The effects of preprocessing on PCA for classification problems
July 6, 2018

Laurisa Maagendans
5526353

Supervised by dr. S.W. Gaaf

Contents

1 Introduction

Have you ever wondered how your smartphone unlocking system is able to recognize your fingerprint? Or how Google can return a whole collection of images that are to some extent identical to one image that you uploaded? These problems have something in common, which is that they all ask for a specific observation to be labeled correctly. A fingerprint image has to be identified as belonging to the owner of the smartphone, or as not belonging to the owner. Google's 'search by image' algorithm aims to find one or more labels that describe the image that you served as input. The algorithm then finds images that have these labels in common. This is to name only two examples of problems belonging to the much larger class of **classification problems** in machine learning.[12]

To solve such problems, a so-called **classifier** is trained on a training data set, containing observations that are pre-classified. The aim of the training stage is to find similarities in observations that are in the same class, i.e. have the same labels. These similarities are key to identifying a new observation as belonging to a certain class. While there are many different kinds of classifiers available, the classifier we will be using is a **Support Vector Machine (SVM)**[4], due to its simplicity and versatility.

In this thesis, we are using **Principal Component Analysis (PCA)**[6] as a preprocessing step. This procedure is commonly used for dimensionality reduction purposes[6] and might prove helpful when dealing with high-dimensional input data, without much loss of information. If high dimensionality is not a problem, a side-effect of performing PCA is that it can denoise the input[13], which can be desirable depending on the quality of the input data. Aside from denoising and helping classifiers handle high-dimensional input by reducing it, another reason for using PCA in this thesis is that in practice, most input data that is fed to a classifier is compressed in one way or another. The results of this thesis might help defend PCA as a suitable compression method for such data, perhaps in combination with other preprocessing steps.

The aim of this thesis is to show that using various preprocessing methods prior to preprocessing our input using PCA can have a significant impact on the accuracy of a classifier. In order to do so, we have assembled a pipeline that loads, preprocesses and performs PCA-based dimensionality reduction on the input data and finally uses this transformed data to test classifier accuracy. As input dataset we are using the MNIST handwritten digit database[10]. Instead of delving headfirst into the specifics of the various components that make up this pipeline, a global overview might be needed, or at least helpful, to gain an understanding of the big picture.

The initial input of this pipeline is a **dataset**, which consists of **data entries**, a set of images in our case, and **labels** for each data entry that specify what class the entry belongs to. This dataset is further split into two parts; the **training dataset** and the **testing dataset**. Both datasets consist of a set of data entries and a set of corresponding labels, in such a way that there is at least one (preferably at least 10 for the training dataset) data entry of each class label present in both datasets.

As mentioned before, we are interested in the effects of preprocessing in combination with

PCA-based dimensionality reduction on the data prior to the whole classification process. A big part of the process of preparing our data is shown as a diagram in Figure 1. In case of images, each data entry is represented as a matrix of (A)RGB values, all of similar dimensions, making the set of data entries a set of matrices. Since both our chain of preprocessing methods and PCA take matrices, containing the whole training or testing dataset, as input, a conversion of the data is necessary. In the 'Extract feature vectors' step of Figure 1, the set of data entries is transformed into a matrix such that each column vector represents a data entry. This is done with a function $f : \mathbb{R}^{a\times b} \to \mathbb{R}^{ab}$ defined as

$$f(P) = f((p_{a,b})) = (p_1, p_2, \ldots, p_{ab})^T \tag{1}$$

where $p_i = p_{i \mod a, \lceil i/a \rceil}$. In words, it truncates the rows of the original image from top to bottom to form it into one ab-dimensional vector. In the field of machine learning, these vectors are referred to as **feature vectors**. Each of these feature vectors is a column in the **feature matrix** A.

