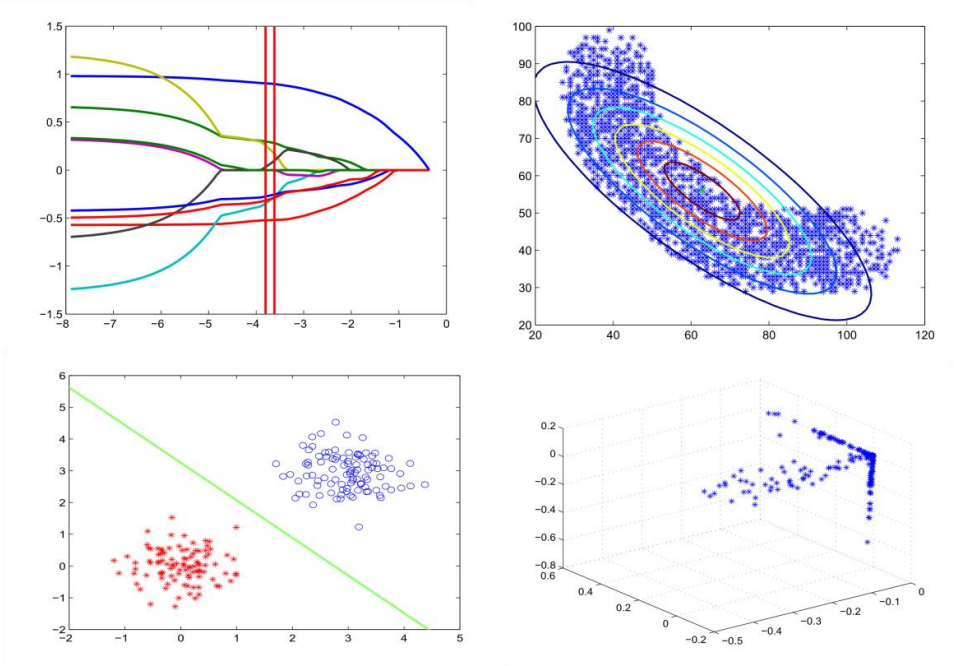
UNIVERSITY OF TROMSØ UIT

FACULTY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF PHYSICS AND TECHNOLOGY

Semi-Supervised Classification using Kernel Entropy Component Analysis and the LASSO



Jonas Nordhaug Myhre

FYS-3921 Master's thesis in Electrical Engineering

December 2011

Abstract

In this thesis we present a new semi-supervised classification technique based on the Kernel Entropy Component Analysis (KECA) transformation and the least absolute shrinkage selection operator (LASSO). The latter is a constrained version of the least squares classifier.

Traditional supervised classification techniques only use a limited set of labeled data to train the classifier, thus leaving a large part of the data practically unused. If we have very little training data available it is obvious that the classifier will have problems generalizing well as too few points will not fully represent the classes we are training the classifier to separate. So creating semi-supervised classifiers that somehow includes information from unlabeled data is a natural extension. This is further confirmed by the fact that labeling of data is very often a boring and time consuming task that can only be done by a few experts on the field in question rather than general pattern recognition experts, while unlabeled data are often abundant and no experts are needed.

One way of taking advantage of unlabeled data, which is the one we will use in this thesis, is to first transform the data to a new space using all data, both labeled and unlabeled, and in this new space use the labeled points to create a classifier. The idea is that when we include all the unlabeled data in the transformation the, often scarcely populated, labeled data set will represent the data better than without the unlabeled points. Previous work have shown very good results using the data transformations Laplacian eigenmaps with ordinary least squares and Data Spectroscopy with the LASSO.

We transform the data with a new method developed at the University of Tromsø called Kernel Entropy Component Analysis and combine it with LASSO classification. Previous work using ordinary least squares and KECA

have shown good results. This transformation preserves entropy components from the input data which has been showed can give much better representation of data than the otherwise almost exclusively used variance measure.

Through different experiments we show that this semi-supervised classifier in most cases performs comparable to or better than the previous results using other data transformations. We also show that the LASSO has an almost exclusively positive effect on classification after data transformation. A deeper analysis of how the LASSO classifier works together with the Kernel Entropy Component Analysis transformation is included, and we compare the results to the closely related Kernel Principal Component Analysis transformation. Finally we test the new classifier on a data set consisting of different facial expression showing that including unlabeled data leads to much better classification result than with straight forward least squares or LASSO classification.

Acknowledgments

First and foremost I would like to thank my supervisor Robert Jenssen. Throughout the process of writing this thesis his expertise, ideas and seemingly endless enthusiasm for the field of machine learning has been extremely contagious and inspiring for me.

I would also like to send a big thank you to my wife Kjærsti who has been extremely patient with me during this process. She even managed to get pregnant amidst all of this, giving me an even bigger incentive to finish in time!

My family and friends also deserves gratitude for supporting me and helping think about other things when I need a break.

As far as my office partner through the last years, Helge, goes, I really don't know what to say. A combination of fruitful discussions and extreme physical pain has accompanied my work in this thesis, so I will let the reader decide whether his influence has been good or bad.

Finally I would like to thank my newly born son and my two cats for being who they are.

Contents

Chapter 1

Introduction

The world today is at the pinnacle of the information age, and we are surrounded by enormous amounts of data. Sources as different as the Internet, new advanced high resolution DNA measurement techniques, gigantic digital text libraries, hyperspectral image sensors and many others provide us with far more data than we are able to process. To cope with this almost uncontrollable amount of data, pattern recognition and machine learning methods are extremely important today.

The norm in machine learning is to operate in either of two directions: *supervised* and *unsupervised* learning [30]. In this thesis we will work in between these different settings and explore the *semi-supervised learning* scheme where the two frameworks are combined with an intention of creating better classifiers by utilizing information available in the data sets normally only used for testing.

In practice labeled data, i.e. data that belongs to a certain class[1], are often scarce as they usually have to be determined by an expert on the field in question rather than the actual person designing the learning algorithm[2]. E.g. doctors examining medical data, geologists defining specific mineral types and so on. Because of this it is an enormous advantage if a learning algo-

[1]E.g. a blood sample that we know indicates a certain type of cancer. An *unlabeled* data point would be a blood sample that we don't know if indicates cancer or not

[2]Today it is not uncommon that statisticians and other pure data analysis experts work with scientific data rather than the actual person who collects the data.

rithm can be trained using both a few known examples and a large quantity of unlabeled examples.

There are two main ideas that motivate taking advantage of unlabeled data in practice: *the cluster assumption* or *low-density separation assumption* and *the manifold assumption* [27], [6]. The cluster assumption, as the name implies, assumes that there exists some kind of natural clusters in the data and that these clusters or classes are separated by low density regions. The manifold assumption expects the data to be suspended along some kind of manifold[3] hidden in the input space, so that with a proper distance measure points that lie on the manifold should be close, e.g. in the same class.

In the traditional binary classification setting we have two classes of data we want to separate, a set of points we know the classes of and an unknown test set. We train the classifier using the test points and use it to classify the test points. Assume that we have 20 labeled test points, 10 from each class. When we think of the cluster- or manifold assumption it seems obvious that only 10 points from each class would probably not represent the given cluster or manifold very well. However if we now somehow could include all the unlabeled test points in the classifier to yield a better representation of the clusters/manifolds corresponding to the different classes we would expect a better result.

This is where the idea stops and the practical implementations start: We have chosen to use the setting presented in [4], [28] and [16]: If we first transform all of our data, *both labeled and unlabeled*, to a new space and then train a classifier in the new space based on our labeled data we have a way of taking advantage of the normally unused unlabeled data. The idea is that when included in the data transformation the unlabeled data helps describe the overall structure of the data. In this way a traditionally small training set will become a much better representative for the entire data distribution yielding a better and more general classifier.

In this thesis we will investigate how the *Kernel Entropy Component Analysis transformation*, [15], developed at the University of Tromsø, behaves in this version of the semi supervised scheme. Previous work has been done using the ordinary least squares classifier [16]. We introduce a new version using the least squares classifier called the LASSO, [31], which has some advantageous

[3] In practice any kind of smooth geometric hypersurface [30].

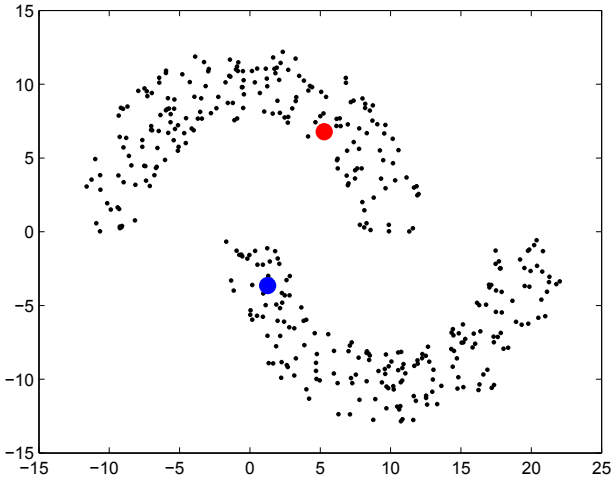


Figure 1.1: A toy data set, often referred to as the 'two moons' data set. The points marked with red and blue are labeled, the rest are unlabeled data.

properties and has shown good results with other dimensionality reducing transformations similar to the Kernel Entropy Component Analysis transformation, [4], [28]. There also exist many other schemes of semi-supervised learning which we will not discuss, but a survey of the most used methods can be found in [33].

To conclude this introduction we include a simple toy example to illustrate how a generic semi supervised way of thought could work in practice. In Figure 1.1 we see a set of data where we only have one labeled point from each class marked with red and blue.

Now if we only work with the two labeled points we would probably create a simple linear classifier as seen in Figure 1.2.

It is evident that when we know how the rest of the data is distributed, this would not give a particularly well result. On the other hand if we actually use the unlabeled data to help us look at the structure of the data we could create a classifier more like we see in Figure 1.3.

When looking at the two figures it is quite obvious that the latter, Figure 1.3, would give a much better classification result. We see that this particular

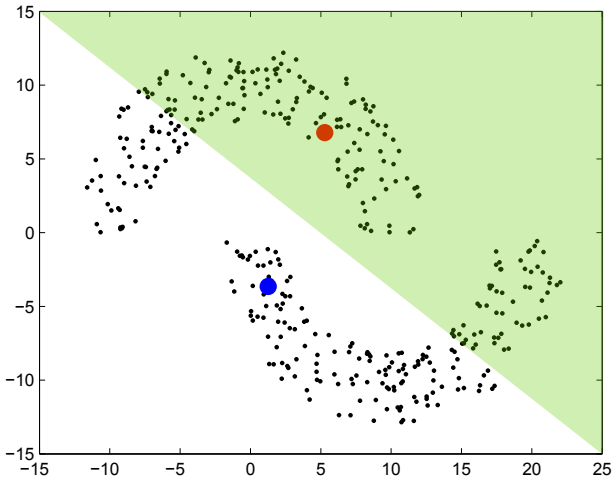


Figure 1.2: The two moons data set with a 'reasonable' classifier if we only take the two known points into account.

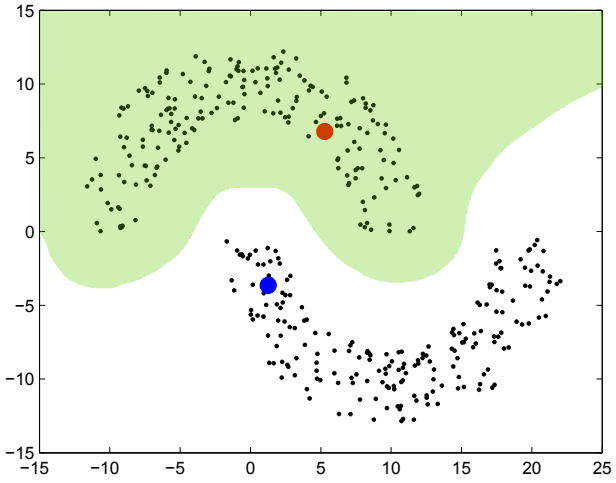


Figure 1.3: The two moons data set with a reasonable classifier if we make use of the unlabeled data in addition to the two known points.

data set can fulfill both the cluster and the manifold assumptions: the cluster assumption is reasonable as there clearly is a low density region between the two moons[4], and the manifold assumption can be related to the fact that the nonlinear moon, or almost banana-like, shapes can be interpreted as two manifolds which each class lies on. Actually in practice the two assumptions often implies the same meaning as classes that lie on different manifolds will have low density regions between them indirectly fulfilling the cluster assumption [6].

1.1 Structure of Thesis

We divide this thesis in two parts, Theory and Experiments. In the Theory part we derive, discuss and illustrate the Kernel Entropy Component Analysis transformation in Chapter 4. We also introduce the Data Spectroscopy algorithm and the Laplacian eigenmaps in Chapter 5 which we will use as comparisons in the Experiments part. Upon deriving the KECA transformation we introduce several important subjects: non-parametric density estimation in Section 4.2 and Information theoretic learning in Section 4.1 as well as the closely related non-centered Kernel Principal Component Analysis transformation and the baseline transformation Principal Component Analysis, in Chapter 3 and Chapter 2 respectively.

One important factor the reader should note is that we in the Theory section only focus on data transformations and their properties. The semi-supervised setting and related subjects are put on hold until the experiment section where the concept of classification is introduced and we tie it up with the theory to introduce the semi supervised learning algorithms we will use.

The Experiments part starts with introducing the concept of classification and the classifiers we will use, least squares and the LASSO in Chapter 7. We continue by explaining the practical considerations we have taken to perform different experiments in MATLAB.

Lastly we present and discuss the results in the Chapter 9 followed by a

[4]We of course notice that the region between the two moons is actually a 'no-density region' as there are no data points there, but we think it illustrates the point in a good fashion.

conclusion and ideas for further work in Chapter 10.

Part I

Theory

Chapter 2

Principal Component Analysis

Principal component analysis (PCA) is perhaps the most well known statistical dimensionality reducing data transformation, and it has been used extensively. The goal of PCA is to remove redundancies in a data set by generating decorrelated features, or dimensions [30][1]. In the derivation of the PCA it will become obvious that the transformation also can be used to reduce the dimension of the input data while still keeping as much of the variance, or energy, of the data as possible.

Let $\mathbf{x} = \{x_1, x_2, \cdots, x_N\}$ be a stochastic variable. Then we define the transformation

$$\mathbf{y} = \mathbf{A}^T\mathbf{x} \tag{2.1}$$

where the transformed data $\mathbf{y} = \{y_1, y_2, \cdots, y_N\}$ is also a stochastic variable and $\mathbf{A}$ is a transformation matrix.

As mentioned the goal is to generate uncorrelated data, and therefore it is natural to look at the correlation matrices of both the input data $\mathbf{x}$ and the resulting output $\mathbf{y}$ [2].

[1]The terms *feature* and *dimension*, when expressed in this context, is used interchangeably in the literature used in this thesis, and is roughly speaking just the number of measurements in an experiment.

[2]In this derivation we assume centered data, that is $E\{\mathbf{x}\} = 0$, and therefore look at the correlation matrix and not the covariance matrix. The non-centered data derivation

The correlation matrix of the input data is given as

$$\mathbf{R_{xx}} = E\{\mathbf{x}\mathbf{x}^T\}. \tag{2.2}$$

The correlation matrix of the output is given as

$$\mathbf{R_{yy}} = E\{\mathbf{y}\mathbf{y}^T\} = E\{\mathbf{A}^T\mathbf{x}(\mathbf{A}^T\mathbf{x})^T\} = \mathbf{A}^T E\{\mathbf{x}\mathbf{x}^T\}\mathbf{A} = \mathbf{A}^T\mathbf{R_{xx}}\mathbf{A}. \tag{2.3}$$

The next step is expressing $\mathbf{R_{xx}}$ as the eigenvalue problem

$$\mathbf{R_{xx}}\mathbf{E} = \mathbf{E}\mathbf{\Lambda} \tag{2.4}$$

where $\mathbf{E}$ is the matrix consisting of the eigenvectors of $\mathbf{R_{xx}}$ as columns, i.e. $\mathbf{E} = [\mathbf{e}_1\ \mathbf{e}_2\ \cdots\ \mathbf{e}_N]$, and $\mathbf{\Lambda}$ is a diagonal matrix with the eigenvalues of $\mathbf{R_{xx}}$ as its diagonal elements.

Rewriting (2.4) yields[3]

$$\mathbf{R_{xx}} = \mathbf{E}\mathbf{\Lambda}\mathbf{E}^{-1} = \mathbf{E}\mathbf{\Lambda}\mathbf{E}^T. \tag{2.5}$$

Using this relation in (2.3) gives the following

$$\mathbf{R_{yy}} = \mathbf{A}^T\mathbf{E}\mathbf{\Lambda}\mathbf{E}^T\mathbf{A}. \tag{2.6}$$

If then the transformation matrix $\mathbf{A}$ is chosen as $\mathbf{E}$ we get

$$\mathbf{R_{yy}} = \mathbf{E}^T\mathbf{E}\mathbf{\Lambda}\mathbf{E}^T\mathbf{E} = \mathbf{\Lambda} \tag{2.7}$$

and since we know that $\mathbf{\Lambda}$ is diagonal we then reach the goal of uncorrelated dimensions in the output $\mathbf{y}$.

---

is a trivial extension.

[3]Here we use the fact that the correlation matrix is symmetric, and therefore has orthogonal eigenvectors, which gives $\mathbf{E}^{-1} = \mathbf{E}^T$.

2.1 Dimensionality reducing property

Now we go a little further and notice that the transformation is a projection of $\mathbf{x}$ onto the space spanned by the eigenvectors of $\mathbf{R_{xx}}$ given by

$$\mathbf{y} = \mathbf{E}^T\mathbf{x} = \begin{bmatrix} \mathbf{e}_1^T \\ \mathbf{e}_2^T \\ \vdots \\ \mathbf{e}_N^T \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} \mathbf{e}_1^T\mathbf{x} \\ \mathbf{e}_2^T\mathbf{x} \\ \vdots \\ \mathbf{e}_N^T\mathbf{x} \end{bmatrix} \tag{2.8}$$

And if we simply use a subset of the eigenvectors to do the transformation it is obvious that we get a dimensionality reduction as the choice of eigenvectors directly affect the dimension of y.

The problem now is what eigenvectors to choose. If we choose some subset of the eigenvectors and look at the mean square error (MSE) of the reconstruction of $\mathbf{x}$, given by $\mathbf{x} = \mathbf{A}\mathbf{y}$ [30], we get the following.

Let $\hat{\mathbf{x}}$ be the projection using only M of the N eigenvectors of $\mathbf{R_{xx}}$ given by

$$\hat{\mathbf{x}} = \mathbf{A}_M\mathbf{y} = \mathbf{E}_M\mathbf{y} = \sum_{i=1}^{M} y_i\mathbf{e}_i, \tag{2.9}$$

where $\mathbf{A}_M$ and $\mathbf{E}_M$ denotes the matrices containing the chosen M eigenvectors.

Then the MSE between the two vectors is

$$\begin{aligned} E\left\{\|\mathbf{x}-\hat{\mathbf{x}}\|^2\right\} &= E\left\{\|\sum_{i=1}^{N} y_i\mathbf{e}_i - \sum_{i=1}^{M} y_i\mathbf{e}_i\|^2\right\} = E\left\{\|\sum_{i=M+1}^{N} y_i\mathbf{e}_i\|^2\right\} \\ &= E\left\{\sum_i\sum_j (y_i\mathbf{e}_i)^T(y_i\mathbf{e}_i)\right\} = E\left\{\sum_i\sum_j y_i^2\mathbf{e}_i^T\mathbf{e}_i\right\} \end{aligned} \tag{2.10}$$

and since we know that the eigenvectors $\mathbf{e}_i$ are orthogonal, i.e. $\mathbf{e}_i^T\mathbf{e}_i = 1$ and $\mathbf{e}_i^T\mathbf{e}_j = 0$, and that $y_i = \mathbf{e}_i^T\mathbf{x}$ we get

$$
\begin{aligned}
E\left\{\|\mathbf{x}-\hat{\mathbf{x}}\|^2\right\} &= E\left\{\sum_{i=M+1}^{N} y_i^2\right\} = E\left\{\sum_{i=M+1}^{N} (\mathbf{e}_i^T\mathbf{x})^T(\mathbf{e}_i^T\mathbf{x})\right\} \\
&= \sum_{i=M+1}^{N} E\left\{\mathbf{x}^T\mathbf{e}_i\mathbf{e}_i^T\mathbf{x}^T\right\} = \sum_{i=M+1}^{N} E\left\{\mathbf{e}_i^T\mathbf{x}\mathbf{x}^T\mathbf{e}_i\right\} \qquad (2.11) \\
&= \sum_{i=M+1}^{N} \mathbf{e}_i E\left\{\mathbf{x}\mathbf{x}^T\right\}\mathbf{e}_i^T.
\end{aligned}
$$

Finally using the orthogonality properties of $\mathbf{E}$ and (2.5) we see that we are left with

$$
\begin{aligned}
E\left\{\|\mathbf{x}-\hat{\mathbf{x}}\|^2\right\} &= \sum_{i=M+1}^{N} \mathbf{e}_i\mathbf{E}\mathbf{\Lambda}\mathbf{E}^T\mathbf{e}_i^T \\
&= \sum_{i=M+1}^{N} \lambda_i.
\end{aligned} \qquad (2.12)
$$

And from this we easily see that if we chose the eigenvectors corresponding to the top M eigenvalues in (2.9) the mean square error will be minimized as it will consist of the sum of the N-M smallest eigenvalues.

Now we will show that this choice of eigenvectors will keep as much variance as possible for the given dimensionality reduction.

Since we have used centered data as a starting point we can use the same derivations for the variance of the dimensionality reduced data as for the MSE.

So from (2.11) and (2.12) we have that [4]

$$
var(y_i) = E\{y_i^2\} = \lambda_i \qquad (2.13)
$$

and using the fact that $y_i = \mathbf{e}_i^T\mathbf{x}$, we see that when using the eigenvectors belonging to the top M eigenvalues to do the transformation the variance will be the maximum it can be for the chosen dimensionality reduction.

[4] We still assume centered data so the formula for the variance is simplified

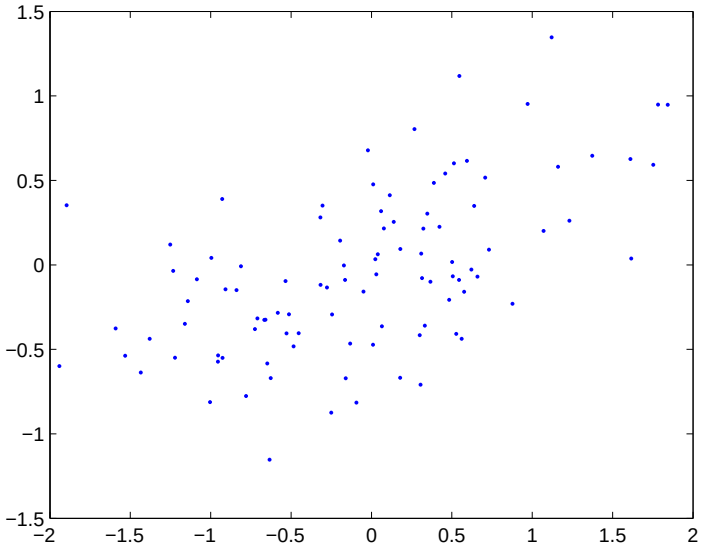
2.2 Examples

To show how PCA works in practice and illustrate the associated problems, we start with some basic examples using toy-data generated in Matlab. We start with a simple Gaussian distributed 'blob' of data to get a visual intuition of how the PCA works.

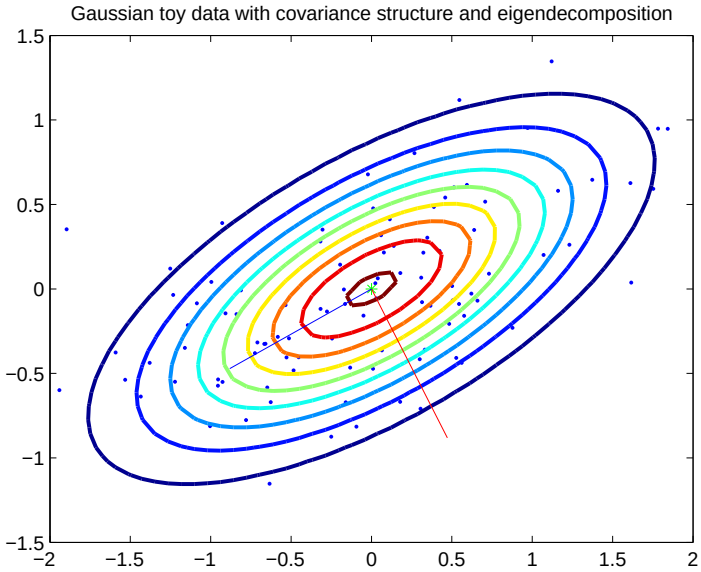
In Figure 2.1(a) we see the Gaussian data and in Figure 2.1(b) we see the covariance structure of the data as well as the two eigenvectors of the covariance matrix. What the PCA does is that it finds the eigenvectors that has the largest eigenvalues, and then project the data onto that axis, in our example that would be the blue vector in Figure 2.1(b). This we can see directly from the data as the variance is clearly largest in that direction, i.e the radius of the elliptical covariance structure is largest in that direction.

In order to investigate what happens when we do the actual projection we project the data to one dimension using the largest eigenvalue of the covariance matrix in Figure 2.1 estimated in matlab. The result can be seen in Figure 2.2.

Now that we have seen how the PCA works in the simplest of cases, it is time to discuss some of the related problems, which we will see form all the major motivations as to why the rest of the methods reviewed in this thesis were created.



(a) Gaussian 'blob' of data.



(b) Covariance structure and eigenvectors.

Figure 2.1: Gaussian toy data showing covariance structure and the eigenvectors spanning the covariance space

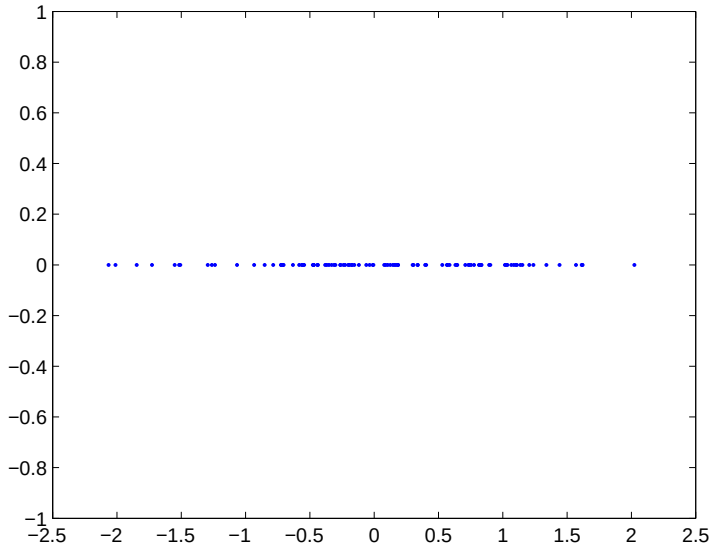


Figure 2.2: Data set from Figure 2.1 projected to first principal component

2.3 The issues of Principal Component Analysis

Even though Principal Component Analysis has been used widely, we notice some serious restrictions.

- The foundation of the PCA is covariance, the second order central moment which in practice is only a descriptor of the spread of the data, see [29] for details. Description of a density using covariance is only optimal for the Gaussian density as it is the only distribution with all moments of order greater than two equal to zero. So from this one can conclude that if a distribution differs greatly from the Gaussian shape, the covariance is a poor estimate and the PCA will perform accordingly.
- In effect the covariance measure is an elliptical measure[5] and thus if we have data with nonlinear structure the covariance has no way of

[5] A hyperellipsoide in dimensions >3.

capturing it and hence it will be lost in a PCA dimensionality reduction.

With these restrictions and illustrations in mind we have a clear picture of what the PCA does and what we need to investigate further to fix the inherent problems we have proposed. In Chapter 3 we look at *kernel methods* which can deal with nonlinearities, and in Chapter 4 we look at *information theoretic* measures as an alternative to the limited variance descriptor.

Chapter 3

Kernel Principal Component Analysis

Kernel principal component analysis (KPCA) is a data transformation which deals with the nonlinearity problems of PCA based on th so-called *kernel methods*. We start by a quick introduction of these methods in general.

3.1 Kernel methods

Kernel methods have been fundamental in the development of data analysis methods that solve nonlinear problems. The most well known example is the *support vector machine* [30], but numerous other examples exist such as, ranking, clustering and regression [24].

When we talk about kernel methods we are really talking about a framework of different algorithms where two steps are involved:

- A nonlinear transformation, sometimes reffered to as mapping, to a new space called the *feature space*.
- The execution of some linear algorithm in the new space.

The idea is that we transform the nonlinear input data set to a new higher dimensional space where the data is linearly separable, Cover's theorem states that the probability of reaching linear separability increases to one as the

dimension of the feature space goes to infinity [10], and then we can use any method from the well established world of linear filters and algorithms that suits the given problem, e.g. classification, clustering, etc. Unfortunately this presents us with some problems, the nonlinear mapping to the feature space is not at all a trivial matter, and in theory the feature space can have infinite dimension. It is in the consideration of these problems we introduce the *kernel functions*, which lets us do the nonlinear mapping *to* and operations *in* the feature space in an elegant way. Before we define the kernel function, we quickly formalize the transformation from the input space to the feature space.

