] describes this method. What they do is that they instead of creating the tangent space, to use the Euclidean regression, as is done in this project, they find a regression model directly on the Riemannian manifold. In this way it ensures no transformation of the data, and thus no data loss, which can improve the precision of the prediction.

The implementation of our regression does not use regularization at the moment. Implementing regularization, e.g. Lasso, can improve the results, because overfitting is a problem, as the number of variables exceeds the number of trials we have.

The implementation of TSLDA suffers from a few issues that might account for the results being worse than MDRM. As noted in Section 5.6, [4] shows TSLDA yielding better results. It was mentioned in Section 5.6, that the q-value given to FDR for the selection of variables, is selected dynamically. Instead of selecting the q-value dynamically, better results might be achieved if the q-value is selected manually or by using cross validation.

7.2 Test Setup

The data gathering in our test setup was performed in a conference room, in a sub-optimal environment. The reason why we think the environment could be improved is based on the following issues:

Screen Placement

The screen showing the user how much time he/she should use to perform a hand movement and how much time is left was awkwardly placed, which often led to the user turning their head, to follow the time, which led to a less relaxed position. Another problem with the screen was the user had to focus a lot on the timer ticking down, which also might influence the brain signals.

Regression Magnitude Calculation

At the moment, the actual speed of our regression method, use the mean over an entire trial. This can be an issue, if the test subject do not move the hand at a stable speed throughout the hand movement. A solution to this can be to use sensor data, which is done by attaching an accelerometer to the test subject's hand and record the acceleration of the hand at each sample. There is, of course, some problems with this solution, namely the problem of synchronizing the sensor gathering to the sampling rate of our EEG reader and the making sure that there is no delay such that the samples of the sensor correspond to the exact sample of the EEG reader.

Time intervals of training setup

Another factor that might affect the regression results is the time intervals we use when gathering test data. By studying the results it seems like the difference between the signal of a fast hand movement and a slow hand movement is too similar. By trial and error we found the slowest realistic movement to be 2 seconds, it could be increased but the hand movement would feel awkward.

BIBLIOGRAPHY

[1] *10/20 Positioning Manual, author =.*

[2] `http://mathworld.wolfram.com/PositiveDefiniteMatrix.html`, 2015. Visited: 16-4-2015.

[3] `https://code.google.com/p/matlabcontrol/`, 2015. Visited: 11-5-2015.

[4] A. Barachant, S. Bonnet, M. Congedo, and C. Jutten. Multiclass brain–computer interface classification by riemannian geometry. *Biomedical Engineering, IEEE Transactions on*.

[5] J. Baztarrica Ochoa. EEG Signal Classification for Brain Computer Interface Applications, 2002.

[6] Berlin BCI. BCI Competition IV. `http://bbci.de/competition/iv/`. Visited: 04-5-2015.

[7] Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 289–300, 1995.

[8] B. Blankertz, R. Tomioka, S. Lemm, M. Kawanabe, and K.-R. Muller. Optimizing spatial filters for robust eeg single-trial analysis. *Signal Processing Magazine, IEEE*, 25(1):41–56, 2008.

[9] G. Dallas. `https://georgemdallas.wordpress.com/2013/10/30/principal-component-analysis-4-dummies-eigenvectors-eigenvalues-and-dimension-reduction/`, 2013. Visited: 06-6-2015.

[10] G. Dougherty. *Pattern Recognition and Classification an Introduction*. Springer, New York, 1st edition edition, 2013.

[11] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[12] Inc. Emotiv. Emotiv EPOC EEG Reader. `https://emotiv.com/epoc.php`. Visited: 18-03-2015.

[13] P. Thomas Fletcher and S. Joshi. Principal Geodesic Analysis on Symmetric Spaces: Statistics of Diffusion Tensors. In M. Sonka, I. Kakadiaris, and J. Kybic, editors, *Computer Vision and Mathematical Methods in Medical and Biomedical Image Analysis*, volume 3117 of *Lecture Notes in Computer Science*, pages 87–98. Springer Berlin Heidelberg, 2004.

[14] T. Fletcher. Geodesic Regression on Riemannian Manifolds. In Xavier Pennec, Sarang Joshi, and Mads Nielsen, editors, *Proceedings of the Third International Workshop on Mathematical Foundations of Computational Anatomy - Geometrical and Statistical Methods for Modelling Biological Shape Variability*, pages 75–86, Toronto, Canada, September 2011.

[15] K. Fukunaga. *Introduction to Statistical Pattern Recognition*. Academic Press, San Diego, 2nd edition edition, 1990.

[16] A. Jensen, D. Bøcker Sørensen, and N. Lund Hasager Kirk. Developing a BCI System. June 2014. Available from http://projekter.aau.dk/projekter/da/.

[17] M. Moakher. A Differential Geometric Approach to the Geometric Mean of Symmetric Positive-Definite Matrices. *SIAM J. Matrix Anal. Appl.*, 26(3):735–747, March 2005.

[18] Pbchem. Fast Fourier Transform Image. `http://commons.wikimedia.org/wiki/File:Time_domain_to_frequency_domain.jpg`. [Online; accessed 30-October-2014].

[19] Ph.D Robert P. Lehr Jr. Brain Function. `http://www.neuroskills.com/brain-injury/brain-function.php`. [Online; accessed 30-October-2014].

[20] Howard J Seltman. Experimental design and analysis. *Online at: http://www.stat. cmu. edu/, hseltman/309/Book/Book. pdf*, 2012.

[21] P. Shirkey. Butterworth Filters in C#. `http://www.centerspace.net/blog/tag/c-butterworth-filter/`. Visited: 16-5-2015.

[22] Robert Tibshirani. Regression Shrinkage and Selection Via the Lasso. *Journal of the Royal Statistical Society, Series B*, 58:267–288, 1994.

[23] O. Tuzel, F. Porikli, and P. Meer. Pedestrian detection via classification on Riemannian manifolds. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(10):1713–1727, Oct 2008.

[24] C. Van Loan. *Computational Frameworks for the Fast Fourier Transform*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

APPENDIX A

EMOTIV EPOC EEG READER SPECIFICATIONS

Emotiv EPOC

	EEG HEADSET
Number of channels	14 (plus CMS/DRL references, P3/P4 locations)
Channel names (International 10-20 locations)	AF3, F7, F3, FC5, T7, P7, O1, O2, P8, T8, FC6, F4, F8, AF4
Sampling method	Sequential sampling. Single ADC
Sampling rate	128 SPS (2048 Hz internal)
Resolution	14 bits 1 LSB = 0.51µV (16 bit ADC, 2 bits instrumental noise floor discarded)
Bandwidth	0.2 - 45Hz, digital notch filters at 50Hz and 60Hz
Filtering	Built in digital 5th order Sinc filter
Dynamic range (input referred)	8400µV (pp)
Coupling mode	AC coupled
Connectivity	Proprietary wireless, 2.4GHz band
Power	LiPoly
Battery life (typical)	12 hours
Impedance Measurement	Real-time contact quality using patented system