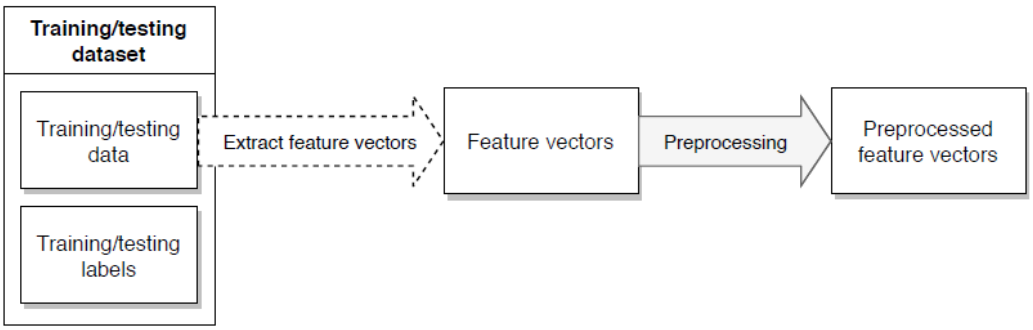

Figure 1: Extracting feature vectors from the training and testing datasets and preprocessing them.

Once A is obtained, it is to be preprocessed prior to doing anything else with it. The 'Preprocessing' step is where this takes place and it is further specified in Figure 2. All preprocessing steps mentioned take a $n \times m$ matrix as input and returns a $n \times m$ matrix as output. Two preprocessing steps that are necessary for PCA to produce the wanted results are **mean centering** and **column normalization**. Both steps will be touched upon in section 2 of this thesis. For the sake of consistency, we require every feature vector to undergo these two preprocessing steps, even if PCA is not being used. After this comes a chain of a combination (including none) of preprocessing methods to experiment with, and hopefully improve classifier accuracy. These steps will all be discussed in detail in section 4. The preprocessed feature vectors, both training and testing, are saved for use in future steps.

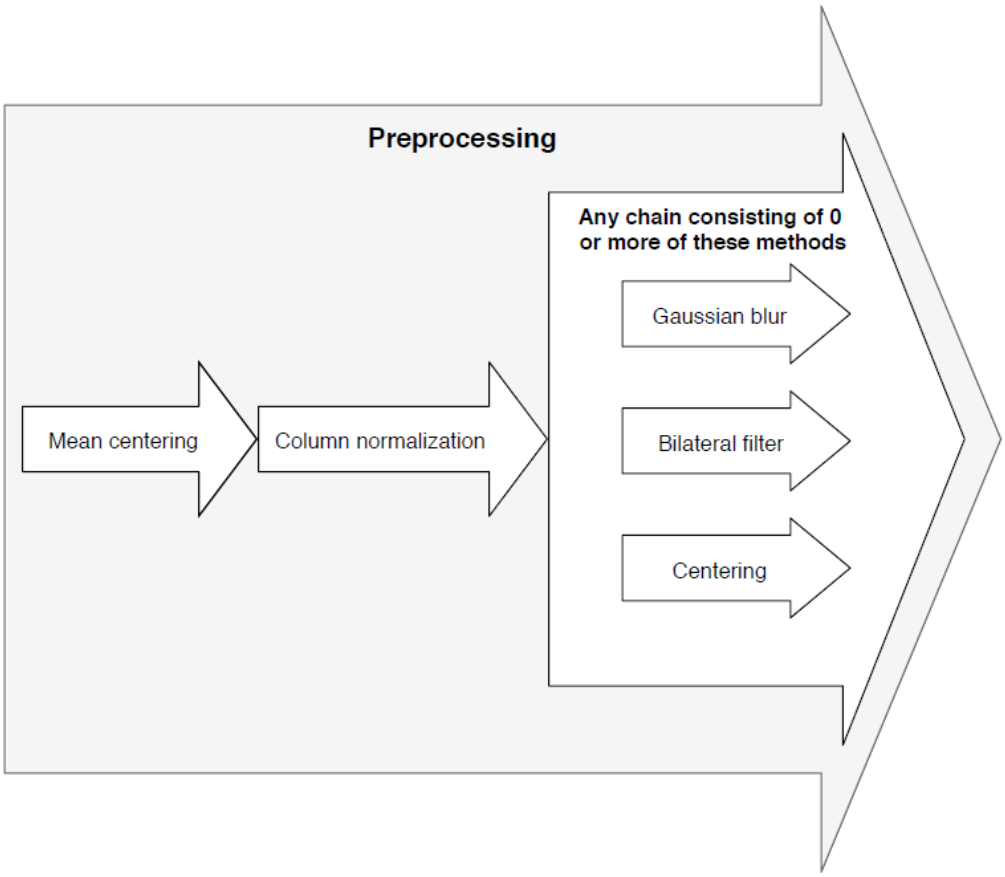

Figure 2: The preprocessing step. This procedure takes a $n \times m$ matrix as input and returns a $n \times m$ matrix as output