If we assume that the input space consists of some $\mathbf{x}_i \in \mathbb{R}^l$ in the set X we denote the nonlinear mapping as

$$\begin{aligned} \Phi : X &\to F \\ \mathbf{x} &\to \Phi(\mathbf{x}) \end{aligned} \tag{3.1}$$

where $F \subseteq \mathbb{R}^M$, $M \geq l$.

3.1.1 Kernel functions

A kernel function is a function κ that satisfies

$$k(\mathbf{x}_i, \mathbf{x}_j) = \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle \tag{3.2}$$

for all $\mathbf{x}_i, \mathbf{x}_j \in X$.

In short this is what is known as the *kernel trick*. As we see in (3.2) it lets us calculate inner products in a possibly infinite dimensional feature space directly without having to deal with the explicit data mapping. This means that any *linear machine learning algorithm that can be expressed via inner products* now can solve nonlinear problems by operating in a high dimensional feature space.

The mathematical foundation of kernel functions is deep and beyond the scope of this paper, but we introduce two key elements: the *Moore-Aronszajn* theorem and *Mercer's* theorem [2], [18], [24].

Theorem 3.1.1 (Moore-Aronszajn's)[21] *For any positive semidefinite function $\kappa(x, y)$ there exists a uniquely determined (Hilbert[1]) space, H, which consists of functions of the input set X given by*

$$\begin{aligned} &1.\ \forall\, \mathbf{x}, \quad \kappa(\cdot, \mathbf{x}) \in H \\ &2.\ \forall\, \mathbf{x}, \quad \forall f \in H, \quad f(\mathbf{x}) = \langle f, \kappa(\cdot, \mathbf{x}) \rangle_{H_\kappa} \end{aligned} \tag{3.3}$$

In the first equation we directly see the mapping to the feature space via the kernel function. From the second equation we can directly derive the kernel trick, [21] by

$$f(\mathbf{x}) = \Phi(\mathbf{x}) = \kappa(\cdot, \mathbf{x}) \tag{3.4}$$

giving

$$\langle \Phi(\mathbf{x}), \Phi(\mathbf{x}')) \rangle = \langle \kappa(\cdot, \mathbf{x}), \kappa(\cdot, \mathbf{x}') \rangle = \kappa(\mathbf{x}, \mathbf{x}'). \tag{3.5}$$

The last property of (3.3) is referred to as the *reproducing* property as it reproduces functions in the Hilbert space via the inner product with $\kappa(\cdot, \mathbf{x})$, because of this the space is limited by the choice of $\kappa(\cdot, \cdot)$ which is the reason for the denotation with the H_κ.

The Mercer theorem simply secures the existence of the feature space.

Theorem 3.1.2 (Mercer's)[30] *Let $\kappa(\mathbf{x}, \mathbf{z})$ be given. Then κ is a valid Mercer kernel, i.e $\exists\, \Phi$ s.t $\kappa(\mathbf{x}, \mathbf{z}) = \Phi(\mathbf{x})^T \Phi(\mathbf{z})$, if and only if for all $\{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_l\}$ the kernel matrix $\mathbf{K}$ is symmetric positive semidefinite.*

Naturally there exist numerous choices of actual kernel functions, but we will just introduce the most commonly used function, namely the Gaussian kernel. Other kernels can be found in [21],[24] or [30]. We adopt the form of the Gaussian kernel in [24] which is defined as follows,

$$\kappa_\sigma(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{||\mathbf{x} - \mathbf{y}||^2}{2\sigma^2} \right), \tag{3.6}$$

where σ is known as the *kernel size*. This choice of kernel actually represents inner products in an infinite dimensional feature space, see [24] or [21] for details.

[1] A Hilbert space is a, possibly infinite dimensional, complete inner product space . A complete vector space is a space where a Cauchy sequence is guaranteed convergence.

Lastly we will introduce the *kernel matrix* $\mathbf{K}$, which contains all the evaluations of the kernel function κ on the input data e.g the input data in 3.1. From the kernel trick we know that this matrix also contains all evaluations of inner products between the data points in the feature space. It is given by

$$\mathbf{K}_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_j) = \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle \tag{3.7}$$

or in matrix form

$$\begin{aligned} \mathbf{K} &= \begin{bmatrix} \kappa(\mathbf{x}_1, \mathbf{x}_1) & \kappa(\mathbf{x}_1, \mathbf{x}_2) & \cdots & \kappa(\mathbf{x}_1, \mathbf{x}_n) \\ \kappa(\mathbf{x}_2, \mathbf{x}_1) & \kappa(\mathbf{x}_2, \mathbf{x}_2) & \cdots & \kappa(\mathbf{x}_2, \mathbf{x}_n) \\ \vdots & \vdots & \ddots & \vdots \\ \kappa(\mathbf{x}_n, \mathbf{x}_1) & \kappa(\mathbf{x}_n, \mathbf{x}_2) & \cdots & \kappa(\mathbf{x}_n, \mathbf{x}_n) \end{bmatrix} \\ &= \begin{bmatrix} \langle \Phi(\mathbf{x}_1), \Phi(\mathbf{x}_1) \rangle & \langle \Phi(\mathbf{x}_1), \Phi(\mathbf{x}_2) \rangle & \cdots & \langle \Phi(\mathbf{x}_1), \Phi(\mathbf{x}_n) \rangle \\ \langle \Phi(\mathbf{x}_2), \Phi(\mathbf{x}_1) \rangle & \langle \Phi(\mathbf{x}_2), \Phi(\mathbf{x}_2) \rangle & \cdots & \langle \Phi(\mathbf{x}_2), \Phi(\mathbf{x}_n) \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \Phi(\mathbf{x}_n), \Phi(\mathbf{x}_1) \rangle & \langle \Phi(\mathbf{x}_n), \Phi(\mathbf{x}_2) \rangle & \cdots & \langle \Phi(\mathbf{x}_n), \Phi(\mathbf{x}_n) \rangle \end{bmatrix}, \end{aligned} \tag{3.8}$$

where n is the number of data points in the input space.

Finally we will list some of the most important properties of the kernel matrix:

- It is positive semidefinite[2].
- It is symmetric.
- It is a Gramian matrix, i.e a matrix of inner products.
- In a way it acts as a funnel[3], the only information we get to work with in the feature space are the inner products, i.e the elements in the kernel matrix are all the information we get [24]!

[2]The positive semidefinite property is a direct implication from Mercer's theorem
[3]This property is naturally also true for the kernel function.

3.2 The Kernel Principal Component Analysis transformation

Now that we have the basics of kernel methods in place, we can utilize the kernel trick to derive a nonlinear version of the principal component analysis in Chapter 2, namely the *Kernel Principal Component Analysis* [22]. As with PCA we want to project the data onto the principal axis of the covariance matrix of the data, but now we transform the data to the aforementioned feature space and in effect doing a PCA in that space.

The derivation of the KPCA is not as simple as the PCA as we have to do some algebra to be able to use the kernel trick making the transformation only dependent on inner products. Otherwise the main steps are similar to those of PCA.

We start by introducing the function Φ which maps the input data to a feature space F.

$$\begin{aligned} \Phi : X &\to F \\ \mathbf{x} &\to \Phi(\mathbf{x}). \end{aligned} \tag{3.9}$$

As with the linear PCA we assume centered data, $E\{\Phi(\mathbf{x})\} = 0$, and the estimated correlation matrix of the input data is given by

$$\mathbf{C} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}\mathbf{x}^T. \tag{3.10}$$

Expressing the correlation matrix in the feature space, $\mathbf{C}_F$, via the mapping Φ gives

$$\mathbf{C}_F = \frac{1}{N} \sum_{i=1}^{N} \Phi(\mathbf{x}_i)\Phi(\mathbf{x}_i)^T. \tag{3.11}$$

Now we follow a similar path as with the derivation of the linear PCA, and we start by defining the eigenvalue problem for the correlation matrix of the data in the feature space F. Remember that the goal is to find the *eigenvectors that*

span the covariance space of the data, in this case in the higher dimensional feature space. The only major difference in the derivations is that we have to tailor the kernel PCA so that it is an expression of inner products in the feature space so that we can utilize the kernel trick.

The eigenvalue problem expressed in the feature space:

$$\begin{aligned}\lambda \mathbf{v} = \mathbf{C}_F \mathbf{v} &= \frac{1}{N}\sum_{i=1}^{N} \Phi(\mathbf{x}_i)\Phi(\mathbf{x}_i)^T \mathbf{v} = \frac{1}{N}\sum_{i=1}^{N} \Phi(\mathbf{x}_i)\left\langle \Phi(\mathbf{x}_i), \mathbf{v}\right\rangle \\ &= \frac{1}{N}\sum_{i=1}^{N} \left\langle \mathbf{v}, \Phi(\mathbf{x}_i)\right\rangle \Phi(\mathbf{x}_i).\end{aligned} \tag{3.12}$$

Where $\langle \cdot, \cdot \rangle$ denotes the Euclidean inner product (dot product).

From this we see that the solutions of $\mathbf{v}$ lies in the span of the transformed input data, $\Phi(\mathbf{x_1}), \cdots, \Phi(\mathbf{x}_N)$, and it is possible to reformulate the eigenvalue problem as

$$\lambda \Phi(\mathbf{x}_k)^T \mathbf{v} = \Phi(\mathbf{x}_k)^T \mathbf{C}_F \mathbf{v} \quad \forall\, k = 1, \cdots, N. \tag{3.13}$$

Using this and the fact that $\mathbf{v}$ can be expressed as (from (3.12))

$$\mathbf{v} = \frac{1}{\lambda N}\sum_{i=1}^{N} \left\langle \Phi(\mathbf{x}_i), \mathbf{v}\right\rangle \Phi(\mathbf{x}_i) = \sum_{i=1}^{N} \alpha_i \Phi(\mathbf{x}_i), \tag{3.14}$$

since $\langle \Phi(\mathbf{x}_i), \mathbf{v} \rangle$ is just a scalar, we get(from (3.13))

$$\lambda \sum_{i=1}^{N} \alpha_i \Phi(\mathbf{x}_k)^T \Phi(\mathbf{x}_i) = \Phi(\mathbf{x}_k)^T \frac{1}{N}\sum_{i=1}^{N} \Phi(\mathbf{x}_i)\Phi(\mathbf{x}_i)^T \sum_{j=1}^{N} \alpha_j \Phi(\mathbf{x}_j). \tag{3.15}$$

To simplify and get a thorough understanding of (3.15), we rewrite the left and right hand sides separately.

The left hand side gives

$$\lambda \sum_{i=1}^{N} \alpha_i \Phi(\mathbf{x}_k)^T \Phi(\mathbf{x}_i) \quad \forall k, \tag{3.16}$$

$$\Updownarrow$$

$$\lambda \begin{bmatrix} \alpha_1 \Phi(\mathbf{x}_1)^T \Phi(\mathbf{x}_1) + \alpha_2 \Phi(\mathbf{x}_1)^T \Phi(\mathbf{x}_2) + \cdots + \alpha_m \Phi(\mathbf{x}_1)^T \Phi(\mathbf{x}_N) \\ \vdots \\ \alpha_1 \Phi(\mathbf{x}_N)^T \Phi(\mathbf{x}_1) + \alpha_1 \Phi(\mathbf{x}_N)^T \Phi(\mathbf{x}_2) + \cdots + \alpha_1 \Phi(\mathbf{x}_N)^T \Phi(\mathbf{x}_N) \end{bmatrix} = \lambda \mathbf{K} \boldsymbol{\alpha}.$$

Where $\boldsymbol{\alpha}$ is a vector containing the coefficients α, $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \cdots, \alpha_N]^T$, and $\mathbf{K}$ is the kernel matrix as stated in (3.7).

Looking at the right side we can rewrite it as follows

$$\begin{aligned} &\Phi(\mathbf{x}_k)^T \frac{1}{N} \sum_{i=1}^{N} \Phi(\mathbf{x}_i) \Phi(\mathbf{x}_i)^T \sum_{j=1}^{N} \alpha_j \Phi(\mathbf{x}_j) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_j \Phi(\mathbf{x}_k)^T \Phi(\mathbf{x}_i) \Phi(\mathbf{x}_i)^T \Phi(\mathbf{x}_j) \\ &= \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_j k(\mathbf{x}_k, \mathbf{x}_i) k(\mathbf{x}_i, \mathbf{x}_j) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{N} \mathbf{K}_{ki} \mathbf{K}_{ij} \alpha_j \quad \forall k = 1, \cdots, N. \end{aligned} \tag{3.17}$$

If we now look at the kernel matrix $\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ and the standard formula for matrix multiplication

$$\mathbf{K}_{kj}^2 = \sum_{i=1}^{M} \mathbf{K}_{ki} \mathbf{K}_{ij} \quad \forall k, j \tag{3.18}$$

and in addition look at the coefficients α and the sum over all j as a multiplication with the vector $\boldsymbol{\alpha}$, given by

$$\mathbf{K}\boldsymbol{\alpha} = \sum_{j=1}^{M} \mathbf{K}_{kj} \alpha_j \ \textit{ for each row } k, \tag{3.19}$$

we see that the right hand side expression of (3.15) simply becomes

$$\frac{1}{N}\mathbf{K}^2\boldsymbol{\alpha}. \tag{3.20}$$

Combining (3.20) and (3.17) we get the eigenvalue problem expressed via the kernel matrix

$$\lambda\mathbf{K}\boldsymbol{\alpha} = \frac{1}{N}\mathbf{K}^2\boldsymbol{\alpha}. \tag{3.21}$$

This is equivalent to

$$N\lambda\boldsymbol{\alpha} = \mathbf{K}\boldsymbol{\alpha} = \lambda_k\boldsymbol{\alpha}, \tag{3.22}$$

where $\lambda_k = N\lambda$ denotes an eigenvalue of the kernel matrix.

Now we have the coefficients α which can be used to find $\mathbf{v}$. The only thing left is to make sure that the norm of $\mathbf{v}$ is equal to one, as the eigenvectors are used to form a basis for the points in the feature space.

The squared length of $\mathbf{v}$ is given by

$$\begin{aligned}\mathbf{v}^T\mathbf{v} &= \sum_{i=1}^{N}\alpha_i\Phi(\mathbf{x}_i)^T\sum_{j=1}^{N}\alpha_j\Phi(\mathbf{x}_j) \\ &= \sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_i\alpha_j\Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}_j) \quad = \boldsymbol{\alpha}^T\mathbf{K}\boldsymbol{\alpha} = \lambda_k\boldsymbol{\alpha}^T\boldsymbol{\alpha} \ = \lambda_k.\end{aligned}$$

Now we can express the eigenvectors that is used to project the data onto the principal axes in the feature space:

$$\mathbf{v} = \frac{1}{\sqrt{\lambda_k}}\sum_{i=1}^{N}\alpha_i\Phi(\mathbf{x}_i). \tag{3.23}$$

The final thing we need to look at is the transformation of an arbitrary point $\Phi(\mathbf{x})$ in the feature space onto the k'th principal axis of the same space which is given by

$$\mathbf{v}_k^T\Phi(\mathbf{x}) = \frac{1}{\sqrt{\lambda_k}}\sum_{i=1}^{N}\alpha_{i,k}\Phi(\mathbf{x}_i)^T\Phi(\mathbf{x}) = \frac{1}{\sqrt{\lambda_k}}\sum_{i=1}^{N}\alpha_{i,k}\kappa(\mathbf{x}_i,\mathbf{x}). \tag{3.24}$$

And if we use only the input data we can rewrite it as follows.

For all input point $\mathbf{x}_j$, $j = 1, \cdots N$, the projection onto the k'th principal component is given as

$$\begin{aligned} \mathbf{v}_k^T \Phi(\mathbf{x}_j) &= \frac{1}{\sqrt{\lambda_k}} \sum_{i=1}^{N} \alpha_{i,k} \kappa(\mathbf{x}_i, \mathbf{x}_j) \\ &\overset{\forall j}{=} \frac{1}{\sqrt{\lambda_k}} \sum_{i=1}^{N} \alpha_{i,k} \mathbf{K} \\ &= \frac{1}{\sqrt{\lambda_k}} \boldsymbol{\alpha_k^T} \boldsymbol{\alpha_k} \lambda_k \boldsymbol{\alpha_k^T} \\ &= \sqrt{\lambda_k} \boldsymbol{\alpha}_k^T \end{aligned} \tag{3.25}$$

From (3.24) and (3.25) we see that we use the eigenvectors of the kernel matrix to do the actual transformation. The choice of eigenvector follows the idea of ordinary PCA, the eigenvectors with the highest belonging eigenvalues are chosen as we have seen that this minimizes the mean squared error and retains the most variance possible for the chosen dimensionality reduction.

To conclude this section we sum up the Kernel PCA transformation in a simple pseudo-code algorithm:

Algorithm 1 Kernel principal component analysis dimensionality reducing algorithm

Input: The complete set of data, $X = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N\}$

1: Calculate the kernel matrix, $\mathbf{K}$, with elements $\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$.
2: Calculate the eigenvectors and eigenvalues, $\boldsymbol{\alpha}$ and λ_k, of $\mathbf{K}$.

Output: Chose the eigenvectors corresponding to the d largest eigenvalues of $\mathbf{K}$ for a d dimensional output.

3.2.1 Examples

We illustrate the properties of the KPCA transformation with a toy example.

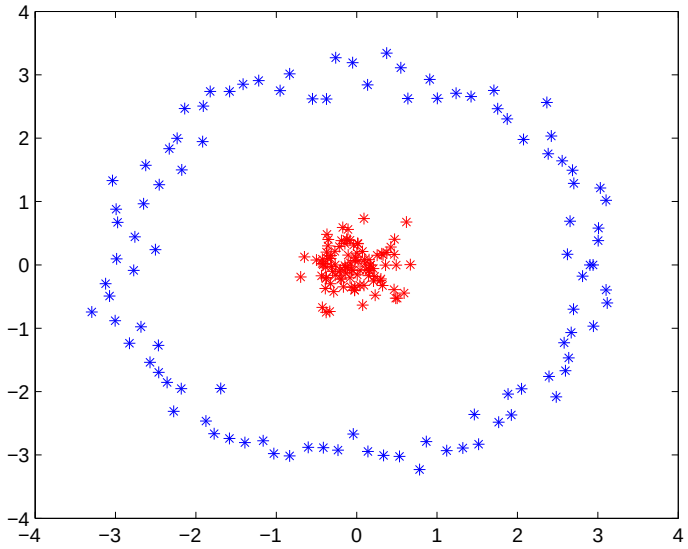


Figure 3.1: The toy data.

In Figure 3.1 we see a typical nonlinear toy data set. The colors are included so we can see how the data behave when transformed. If we perform ordinary PCA we get the result displayed in Figure 3.2 which is clearly undesired as the two structures, marked with red and blue, in the data are totally gone after the transformation. In Figure 3.3 and Figure 3.4 we see the dataset transformed to two and one dimensions respectively using kernel PCA. In both transformations we see that the data are now at least linearly separable[4], which is good for clustering or classification, and the nonlinear structure has been reduced significantly.

[4]Linearly separable means that they can be separated by a line, plane or hyperplane, depending on the dimension. See [30].

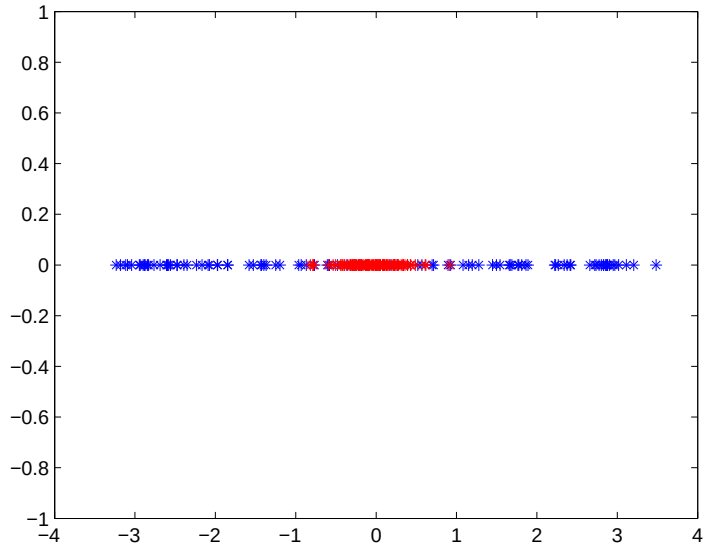


Figure 3.2: Ordinary PCA performed on the nonlinear data. Dimension reduced to 1.

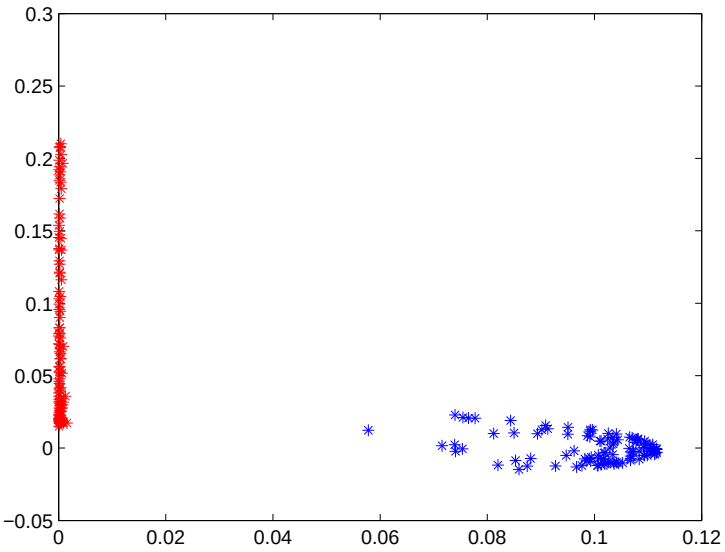


Figure 3.3: Example of Kernel PCA on the data in Figure 3.1.

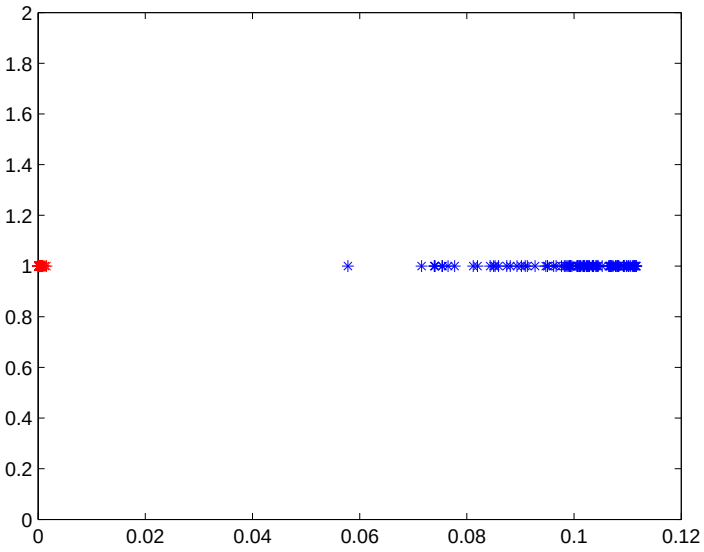


Figure 3.4: Kernel PCA with dimension reduced to 1.

Chapter 4

Kernel Entropy Component Analysis

Kernel Entropy component analysis is a data transformation that is very closely related to the kernel principal component analysis, but as the name implies, it focuses on *entropy components* instead of the principal, variance describing, components in PCA and KPCA. The term *entropy* stems from a branch of statistics and machine learning called *information theoretic learning*, so we start with introducing this framework.

4.1 Information theoretic learning

Information theoretic learning (ITL) is an approach to machine learning where *information theoretic* learning criteria, or cost functions as in many pattern recognition techniques, form the backbone instead of more classical approaches such as *mean square error*, minimizing *Euclidean distance* or similar. We will look at the basic concepts of this framework after a quick motivation.

4.1.1 Motivation

The main idea of using information theory in machine learning and pattern recognition is that the the widespread use of second order statistics, i.e. terms such as mean square error, variance, correlation, l_2 norm, etc, as a learning criterion, or cost function, is only optimal when assuming Gaussian or symmetric 'Gaussian-like' distributions.

As stated in [11], '*Learning in artificial neural networks and adaptive filters has used almost exclusively correlation (the l_2 norm or mean-square error) as a criterion to compare the information carried by the signals and the response of the learning machine'...*

In real-life applications there are many examples of problems which need a statistical foundation beyond the second order. For instance simple things like error densities with fat tails or severe outliers may cause problems if only second-order statistics are used [21]. In Figure 4.1 we see three distributions with equal variance and it is obvious that these three distributions behave quite differently even though they share the exact same variance. So for example if we have an optimization problem based on minimizing the mean square error, these three distribution will be treated as equals, which is not good.

In Figure 4.2 we see a toy data set with an estimate of the covariance structure plotted on top. Here we can see that the variance measure fails completely when trying to describe the dataset. This is of course a toy data set created for illustration only and it also has a nonlinear aspect which we will not comment here, but it clearly illustrates the problems of the second order statistical 'way of thought'.

It is worth mentioning that even though the apparent difficulties of the traditional 'Gaussian assumption - second order statistics' concept, the framework has been used, and still is, with great success. Erdogmus and Principe [7] lists some of the main reasons to why this has been so widely used:

- The success of linear filters combined with second-order statistics.
- A well established framework, with a wide arsenal of efficient algorithms.

So to conclude this section:

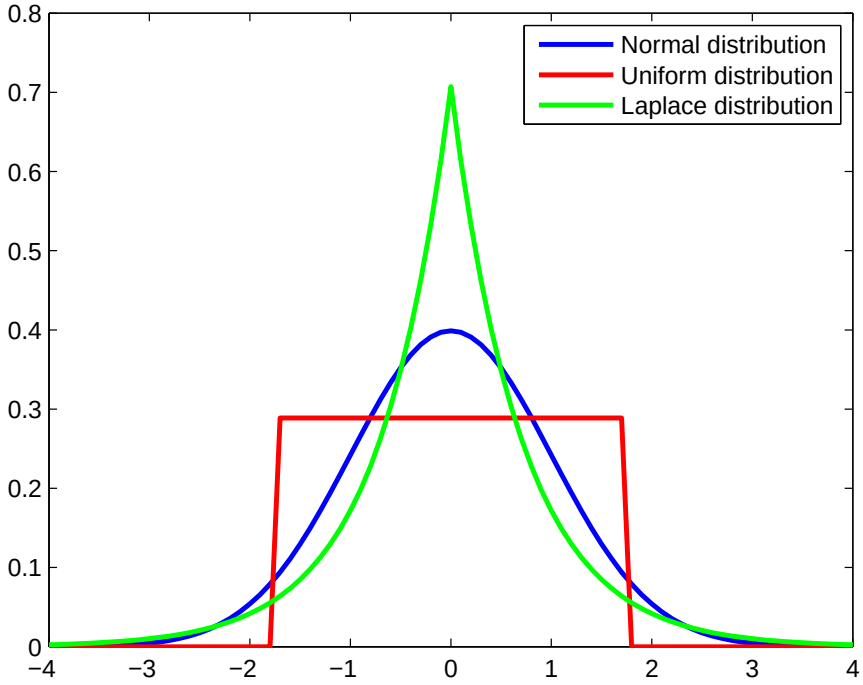


Figure 4.1: Three distributions, normal, uniform and Laplace, with variance equal to one.

Second order statistics can be used to solve many problems, but in a wide array of problems it simply is not enough. In the next section we will briefly go through the fundamental concepts of information theory relevant to ITL, and look at how it can be used in machine learning and pattern recognition.

4.1.2 Fundamentals of information theoretic learning

In ITL two fundamental statistical descriptors are used: *entropy* and *divergence*. In short terms these can be thought of as basic nonparametric measurements performed on probability density functions. Entropy can be thought of as the general uncertainty of a pdf, and divergence is simply a dissimilarity measure between two densities.

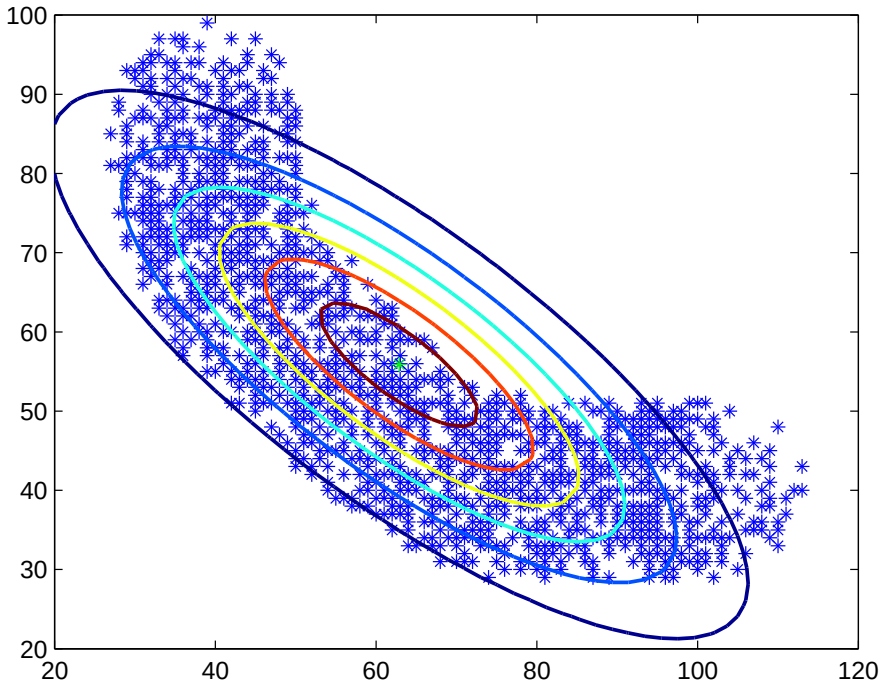


Figure 4.2: A toy data set

In this thesis we will not use the divergence quantity so we will leave it out and focus on entropy, often called the most important quantity in ITL [14].