PCA-based dimensionality reduction is carried out in two stages. During the first stage, as shown in Figure 3, the training feature vectors are used to determine the first p principal components, as they correspond to the (orthogonal) axes with the highest variation in the set of feature vectors.

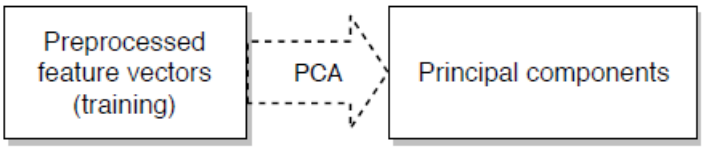

Figure 3: Using preprocessed feature vectors from the training set to find principal components

Shown in Figure 4 is the second stage, where PCA performs its dimensionality reduction magic. Both the (preprocessed) training and testing feature vectors are to go through this stage. Here, the feature vectors are projected onto the principal components, in order to write each feature vector as a linear combination of principal components that best approximates the original feature vector. The specifics of PCA will be explained in more detail in section 2.

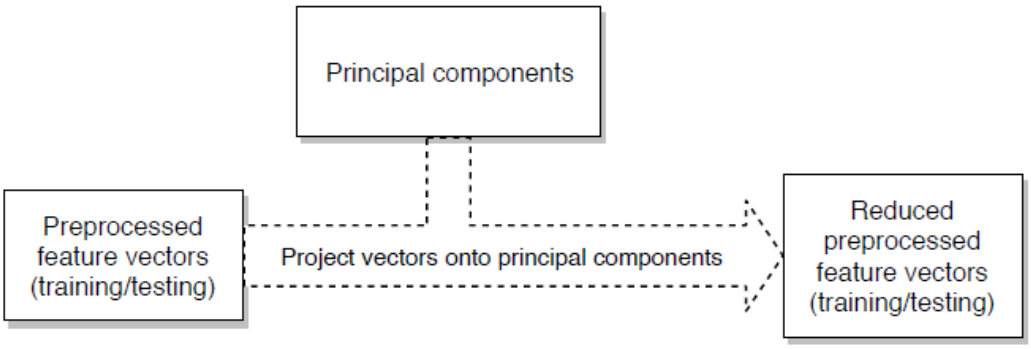

Figure 4: Reducing the dimensionality of preprocessed feature vectors by projecting them onto principal components

The last big part of the process is the actual classification step. As mentioned before, this step consists of two phases, the **training phase** and the **testing phase**. The training phase is portrayed in Figure 5. During this phase, an SVM is trained on the training feature vectors, that have already gone through preprocessing, and their corresponding labels. The output is a set of so-called **weight vectors** and their corresponding **support vectors**.

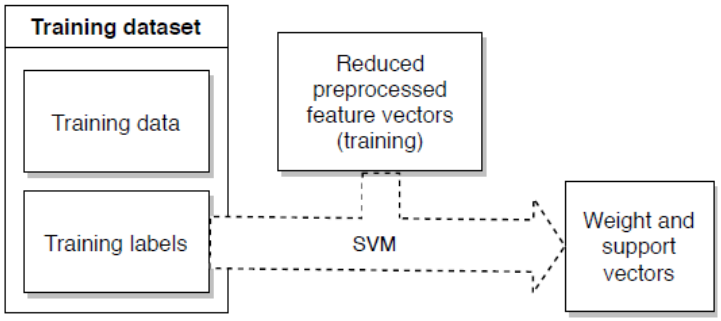

Figure 5: Classification with SVM: Training phase

These vectors can then be used in the testing phase, which is shown in Figure 6. With these vectors, the SVM is then able to classify the testing feature vectors that are already preprocessed and reduced. Afterwards, these classifications are checked on correctness by checking them against the testing labels that correspond to the testing feature vectors. The percentage of testing feature vectors that have been classified correctly is then what we call the accuracy of the SVM.