Entropy

The concept of entropy in statistics was first introduced by Shannon as a measure of statistical uncertainty used in communication. He defined the entropy of a random variable X as the sum of the uncertainty, or information, in each message (a realization, $\mathbf{x}_k$, of X) weighted by the probability of each message

$$H_s(\mathbf{x}) = -\sum_k p(\mathbf{x}_k) \log p(\mathbf{x}_k), \tag{4.1}$$

or

$$H_s(\mathbf{x}) = -\int p(x)\log(x)dx, \tag{4.2}$$

where the quantity $-\log p(\mathbf{x}_k)$ or $\log \frac{1}{p(\mathbf{x}_k)}$ is called Hartley's information content of $\mathbf{x}_k$[1] [21].

A fundamental property of entropy is that it is a single scalar measuring the uncertainty in a probability density. It is also a very robust, or 'naturally balanced' as stated in [21], measure. An unlikely realization, i.e a low value of $p(\mathbf{x}_k)$, is weighted up by its high information content, $\log \frac{1}{p(\mathbf{x}_k)}$, and similarly a more probable realization is weighted down. This balance is really the essence of entropy as it makes it better suited to capture the true information in probability densities. For example a very skewed density may have almost the same entropy, or information content, as a totally symmetric distribution, something a variance measure would have serious problems describing.

The last thing we will mention is that the entropy measure has also been showed to be a good descriptor of the hypervolume spanned by a high dimensional probability density, making it usable in high dimensional data which is very seldom the case for other descriptors [21].

There exist many different definitions of entropy, but the information theoretic framework relevant to this thesis focuses on a definition called the Renyi entropy [15]. Therefore we move on to look at Renyi's definition of entropy, which is computationally simple and easy to estimate [21].

Renyi's entropy

Renyi's entropy of order α of a random variable X is given as

$$H_\alpha(X) = \frac{1}{1-\alpha}\log\left(\sum_{k=1}^{N} p_k^\alpha\right) \tag{4.3}$$

or

[1] Similarly for the continuous definition.

$$H_\alpha(X) = \frac{1}{1-\alpha} \log \int p^\alpha(x) dx, \tag{4.4}$$

where $\alpha \geq 1$. Complete derivation can be found in [11].

In the information theoretic framework presented in [21], $\alpha = 2$ is chosen as the fundamental descriptor because it gives us an easy and computationally effective estimator of entropy, and thus we get *Renyi's quadratic entropy*

$$H_2(X) = -\log \int p^2(x) dx = -\log V_2(x), \tag{4.5}$$

where $V_2(X)$ is called the *quadratic information potential* and can be expressed as

$$V_2(X) = \int p^2(x) dx = E\left\{p(x)\right\}. \tag{4.6}$$

Similarly for the discrete case we get

$$H_2(X) = -\log \left(\sum_k p^2(x_k) \right). \tag{4.7}$$

The reason for choosing $\alpha = 2$ is that it gives a simple, but elegant way to estimate the quadratic information potential directly from data samples as we will see in Section 4.3 where the continuous definition in (4.5) is used to create an estimator using a nonparametric density estimate called a Parzen window[2].

One of the most important results directly relevant to Renyi's entropy is found in [17] where it is shown that the Gaussian distribution is the only distribution that has a finite number[3] of *cumulants*. The cumulants of a probability density function are simply the coefficients, similar to the *moments*, created from the *logarithm of the moment generating function*, and

[2]See section 4.2.

[3]As we know the Gaussian distribution has only two moments, the mean and the variance, and therefore also only two cumulants.

any probability density function can be expressed as a function of its cumulants [5]. So from this we see that since the Renyi entropy can be viewed as a monotonic function of the expectation of the probability density itself it is a measure which contains *all order moments of the density*. Because of this the entropy measure is a completely general measure, which can be used in principle for any density.

A good example where the entropy measure has been used with great success is in the blind source separation method called *independent component analysis* (ICA) [13]. In this method the goal is to find independent latent mixture components in signals. A simple example would be two people talking independently in a room recorded with two microphones, and the goal would be to separate the two monologues, this is called *the cocktail party problem*. We will not go into details, but the solution comes from searching after statistically independent component which are non-Gaussian. Because of this a measure of non-gaussianity is needed, and since Gaussian distributions have the highest possible entropy of all distributions with equal covariance structure, a measure based on entropy has been used, called *negentropy*. It is stated as

$$J(y) = H(y_{Gauss}) - H(y), \tag{4.8}$$

where y_{Gauss} is a random variable with the same covariance as y and H is some entropy measure. Because $H(y_{Gauss})$ has the largest possible entropy of y $J(y)$ will be a nonnegative measure describing how dissimilar the distribution of y is from a Gaussian distribution.

4.2 Non-parametric density estimation

Nonparametric density estimation is a framework of methods that estimates the probability distribution of a given dataset with only the dataset itself to work with, no assumptions of shape or belonging parameters are made. There are several reasons to why nonparametric density estimation is a central subject in this thesis.

1. The methods we are dealing with are *unsupervised*, i.e in principle we know nothing about the data.
2. All of the information theoretic measures mentioned requires an estimate of the probability density to be calculated.
3. The non-parametric estimator called the Parzen window (or kernel density estimator) can in some cases actually be viewed as a kernel function and thus creates a strong connection between information theoretic learning and kernel methods.

We will start with a crude, but widely used estimator, namely the histogram.

4.2.1 The histogram

The histogram is a straightforward non-parametric estimator which simply counts the number of observations occurring in some interval surrounding a given set of equidistant points. We adopt the formal definition given in [26].

Given origin x_0 and a point-surrounding interval h, the *bins* of the histogram is given as $[x_0 + mh, x_0 + (m+1)h]$, $m \in \mathbb{Z}$, yielding the density estimate

$$\hat{f}(x) = \frac{1}{Nh}(\#\text{ of } x_i \text{ in the same bin as } x). \tag{4.9}$$

Here we see that we have a set of points, e.g $x_m = x_0 + mh + \frac{h}{2}$, and simply count the number of observations inside the interval $x_m \pm \frac{h}{2}$.

The histogram has been predominantly used as a visualization tool for uni-,bi- and in some cases trivariate densities, as it has poor analytical properties being a discontinuous estimator. But is simple to compute and gives a quick

way to visualize the densities. Because of this it has not been used extensively in methods requiring density estimates. Another problem with the histogram is that we have to chose both bin width, h, and origin, x_0 which in many cases can give drastically different results. We will not go into further details, but they can be found in [26].

4.2.2 The naive density estimator

The naive density estimator is a density estimator stemming from the definition of calculating probability from a density function, that is

$$f(x) = \lim_{h\to\infty} \frac{1}{2h} P(x - h < X < x + h). \tag{4.10}$$

From this equation it is intuitive that we can choose an interval h and, as with the histogram, count the number of observations inside the interval for each x in $f(x)$. In [26] the following definition is given

$$\hat{f}(x) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{h} w(\frac{x - x_i}{h}), \tag{4.11}$$

where

$$w(x) = \begin{cases} \frac{1}{2} & ; \ |x| < 1 \\ 0 & ; \ \text{otherwise} \end{cases} . \tag{4.12}$$

To give a quick interpretation of this we see that we place a box over each of the data points and the density estimate, $\hat{f}(x)$, is the sum of the contributions of all boxes at a given point x. The naive estimator can give better results than the histogram, though in a special case it gives exactly the same, but the estimation is still discontinuous and therefore we move directly on to the generalization called *the Parzen window* and leave the further details of the naive estimator that can be found in [20],[26] and [30].

4.2.3 The Parzen window/kernel density estimator

In the Parzen window density estimator the $w(x)$ in (4.11) is replaced with a more general kernel function K, or Parzen window, which satisfies a number of conditions [20],[26]. The most notable conditions,[30], are that if

$$\int_{-\infty}^{\infty} K(x) = 1, \tag{4.13}$$

i.e the kernel function is a probability density itself, and

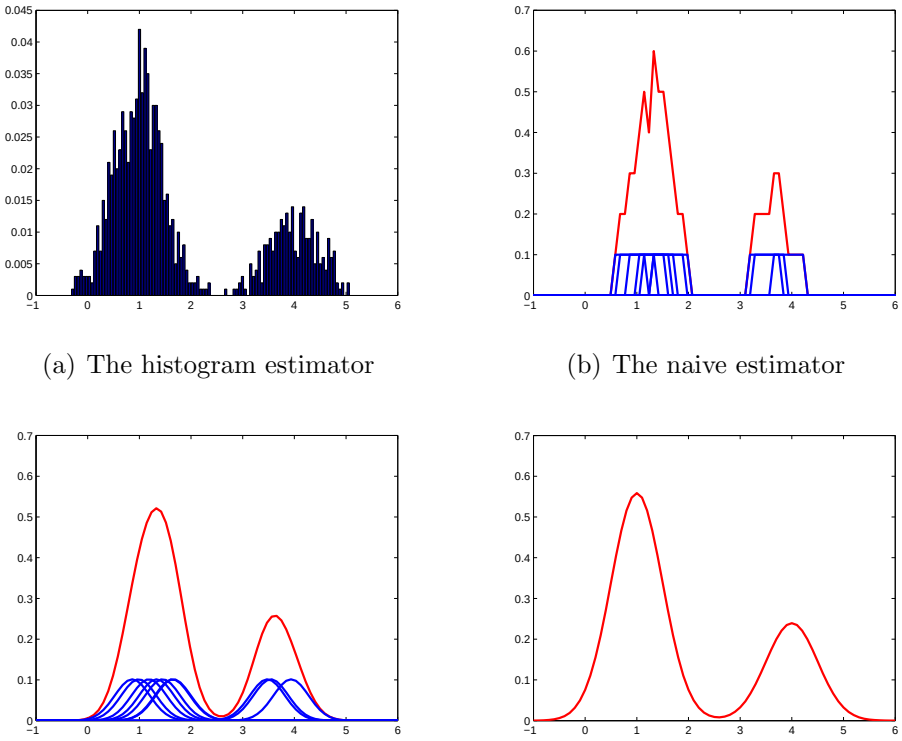
$$K(x) \geq 0 \tag{4.14}$$

then the density estimate

$$\hat{f}(x) = \frac{1}{Nh} \sum_{i=1}^{N} K(\frac{x - x_i}{h}) \tag{4.15}$$

will be a true density function. The kernel is also generally assumed to be symmetric [23]. Exact mathematical details can be found in [20]. If we take a quick look at the intuition behind this estimator we see that is the same as with the naive estimator, but now instead of placing a box on each observation we place an almost 'true' density function on each observation, the only difference is that they are 'weighted down' so all of them sum to a total of one. Different choices of the density kernel used can be found in [20]. One problem with this approach is that the 'density bumps' are placed on every observation regardless of where in the distribution it is, and therefore in e.g long tailed distributions problems with noise in the tails can occur [26]. Another disadvantage with the Parzen window, which also applies to the other two estimators, is that if we are to estimate a multivariate distribution, the number of observations relative to the dimensionality decreases drastically as the dimension increases, so what we in effect are doing is just placing some bumps on a few points which would not sum up to anything but scattered noise[21],[23]. A bivariate Parzen window estimator, with a straightforward extension to multivariate densities, is given in [23] as

$$\hat{f}(x, y) = \frac{1}{N} \sum_{i=1}^{N} K_{h_x}(x - x_i) K_{h_y}(y - y_i), \tag{4.16}$$



(a) The histogram estimator

(b) The naive estimator

(c) The Parzen window

(d) The true density function

Figure 4.3: Three estimations of a mixture of Gaussians distribution using 10 observations, in 4.3(d) we see the true density function

where $K_{h_x}(x-x_i) = K(\frac{x-x_i}{h_x})$. The problem of sparsity in higher dimension is apparent for density estimation as we would like to get as much of the support of the distribution included in the estimate, but this is not a major problem when dealing with entropy because as previously mentioned it gives a good description of the 'volume' of densities in higher dimensions. Seen in this context it means that the entropy measure is not as dependent on covering as much of the support as possible as the density estimation techniques are.

In Figure 4.3 we see the estimation of a mixture of Gaussians distribution by a histogram, the naive estimator and the Parzen window method. In this example only 10 data points are used for illustration purposes, which in a real situation would probably be insufficient, but we see the general idea.

4.3 The Kernel Entropy Component Analysis transformation

The kernel entropy component transformation is the main goal of this article. From Chapter 2 we remember the issues of the regular PCA transformation, and in Chapter 3 we dealt with the problem of nonlinear data. With these notions in hand it is clear that an intuitive way of improving the *principal components* way of thought is to exchange the estimation of covariance in PCA and KPCA with an estimation of entropy, which is exactly what kernel entropy component analysis does [15]. The particular choice of entropy measure is the quadratic Renyi entropy from Chapter 4 as it has a nice estimator given by the Parzen window density estimator.

The derivation of kernel entropy component analysis is not very similar to the derivations of PCA and KPCA, the starting point is completely different and it is actually quite surprising that it leads to a method so similar to KPCA, but we will dive straight in and rather give some comments on the similarities and differences later in Section 4.3.1.

We start by expressing an estimate of the continuous Renyi's quadratic entropy based on the kernel density estimator.

Renyi's quadratic entropy is given by:

$$H(p) = -\log \int p^2(\mathbf{x}) d\mathbf{x}. \tag{4.17}$$

For estimation purposes one can consider the simplified expression since the logarithm is a monotonic function.

$$V(p) = \int p^2(\mathbf{x}) d\mathbf{x}. \tag{4.18}$$

Assume $\mathbf{x} = \{x_1, x_2, \cdots, x_N\}$ is a stochastic variable. To estimate $p(\mathbf{x})$ in (4.18) a Parzen window estimator is used.

If we look back to Chapter 3 we remember that a kernel function has to be positive semi definite to obey the conditions in Mercer's theorem. So from this we notice that if the Parzen window, or now more appropriately called

kernel function, is chosen to be positive semi definite function, the Parzen window estimator is in fact a realization of the kernel trick, that is, it can be viewed as a sum of inner products calculated in the unknown feature space [14]. Using this we can rewrite the Parzen window estimator using the kernel notation used in Chapter 3[4].

$$\hat{p}(\mathbf{x}) = \frac{1}{N\sigma}\sum_{i=1}^{N} K\left(\frac{\mathbf{x}-\mathbf{x}_i}{\sigma}\right) = \frac{1}{N}\sum_{i=1}^{N} k_\sigma(\mathbf{x},\mathbf{x}_i) \tag{4.19}$$

where $k_\sigma(\mathbf{x},\mathbf{x}_i)$ is the kernel of the Parzen window estimator and σ is the kernel size.

As we remember $V(p)$ can be viewed as $E\{p(\mathbf{x})\}$ and if the sample mean is used as an estimator for $E\{\cdot\}$, we get the following estimate.

$$\hat{V}(p) = \frac{1}{N}\sum_{i=1}^{N}\hat{p}(\mathbf{x}_i) = \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N} k_\sigma(\mathbf{x}_i,\mathbf{x}_j) \tag{4.20}$$

$$= \frac{1}{N^2}[k_\sigma(\mathbf{x}_1,\mathbf{x}_1) + k_\sigma(\mathbf{x}_1,\mathbf{x}_2) + \cdots + k_\sigma(\mathbf{x}_1,\mathbf{x}_N)] \tag{4.21}$$

$$+ \cdots + [k_\sigma(\mathbf{x}_N,\mathbf{x}_1) + k_\sigma(\mathbf{x}_N,\mathbf{x}_2) + \cdots + k_\sigma(\mathbf{x}_N,\mathbf{x}_N)] \tag{4.22}$$

$$= \frac{1}{N^2}\mathbf{1}^T\mathbf{K}\mathbf{1}, \tag{4.23}$$

where $\mathbf{K}$ is the Kernel matrix as stated in section 3.1. The kernel matrix can be expressed as the eigendecomposition $\mathbf{K} = \mathbf{E}\mathbf{\Lambda}\mathbf{E}^T$ which, when inserted in (4.23) gives

$$\hat{V}(p) = \frac{1}{N^2}\mathbf{1}^T\mathbf{K}\mathbf{1} \tag{4.24}$$

$$= \frac{1}{N^2}\mathbf{1}^T\mathbf{E}\mathbf{\Lambda}\mathbf{E}^T\mathbf{1}. \tag{4.25}$$

$$\tag{4.26}$$

[4]We also exchange the h in the Parzen window with a σ for notation purposes.

The eigenvector matrix $\mathbf{E}$ contains, similar as in the kernel PCA derivation, the eigenvectors $\boldsymbol{\alpha}_i$ of $\mathbf{K}$ as its columns.

To get a full understanding we rewrite (4.26) as

$$\hat{V}(p) = \frac{1}{N^2}\mathbf{1}^T\mathbf{E}\boldsymbol{\Lambda}\mathbf{E}^T\mathbf{1} \tag{4.27}$$

$$= \frac{1}{N^2}\begin{bmatrix}1 & 1 & \cdots & 1\end{bmatrix}\begin{bmatrix}\boldsymbol{\alpha}_1 & \boldsymbol{\alpha}_2 & \cdots & \boldsymbol{\alpha}_N\end{bmatrix}\begin{bmatrix}\lambda_1 & 0 & \cdots & 0\\ 0 & \lambda_2 & \cdots & 0\\ \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & \cdots & \lambda_N\end{bmatrix}\begin{bmatrix}\boldsymbol{\alpha}_1^T\\ \boldsymbol{\alpha}_2^T\\ \vdots\\ \boldsymbol{\alpha}_N^T\end{bmatrix}\begin{bmatrix}1\\ 1\\ \vdots\\ 1\end{bmatrix} \tag{4.28}$$

$$= \frac{1}{N^2}\begin{bmatrix}\lambda_1\mathbf{1}^T\boldsymbol{\alpha}_1 & \lambda_2\mathbf{1}^T\boldsymbol{\alpha}_2 & \cdots & \lambda_N\mathbf{1}^T\boldsymbol{\alpha}_N\end{bmatrix}\begin{bmatrix}\boldsymbol{\alpha}_1^T\mathbf{1}\\ \boldsymbol{\alpha}_2^T\mathbf{1}\\ \vdots\\ \boldsymbol{\alpha}_N^T\mathbf{1}\end{bmatrix} = \sum_{i=1}^{N}(\sqrt{\lambda_i}\boldsymbol{\alpha}_i^T\mathbf{1})^2. \tag{4.29}$$

From this we see that the entropy estimate consist of a *sum of all the components contained in the individual kernel principal component axes*, $\sqrt{\lambda_k}\boldsymbol{\alpha}_k^T$, of the input data as seen in (3.24) which again is *summed up to create the total entropy estimate*. If we think of each element in the sum in (4.29) as an *entropy component* similar to the *principal component* in PCA and KPCA, and the fact that the eigenvectors in 4.26 are the same used in KPCA we can denote the Kernel Entropy Component transformation by using the top k eigenvectors that contribute to the entropy estimate as the projecting vectors. It is straightforward to see that using this will keep as much entropy as possible after the transformation from (4.29).

Using the k'th entropy contributing eigenvector to transform an arbitrary point in the feature space gives the following transformation.

$$\mathbf{v}_{k_H}^T\boldsymbol{\Phi}(\mathbf{x}) = \frac{1}{\sqrt{\lambda_{k_H}}}\sum_{i=1}^{N}\alpha_{i,k_H}\boldsymbol{\Phi}(\mathbf{x}_i)^T\boldsymbol{\Phi}(\mathbf{x}) = \frac{1}{\sqrt{\lambda_{k_H}}}\sum_{i=1}^{N}\alpha_{i,k_H}\kappa(\mathbf{x}_i,\mathbf{x}), \tag{4.30}$$

where the subscript k_H denotes the entropy preserving component as opposed to the principal components from the KPCA expression (3.24). Similarly if we only use the input points, the transformation reduces to

$$\mathbf{v}_{k_H}^T \mathbf{\Phi}(\mathbf{x}_j) \overset{\forall j}{=} \sqrt{\lambda_{k_H}} \boldsymbol{\alpha}_{k_H}^T \tag{4.31}$$

To conclude this section we present the Kernel entropy component analysis algorithm we will use to create examples and conduct experiments.

Algorithm 2 Kernel entropy component analysis dimensionality reducing algorithm

Input: The complete set of data, $X = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N\}$

1: Calculate the kernel matrix, $\mathbf{K}$, with elements $\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j$.
2: Calculate the eigenvectors, $\boldsymbol{\alpha}$, of $\mathbf{K}$.
3: Calculate the entropy components of $\mathbf{K}$ using $\sum_{i=1}^{N} (\sqrt{\lambda_i} \boldsymbol{\alpha}_i^T \mathbf{1})^2$

Output: Chose the eigenvectors corresponding to the largest d *entropy* components for a d dimensional output.

4.3.1 Comments, observations and examples

Component selection

The first thing we notice is a clear difference with respect to the principal components methods, in kernel entropy component analysis we see that if we are to do a projection onto the most dominant feature we have to look at the component which contributes most to the entropy measurement. This difference lies in the fact that the entropy measure for each feature contains the sum of the components in the individual eigenvectors, not only the corresponding eigenvalue. This results in *not necessarily choosing the top eigenvectors of the kernel matrix.*

Input space vs feature space

Another very important thing we have to note are the starting points of the two methods. The KPCA started with transforming the data to some possibly infinite dimensional feature space and then projecting the data onto the axes which keeps as much of the variance in that space as possible via the kernel trick. The KECA on the other hand, started simply with a measure of entropy in the input space using Renyi's entropy and a Parzen window estimate. So we see that the KECA transformation with the top n entropy components can be viewed as an entropy preserving transformation in the *input space* as opposed to the KPCA which, keeping n principal components, is a variance preserving tranformation in an *unknown feature space*.

4.3.2 Comparative illustrations

We now look at some examples of data transformed using both the well known kernel PCA method and the kernel ECA method. We start with approximately the same data set as used in Section 3.2.1 and do the KECA and KPCA transformations with the Gaussian kernel, as stated in (3.6), and two different kernel sizes, 1 and 0.3. In Figure 4.4 and Figure 4.5 we see that the result is exactly the same for both KPCA and KECA with kernel size, $\sigma = 1$. This is also reflected in the component choices. With this kernel size the entropy preserving and variance preserving axes are the same.

On the other hand, in Figure 4.6 and Figure 4.7 we see that the results from the KECA and KPCA are not the same. In this example the entropy contribution was highest in the first and fourth axis. The same Gaussian kernel is used, but the kernel size is changed to $\sigma = 0.3$. As we see the KECA has projected the data more along the axes, whereas the KPCA has gathered the Gaussian blob of data in the origin while we see the ring spread out in the right side of the figure.

These examples show us that the choice of kernel size is very important, and as it turns out, there is no well established practical way of choosing the kernel size [30]. So the fact that the Kernel Entropy Component Analysis performs different from kernel principal component analysis on some choices of kernel size can be very useful [15].

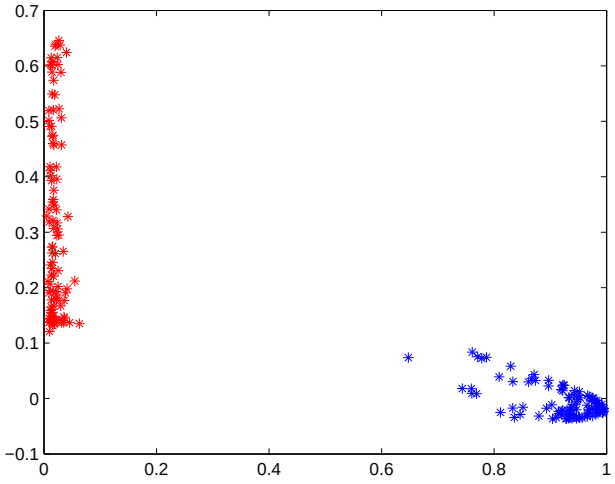


Figure 4.4: Kernel *entropy* component analysis of toy data in Figure 3.1 with Gaussian kernel and kernel size 1. The entropy is highest in the first and second kernel principal axis.

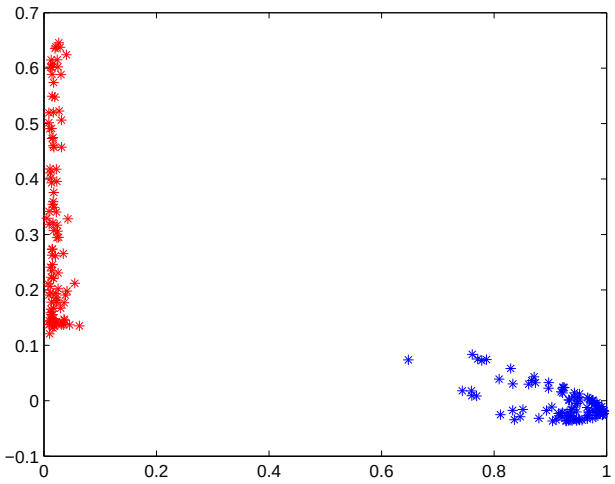


Figure 4.5: Kernel *principal* component analysis of toy data in Figure 3.1 with Gaussian kernel and kernel size 1.

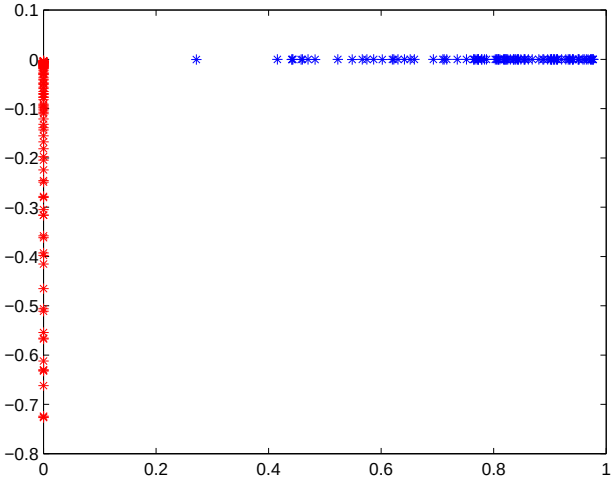


Figure 4.6: Kernel *entropy* component analysis of toy data in Figure 3.1 with Gaussian kernel and kernel size 0.3. The entropy is highest in the first and fourth kernel principal axis.

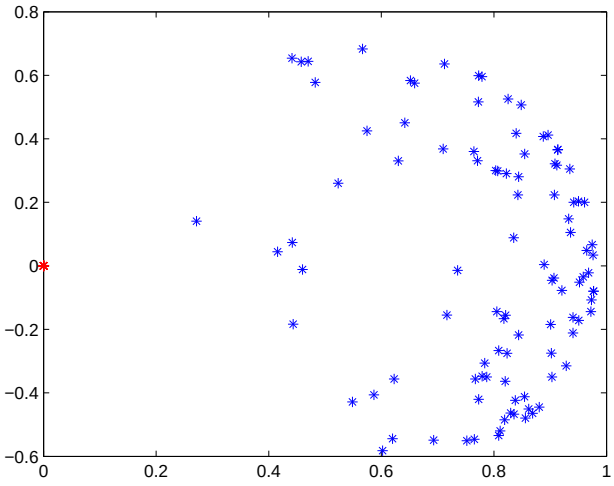


Figure 4.7: Kernel *principal* component analysis of toy data in Figure 3.1 with Gaussian kernel and kernel size 0.3.

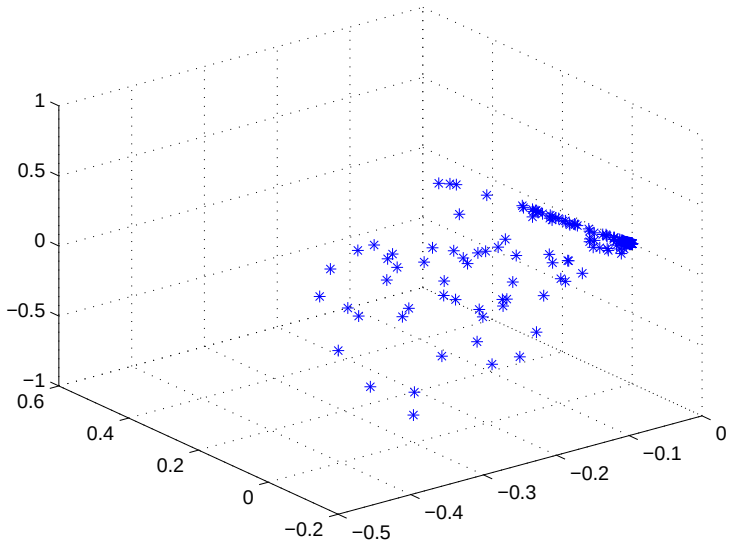
As a final example of the transformations we include the KECA and the ECA performed on a data set from the UCI machine learning repository[5] [1]. The data set consists of the chemical analysis of three different wines from the same region in Italy[6]. We do the transformation with the Gaussian kernel and two different kernel sizes, $\sigma = 0.91$ and $\sigma = 1.1$.

Using the $\sigma = 0.91$ kernel we see in Figure 4.8 that the two projections have created different results. The KECA used the first, second and fourth eigenvector. Since we know from before that this data set contains three classes, the KECA result gives much more sense compared to the KPCA.

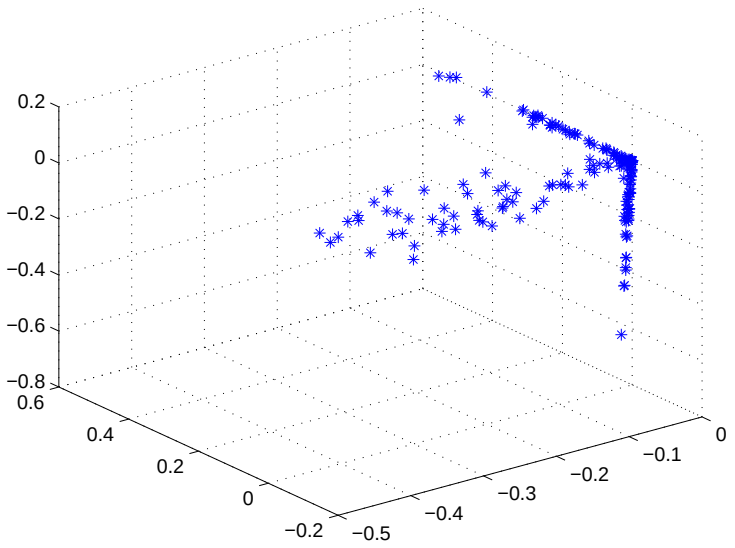
With the kernel size changed to $\sigma = 1.1$ in Figure 4.9 we notice that the KPCA and the KECA gives the same result, confirmed with the KECA choosing the first, second and third kernel principal axis.

In these examples we have seen that the Kernel Entropy Component Analysis transformation in some cases gives the same result as with Kernel Principal Component Analysis and some times completely different results. The difference often arises when the kernel sizes are small, as can be seen in [15]. We also notice the angular structure of the data after the KECA transformation seen in all the figures in this section. This property is further illustrated in [15].

[5] http://archive.ics.uci.edu/ml/.
[6] http://archive.ics.uci.edu/ml/datasets/Wine.

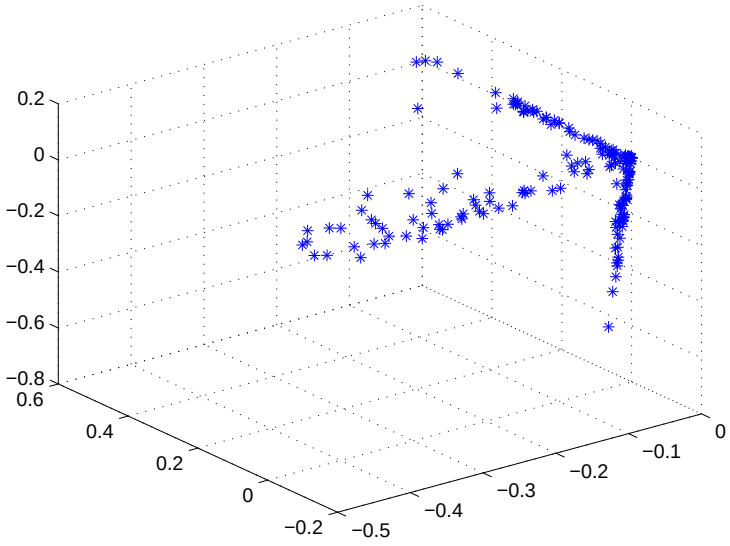


(a) KPCA on the wine dataset, $\sigma = 0.91$

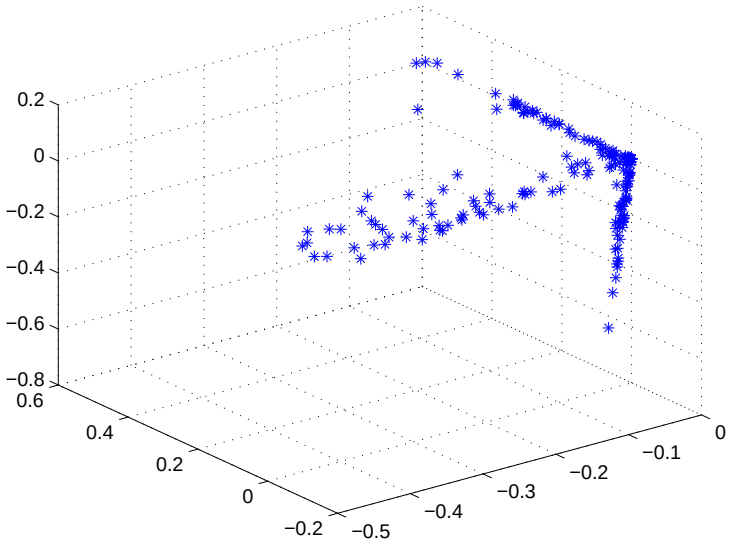


(b) KECA on the wine dataset, $\sigma = 0.91$, kernel PCA axes 1,2 and 4 used in the projection

Figure 4.8: KPCA and KECA on the wine dataset with $\sigma = 0.91$ and Gaussian kernel



(a) KPCA on the wine dataset, $\sigma = 1.1$



(b) KECA on the wine dataset, $\sigma = 1.1$, kernel PCA axes 1,2 and 3 used in the projection

Figure 4.9: KPCA and KECA on the wine dataset with $\sigma = 1.1$ and Gaussian kernel

Chapter 5

Other spectral dimensionality reducing data transformations

In this chapter we will introduce two different data transformations which work similarly to the kernel component transforms (KECA and KPCA), but has different theoretical starting points, namely the *Laplacian eigenmaps* [3] and the *Data spectropy* [25]. Both of these methods use the eigenvectors of the kernel matrix.

The latter is originally a clustering algorithm, but it also involves a selection of eigenvectors from the kernel matrix. It is also is the only other algorithm than the KECA method which does not necessarily choose the top eigenvectors of the kernel matrix. We begin with the Data Spectroscopy algorithm.

5.1 Data Spectroscopy

The Data Spectroscopy algorithm, or simply DaSpec, bases its foundation on the spectral properties of the distribution dependent convolution operator [25], for which the kernel matrix, with some assumptions, is the empirical version.

Notation-wise this is not in conjunction with the rest of this thesis as we have only used and dealt with eigenvectors and matrices, but we chose to follow

the notation of [25] and take note that all the previous methods also have a theoretical background in the continuous convolution operator.

We start by defining the distribution dependent convolution

$$\mathcal{K}_P f(x) = \int \kappa(x,\tau) f(\tau) p(\tau) d\tau = \int \kappa(||(x-\tau)||) f(\tau) p(\tau) d\tau. \tag{5.1}$$

As we see this is simply a convolution weighted by the probability density of the input data.

The motivation for the DaSpec algorithm is based on the *eigenfunctions* of this convolution operator and their connection to the underlying distribution of the data in question. In practice we will only deal with eigenvectors, but the theoretical base is founded in the continuous eigenfunctions so we will introduce the very similar eigenfunction/eigenvalue equation:

$$\mathcal{K}_P \phi = \lambda \phi. \tag{5.2}$$

There are two properties/theorems of this eigenfunction problem listed in [25] that form the foundation of the DaSpec algorithm. We will not state the theorems in full, but simply list the implicating properties from them:

- The eigenfunction of $\mathcal{K}_P$ decays fast when moving away from the high density regions of P if the tails of the kernel function, K, decays fast enough.
- The top eigenfunction of the convolution operator is the only with no sign change on $\mathbb{R}^d$
- The top eigenfunction of the convolution operator is nonzero on the support of P.
- For a mixture of distributions, which we assume is true in many cases of classification and clustering, if we chose an appropriate kernel, the eigenfunctions of each of the mixtures will be contained in the eigenfunctions of the total mixture distribution.

As mentioned in practice we only deal with the empirical equivalents, the kernel matrix and its eigenvectors, as they under certain conditions converge to the eigenfunctions and corresponding eigenvalues of the convolution operator. We will not go further into details, but they can be found in [25].

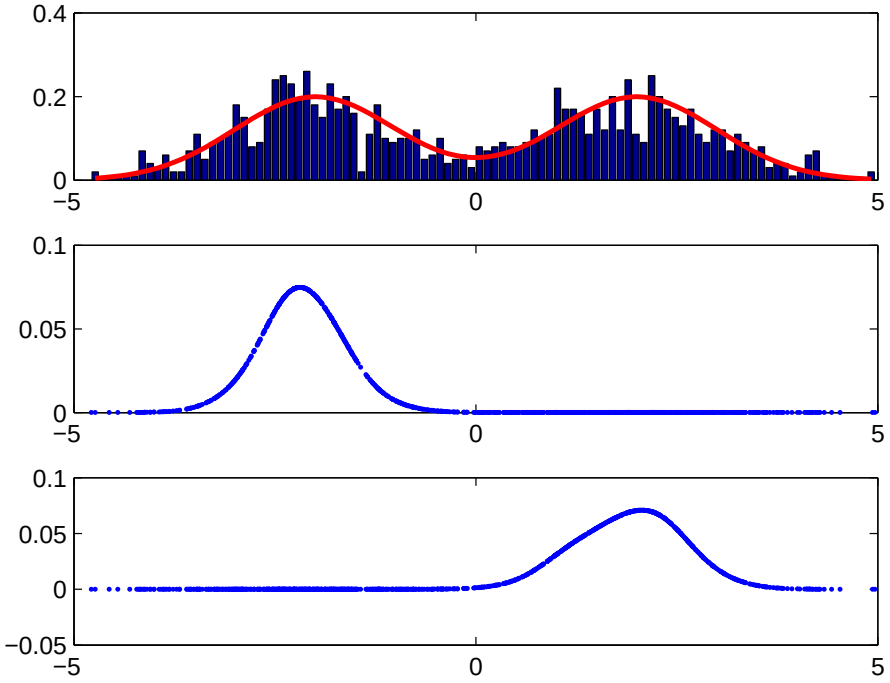


Figure 5.1: *Top*: Histogram of mixture of gaussians with the true density in red. *Middle*: The first eigenvector of the Kernel matrix of the mixture data. *Bottom*: The second eigenvector of the kernel matrix of the data mixture.

These properties can be seen clearly with a simple Gaussian mixture; $0.5\mathcal{N}(-2,1)+0.5N(-2,1)$. We use a Gaussian kernel, (3.6), and create the kernel matrix and find its eigenvectors. In Figure 5.1 we see the histogram of the mixture and the top two eigenvectors of the kernel matrix.

We see that the eigenvectors fulfill the aforementioned properties; no sign change, nonzero over the support of the mixture components, rapid decay when the density is low and it is clear that the eigenvectors clearly stems from each of the mixture components even though they are eigenvectors of the entire mixture distribution. In [25] the authors have created a clustering algorithm based on these properties, but this algorithm also contains an indirect dimensionality reduction which in [28] is used in combination with

the LASSO[1] to create a semi-supervised classifier. We leave out the clustering part of the algorithm in [25] and use only the dimensionality reducing part, summarized in Algorithm 3.

Algorithm 3 DaSpec dimensionality reducing algorithm

Input: The complete set of data, $X = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N\}$

1: Calculate the Kernel matrix $\mathbf{K}$ from all available data, labeled and unlabeled.
2: Calculate the eigenvectors of the matrix $\mathbf{K}$.
3: Select all N_ϵ eigenvectors $\boldsymbol{\alpha}_i$ of $\mathbf{K}$ that has no sign change up to precision ϵ.

Output: The eigenvectors form the reduced dataset$\{\boldsymbol{\alpha}_i\}_{i=1}^{N_\epsilon}$

The last and perhaps most important thing we need to notice is that the top eigenvectors in question does not necessarily correspond to the top eigenvalues. This makes the Daspec algorithm comparable to Kernel ECA in that it also differs from the principal component way of thought, automatically choosing the eigenvectors corresponding to the top eigenvalues.

5.2 Laplacian eigenmaps

The final data transformation we will present are the Laplacian eigenmaps. This transformation actually ends up being closely related to Kernel PCA, but the starting point is a completely different story.

The intuition behind the Laplacian eigenmaps is that we assume that the high dimensional data lies on a manifold with much lower dimension suspended in the high dimensional space and try to take advantage of this to reduce the dimension of the data [3]. To illustrate this idea the so called 'swiss roll' is often used, seen in Figure 5.2. We see that if we could somehow stretch out the data lying on the roll we would get a two dimensional data set which intuitively would seem much easier to work with.

[1] A least squares classifier with a linear constraint. We will look closer at this in section 7.1.3.

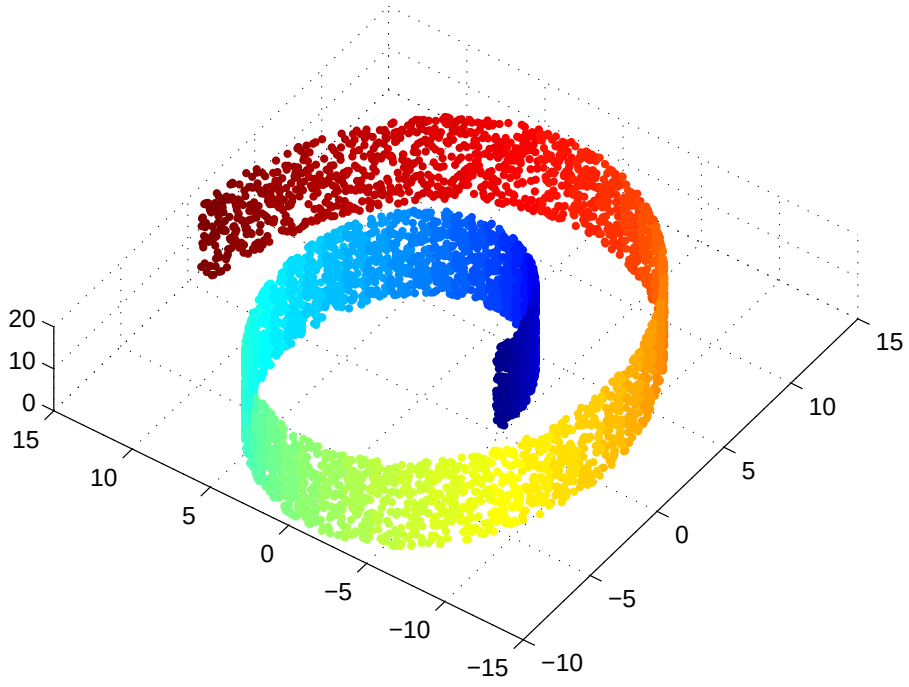


Figure 5.2: 'Swiss roll' toy data

The actual transformation tries to preserve the neighborhood structure in the data when reducing the dimension and in that way keeping the geometric structure of the underlying manifold [3],[30]. This is done via the *Laplacian* of an *adjacency graph* created on the data. We will not go into further details of the algorithm as its purpose in this thesis is to serve as a comparison to the Kernel ECA transformation in the semi-supervised setting. The algorithm is presented in [3].

We create a simplified version of the algorithms presented in [30] and [3] which represents how we use the transformation in this thesis shown in Algorithm 4.

The adjacency weight matrix $\mathbf{W}$ is for all our intents and purposes exactly the same as the kernel matrix $\mathbf{K}$ used in DaSpec and Kernel ECA/PCA. This is because we see the *graph* as a fully connected with a Gaussian kernel

Algorithm 4 Laplacian eigenmaps dimensionality reducing algorithm

Input: The complete set of data, $X = \{\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N\}$

1: Create a connected *graph* over all points in the input set.
2: Create the matrices $\mathbf{W}$ and $\mathbf{D}$, where $\mathbf{W}_{ij} = \exp\left(-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{\sigma^2}\right)$ and $\mathbf{D}$ is a diagonal matrix with the sum of each row or column[2] of W as the diagonal elements.
3: Create the matrix $\mathbf{L} = \mathbf{W} - \mathbf{D}$ and solve the eigenvalue problem $\mathbf{D}^{-1}\mathbf{L}\mathbf{y} = \lambda\mathbf{y}$

Output: Chose the *smallest* $d+1$, corresponding to the wanted dimension, eigenvectors and ignore the smallest one as it represents projecting all points to a single point.

deciding the weight of the neighborhood between points in the graph. If we rewrite the eigenvalue problem in Algorithm 4 as

$$\mathbf{D}^{-1}\mathbf{L}\mathbf{y} = \mathbf{D}^{-1}(\mathbf{W} - \mathbf{D})\,\mathbf{y} = (\mathbf{D}^{-1}\mathbf{W} - \mathbf{I})\mathbf{y} = \lambda\mathbf{y} \tag{5.3}$$

we see that we can choose the *largest* eigenvectors of $\mathbf{D}^{-1}\mathbf{W}$, basically the kernel matrix with a diagonal term indicating the close relationship to Kernel PCA.

5.3 Quick summary

This concludes the Theory section of this thesis. We have looked at several different data transformations and some background theory. In the next section we will present the algorithms used to test how the transformations work under the semi supervised setting and take note that we only use the kernel matrix-based transformations from now on (i.e. all except ordinary PCA). We have also chosen to define the classifiers we use in the experiment section instead of the theory section as they can be seen as a set of tools used to perform the experiments.

Part II

Experiments

Chapter 6

Introduction and connection to Theory part

To summarize what we have presented in the Theory section and connect it to the semi-supervised scheme presented in the introduction, we note the following:

- We want to investigate how classification benefits from including unlabeled data.

Which in this setting implies *including the unlabeled points we have available in the building of the kernel matrix* that all the transformations we have presented are based on, and then apply a classifier on the transformed data. We will use the LASSO classifier, presented in Section 7.1.3.

Initially the intuition that adding unlabeled data makes sense is perhaps stronger for the Laplacian eigenmaps and DaSpec method than for the KECA and KPCA transformations so we try to shed some extra light on this before we move on to the experiments.

The Laplacian method is a transformation based directly on points lying on manifolds in the input space, so the intuition that adding extra data points in the matrix describing the manifold neighborhood helps is quite clear. The more data, the better description of the manifolds and thus better class

description[1]. We note almost the same for the DaSpec algorithm, it searches for eigenvectors that has a set of properties flagging them as representatives for mixture components in the data set. From Figure 5.1 it is obvious that if we have very few (labeled) data points, the algorithm probably would not be able to find such components.

For the KECA transformation we remember that it is based on a Parzen window estimate of Renyi's quadratic entropy. A setting with few labeled points will give poor entropy estimates probably leading to the KECA transformation suggesting a basis of eigenvectors containing very little useful information. So to include the unlabeled points give meaning in that it hopefully yields good entropy estimates leading to a choice of eigenvectors that describes the data well which again could help the classifiers we use. The same intuition applies for the KPCA, except that we now need the unlabeled data points to build up the diagonalized covariance matrix in the feature space.

In the rest of this part of the thesis we will conduct different experiments to illustrate the Kernel ECA Semi-Supervised Learning LASSO classifier, presented in section 7.1.4. Both toy data and real 'benchmark' datasets will be used and we will compare with results based on the other three dimensionality reducing techniques discussed in the theory section.

[1]We now assume that each class in a classification problem lies on different manifolds, not an unreasonable assumption, [6].

Chapter 7

Classification and the semi-supervised learning classifiers

Before we can start the actual testing of the KECA transformation in the semi-supervised learning (SSL) scheme we have to do some choices on what kind of classification methods we will use. Inspired by [16], [28] and [4] we choose to use the simple linear *ordinary least squares classifier* and a constrained version known as the LASSO which has been shown to give better results in [28]. We start by giving a short introduction to classification and a description of the two classifiers with some simple illustrations.

Normally supervised learning is divided into two main categories, *regression* and *classification*. The former tries to learn a function from the data we are given which can take continuous values, while the latter, classification, tries to learn a discrete function from the data which separates the data set in different *classes*. Common to both are the need for some knowledge about the data, either as *labels* indicating classes or patterns in the data, or as a continuous *response* representing the output of the function we want to learn.

In this thesis we will only work with classification, but we keep with us the notion that in practice the only difference is that in regression the function we try to learn is a continuous function that fits the structure of the data rather than trying to separate the data in discrete values.

7.1 Classification

As mentioned the goal of classification is to assign a discrete value to each data point, and by that assigning each data point to a particular *class*.

We start by introducing the simplest form of classification: binary classification of two well separated groups of data. A typical example of such data is shown in Figure 7.1 (a). As we see there are two distinct classes in the data, marked with blue and red, and we want to design a function that tells us which class, in this case either 'blue' or 'red', a new data point from the same source belongs to. A rough mathematical representation could be:

Given a set of data $X_{TRAINING}$, X_{TEST} and labels $Y \in$ ['red', 'blue'], we want to find an f, using $X_{TRAINING}$ and Y, such that

$$f(X_{TEST}) \in \text{['red', 'blue']}. \tag{7.1}$$

The design of the function f is reffered to as *training* the classifier. In practice there are extremely many ways of designing the classifiers, ranging from simple linear classifiers to advanced multidimensional non-linear networks. The most well known are perhaps the *support vector machine*, *the naive Bayes classifier*, *the least squares classifier* and *the multilayer perceptron*[1] [30].

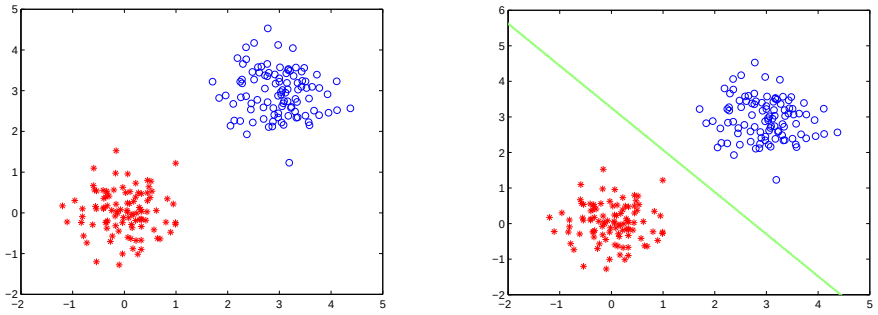
In this thesis we will limit ourselves to only use simple linear classifiers and use a variant of the *least squares (LS) classifier* called the LASSO. We start by defining the LS classifier and then add some constraints to the solution to obtain the LASSO classifier.

7.1.1 Short introduction to linear classifiers

We follow the concept of linear classifiers given in [30] where the goal of the classifier is to design a hyperplane which separates the data as best it can. The hyperplane is defined as:

$$f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0 = \mathbf{w}_a^T\mathbf{x}_a = 0. \tag{7.2}$$

[1] Also known as a *neural network*



(a) Data set with two classes of data marked in blue circles and red stars.

(b) The data separated by a line (the '2-D equivalent' of a hyperplane).

Figure 7.1: Typical binary classification

Where the augmented quantities $\mathbf{x}_a = \begin{bmatrix}\mathbf{x} & 1\end{bmatrix}$ and $\mathbf{w}_a = \begin{bmatrix}\mathbf{w} & w_0\end{bmatrix}$ are introduced for notational simplicity. Later we simply denote $\mathbf{x}_a$ as $\mathbf{x}$ and $\mathbf{w}_a$ as $\mathbf{w}$

The hyperplane, $\mathbf{w}$, we design such that points on each side of the plane takes on values $\gtrless 0$ and exactly 0 if the point lies on the plane. A typical example can be seen in Figure 7.1 (b). In (7.2) we see that the linear classifier is simply an inner product with a weight vector $\mathbf{w}$ in addition to a threshold w_0, which maps each point to either a positive or negative value. All we have to do in practice is to utilize the weight vector plus threshold and then check the sign of the result. The only thing left now is how to find the weight vector. We start with one of the most basic, but still very widely used classifier: the *least squares classifier*.

7.1.2 The least squares classifier

The least squares classifier, [30], is based on the optimization of the following cost function[2]:

[2]We remember the augmented vectors introduced including the threshold w_0 in $\mathbf{w}$.

$$J(\mathbf{w}) = \sum_{i=1}^{N} (y_i - \mathbf{x}_i^T \mathbf{w})^2. \tag{7.3}$$

Here we see that we measure the squared error of the difference between the labels, y_i, and the classified points $\mathbf{x}_i^T \mathbf{w}$. To optimize this function we differentiate with respect to $\mathbf{w}$ and force the derivative to zero:

$$\frac{dJ(\mathbf{w})}{d\mathbf{w}} = \sum_{i=1}^{N} \mathbf{x}(y_i - \mathbf{x}_i^T \mathbf{w}) \stackrel{!}{=} 0 \tag{7.4}$$

$$\sum_{i=1}^{N} (\mathbf{x}_i^T \mathbf{x}_i)\mathbf{w} = \sum_{i=1}^{N} \mathbf{x}_i y_i \tag{7.5}$$

For simplicity we rewrite the sums using matrix notation. The matrix X is a matrix containing all $\mathbf{x}$ as rows and the vector $\mathbf{y}$ contains all labels, y_i, associated with the points $\mathbf{x}_i$. With this we get the famous closed form solution for $\hat{\mathbf{w}}$ in the least squares problem [30]:

$$X^T X \hat{\mathbf{w}} = X^T \mathbf{y} \tag{7.6}$$

$$\hat{\mathbf{w}} = (X^T X)^{-1} X^T \mathbf{y} \tag{7.7}$$

This closed form solution gives a hyperplane that separates two classes with the lowest squared error possible. If we have more than two classes we need to create one hyperplane for each class separating the class in question from all the other classes. This is known as the one versus all strategy for multi class classification problems [30].

7.1.3 The LASSO

When working with large multivariate data sets and least squares classifiers there are two problems besides the obvious linearity constraints that dominate the results: *Prediction accuracy* and *model interpretation* [8][31].

The most used techniques to deal with these problems are, respectively, *ridge regression* and *subset selection*, [12]. Ridge regression puts a constraint on the squared sum of the coefficients in the classifying function, while subset selection simply chooses the subset of coefficients which gives the best performance.

Straight forward subset selection is with most real world data sets computationally unfeasible as it must iterate through all combinations of features to find the optimal one. Also, because it is a discrete process that simply removes features that does not contribute to lower prediction error it does not do anything with possible noise problems causing large variance in the predictions.

Ridge regression can give better prediction results by lowering prediction variance caused by extreme values and noise, but does not exclude any features and thus does not leave any room for model interpretation [31].

With these disadvantages with the usual solutions to the problems at hand the *least absolute shrinkage selection operator*, or simply LASSO, was introduced [31]. It can be seen as a combination of ridge regression and subset selection, it shrinks some coefficients and sets some to zero thus excluding them. The LASSO cost function, [31], is defined as:

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \sum_{i=1}^{N} (y_i - \beta_0 - \sum_{j=1}^{d} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{d} |\beta_j|, \tag{7.8}$$

[3]

where $\boldsymbol{\beta}$ is used to avoid confusion with the hyperplane coefficients $\mathbf{w}$ of the least squares classifier. We also note that the shrinkage parameter λ uses the same symbol as the eigenvalues used different places in the Theory section, so when we speak of the LASSO the λ means the shrinkage factor of the coefficients unless otherwise specified.

In this definition we see that the LASSO is simply a constrained version of the ordinary least squares classifier. Closely related to the square constraints of ridge regression the LASSO shrinks the parameters of β, but also sets some to zero. This is due to the absolute constraints of the LASSO creating

[3] In the LASSO cost function it is not normal to use augmented variables because the threshold β_0 is not included in the optimization as shrinking it would simply move the hyperplane away from the classification problem.

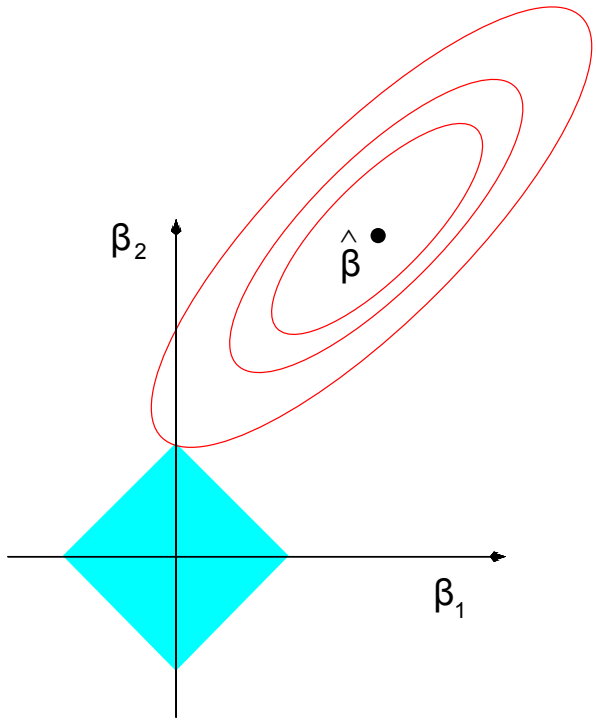


Figure 7.2: The least squares solution space contours (red) and the LASSO constraints shown in cyan. Figure taken from [8].

a nonlinear optimum in the solution space of the least squares cost function, seen in Figure 7.2 where we see that the solution obeying the absolute constraint closest to the LS solution $\hat{\boldsymbol{\beta}}$ has one coefficient set to zero. We also note that the problem posed in (7.8) is a quadratic programming problem and thus does not have a trivial closed form solution such as the least squares classifier [8]. In this thesis we will not go into the details of how the LASSO is optimized, we will use a software package to do the optimization, see Section 8.1.5.