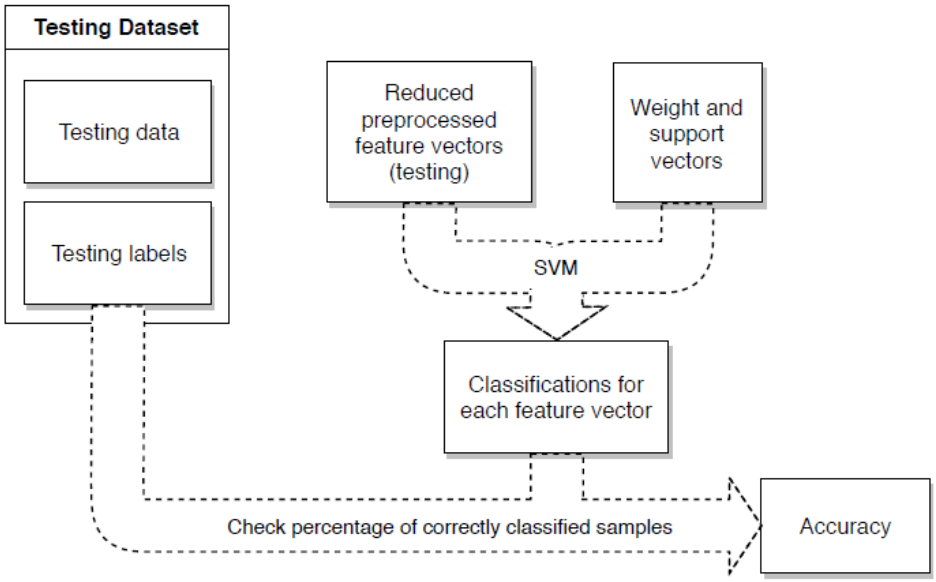

Figure 6: Classification with SVM: Testing phase and accuracy calculation

In order to perform accurate classification, the SVM has to be able to assume that both the training and the testing feature vectors belonging to a certain class share common properties. The hope is to find a combination of preprocessing methods that, intuitively speaking, make those common properties more prevalent so they become easier for the SVM to pick up on.

While Figure 3 and 4 make PCA-based dimensionality reduction seem like a simple process, we will go beyond considering it a black box. The following section shall touch on the specifics of PCA, explaining why and how it works.

2 Principal Component Analysis

PCA is a procedure best known for its uses in dimensionality reduction.[6] As input, we are considering a feature matrix $A \in \mathbb{R}^{n \times m}$, with the columns corresponding to the feature vectors. In almost all classification problems, the amount of samples is way smaller than the size of the samples themselves. This is also true for the datasets we are using, so $n < m$. Providing PCA with such a matrix A as input, it returns a set of $p < n$ orthogonal m-dimensional vectors; the so-called **principal components**. These are chosen in such a way that they are orthogonal to any previous principal component and the average distance of all data points to this principal component are minimized, which means that the variance along the principal component is maximized.

2.1 Finding principal components

To determine the principal components of A, one could either use the covariance matrix of A or the correlation matrix of A. In fact, both ways are similar in essence, since the correlation matrix of A can be considered a standardized version of the covariance matrix of A.[17] Usually, standardization is desirable when dealing with images that are recorded under different circumstances, if those differences do not hold much meaning, such as a difference in lighting when classifying faces. In those cases, it makes sure that PCA weighs each entry Since the datasets we are using has exclusively grayscale images, standardization would amount to a change in contrast in each picture. For the MNIST dataset, the data entries have been recorded in circumstances that are similar enough for standardization to not have any benefits. Standardizing the data could even lead to loss of valuable information. For MNIST, the digits are written with the same kind of (digital) pen, so the area of the surface that is drawn on in an image is what mostly determines the variance within an image. One can imagine that the covered area when writing an '8' is likely much bigger than when writing a '1', hence there is some sort of meaning attached to the covered area, which would get lost during standardization. This is a reason for us to use the covariance matrix of A instead.

In our case, PCA is performed on preprocessed feature vectors. Regardless of which of the optional preprocessing steps the feature vectors have gone through, all of them have

been mean centered and normalized. Mean centering the columns of A would result in a new matrix A_{centered} such that

$$A_{\text{centered}} = A - \begin{bmatrix} \sum_{i=1}^{m} a_{1,i}/m & \dots & \sum_{i=1}^{m} a_{n,i}/m \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{m} a_{1,i}/m & \dots & \sum_{i=1}^{m} a_{n,i}/m \end{bmatrix} \tag{2}$$

In words, the mean of each column of A is subtracted from each entry of that column in order to ensure that the mean of the column is 0. Normalizing A_{centered} ensures that each column vector is an unit vector. This results in a new matrix $A_{\text{normalized}}$ such that

$$A_{\text{normalized}} = A_{\text{centered}} \begin{bmatrix} \frac{1}{||\vec{a}_1||} & & \\ & \ddots & \\ & & \frac{1}{||\vec{a}_n||} \end{bmatrix} \tag{3}$$

where $\vec{a}_i$ is the ith column vector of A_{centered}. From this point on, we shall refer to $A_{\text{normalized}}$ simply as A. That is, we are assuming that the column vectors of A already have zero mean and are unit length.

When A is a feature matrix with feature vectors of zero mean, its **covariance matrix** is given by

$$C_A = \frac{1}{n} A A^T \tag{4}$$

[17]

The eigenvectors of this covariance matrix is what PCA wishes to find, as they lie on the axis of maximal variance under an orthogonality constraint.[17]

A naive way of finding the principal components of A would be to first calculate the covariance matrix $\frac{A^T A}{n}$ and then using this matrix to find its eigendecomposition. This is a viable strategy when working with relatively small feature matrices, but it quickly becomes slower than a snail traveling through peanut butter once bigger feature matrices are involved. To show this, some of the asymptotically fastest matrix multiplication algorithms for $n \times n$ matrices to date are the Optimized CW-like algorithms, with a time complexity of $\mathcal{O}(n^{2.373})$[5]. Our feature matrices are very likely not square to begin with, so we would have to deal with a time complexity of $\mathcal{O}(n^2 m)$[14] with a feature matrix of size $n \times m$. Either way, it is needless to say that this method does not scale well.

A better approach, and the one we are using, would be to compute the **singular value decomposition (SVD)** of the data matrix. The next subsection intends to briefly introduce the concept of SVDs, reduced forms of the SVD and finally, the iterative methods we are using to approximate SVDs.

2.2 Singular Value Decomposition

This subsection serves as a brief introduction to SVDs and can be skipped entirely by readers who are familiar with it. We will start by introducing the definitions of singular values and singular vectors.

Definition 2.1 (Singular value decomposition (SVD))**.** *A **singular value decomposition** of a real $n \times m$ matrix A is a factorization of the form $U\Sigma V^T$, such that U is a $n \times n$ orthogonal matrix, Σ is a diagonal $n \times m$ matrix and V is a $m \times m$ orthogonal matrix.*
*The values on the diagonal of Σ are called the **singular values** of A.*
*The columns vectors of U and the columns vectors of V are called the **left** and **right singular vectors** of A, respectively.*

[1]

A nice property of SVDs is that unlike eigendecompositions, that do not need exist for every real $n \times m$ matrix A[], an SVD does exist for any such matrix A.

Theorem 2.2 (Singular value decomposition (SVD))**.** *Every real $n \times m$ matrix A has a singular value decomposition.*

A proof of this theorem can be found in [7], chapter 2.4.1. Aside from the obvious observation that both are matrix decompositions, the concept of SVDs is very closely related to that of eigendecompositions. The following lemma helps clear up that relation.