There are several reasons for choosing the LASSO as classificator in this setting. Firstly it has given good results in [28] combined with DaSpec algorithm. Secondly the properties of the LASSO predictions are particularly suited for complementing the data transformations involved in the semi-supervised setting we are using:

- It helps prediction error by constraining the parameter vector deciding the 'tilt' of the classifying hyperplane preventing extreme points and outliers from having too much influence.
- When we have transformed the data using some kind of transformation in many cases the exact dimension/number of eigenvectors to use is very hard to determine and we have to resort to heuristic methods. The LASSO can remove unnecessary eigenvectors/dimensions included by these methods revealing a stronger choice of features used in the final classifier.

In the rest of the thesis we will refer to the LASSO and least squares classifiers as the baseline classifiers as they will be compared to the semi-supervised classification algorithms.

7.1.4 The semi-supervised classifiers

Now that we have presented the least squares classifier and the LASSO classifier we are ready to define the semi-supervised classifiers we will use. To simplify things we define two generic * SSL classifiers[4] which can be combined with any of the four kernel based data transformations we have presented in the Theory part, Kernel ECA, Kernel PCA, Data Spectroscopy and Laplacian eigenmaps:

General SSL LS classifier:

General SSL LASSO classifier:

[4]The * can be either KECA, DaSpec, Laplacian or KPCA in this setting.

Algorithm 5 * SSL LS classifier

Input: Given data arranged in a labeled set L and an unlabeled test set U

1: Create the (Gaussian) kernel matrix $\mathbf{K}$, with elements $\mathbf{K}_{ij} = \exp\left(-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{\sigma^2}\right)$, using all input points, $L + U$.
2: Based on $\mathbf{K}$, transform the data to a new space, f, using *.
3: Train a least squares classifier in f using the labeled set L.

Output: A least squares classifier in f that has included information from the unlabeled data in the test set, hopefully improving classification performance.

Algorithm 6 * SSL LASSO classifier

Input: Given data arranged in a labeled set L and an unlabeled test set U

1: Create the (Gaussian) kernel matrix $\mathbf{K}$, with elements $\mathbf{K}_{ij} = \exp\left(-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{\sigma^2}\right)$, using all input points, $L + U$.
2: Based on $\mathbf{K}$, transform the data to a new space, f, using *.
3: Train a LASSO classifier in f using the labeled set L.

Output: A constrained least squares classifier in f that has included information from the unlabeled data in the test set, hopefully improving classification performance.

Chapter 8

Experiment setup

We will conduct experiments inspired by the work in [28], [25], [4] and [16]. We compare the Kernel ECA SSL LASSO classifier with SSL versions of the methods reviewed in the Theory part: Kernel PCA, Laplacian Eigenmaps and the DaSpec algorithm.

In the basic experiment setup we will draw a number of data points at random from the particular data set we are analyzing and treat them as our labeled data/training set L. The rest of the data will act as unlabeled data/test set U. Unless otherwise noted this process is repeated 20 times and average classification error rates and standard deviations are calculated. It is important to note that in our case the set U will act as both unlabeled data for testing and test examples at the same time. This means that different experiments will have very different statistical properties as the ratio between labeled and unlabeled data points can vary.

8.1 Setup and parameter choices

Before we start the actual testing we need to define what exactly we want to test and because of the number of free parameters that can be tweaked we need to define restrictions on the experiments[1].

[1] E.g. kernel size, number of eigenvectors/dimensions/features to be used, the number of unlabeled points vs labeled points, the shrinkage factor λ in the LASSO and so on.

8.1.1 Choice of kernel

In theory any symmetric and positive semidefinite function could be used, [21], so the choices are almost endless, but we make a choice and use the well known Gaussian kernel:

$$\kappa(\mathbf{x}, \mathbf{y}) = \exp\left(-\frac{||\mathbf{x} - \mathbf{y}||^2}{\sigma^2}\right). \tag{8.1}$$

Throughout the experiments this is the only kernel we will use. This choice is also motivated by the fact that most related work almost exclusively use this kernel [28], [25], [4], [16].

8.1.2 Statistical normalization

We normalize all the datasets to zero mean and unit variance before we do any kind of transformations or classifications. This is a common practice when using spherical kernels such as the Gaussian and also helps scale the cost function space of the LS solution, making the structure of the data decide the importance of the coefficients rather than the scaling of the individual features, an important point when we deal with coefficient manipulating operators such as the LASSO [16], [9], [30].

8.1.3 Kernel size

Working with the UCI datasets we of course know all the labels of the data, we only simulate unknown labels. So to make the experiments best reflect the potential of the methods we will use, we iterate through the data with an entire spectrum of eigenvalues instead of focusing on single kernel sizes of different heuristic methods with different outcomes.

8.1.4 The number of dimensions/features/eigenvectors

The number of eigenvectors used in the different data transformation dictates the dimension of the output and hence the dimension of the space where the

classification takes place. So it is obvious that the choice of eigenvectors, and hence the dimension of the output has to be chosen carefully. We have chosen to both use former empirical observations which has given good results and straightforward heuristic methods. Below is a list of the different choices we have included and will use.

- 'Knee method', suggested for Kernel ECA in [16].
- In [4] an approximate number of eigenvectors of 20% of the number of labeled data points is suggested as it has given good results in a number of different experiments using the Laplacian LS SSL method.
- The number of eigenvectors equal to the number of classes, a common method used in e.g. [15].

If nothing specific is stated in the experiments we use the methods given in the original papers associated with the different algorithms, i.e.:

- The knee method for the Kernel ECA [16].
- The 20% rule for the Laplacian eigenmaps [4].
- The DaSpec is dictated by the number of eigenvectors fulfilling the properties listed in Section 5.1, so no choice can be made [28], [25].

For the KPCA SSL classifiers we use the same dimension as used for KECA because of their close relation and sometimes identical results, as shown in Section 4.3.1.

8.1.5 LASSO parameters

To do the actual LASSO shrinkage we use the 'glmnet' package, originally programmed in R, compiled for MATLAB[2], details of the algorithm can be seen in [9]. The choice of the shrinkage parameter λ is usually found by cross validation, [31], or some similar technique, but since we already know the full truth about the data set we can test empirically which λ gives the best results. Each time the 'glmnet' package solves the LASSO optimization it gives out a set of λ values ranging from small, resulting in the LS solution, to very large, resulting in all parameters reduced to zero. So for each test this λ-set can be

[2] http://www-stat.stanford.edu/~tibs/glmnet-matlab/

evaluated manually to see which is best. This is not a very practical solution, but the set does not usually contain more than ~ 60 different λ values so it is computationally feasible and lets us show the maximal potential of the LASSO.

8.1.6 Plotting

In this thesis we have used color as the main separator between different data in the same plot, so we strongly encourage the reader to view the plots in color.

8.2 Data sets

In the results section, Section 9, the different classifiers are tested on different data sets from the UCI machine learning repository [1]. We have chosen the sets IONOS, ADULT, PIMA, WINE and PEN as they are used in many similar publications [16], [4], [28]. In Section 9.2 we also use the 'two moons' toy data set from the introduction and in the last section we use the so called Frey faces data set also used in [15].

Chapter 9

Results

We begin by comparing the Kernel ECA SSLLASSO classifier to all the other methods from the theory section: DaSpec SSL LASSO, Laplacian SSL LASSO and Kernel PCA SSL LASSO.

9.1 Classification error over a range of kernel values

To look at the general behavior of the classifiers in the semi-supervised setting using the LASSO, we start by letting the kernel sizes vary over a range of values and test the classifiers with a different number of labeled points drawn at random from the original set. The random points are drawn half and half from each class in binary cases and equal numbers from each class in multi class settings. The rest of the data is treated as both unlabeled data and the test set. In this way we get to see the full potential of the methods, not suppressed by the difficult choice of kernel size we meet in practice.

In figures 9.1, 9.2, 9.3 and 9.4 we see each of the sets, IONOS, PIMA, ADULT and WINE classified with 20 labeled points for all methods using the both the SSL LASSO and the SSL LS algorithms. The best results, i.e. lowest prediction error, for all the data sets with 10, 20 and 30 number of labeled points and all unlabeled data included in the data transformation are sum-

marized in Table 9.1[1]. In Appendix C the dimension/number of eigenvectors and average sparsity of the LASSO for the three binary data sets are shown.

The choice of number of labeled points is inspired by [28], [16] and [4] for comparisons in addition to the fact that 10, 20 or 30 points is not an unreasonable number of data which an expert could label in a real life case. The results from the Lasso classifiers are the results of the λ values which gives lowest prediction error found by brute force iterations over all λ values for each iteration at each kernel size.

9.1.1 IONOS data set

In Figure 9.1 we see the IONOS data set classified using 20 known labels and the rest of the data treated as unlabeled test data using all four methods.

For the KECA and KPCA we clearly see a window of kernel sizes in the range $\sigma \in [1, 3]$ giving the lowest classification errors. For larger kernel sizes the two methods does not seem to give any improvement over the baseline LASSO classifier.

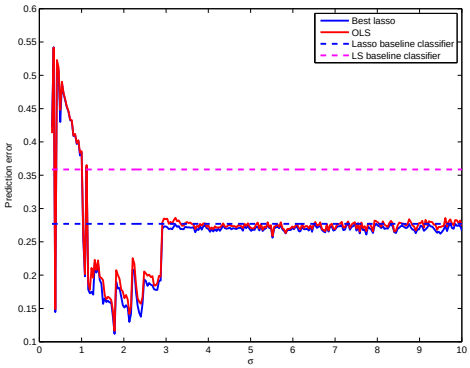
The Laplacian classifier seems to work best with larger kernel sizes than the rest with the best results appearing at σ greater than approximately 3.5. Results vary much and are sometimes well above the baseline results for the lower range of the kernel sizes, $\sigma \in [1, 4]$. An interesting observation is that this poor performance appears in almost the same range as the KECA and KPCA algorithms works best and vice versa when the KECA/KPCA works poorly. In Appendic C, Figure C.1 we can see that the differences in chosen number of dimensions is highest in this region.

We also observe that for the kernel size range where the Laplacian method works best, the LASSO also has an obvious effect lowering error, which is not as evident in the other methods.

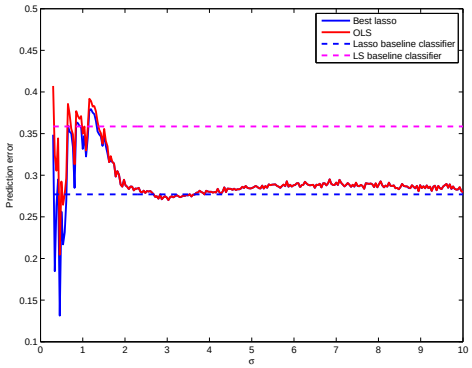
The DaSpec classifier has only a small window where it performs better than the baseline LASSO classifier, but in this area with $\sigma \approx 0.5$ it performs comparable to the best results of the other methods. For the larger kernel sizes we assume that the algorithm has problems finding the eigenvectors of the relevant components in the data explaining the poor results.

[1]Note that the values in the table are given in %, not error rates as the figures.

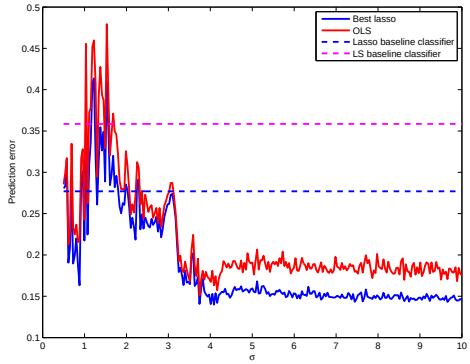
In Table 9.1 we note that the KECA and KPCA seem to have the lowest classification error for the IONOS data set.



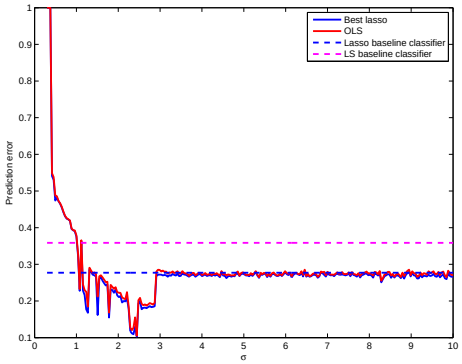
(a) Kernel ECA SSL classifier



(b) Daspec SSL classifier



(c) Laplacian eigenmaps SSL classifier



(d) Kernel PCA SSL classifier

Figure 9.1: IONOS data set: The different SSL classifiers tested over a range of kernel sizes with 20 known points. Lasso classification results: blue, LS classification results: red. Baseline ordinary least squares and lasso classifications with 15 labels are shown in magenta and blue dotted lines.

9.1.2 PIMA data set

Classification results for the PIMA dataset are presented in Figure 9.2.

The first thing we note is that the baseline LASSO classifier actually performs poorer than the least squares, indicating that the original least squares classifier gives the best linear result possible in the input space.

We see that both the KECA and the KPCA algorithms perform well with kernel sizes in the range larger than 5, and poorer in the range 0-2. A spike in the LS SSL classifiers occur at a kernel size slightly larger than 4, but with a much better LASSO result. In this case we see that the LASSO has helped the classifier significantly, we assume that the spike is caused by the knee method selecting very many dimensions and thus disturbing the least squares classifier, see Appendix C, Figure C.2. In section 9.4 we study this spike further.

The Laplacian SSL classifiers performs good with a fairly low classification error for kernel sizes approximately larger than 2, and generally poorer with very varied results in the kernel size range 0-2.

The DaSpec algorithm has only a small window where it performs better than the baseline classifiers in the kernel range around 1-3. Smaller kernel sizes yield quite poor results, with errors higher than the baseline. We also note basically no difference between the LASSO and the LS methods, indicating that the DaSpec only finds one eigenvector thus rendering the LASSO useless. The latter can be seen in Appendix C, Figure C.2.

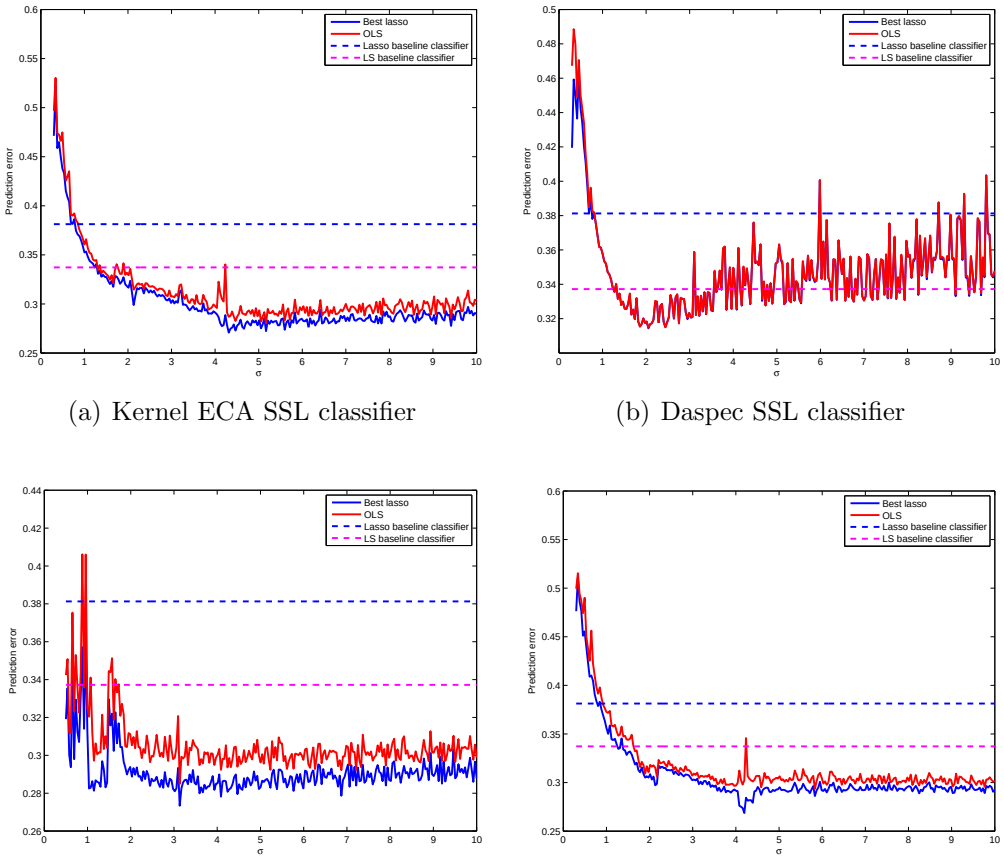
In general the LASSO has more impact in the KECA and KPCA cases for this data set than for the IONOS set, but it still seems to have the greatest general effect on the Laplacian method.

9.1.3 ADULT data set

Classification results for the ADULT data set are displayed in Figure 9.3.

For the KECA classifier results we see that the prediction error varies extremely, sometimes well below the baseline classifiers and sometimes far worse. It is hard to say what causes this, but it is probably related to the knee method selecting very different numbers of eigenvectors for different kernel sizes. See Figure C.3 in Appendix C.

The KPCA classifier does not seem to have the same problems with the extreme variance as the KECA classifier has, which probably is related to



(a) Kernel ECA SSL classifier

(b) Daspec SSL classifier

(c) Laplacian eigenmaps SSL classifier

(d) Kernel PCA SSL classifier

Figure 9.2: PIMA data set: The different SSL classifiers tested over a range of kernel sizes with 20 known points. Lasso classification results: blue, LS classification results: red. Baseline ordinary least squares and lasso classifications with 15 labels are shown in magenta and blue dotted lines.

the difference in the eigenvectors chosen compared to the KECA. Given that we know there are two classes in this data set and assume that they fall under the cluster assumption there should be at least two eigenvectors with a strong response over each of the classes [25]. If these two should happen to be the two top eigenvectors, and the knee method selects 20 eigenvectors as the dimension, it seems probable that the KPCA method chooses eigenvectors with less information than the KECA method which can lead to a non-general classifier or overfitting in the KECA case. In [28] the opposite is proposed; an extra non-informative KPCA eigenvector can lead to overfitting, but with the notions presented in the Theory section regarding the variance being a limited descriptor we leave this question open.

The DaSpec algorithm performs similar as in the IONOS set, a small window of good and low classification errors at small kernel sizes. The opposite is true for the Laplacian method, also similar to the IONOS results, a window of high error rate in the lower kernel size region and stable low error rates as the kernel size increases.

In these results we notice that the LASSO improves the classifications at almost all kernel sizes, except for the DaSpec at larger sizes where again only one eigenvector is chosen.

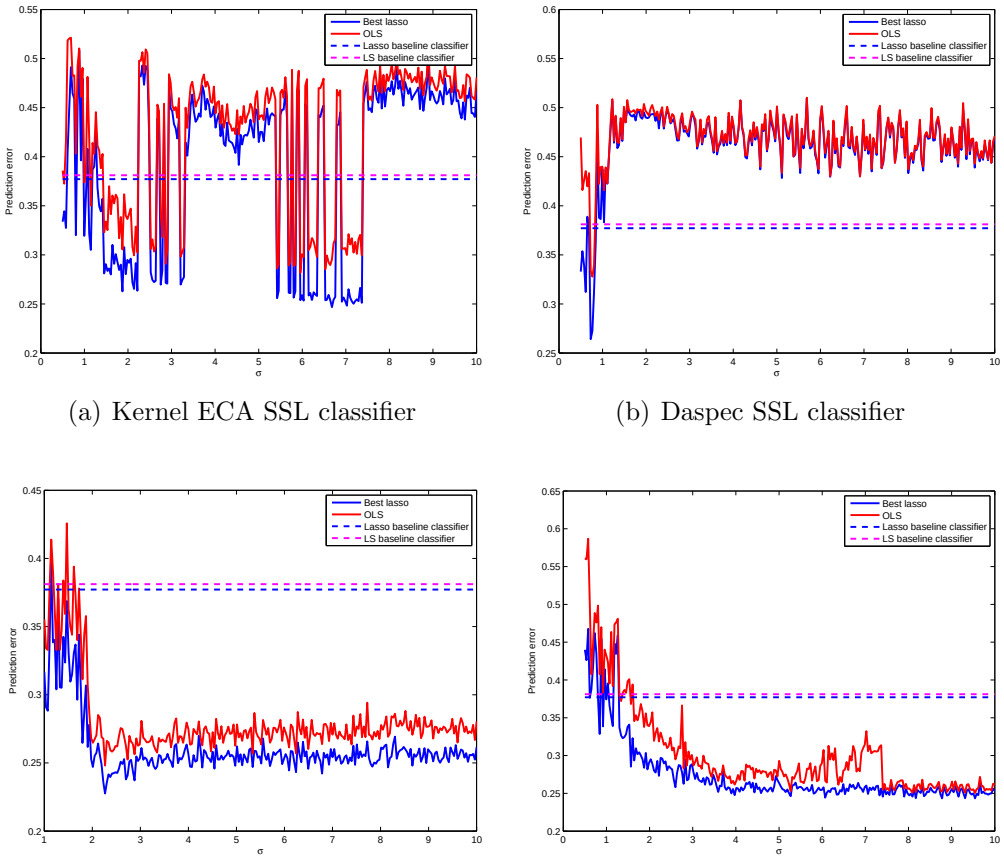
9.1.4 WINE data set

For the wine data set we see that the KECA method works very well for almost all kernel sizes with a result far below the baseline results. In the range from $\sigma \in [0.5, 2]$ we see somewhat higher error rates, but still a good result.

The KPCA also gives good results, best in the kernel size range above 2. For lower kernel sizes we see that the KECA result gives lower prediction error than KPCA, indicating that the KECA has chosen a different set of eigenvectors than KPCA as they have the same number of dimension according to the knee method.

The Laplacian method has good results for kernel sizes greater than approximately 1.5, but in the range below the error rates increase.

DaSpec SSL classification yields a bit higher error than the rest of the meth-



(a) Kernel ECA SSL classifier

(b) Daspec SSL classifier

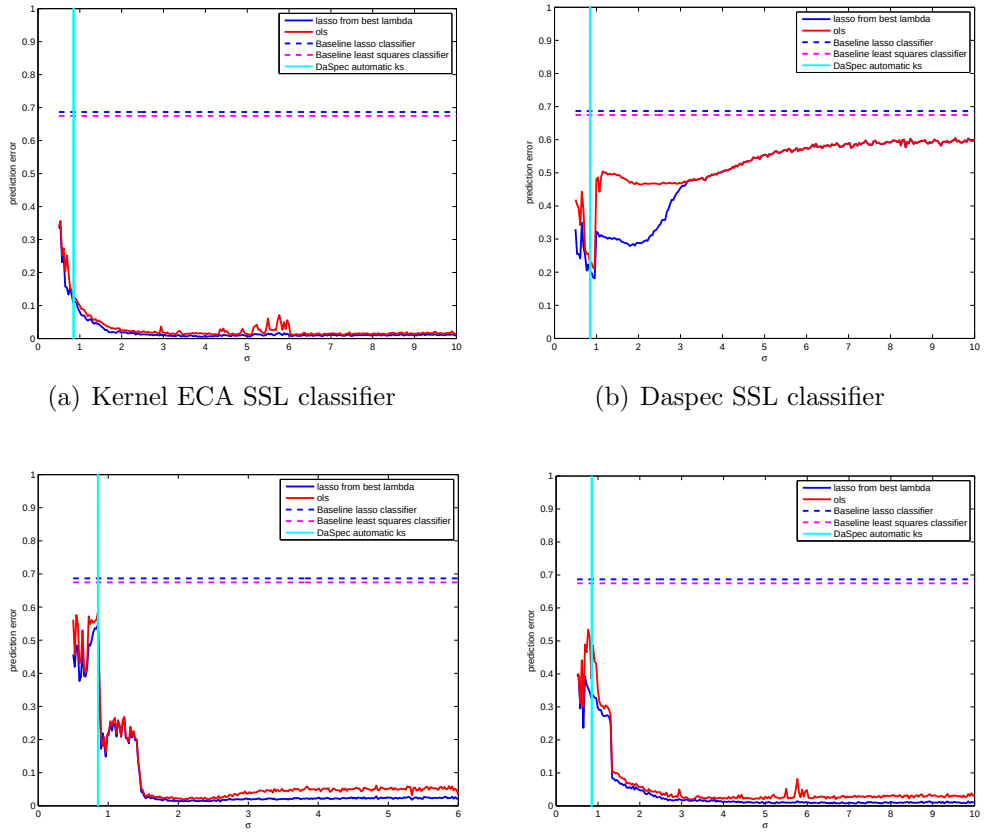
(c) Laplacian eigenmaps SSL classifier

(d) Kernel PCA SSL classifier

Figure 9.3: ADULT data set: The different SSL classifiers tested over a range of kernel sizes with 20 known points. Lasso classification results: blue, LS classification results: red. Baseline ordinary least squares and lasso classifications with 15 labels are shown in magenta and blue dotted lines.

ods and again has a small range of kernel sizes where it works best.

We see that the LASSO improves all classifiers at some points. In the DaSpec results we note a drastic improvement in the kernel size range of approximately 1-3. The KECA and KPCA experiences some spikes in the error rates at $\sigma \in [5, 6]$ removed completely by the LASSO. At kernel sizes greater than around 3, the LASSO helps the overall classification for all kernel sizes for the Laplacian and KPCA methods.



(a) Kernel ECA SSL classifier

(b) Daspec SSL classifier

(c) Laplacian eigenmaps SSL classifier

(d) Kernel PCA SSL classifier

Figure 9.4: WINE data set: The different SSL classifiers tested over a range of kernel sizes with 21 known points. Lasso classification results: blue, LS classification results: red. Baseline ordinary least squares and lasso classifications with 15 labels are shown in magenta and blue dotted lines.

9.1.5 Summary of comparison results

In all data sets we see that the semi- supervised classifiers performs better than the supervised baseline LASSO and LS classifiers. The results are of course very dependent on kernel size, but all methods perform better than the baseline classifiers at some kernel size indicating that using the unlabeled test data has improved the classification.

It also seems that our results directly contradicts the results of [28] regarding the differences between KPCA LASSO and DaSpec LASSO, something that most likely is related to our choice of conducting these experiments over a whole range of kernel sizes. In the figures showing the classification results on the WINE data set, seen in Figure 9.4, we have included kernel size generated by an automatic algorithm presented in [25][2], marked in cyan color, showing that the results from [28] gives good meaning for this particular size, but as far as we can see does not generalize very well.

[2]Also presented quickly in Appendix A.2.

DATA SET	IONOSPHERE			PIMA			ADULT			WINE		
# Labeled points	10	20	30	10	20	50	10	20	50	9	21	30
KECA LASSO SSL	14.1591 ± 4.4101	11.1790 ± 3.0976	11.5349 ±3.0412	28.1595 ±1.6652	27.0833 ±1.5061	25.6437 ±1.1093	25.9396 ±4.1945	24.6755 ±2.0146	23.6696 ±1.4616	0.9272 ±0.7894	0.4928 ±0.4383	0.4167 ±0.5570
KECA LS SSL	16.8580 ± 2.9680	11.6506 ± 2.9779	11.8671 ± 2.9968	29.6168 ±2.9543	28.0357 ±1.3757	26.5719 ±1.3278	31.9570 ±6.6839	28.1595 ±3.7553	25.0605 ±1.5421	1.7881 ±0.8368	0.9855 ±0.6749	0.7576 ±0.6212
DaSpec LASSO SSL	14.2699 ± 3.9857	13.1404 ±2.4250	17.8580 ±8.2279	31.6310 ±0.5297	31.4377 ±1.0195	31.1377 ±0.7142	31.4449 ±12.4556	26.4117 ±8.4032	23.2100 ±3.1594	19.0508 ±3.2800	18.1159 ±2.3409	16.7424 ±2.6847
DaSpec LS SSL	24.7231 ± 8.1066	20.4502 ± 4.6708	21.3173 ± 8.8626	31.6310 ±0.5343	31.4744 ±1.0316	31.1377 ±0.7142	36.4609 ±13.7951	32.7816 ±11.3867	29.6033 ±10.7395	23.2671 ±5.8997	21.3333 ±4.1967	18.5227 ±3.0891
Laplacian LASSO SSL	20.2115 ± 8.6634	13.9871 ±2.6592	10.3093 ± 7.7773	29.1934 ±1.3417	27.3397 ±1.2633	24.5259 ±1.4440	24.1632 ±3.3058	22.7585 ±2.6962	21.0087 ±1.0718	1.8543 ±0.4731	1.2754 ±0.6749	1.2879 ±0.9310
Laplacian LS SSL	22.5579 ± 7.5690	15.0911 ± 3.6092	11.9931 ± 8.5696	30.4456 ±1.6881	28.4844 ±1.6902	25.6287 ±1.2341	25.4092 ±3.4735	24.8053 ±3.7112	24.5041 ±2.7238	1.9868 ±0.5768	1.7971 ±1.1172	2.4621 ±1.8894
KPCA LASSO SSL	11.5408 ± 4.0334	10.2894 ± 3.2921	8.6827 ± 2.4365	28.4848 ±2.4875	26.8681 ±1.8452	25.5339 ±1.5914	24.1975 ±3.1344	24.3370 ±2.5900	23.8921 ±1.6117	1.2583 ± 1.1183	0.6957 ±0.7001	0.5303 ±0.6493
KPCA LS SSL	14.1893 ± 4.9576	10.5252 ± 3.2671	9.2554 ± 2.6108	30.3387 ±1.2856	29.3407 ±1.0409	26.7515 ±2.1530	25.1463 ±2.8175	25.0533 ±2.4434	24.1993 ±1.9097	2.1854 0.8894±	1.9130 ±1.0808	1.4394 ±0.8405
Baseline LASSO	32.5904 ±7.6311	27.7039 ±6.5315	25.3479 ±4.8720	42.1680 ±11.9576	38.1283 ±8.2688	34.2433 ±4.4171	41.3147 ±11.7990	37.7138 ±5.7960	34.3044 ±2.9964	64.6943 ±7.5527	68.6624 ±12.7404	64.6943 ±13.5477
Baseline LS	40.5572 ± 9.2249	35.8610 ±6.6248	38.2347 ± 8.2710	41.4468 ±6.5306	33.7210 ± 3.3764	29.7400 ±2.0667	46.1331 ±6.8679	38.1139 ±5.4660	33.1232 ±3.7643	65.9566 ±3.8150	67.4522 ±5.3770	65.9566 ±8.3548

Table 9.1: Prediction error(in %) and standard deviation for several UCI data sets.