Lemma 2.3. *For any real $n \times m$ matrix A with SVD $U\Sigma V^T$, the following are true:*

1. $A^T A = V\Sigma^2 V^{-1}$
2. $AA^T = U\Sigma^2 U^{-1}$

Proof. Assume that A is a real $n \times m$ matrix with SVD $U\Sigma V^T$.

1. Substituting A with its SVD shows that $A^T A = (U\Sigma V^T)^T(U\Sigma V^T) = V\Sigma U^T U\Sigma V^T$. Since U is orthogonal, $U^T U$ is the identity matrix, hence $V\Sigma U^T U\Sigma V^T = V\Sigma^2 V^T$. Since V is orthogonal as well, we have $V\Sigma^2 V^T = V\Sigma^2 V^{-1}$.
2. This proof is fairly similar to the previous proof. Substituting A with its SVD shows that $AA^T = (U\Sigma V^T)(U\Sigma V^T)^T = U\Sigma V^T V\Sigma U^T$. Orthogonality of V gives us $U\Sigma V^T V\Sigma U^T = U\Sigma^2 U^T$. Finally, since U is orthogonal, we can conclude the proof as this implies that $U\Sigma^2 U^T = U\Sigma^2 U^{-1}$

□

Notice how the right sides of the equations in this lemma form eigendecompositions of $A^T A$ and AA^T, since Σ^2 is a diagonal matrix. This allows us to draw a few conclusions about singular values and vectors.

Corollary 2.3.1. *Let A be a real $n \times m$ matrix, then the following are true:*

1. *The singular values of A are the square roots of the eigenvalues of both A^TA and AA^T.*
2. *The left and right singular vectors of A are the eigenvectors of AA^T and A^TA respectively.*

Because of Lemma 2.3, it is quite easy to notice the connection between the SVD of a matrix A and its covariance matrix. If we define $B = \frac{1}{\sqrt{n}}A^T$[17], we will see that $B^TB = (\frac{1}{\sqrt{n}}A^T)^T(\frac{1}{\sqrt{n}}A^T) = \frac{1}{n}AA^T$, which is the covariance matrix of A. Together with Corollary 2.3.1, this means that the eigendecomposition of the covariance matrix of A can be easily obtained when the SVD of B is known. Assuming that the SVD of A is $U\Sigma V^T$, we can show that $B = \frac{1}{\sqrt{n}}A^T = \frac{1}{\sqrt{n}}(U\Sigma V^T)^T = V(\frac{1}{\sqrt{n}}\Sigma)U^T$.

It can be easily shown that A^TA is Hermitian and positive semi-definite. Indeed, we have $(A^TA)^T = A^T(A^T)^T = A^TA$, and its values are all sums of squares of values of A. Hence, singular values of a matrix A are real and non-negative. This allows us to order the singular values of a $m \times n$ matrix as $0 \leq \sigma_n \leq ... \leq \sigma_2 \leq \sigma_1$ and we shall use this notation throughout this section.

2.3 Reduced forms of SVD

Before we jump off to methods for computing the SVD, we will first cover reduced forms of the SVD, since we generally strive to avoid any unnecessary computations. In our case, we are only interested in finding principal components that are significant enough, or only the p most significant principal components if memory limitations are involved. Several smaller versions of the SVD exist that might help reduce the problem a little.

Let A be an $n \times m$ matrix of rank r with SVD $A = U\Sigma V^*$, where $n < m$. Let U_t and V_t be the matrices composed of the first t columns of U resp. V and let Σ_t be a $t \times t$ matrix containing the first t singular values. We can now define these smaller versions of the SVD.

Definition 2.4 (Thin SVD)**.** *The thin SVD of a matrix A is $A = U_n\Sigma_nV^*$.[1]*

For the thin SVD, only the column vectors of U that correspond to row vectors of V are computed, since the SVD has at most n singular values. When $n \ll m$, this can result in a significant decrease in space and computation time. When A has rank r, it will have at most r non-zero singular values, which allows us to omit even larger parts of the decomposition. We call this reduced form of SVD the compact SVD.