9.2 Influence of adding unlabeled data

In the previous section we saw that by including the unlabeled test points in the data transformations the LASSO SSL classifiers outperformed the baseline classifiers with at least 10% in all cases, and sometimes much better. Now we test what effect adding different amounts of unlabeled points in the transformation has.

Most of the data sets in the previous results are quite small so we introduce a new data set, the handwritten PEN digits, also from [1], consisting of handwritten digits 1-9. Besides being a larger data set than the previous ones, this set is also useful because of the fact that because the classes in the set consists of handwritten digits it is natural to assume that they fulfill the cluster assumption, and that adding unlabeled data would improve the 'capturing' of these clusters in the kernel matrix of the transformation.

To reduce computation time we use an automatic kernel size algorithm from [25][3], 20 known points and 4 eigenvectors by the 20% rule from [4]. By doing this we might not get the optimal results classification wise, but we are mostly interested in showing the effects of using different amounts of unlabeled data. To get the full effect of adding unlabeled points we average over 200 repetitions drawing $30+N_{unknown}$ points from the dataset each time. The classification error is calculated and in Figure 9.5 the results are shown as a function of the number of unlabeled points included. We note that the number of unlabeled points are also the test points, so the test sets used vary in size accordingly.

In Figure 9.5 we see that adding more unlabeled data in the KECA LASSO SSL classifier improves the classification. From only 100 labeled points to full kernel matrix we see an improvement of about 3%. Compared to the baseline LASSO classifier which has a classification error of $\sim 60\%$ shown below, the improvement is drastic.

Baseline LASSO classifier with 50 known points from the PEN data set:

Error: 0.6482
Standard deviation: 0.0018

We also classify the 'two moons' data set, as seen in the introduction in Figure

[3]explained briefly in Appendix A.2.

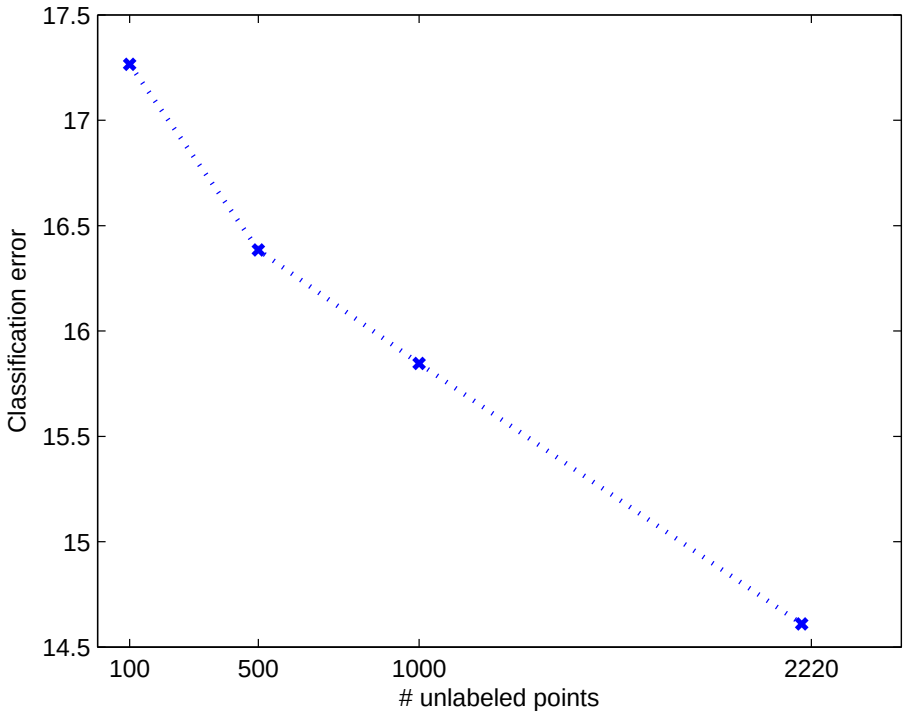


Figure 9.5: The PEN digits data set with the numbers 4, 5 and 6 classified with 100, 500, 1000 and 2220 unlabeled test points and 20 labeled points. 2220 unknown points equals full kernel matrix.

1.1, including different numbers of unlabeled points in the kernel matrix. The set consist of a total of 2500 data points. The same motivation as for the PEN digits set holds here, perhaps even stronger: the set is clearly divided into two clusters and the idea of including the unlabeled data should help the classifier represent the data better. In Figure 9.6 we see the effects of adding unlabeled data when classifying the set with 50 and 200 labeled points using the KECA SSL LASSO classifier.

The baseline LASSO classifier gives the following results:

50 known points:
Error: 0.1501
Standard deviation: 0.0283

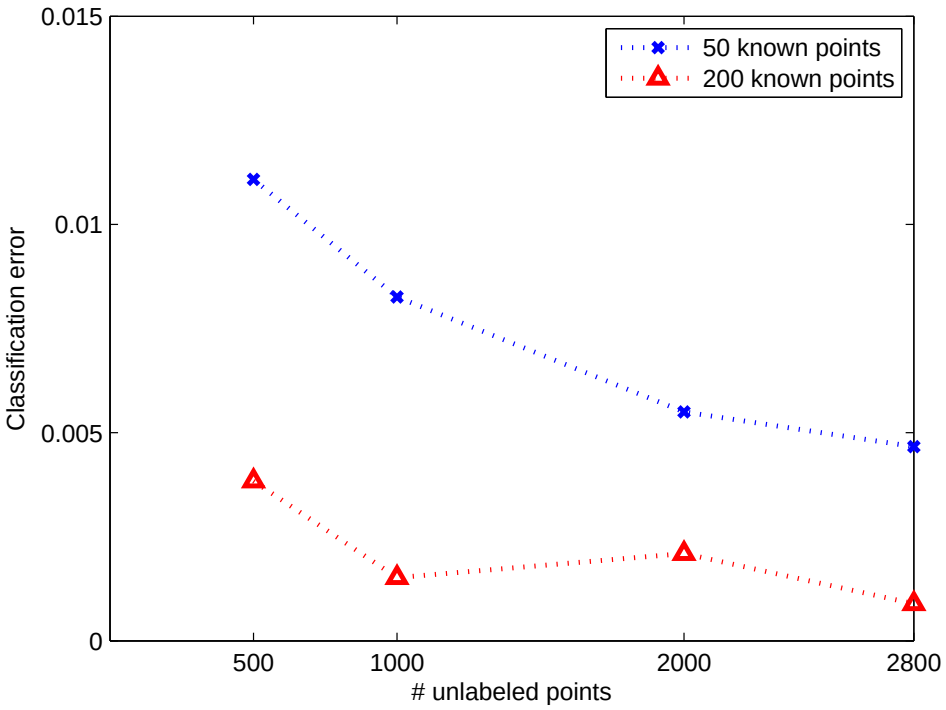


Figure 9.6: Two moons data set classified using 50 and 200 labeled points and varying amounts of unlabeled data.

200 known points:
Error: 0.1366
Standard deviation: 0.0167

In this example we also see the effects of adding more unlabeled data, albeit on a much smaller scale. We also notice that the error rises a bit with 2000 unlabeled compared to 1000 unlabeled data points, but we address this as statistical uncertainty as the scale already is quite small.

9.3 KECA vs KPCA

In this section we look at the differences when using KECA and KPCA to do the transformations in the SSL algorithms as they are closely related. We remember that the only practical difference is the choice of eigenvectors from the kernel matrix of the input data. To simplify things and avoid confusion regarding the number of dimensions chosen we use the same number of eigenvectors used with the Laplacian SSL LASSO classifier in the previous section, 20% of the number of known points. Upon manual inspection during the testing we have noted that the IONOS data set in some cases favor the KPCA LASSO SSL classifier and in some cases the KECA LASSO SSL classifier. Because of this we choose examples from the IONOS set as the basis for the comparisons. In Figure 9.7 we see the classification results over a range of kernels with 30 labeled points and 6 eigenvectors and we see that for some kernel sizes there are some significant differences. The KECA performs best in the area of $\sigma \in [0, 2]$ and the KPCA performs better for $\sigma \in [5, 7]$.

To compare the methods we manually choose a few kernel sizes where the differences are notable and compare the eigenvector choices and the coefficients chosen by the LASSO and least squares classifiers.

The first value we chose is 6.1466 which gives the classification errors shown in Table 9.2.

	KECA	KPCA
Error	0.2736	0.0587
Standard deviation	0.0172	0.0126

Table 9.2: LASSO SSL classification errors with 30 known points drawn at random, averaged over 20 iterations.

In this case the KPCA SSL LASSO classifier outperforms the KECA SSL LASSO classifier in terms of lower prediction error. For each calculation of classification error we have 20 sets of coefficients for both the LASSO and the LS. The LASSO coefficients are the ones giving the lowest classification error in each case. In Table 9.3 and 9.4 we see the coefficients averaged over the 20 sets.

In the tables 9.3 and 9.4 we see that for the KPCA both the LASSO and the

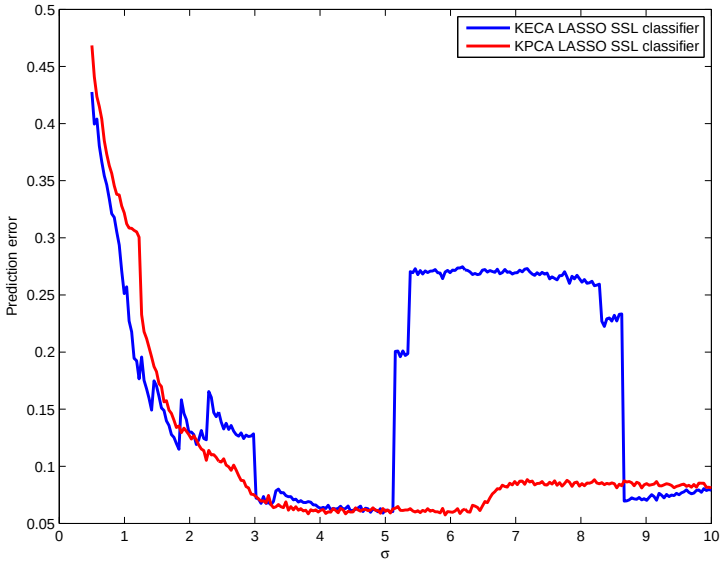


Figure 9.7: The IONOS data set classified using both the KECA and the KPCA LASSO SSL classifiers with 30 known points and 6 eigenvectors.

KECA dimension	KECA coeffs	KPCA dimension	KPCA coeffs
1	-0.5670	1	-0.5475
2	0.0081	2	-0.0521
9	-0.0314	3	0.0679
10	0.0021	4	0.5423
11	0.0020	5	0.1397
12	0.0435	6	-0.2384

Table 9.3: Least squares coefficients

KECA dimension	KECA coeffs	KPCA dimension	KPCA coeffs
1	-0.4809	1	-0.4742
2	-0.0055	2	-0.0223
9	-0.0074	3	0.0363
10	0.0125	4	0.4786
11	0.0060	5	0.0859
12	0.0309	6	-0.1748

Table 9.4: LASSO coefficients

LS places a significant weight on the 4th eigenvector, an eigenvector that is not chosen at all with the KECA. The KECA coefficients focus mainly on the first eigenvector, in practice almost creating a one dimensional classifier on the data projected to the first principal axis of the feature space. This illustrates that sometimes the KECA transformation can choose eigenvectors that does not contribute well in a linear classification setting based on least squares.

To compare the other way around, where the KECA SSL LASSO outperforms the KPCA SSL LASSO, we use the same experiment as the previous results and chose a kernel size of 1.8353 which gives the classification errors shown in Table 9.5.

	KECA	KPCA
Error	0.1149	0.1353
Standard deviation	0.0240	0.0176

Table 9.5: LASSO SSL classification errors with 30 known points drawn at random, averaged over 20 iterations.

In this case the KECA performs best and we do a similar analysis as when the KPCA performed better. The averaged least squares and best LASSO coefficients can be seen in tables 9.6 and 9.7.

KECA dimension	KECA coeffs	KPCA dimension	KPCA coeffs
1	0.8398	1	-0.8071
3	-0.3315	2	0.1947
2	0.2000	3	-0.3253
6	0.5328	4	0.4761
8	0.1778	5	0.0049
4	0.4401	6	0.5258

Table 9.6: Least squares coefficients

The first thing we notice here is that the weights are generally higher in this case, i.e. all the 6 dimensions contribute to the classification in the KECA case. For the KPCA the 5th dimension is weighted down almost completely, whilst the 5th largest entropy component, corresponding to the 8th KPCA eigenvector, has a significant weight and thus also an impact on the classifier.

KECA dimension	KECA coeffs	KPCA dimension	KPCA coeffs
1	0.7758	1	-0.7321
3	-0.2624	2	0.1422
2	0.1565	3	-0.2484
6	0.4820	4	0.4087
8	0.1480	5	0.0070
4	0.3782	6	0.4560

Table 9.7: LASSO coefficients

In this case the LASSO agrees with the eigenvectors chose by the KECA, indicating that the KECA chosen axes presents a good subset selection which in combination with a small shrinkage of all coefficients gives the best result with the chosen kernel size. For the KPCA we see that the 5th eigenvector has been almost zeroed out, and in addition to the rest of the somewhat shrunk coefficients the KPCA is left with a subset with a slightly higher prediction error.

To summarize; the differences between the KECA and KPCA LASSO SSL algorithms comes as expected from the differences in choices of eigenvectors. The LASSO itself is of course very sensitive to the choice of eigenvectors so it will be helpless if the choice of eigenvectors does not contain good classification properties at all.

The last thing we need to note in this case are the different choices of kernel sizes, 1.8353 and 6.1466. In practice there exist only heuristic methods for choosing kernel size and calculating the kernel size based on a selection of methods presented in [15] and [25] gives:

Method	Kernel size
20% of median*:	1.5572
20% of mean*:	1.5039
20% of total range*:	3.6906
Data Spectroscopy kernel size algorithm:	1.3444

* of total euclidean distance range between all input points.

Table 9.8: Different heuristic kernel sizes calculated for the standardized IONOS data set.

So in practice we see that the kernel size favoring KECA is the closest to the sizes created by the heuristic methods implying that practical methods often will be worse than the best results for each method shown in this thesis.

9.3.1 Comments to differences between KECA and KPCA shown in Theory section

In [15] as well as our illustrations in Chapter 4 we have seen examples where the KECA transformation gives distinctly different results than the KPCA transformation. In many cases, especially the ones where the KECA results in a very reasonable angular structure, we have inferred that the KECA transformation reveals more information about the data than the KPCA transformation. A clustering algorithm based on a cosine cost function after KECA transformations was presented in [15] using this intuition with good results. From the results in Section 9.1 we see that when we use the LASSO and least squares classifiers, these differences does not seem to come in to play as significantly as expected. In many cases we see that the KECA performs better at smaller kernel sizes[4], but in almost all cases the best results occurs at larger kernel sizes where the KPCA and KECA yields the same transformations.

This is further confirmed in Appendix B where we have classified the same data sets as in the previous section using both algorithms with a fixed number of eigenvectors equal to the number of classes, the Laplacian 20% method and a fixed number of 20 dimensions (mainly to let the LASSO decide which eigenvectors to use). We chose two ways of doing this. First, to illustrate the tendency of the best LASSO results coming at larger kernel sizes and creating almost equal reults for KPCA and KECA, we use the same kernel size range as in the examples in Section 9.1, $\sigma \in [0, 10]$. In this case we use 20, 30 and 50 labeled points. Second, we test the two methods with the different heuristic kernel size methods presented in the previous section with 50 known points. The best results with full kernel size range are presented in tables B.1, B.2 and B.3. The results from the heuristic kernel size methods are presented in tables B.4, B.5, B.6 and B.7.

When using eigenvectors equal to the number of classes and full kernel size

[4] E.g. Figure 9.4 (a) and (c).

range, Table B.1, KPCA and KECA basically gives the same best LASSO results, except for the ADULT set where the KECA does slightly worse. Almost the exact same can be seen for the Laplacian 20% method with full kernel range, Table B.2: KECA and KPCA has the same best LASSO results, except for the ADULT set. With both these methods we also note that KECA does better on the IONOS set when there are only 10 labeled points. When using 20 eigenvectors and full kernel range, Table B.3, we get the same results, almost all best results are equal, but now the ADULT set is as good for the KECA as for KPCA. In this case we would expect the results to be very similar as we have a total of 20 eigenvectors, and with so many vectors we assume that both KPCA and KECA have chosen, if not all, at least very many of the same vectors.

The results using the heuristic kernel size methods give a different picture. For the WINE data set, Table B.7, the KECA gives consistently better results except for the largest kernel size (15% of the total median range) and when 20 eigenvectors are used. The ADULT and PIMA set, tables B.5 and B.6, yields variying results, but mostly the same results for KPCA and KECA. The IONOS set, Table B.4, behaves much like the wine set, different results for the smaller kernel sizes, and equal results at the larger kernel sizes and when 20 eigenvectors are used.

To summarize this section we see that the differences between KECA and KPCA as presented in Section 4.3.1 and [15] does not seem to be taken full advantage of when using the LASSO or least squares classifiers. We also note that in general the best results using the LASSO and all four transformations, DaSpec, Laplacian eigenmaps, KPCA and KECA, comes from KECA and KPCA at higher kernel sizes where the chosen eigenvectors are mostly the same.

In the next section we will further investigate the LASSO and compare cases where the KECA SSL LASSO and KECA SSL LS perform differently.

9.4 Effects of the LASSO

In this section we will investigate further how the LASSO influences and works together with the KECA transformation in the KECA LASSO SSL

classifier. We choose the PIMA data set as we can observe a significant difference between the LASSO and least squares classification in Figure 9.2 (a) with a kernel size in the range ~ 4 and redo the same experiment using only the KECA classifier collecting all LASSO coefficients and errors. To get a better overview over the individual LASSO coefficient sets we reduce the averaging from 20 to 3 for each kernel size, the classification result is shown in Figure 9.8.

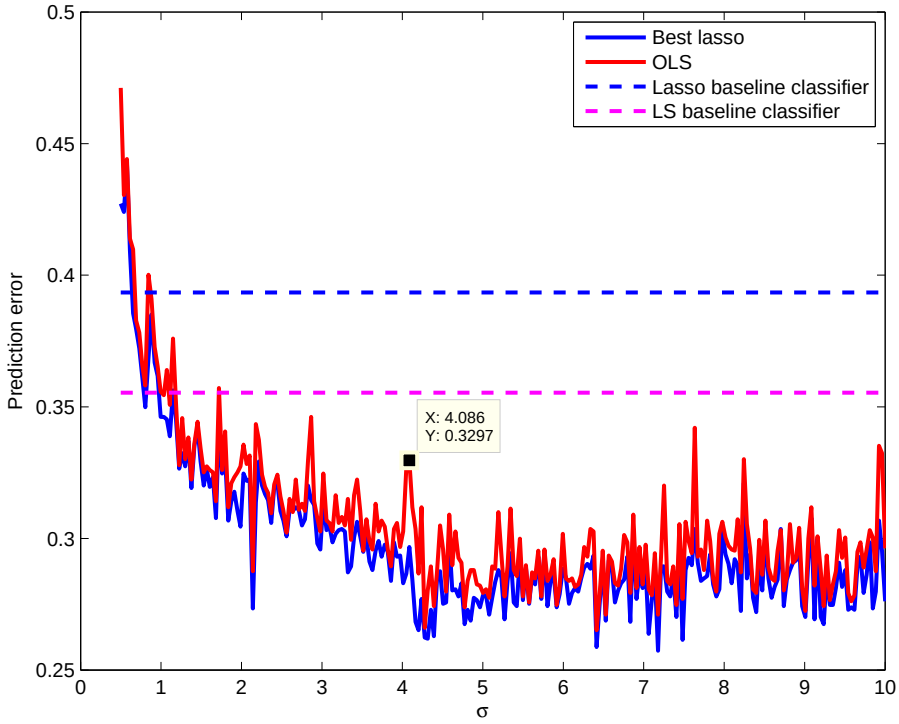


Figure 9.8: Classification error with kernel ECA, 20 known points and 748 unknown points over a range of kernel sizes.

From closer inspection of the classification results we see that a kernel size of 4.086, marked with a data label in Figure 9.8, gives a large difference between the LASSO and the least squares result:

LASSO: 29.67%
Least squares: 32.97%

We now take a look at how the shrinkage of the LASSO interacts with the eigenvectors chosen by the KECA transformation. In Figure 9.9 we see the coefficients of the LASSO classifier with the chosen kernel size as a function of the logarithm of shrinkage parameter λ. The coefficient sets with best prediction errors are marked by thin red lines and can also be seen in Table 9.9.

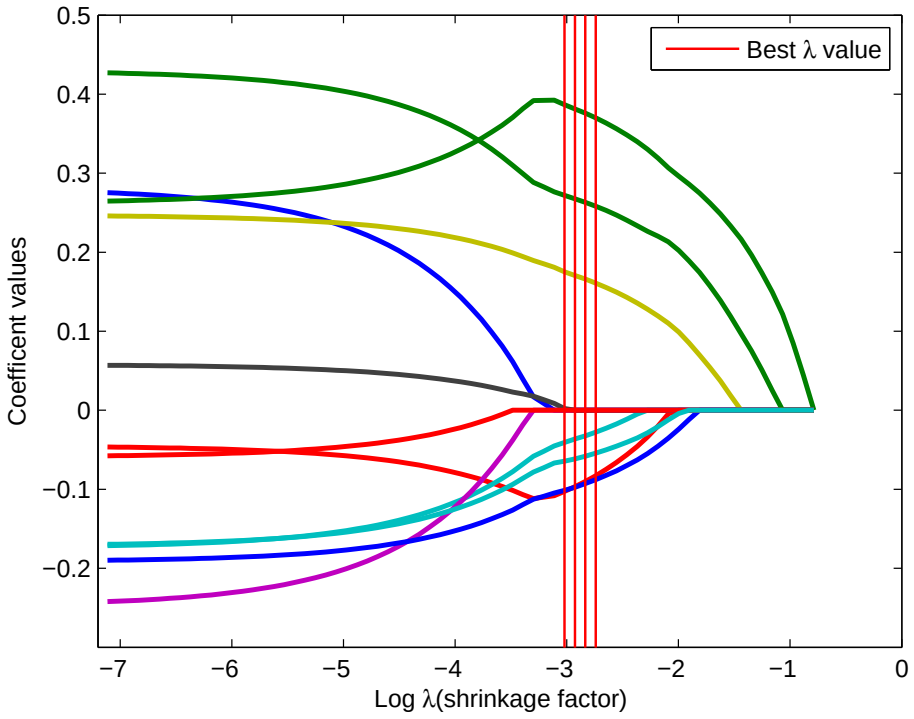


Figure 9.9: The coefficents of the LASSO

We also include a stem plot of the coefficients as function of the KECA eigenvector index seen in Figure 9.10.

From Table 9.9 we see that the LASSO favors eigenvectors 2, 7 and 9. We also note that four of the dimensions are set to zero and that dimensions 4,5,6 and 12, seen in Figure 9.10 with values below zero, have been shrunk quite a bit.

In the case of the LASSO used together with the data transformation it is

λ	0.065	0.059	0.054	0.049
KECA eigenvectors				
1	0	0	0	0
2	0.370	0.376	0.381	0.387
4	-0.083	-0.090	-0.097	-0.103
5	-0.028	-0.032	-0.036	-0.041
10	0	0	0	0
7	0.161	0.166	0.171	0.176
8	0	0	0	0.002
6	-0.088	-0.093	-0.097	-0.101
9	0.258	0.263	0.268	0.272
16	0	0	0	0
12	-0.054	-0.058	-0.062	-0.064

Table 9.9: The LASSO coefficients and KECA dimensions at three different shrinkage factors which gives the best classification error.

most intuitive to focus on the subset selection operator part of the LASSO as all correlation already has been removed via projecting onto the orthogonal principal axes of the feature space thus reducing the possible effects of the shrinkage [8], [31], [9]. But as we have seen in most examples presented in this thesis all LASSO results showing better results than the least squares without any dimensions removed have lower coefficients indicating shrinkage. We will only treat this as an empirical observation and not analyze it further.

The same effects can be seen in a case with the IONOS data set where we test the KECA SSL classifier over a range of kernel values with 20 known points and all other used as unlabeled test points. We now include 10 eigenvectors to show that the LASSO can act as a practical subset selection method. The test reveals a significant improvement in classification error when using the LASSO at the kernel size 2.2932:

LASSO: 6.43%
Least squares: 16.08%

There are two sets of coefficients that gives this result shown in Table 9.10. We have chosen to include the least squares coefficients in the table to get insight in both the subset selection by setting coefficients to zero and the

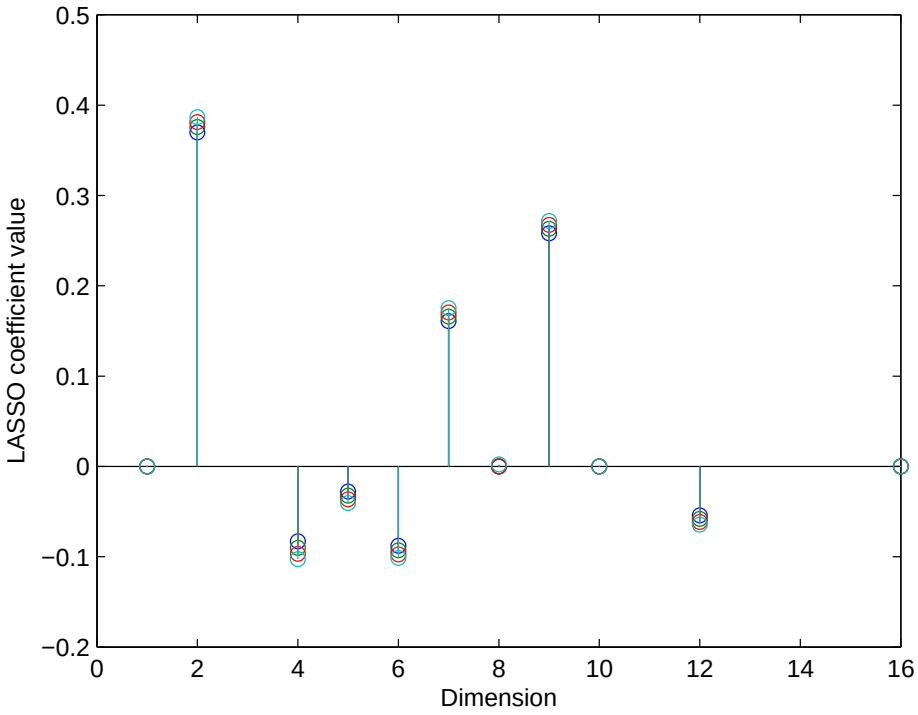


Figure 9.10: The value of the best LASSO coefficients at each KECA dimension for the PIMA dataset with a kernel size of 4.086 and 20 labeled points.

shrinkage of the remaining coefficients compared to the original least squares result.

Figure 9.11 shows the coefficients as a function of the logarithm of the shrinkage factor λ.

In this case we see that the LASSO has kept all, but two dimensions, the 7th and 13th eigenvectors. We also in Figure 9.11 combined with Table 9.10 see that all coefficients except the first and third have been shrunk quite significantly. This we can interpret as the KECA SSL algorithm have taken full advantage of the LASSO, with both subset selection and shrinkage of coefficients yielding a lower classification error.

To get a more general idea of how the LASSO interacts with the semi-

KECA eigenvector index	LS coeffs	LASSO coeffs	LASSO coeffs
1	0.9816	0.8969	0.9040
2	0.6693	0.2816	0.2982
3	-0.5722	-0.5204	-0.5239
5	-1.2779	-0.2805	-0.3473
7	0.3313	0	0
4	1.2222	0.1706	0.2560
9	-0.7303	0.0975	0.0314
8	-0.4274	-0.2540	-0.2752
13	0.3474	0	0
6	-0.5015	-0.2886	-0.3169

Table 9.10: LASSO and LS coefficient for KECA SSL LASSO algorithm on the IONOS dataset with 10 eigenvectors.

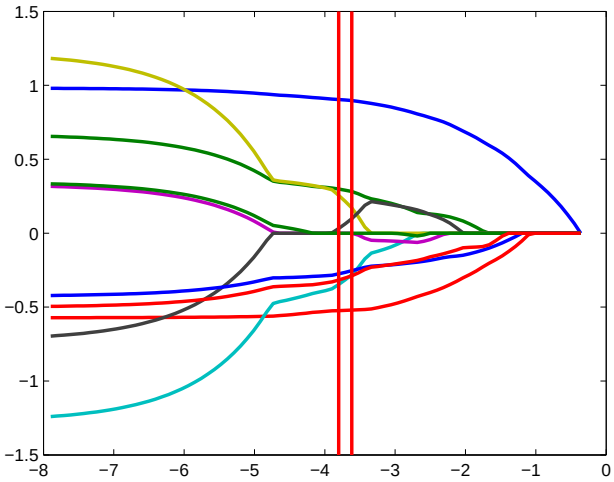


Figure 9.11: Lasso coefficients for the KECA LASSO SSL classifier with 20 random points acting as labels. The coefficients that give best prediction error are marked with red vertical lines.

supervised algorithms we note how many dimensions the LASSO sets to zero in each case in the experiments in Section 9.1. In Figure 9.12 we see the dimension of the algorithm and the average sparsity[5] of the best LASSO results for the particular PIMA example used previously to show the LASSO interaction. In the experiments we used the so called 'knee method' for choosing the number of eigenvectors for the KECA and KPCA SSL LASSO algorithms and we clearly see that the number of eigenvectors chosen by this method varies much, but that the LASSO in most cases suppress much of the extreme dimension choices. Figure 9.13 shows the same results, knee dimension choice for the KECA method and average sparsity, for the data sets IONOS, PIMA and ADULT with KPCA LASSO SSL included for the IONOS set. Plots of the dimension of all methods for the datasets, IONOS, PIMA and ADULT, can be seen in Appendix C confirming the LASSO behavior.

We also remember the very strange result with the KECA classifier and the ADULT data set in Figure 9.3 (a), and note that in very many of the kernel sizes the results from the LASSO showed significant improvement over the LS version. In (c) of Figure 9.13 we see that in many cases the dimension is very high, but the LASSO sparsity forces the dimension down keeping the best subsets of eigenvectors.

[5]The number of dimensions set to 0.

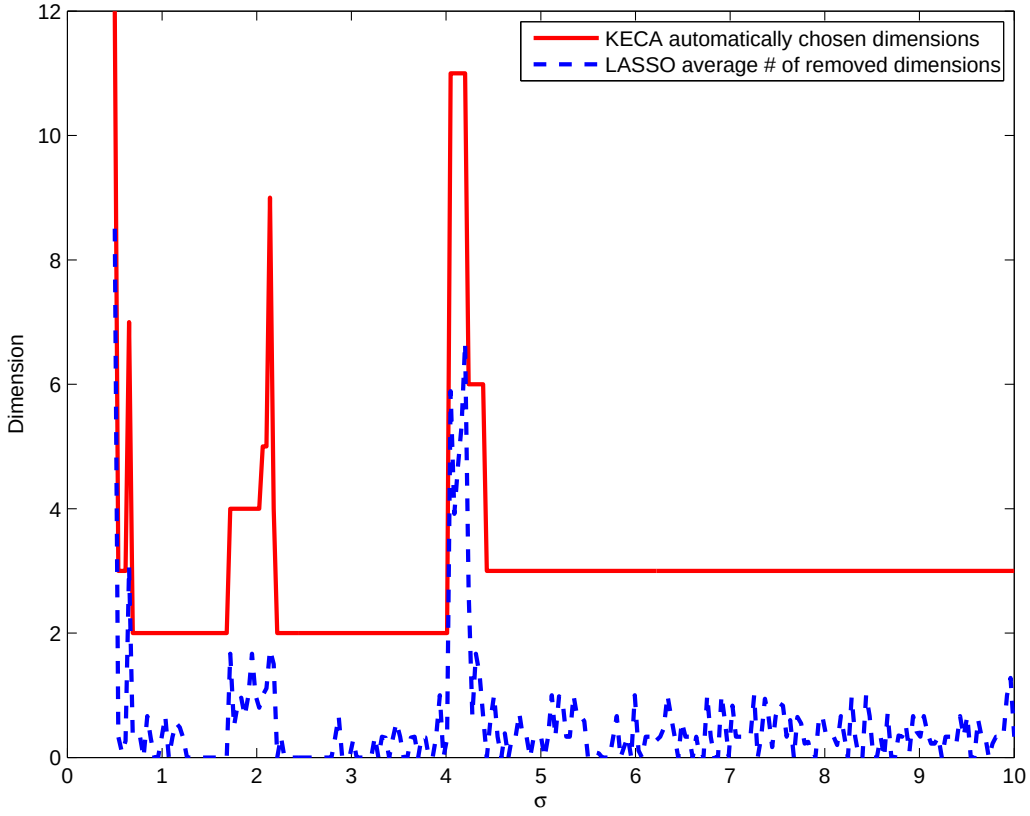
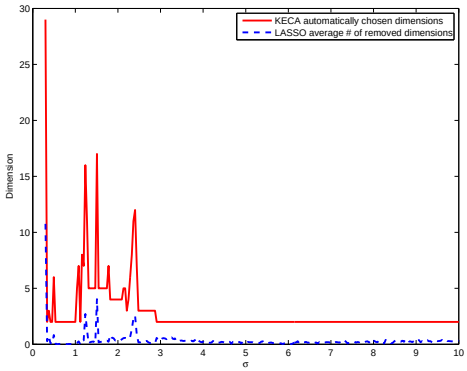
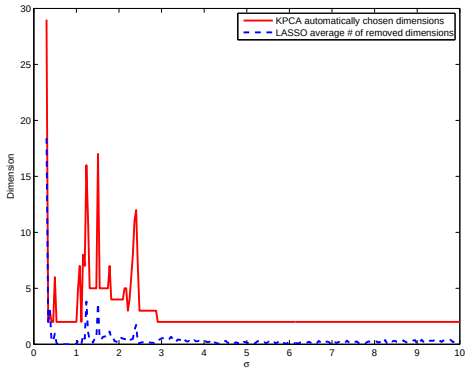


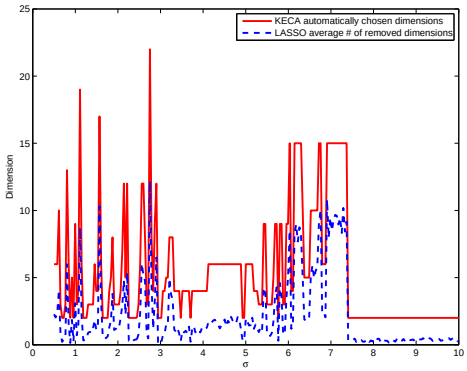
Figure 9.12: The dimension of the KECA transformation as chosen by the 'knee method' and the average number of dimensions set to zero by the LASSO.



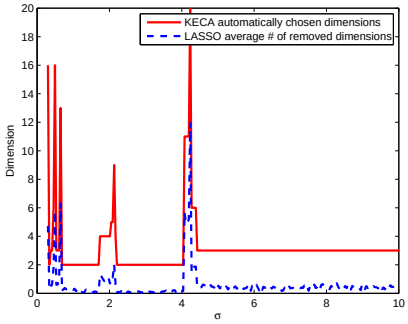
(a) IONOS KECA dimensions and LASSO sparsity



(b) IONOS KPCA dimensions and LASSO sparsity



(c) ADULT KECA dimensions and LASSO sparsity



(d) PIMA KECA dimensions and LASSO sparsity

Figure 9.13: Knee method-chosen dimension and average sparsity of the best LASSO solutions.

9.5 The Frey faces

We conclude the results by doing a simple real world example using the so called *Frey faces* data set[6]. In [15] this data set was clustered using a kernel ECA based clustering algorithm with good results using a median based kernel size, taking a kernel size corresponding to $10 - 20\%$ of the median of the euclidean distance between all points. We experimented with different choices within this kernel size and after standardizing the data set we settled for kernel size $\sigma \approx 15$. This actually corresponds to closer to 50% of the median range, but as we will see it gave reasonable results and this is just an example for illustrational purposes so we choose to leave it as is.

As we have no prior knowledge of the data we have to manually create labels. We chose 10 smiling faces and 10 'sour' faces as our labeled points and opt for a binary classification dividing the data set in a smiling set and a 'sour' face set.

(a) The smiling class (b) The sour class

Figure 9.14: The two sets of labeled training points from the Frey faces data set.

We start by classifying the whole set using the baseline ordinary least squares classifier. The classification results are shown in figures 9.15 and 9.16, with the figures showing the happy class and the sour class respectively. Because of the size of the images, they will extend beyond the margins of the pages on which they are displayed, but reducing the size further would make them very hard to read so we chose to leave them as is.

[6] http://cs.nyu.edu/~roweis/data.html.

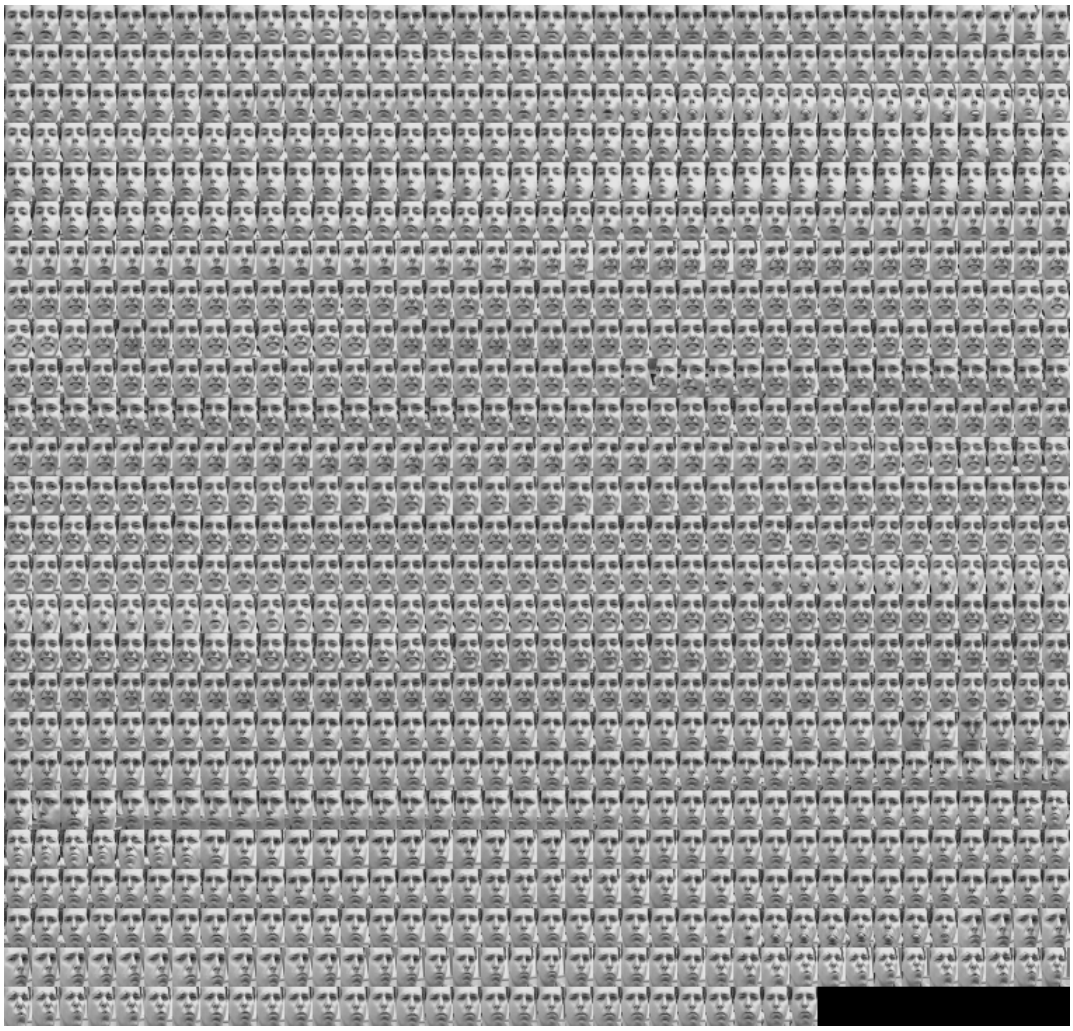


Figure 9.15: The happy faces class as predicted by the baseline least squares classifier.

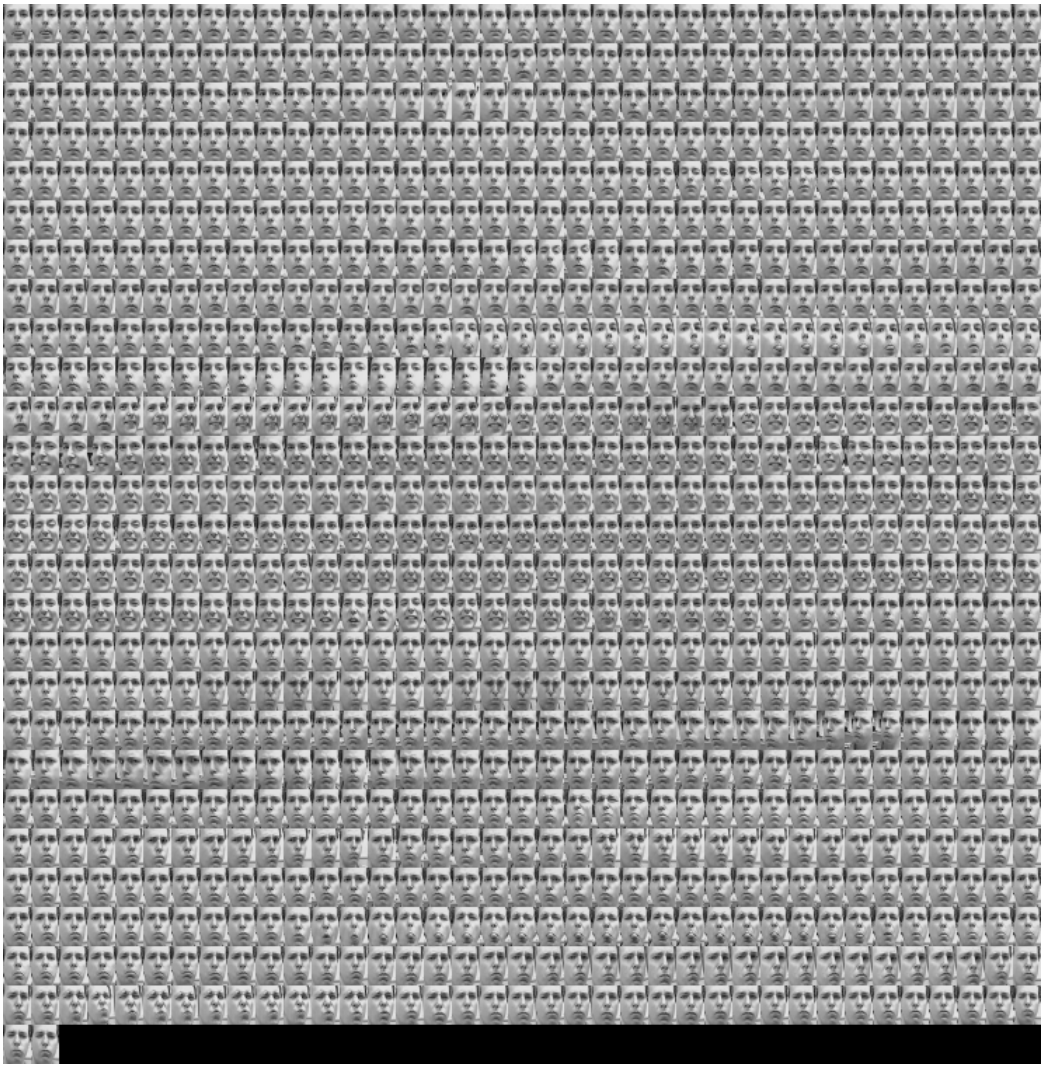


Figure 9.16: The sour faces class as predicted by the baseline least squares classifier.

As we see in figures 9.15 and 9.16 the ordinary least squares classifier does not seem to work very well. There are many smiling and many sour faces in both images, indicating that our goal of classifying the set into two distinct classes have failed.

Now we use the KECA LS SSL classifier with 4 eigenvector corresponding to the 20% rule in [4] and include all images in the kernel matrix. The classification results are shown in figures 9.17 and 9.18, and we note that the KECA algorithm chose eigenvectors 1, 6, 2 and 4 of the kernel matrix.

We note a clear improvement when using the KECA LS SSL classifier. Especially the 'sour' face class seem to be well separated as the image almost exclusively contains frowning and more or less serious faces. The smiling class does also represent a reasonable structure as it seems to have a surplus of smiling faces.

The last thing we do is test the KECA LASSO SSL classifier with the same kernel size and number of eigenvectors as before. As we have no labels we need to find a proper shrinkage factor λ to use with the LASSO. As it turns out the 'glmnet' software package we have used to calculate the LASSO coefficients and shrinkage has a built in cross-validation function. So we perform a leave-one-out, or 20-fold in this case, cross-validation, [8], and get $\lambda = 0.0547$. Using this value we get the classification results shown in figures 9.19 and 9.20.

Also in the case of KECA SSL LASSO we get much more reasonable results compared to the baseline classifiers.

Without true labels we cannot calculate prediction errors, so we will not analyze the figures or differences between the LS SSL and LASSO SSL classifiers any further except noting that in the LASSO case (almost) removed two dimensions, seen in Table 9.11.

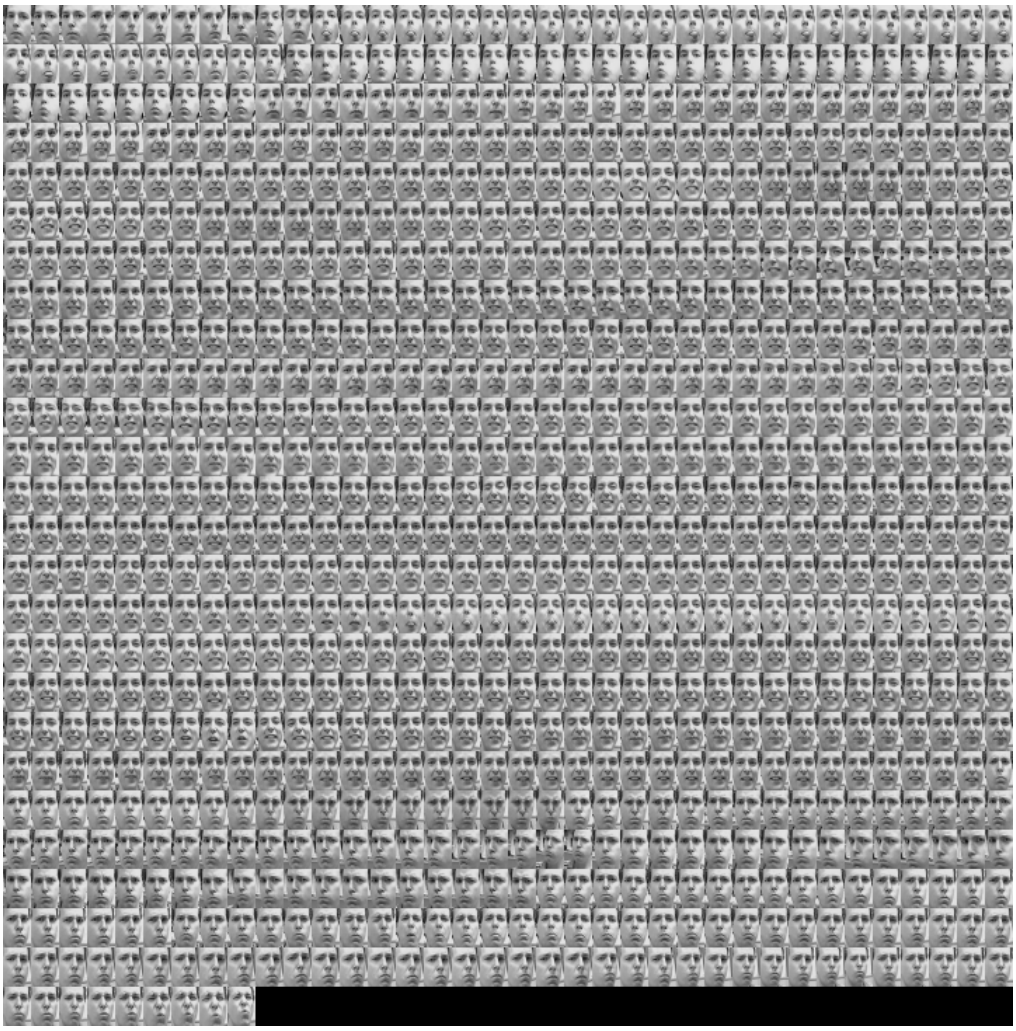


Figure 9.17: The happy faces class as predicted by the KECA LS SSL classifier.

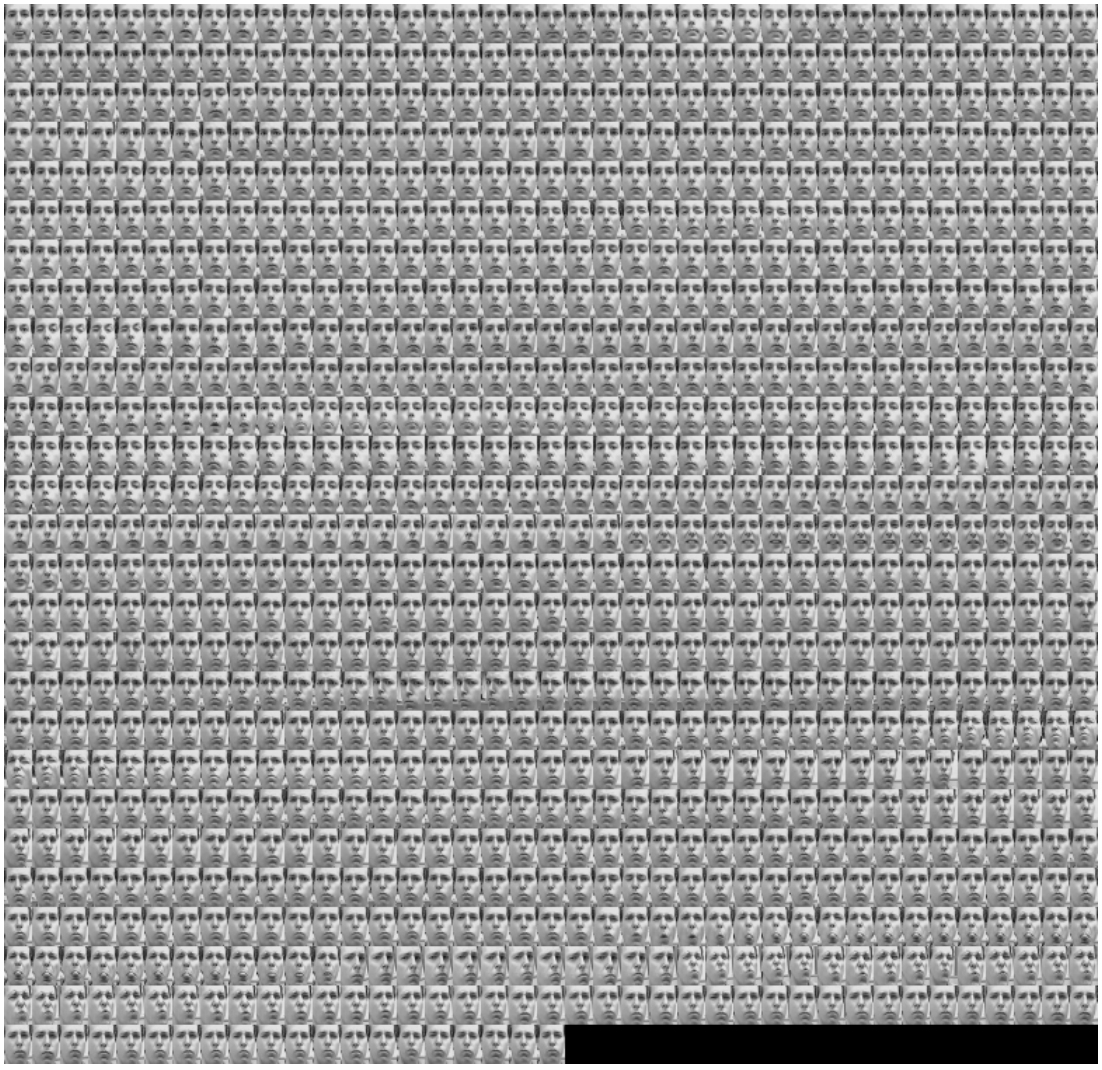


Figure 9.18: The sour faces class as predicted by the KECA LS SSL classifier.

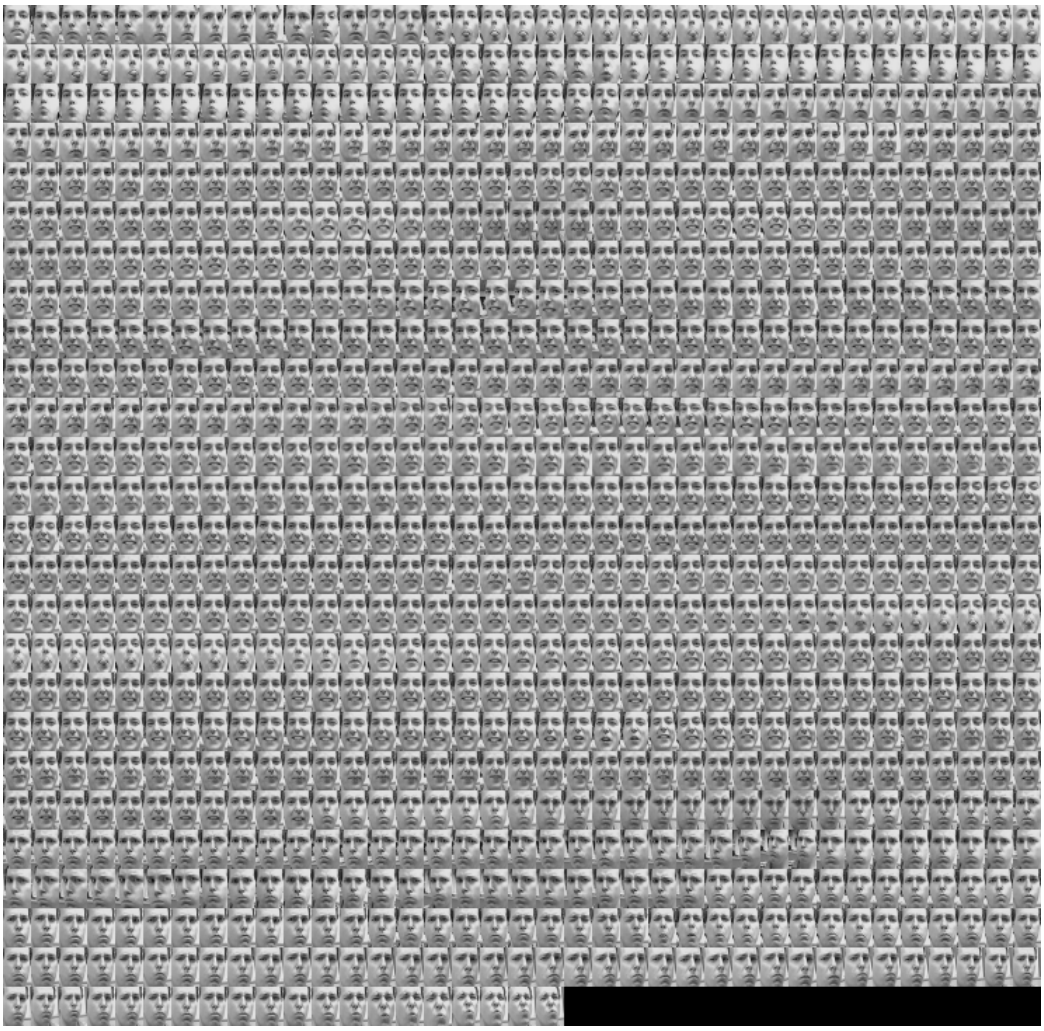


Figure 9.19: The happy faces class as predicted by the KECA LASSO SSL classifier.

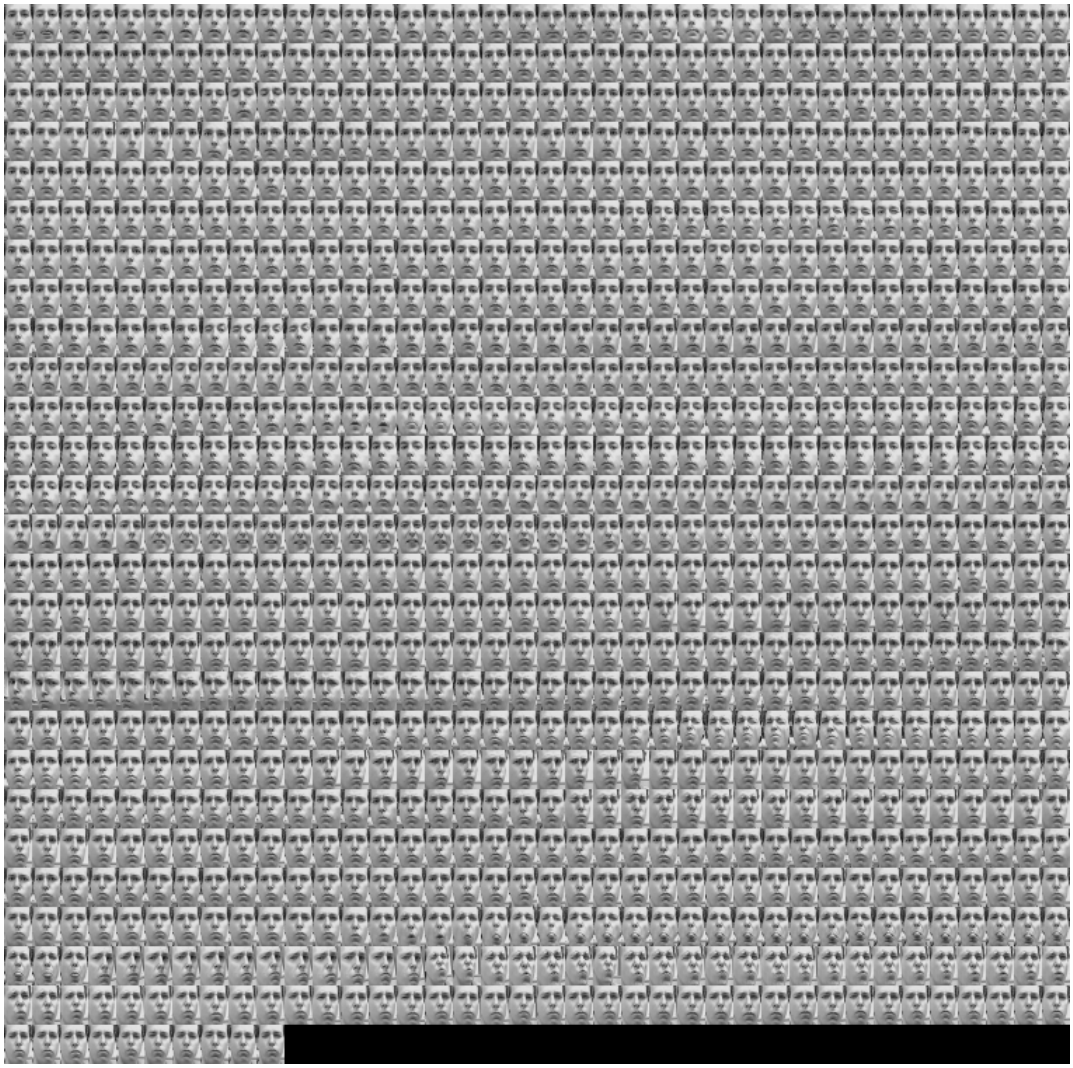


Figure 9.20: The sour faces class as predicted by the KECA LASSO SSL classifier.

KECA dimensions	LASSO coeffs	LS coeffs
1	-2.4597	-2.9899
6	0.0522	0.5413
2	-2.8245	-2.9171
4	0	0.4161

Table 9.11: Coefficients of the LASSO SSL and LS SSL algorithms for the Frey faces.

Chapter 10

Conclusion

In this thesis we have presented a new semi-supervised classifier combining the Kernel Entropy Component Analysis transformation developed at the University of Tromsø with the linear LASSO classifier. The results show that the classifier is comparable and in many cases even better than the benchmark work in [28] and [4]. This method is a further development of the semi-supervised classifier presented in [16] using KECA and least squares, and we have shown that the LASSO almost exclusively gives better classification results.

We have also shed some light on how the LASSO influences the Kernel ECA transformation and shown that both properties of the LASSO comes into play when used in practice: subset selection of the eigenvectors of the kernel matrix and shrinkage of the coefficients.

In the next section we summarize some of the most important observations made in this thesis.

10.0.1 Important observations

- When there is difference between the KECA and KPCA LASSO SSL classifiers the difference mostly consist of the eigenvector choices.
- Significant improvement between LASSO and LS results are often due to subset selection. In most cases the LASSO has removed some of the

eigenvectors.

- LASSO improves prediction accuracy in almost all cases in the SSL setting. This is not always true in the baseline setting where the ordinary least squares in some cases outperformed the LASSO in terms of lower classification errors.
- The LASSO classifier seems to give best results when the KECA and KPCA yields the same results. This indicates that the favorable properties of the KECA over the KPCA mentioned in this thesis and in [15] is not fully utilized by least squares methods.
- The KECA LASSO SSL classifier is comparable and in many cases better than benchmark work using the Laplacian eigenmaps and DaSpec algorithms.
- It seems that the choice of kernel size is by far the most important parameter choice in all data transformations used in this thesis. All the methods have kernel size windows where they perform well, but in most cases these do not overlap at all, and they do not seem to coincide with known heuristic methods.

10.1 Suggestion for further work

There are many interesting directions the work beyond this thesis could take. The most obvious is perhaps to move beyond the choice of linear classifiers, LS and LASSO, and test more advanced classifiers. On the other hand different versions of the semi-supervised scheme could be tested. We present a list of things we find interesting as options for further work.

- In [15] it was shown that the Kernel ECA transform in many cases resulted in a distinct angular structure and a clustering algorithm using k-means and an angular cosine measure was proposed. From these results it could be interesting to try the semi-supervised direction called 'self-training' where the EM[1] algorithm has been used. The idea is to train a classifier using a set of labeled data, classify the unlabeled data and then train a new classifier based on the newly classified points

[1]Expectation Maximization algorithm, details can be found in [30].

[19]. This process is repeated until a convergence criteria is reached, e.g the classification error does not improve for each iteration. The k-means is a special case of the EM algorithm so to implement the EM algorithm in the feature space using an angular cost function could be of interest. Alternatively one could simply test a classification in the feature space using some kind of angular cost function as we have seen that the LASSO does not take advantage of the sometimes distinct angular axes of the KECA transformation.

- One of the biggest problems with using the Kernel ECA in the semi-supervised setting is that we have to calculate the eigenvectors of the kernel matrix. We have used MATLAB which sets the limit at about 5000 data points. In [16] the Epanechnikov kernel is used leading to a sparse kernel matrix, making the Kernel ECA SSL classifiers capable of handling much larger data sets. Another possibility is testing kernel ECA and classification using a nearest neighbor-based entropy measure yielding a sparse kernel matrix. The latter has not yet been implemented, but work is being done at the University of Tromsø. This nearest neighbor setting could also be tested implicitly with the *Semi-definite Embedding* version of kernel PCA, presented in [32][2].

- In [31] it is pointed out that for the different true underlying models of a data set, the LASSO, ridge regression and subset selection will each have a scenario where they work best, and in [9], which is the basis of the algorithm used to calculate the LASSO parameters, the 'elastic net' is proposed, a combination or compromise between the LASSO and ridge regression. Implementing this is not more computationally demanding than the LASSO and because we have shown the LASSO to improve the classification it would be interesting to see if the elastic net could improve the classification further.

- The choice of linear and quadratic error based classifiers is perhaps a strange choice as we in the Theory part of this thesis have proposed possible drawbacks when using such methods, so classifying with an entropy based cost function in the KECA feature space would be interesting as the Kernel ECA preserves entropy. In some cases we have

[2]The connection to entropy is not directly evident as of now and would have to be explored further, but the matrix is a kernel matrix used for KPCA so the connection to Kernel ECA should still hold.

observed that the KPCA based classifiers have outperformed the KECA based classifiers, something that is perhaps not so strange as it is a variance preserving transformation and the LASSO and least squares are based on squared errors.

Appendix A

A selection of extra parameter estimates

A.1 KECA knee method

In this section we introduce the method of choosing the number of eigenvectors in a KECA data transformation we use. In [16] it is pointed out that in many cases a notable drop, or 'knee', can be noted when plotting the entropy values and an automated procedure for finding this knee is proposed:

- Compute the differences between the entropy values, a discrete first order derivative.
- Compute the ratio between the derivatives.
- If the ratio when going from entropy value k to $k+1$ is lower than 15% a 'knee' is noted and the dimension is set to k.
- A lower limit of k is set equal to the number of classes in the data set in classification cases as the drop from the first to second entropy value can in certain settings be extremely high resulting in only two dimensions chosen.

A.2 DaSpec kernel size algorithm

In this section we present a kernel size method from [25].

The idea is that when a kernel operator works on a data point it should cover at least 5% of the neighbouring points, and this should be the case for as much of the data as possible, 95% is used in [25]. This is reflected in a length quantity l such that $P(||X|| < l) = 0.95$

l can be found as:

- an array of 5% quantile of distance from each point to all other points.
- 95% quantile of this array.

We now look at $P(||X|| < l) = 0.95$.

- We know that the Gaussian kernel is $\propto \mathcal{N}(x, \sigma^2)$ which gives $||X|| \propto \frac{1}{\omega}\mathcal{N}(\sum \sqrt{x}, 1) = \frac{1}{\omega}\sqrt{(}\chi^2_d)$
- The latter gives, using the inverse χ^2_d, $l = \sigma\sqrt{95\% \text{ quantile of } \chi^2_d}$

From this we find the kernel size $\sigma = \frac{l}{\sqrt{95\% \text{ quantile of } \chi^2_d}}$

Appendix B

Additional classification results using KECA and KPCA

#labeled points		10		20		30	
Data set		KECA	KPCA	KECA	KPCA	KECA	KPCA
IONOS							
	Error	0.2189	0.2189	0.1982	0.2177	0.1769	0.2182
	Standard deviation	0.0191	0.0191	0.0775	0.0238	0.0561	0.0183
PIMA							
	Error	0.2979	0.2923	0.2903	0.2859	0.2849	0.2804
	Standard deviation	0.0232	0.0212	0.0243	0.0230	0.0192	0.0147
ADULT							
	Error	0.2491	0.2175	0.2461	0.2086	0.2490	0.2090
	Standard deviation	0.0165	0.0149	0.0166	0.0125	0.0107	0.0137
WBC							
	Error	0.0314	0.0314	0.0308	0.0308	0.0301	0.0301
	Standard deviation	0.0060	0.0060	0.0063	0.0063	0.0036	0.0036

Table B.1: KECA and KPCA classification errors with 2 eigenvectors fixed as output dimension.

	#labeled points	20		30		50	
	# eigenvectors	4		6		10	
Data set		KECA	KPCA	KECA	KPCA	KECA	KPCA
IONOS							
	Error	0.0647	0.0924	0.0628	0.0628	0.0568	0.0575
	Standard deviation	0.0117	0.0198	0.0169	0.0163	0.0109	0.0062
PIMA							
	Error	0.2769	0.2758	0.2649	0.2732	0.2504	0.2504
	Standard deviation	0.0247	0.0140	0.0200	0.0199	0.0125	0.0125
WBC							
	Error	0.0292	0.0292	0.0303	0.0296	0.0303	0.0300
	Standard deviation	0.0036	0.0036	0.0037	0.0065	0.0029	0.0036
ADULT							
	Error	0.2361	0.2132	0.2327	0.2104	0.2027	0.2034
	Standard deviation	0.0096	0.0144	0.0151	0.0116	0.0129	0.0145

Table B.2: KECA and KPCA classification errors with 20% of # of labeled points eigenvectors as output dimension.

	#labeled points	10		20		50	
	# eigenvectors	20		20		20	
Data set		KECA	KPCA	KECA	KPCA	KECA	KPCA
IONOS							
	Error	0.2028	0.2026	0.1150	0.1122	0.0621	0.0596
	Standard deviation	0.0694	0.0861	0.0371	0.0531	0.0129	0.0140
PIMA							
	Error	0.3104	0.3109	0.2844	0.2828	0.2553	0.2569
	Standard deviation	0.0385	0.0432	0.0250	0.0209	0.0170	0.0180
WBC							
	Error	0.0366	0.0377	0.0336	0.0339	0.0301	0.0304
	Standard deviation	0.0080	0.0114	0.0064	0.0116	0.0061	0.0059
ADULT							
	Error	0.2334	0.2355	0.2215	0.2203	0.2053	0.2032
	Standard deviation	0.0187	0.0214	0.0173	0.0209	0.0120	0.0123

Table B.3: KECA and KPCA classification errors with fixed # eigenvectors of 20 as output dimension.

Kernel size method	DaSpec method		15 % median range		15 % mean range		15 % total range	
	error	std	error	std	error	std	error	std
# eigenvectors	2		2		2		2	
KECA SSL LASSO	0.0822	0.0089	0.1461	0.0101	0.1488	0.0160	0.0293	0.0032
KPCA SSL LASSO	0.0864	0.0095	0.1824	0.0103	0.1837	0.0115	0.0298	0.0029
# eigenvectors	10		10		10		10	
KECA SSL LASSO	0.0619	0.0051	0.0780	0.0061	0.0768	0.0068	0.0301	0.0026
KPCA SSL LASSO	0.0614	0.0043	0.0976	0.0104	0.0975	0.0089	0.0283	0.0033
# eigenvectors	20		20		20		20	
KECA SSL LASSO	0.0579	0.0070	0.0840	0.0165	0.0781	0.0088	0.0297	0.0027
KPCA SSL LASSO	0.0607	0.0080	0.0840	0.0134	0.0858	0.0138	0.0292	0.0026

Table B.4: Classification errors of IONOS data set with 50 labeled points with different heuristically found kernel sizes.

Kernel size method	DaSpec method		15 % median range		15 % mean range		15 % total range	
	error	std	error	std	error	std	error	std
# eigenvectors	2		2		2		2	
KECA SSL LASSO	0.3646	0.0116	0.4422	0.0141	0.4332	0.0195	0.3142	0.0068
KPCA SSL LASSO	0.3717	0.0132	0.4476	0.0181	0.4430	0.0181	0.3194	0.0114
# eigenvectors	10		10		10		10	
KECA SSL LASSO	0.3646	0.0116	0.4422	0.0141	0.4332	0.0195	0.3142	0.0068
KPCA SSL LASSO	0.3717	0.0132	0.4476	0.0181	0.4430	0.0181	0.3194	0.0114
# eigenvectors	20		20		20		20	
KECA SSL LASSO	0.3531	0.0191	0.4135	0.0175	0.4057	0.0185	0.2896	0.0206
KPCA SSL LASSO	0.3481	0.0166	0.4257	0.0193	0.4183	0.0131	0.2935	0.0140

Table B.5: Classification errors of PIMA data set with 50 labeled points with different heuristically found kernel sizes.

Kernel size method	DaSpec method		15 % median range		15 % mean range		15 % total range	
	error	std	error	std	error	std	error	std
# eigenvectors	2		2		2		2	
KECA SSL LASSO	0.3456	0.0153	0.3751	0.0216	0.3875	0.0190	0.2813	0.0202
KPCA SSL LASSO	0.3448	0.0167	0.3659	0.0200	0.3809	0.0131	0.2821	0.0177
# eigenvectors	10		10		10		10	
KECA SSL LASSO	0.1406	0.0222	0.1847	0.0225	0.1970	0.0218	0.0697	0.0170
KPCA SSL LASSO	0.1482	0.0166	0.1807	0.0200	0.1948	0.0180	0.0643	0.0178
# eigenvectors	20		20		20		20	
KECA SSL LASSO	0.1345	0.0139	0.1530	0.0183	0.1552	0.0180	0.0594	0.0211
KPCA SSL LASSO	0.1375	0.0181	0.1669	0.0227	0.1795	0.0118	0.0783	0.0231

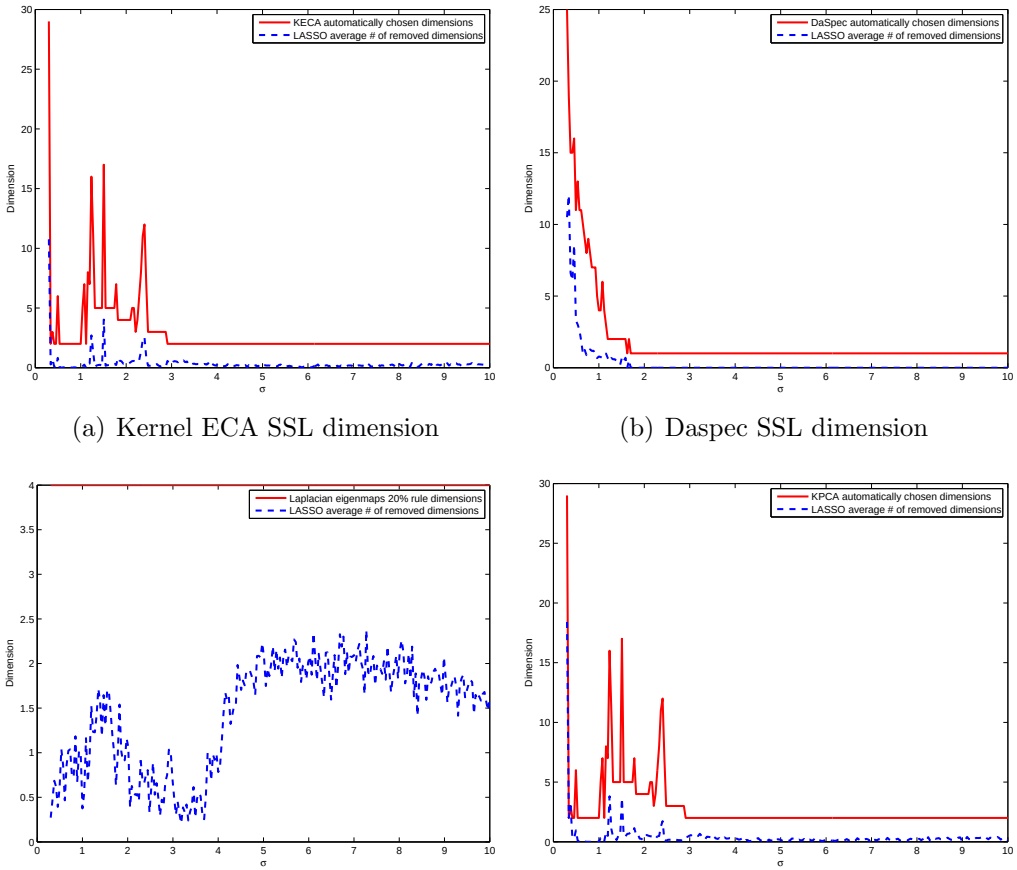
Table B.6: Classification errors of ADULT data set with 50 labeled points with different heuristically found kernel sizes.

Kernel size method	DaSpec method		15 % median range		15 % mean range		15 % total range	
	error	std	error	std	error	std	error	std
# eigenvectors	2		2		2		2	
KECA SSL LASSO	0.1466	0.0191	0.1877	0.0293	0.2045	0.0191	0.2777	0.0119
KPCA SSL LASSO	0.3221	0.0311	0.3476	0.0289	0.3481	0.0147	0.2885	0.0302
# eigenvectors	10		10		10		10	
KECA SSL LASSO	0.1130	0.0339	0.1296	0.0314	0.1388	0.0366	0.0228	0.0102
KPCA SSL LASSO	0.1216	0.0342	0.1411	0.0417	0.1509	0.0485	0.0261	0.0173
# eigenvectors	20		20		20		20	
KECA SSL LASSO	0.1015	0.0337	0.1193	0.0346	0.1546	0.0343	0.0233	0.0137
KPCA SSL LASSO	0.1318	0.0274	0.1551	0.0542	0.1599	0.0343	0.0248	0.0167

Table B.7: Classification errors of WINE data set with 45 labeled points with different heuristically found kernel sizes.

Appendix C

Extra dimensionality plots from the UCI kernel size full range experiments



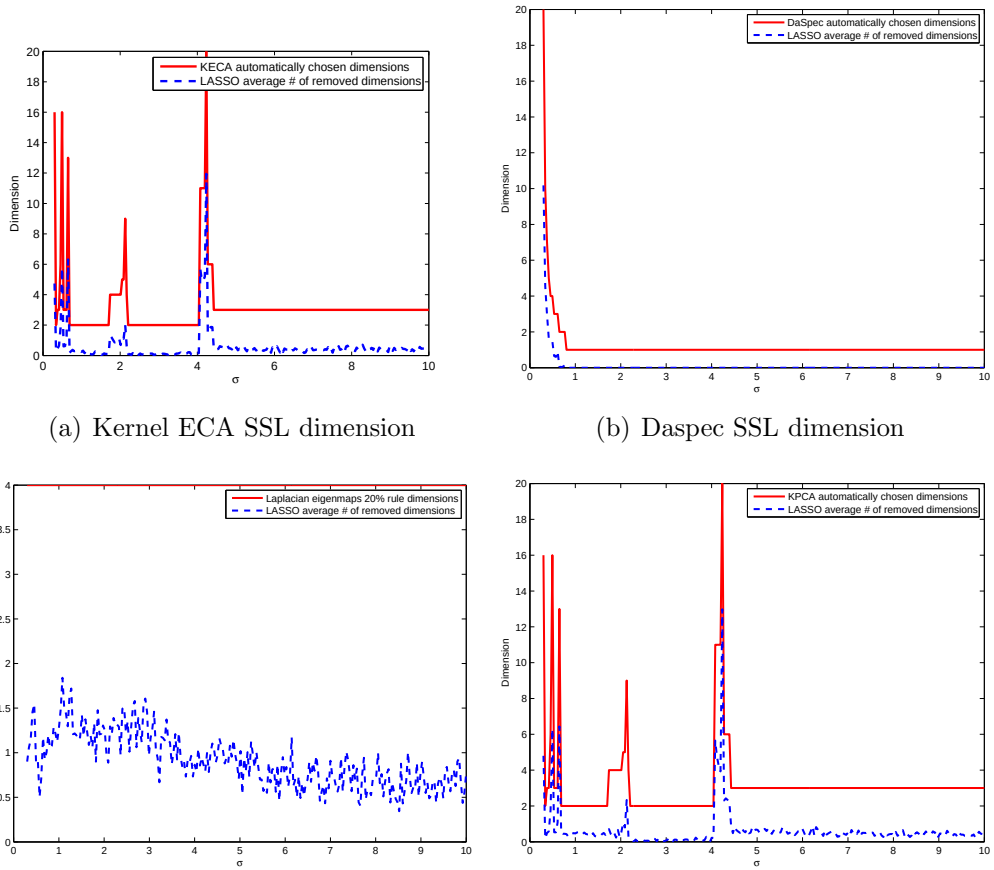
(a) Kernel ECA SSL dimension

(b) Daspec SSL dimension

(c) Laplacian eigenmaps SSL dimension

(d) Kernel PCA SSL dimension

Figure C.1: IONOS data set: Dimension for all SSL methods with average sparsity of the LASSO coefficients.



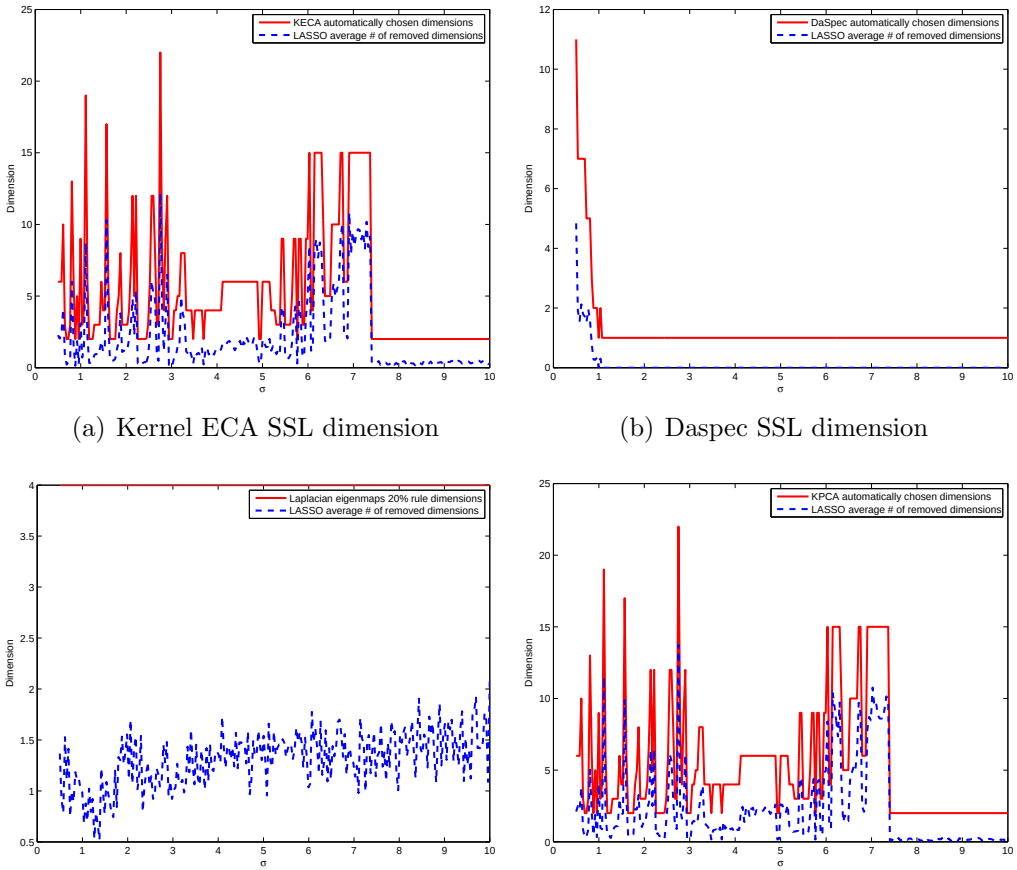
(a) Kernel ECA SSL dimension

(b) Daspec SSL dimension

(c) Laplacian eigenmaps SSL dimension

(d) Kernel PCA SSL dimension

Figure C.2: PIMA data set: Dimension for all SSL methods with average sparsity of the LASSO coefficients.



(a) Kernel ECA SSL dimension

(b) Daspec SSL dimension

(c) Laplacian eigenmaps SSL dimension

(d) Kernel PCA SSL dimension

Figure C.3: ADULT data set: Dimension for all SSL methods with average sparsity of the LASSO coefficients.

Appendix D

List of figures

List of Figures

Appendix E

List of tables

List of Tables

Bibliography

[1] Frank. A and Asuncion. A. Uci machine learning repository. http://archive.ics.uci.edu/ml, 2010.

[2] N. Aronszajn. *Theory of reproducing kernels*. Harvard University, 1951.

[3] M. Belkin and P. Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation*, 15(6):1373–1396, 2003.

[4] M. Belkin and P. Niyogi. Semi-supervised learning on riemannian manifolds. *Machine Learning*, 56(1):209–239, 2004.

[5] M.N. Berberan-Santos. Expressing a probability density function in terms of another pdf: A generalized gram-charlier expansion. *Journal of Mathematical Chemistry*, 42(3):585–594, 2007.

[6] O. Chapelle and A. Zien. Semi-supervised classification by low density separation. 2004.

[7] D. Erdogmus and JC Principe. From linear adaptive filtering to nonlinear information processing-The design and analysis of information processing systems. *Signal Processing Magazine, IEEE*, 23(6):14–33, 2006.

[8] J. Friedman, T. Hastie, and R. Tibshirani. The elements of statistical learning, 2008.

[9] J. Friedman, T. Hastie, and R. Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of statistical software*, 33(1):1, 2010.

[10] S Haykin. *Neural Networks and Learning Machines, Third edition*. Prentice Hall, 3rd edition, 2008.

[11] Simon Haykin, editor. *Unsupervised adaptive filtering*. Wiley, 2000.

[12] A.E. Hoerl and R.W. Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, pages 55–67, 1970.

[13] A. Hyvärinen and E. Oja. Independent component analysis: algorithms and applications. *Neural networks*, 13(4-5):411–430, 2000.

[14] R. Jenssen. Information theoretic learning and kernel methods. In F. Emmert-Streib and M. Dehmer, editors, *Information theory and statistical learning*, pages 209–230. Springer-Verlag New York Inc, 2009.

[15] R. Jenssen. Kernel entropy component analysis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(5):847–860, 2010.

[16] Robert Jenssen. Kernel entropy component analysis: New theory and semi-supervised learning. In *IEEE Workshop on Machine Learning for Signal Processing, 18-21 September*, 2011.

[17] J. Marcinkiewicz. Sur une propriete de la loi de gauss. *Mathematische Zeitschrift*, 44(1):612–618, 1939.

[18] EH Moore. On properly positive hermitian matrices. *Bull. Amer. Math. Soc*, 23(59):66–67, 1916.

[19] K. Nigam, A.K. McCallum, S. Thrun, and T. Mitchell. Text classification from labeled and unlabeled documents using em. *Machine learning*, 39(2):103–134, 2000.

[20] E. Parzen. On estimation of a probability density function and mode. *The annals of mathematical statistics*, 33(3):1065–1076, 1962.

[21] J.C. Principe. *Information theoretic learning: Renyi's entropy and Kernel perspectives*. Springer, 2010.

[22] B. Schölkopf, A. Smola, and K.R. Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural computation*, 10(5):1299–1319, 1998.

[23] D.W. Scott. *Multivariate density estimation*, volume 139. Wiley Online Library, 1992.

[24] J. Shawe-Taylor and N. Cristianini. *Kernel methods for pattern analysis*. Cambridge Univ Pr, 2004.

[25] T. Shi, M. Belkin, and B. Yu. Data spectroscopy: Eigenspaces of convolution operators and clustering. *The Annals of Statistics*, 37(6B):3960–3984, 2009.

[26] B.W. Silverman. *Density estimation for statistics and data analysis*, volume 26. Chapman & Hall/CRC, 1986.

[27] K. Sinha and M. Belkin. The value of labeled and unlabeled examples when the model is imperfect. *Advances in Neural Information Processing Systems*, 20:1361–1368, 2008.

[28] K. Sinha and M. Belkin. Semi-supervised learning using sparse eigenfunction bases. *Advances in Neural Information Processing Systems*, 23, 2009.

[29] H. Stark and J.W. Woods. *Probability and random processes with applications to signal processing*. Prentice Hall Upper Saddle River, NJ:, 2002.

[30] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern Recognition, Fourth Edition*. Academic Press, 4th edition, 2008.

[31] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[32] K.Q. Weinberger, F. Sha, and L.K. Saul. Learning a kernel matrix for nonlinear dimensionality reduction. In *Proceedings of the twenty-first international conference on Machine learning*, page 106. ACM, 2004.

[33] X. Zhu. Semi-supervised learning literature survey. 2005.