Definition 2.5 (Compact SVD)**.** *The compact SVD of a matrix A is $A = U_r\Sigma_rV_r^*$.[1]*

When $r \ll n$, this too can result in a significant decrease in space and computation time as compared to the thin SVD. Even smaller is the truncated SVD, which can be chosen to compute when it is known that we need at most $t < n$ singular values and vectors.

Definition 2.6 (Truncated SVD)**.** *The truncated SVD of a matrix A is $A_t = U_t\Sigma_tV_t^*$.[1]*

The matrix A_t is uniquely determined such that $\|A - A_t\|_F$ is minimized[20]. When $t \ll r$, this reduced form takes much less space and time to compute than the compact SVD.

In our case, we are choosing the desired amount of singular values and vectors beforehand. Therefore, using the truncated SVD would be a good fit for our situation. One drawback of the truncated SVD is that it is an approximation of the full SVD of A, though the best approximation of size t. If our singular values of interest had been the smallest few, this would have been a reason not to use the truncated SVD. However, we will not have to deal with this problem as we are interested in the singular vectors corresponding to the largest singular values.

2.4 Calculating the SVD

As mentioned before, calculating A^*A directly in order to calculate its eigendecomposition is not an option (unless A is sufficiently small) and in order to bypass this, we will have to take a look at other methods. Luckily, A^TA and AA^T are both Hermitian, which allows for the use of more specific eigenproblem algorithms.

Most of these algorithms involve transforming the SVD problem $A = U\Sigma V^T$ into a SVD problem $B = U_B\Sigma_B V_B^T$, where B is bidiagonal. This property of B gives us the big advantage that B^*B is then symmetric tridiagonal, which makes its SVD much easier to find, since there are algorithms available for solving the eigenproblem of symmetric tridiagonal matrices. Using the solution for the SVD of B obtained through such a method, a SVD for A can then be found.

We too will be using such an algorithm. To bidiagonalize our initial SVD problem, the Golub-Kahan Upper Bidiagonalization method[1] is used on a truncated feature matrix A_t.

2.4.1 Golub-Kahan-Lanczos Upper Bidiagonalization

The derivation of the Golub-Kahan-Lanczos Bidiagonalization procedure as shown in this chapter closely follows that of [1], node 198 on their webpage. This procedure is used to find a bidiagonal matrix B and orthogonal matrices U_1 and V_1 such that $U_1^T A_t V_1 = B$, where $A_t = U_t\Sigma V_t^T$ is the original SVD we wish to find. By then finding the SVD of $B = U_2\Sigma V_2^T$, we will be able to solve the initial SVD problem, as $U_1^T A_t V_1 = B = U_2\Sigma V_2^T$, which implies that $A_t = U_1U_2\Sigma(V_1V_2)^T$, since U_1, U_2, V_1 and V_2 are orthogonal by definition.

The first step is to find an upper bidiagonal matrix B, which means that it is of the form

$$B = \begin{bmatrix} \alpha_1 & \beta_1 & & & \\ & \alpha_2 & \beta_2 & & \\ & & \ddots & \ddots & \\ & & & \alpha_{n-1} & \beta_{n-1} \\ & & & & \alpha_n \end{bmatrix} \tag{5}$$

, such that $U_1^T A_t V_1 = B$. Let u_k and v_k be the kth column vector of U_1 and V_1 respectively. Multiplying both sides by U_1 results in $A_t V_1 = U_1 B$, which gives us for the kth column vectors

$$A_t v_k = \beta_{k-1} u_{k-1} + \alpha_k u_k \tag{6}$$

or equivalently

$$\alpha_k u_k = A_t v_k - \beta_{k-1} u_{k-1} \tag{7}$$

where $\beta_0 = 0$. Taking the transpose of both sides, the equality $U_1^T A_t V_1 = B$ becomes $V_1^T A_t^T U_1 = B^T$. Multiplying both sides by V_1 results in $A_t^T U_1 = V_1 B^T$, giving us for the kth column vectors

$$A_t^T u_k = \alpha_k v_k + \beta_k v_{k+1} \tag{8}$$

or equivalently

$$\beta_k v_{k+1} = A_t^T u_k - \alpha_k v_k \tag{9}$$

Remember that U_1 and V_1 are orthogonal matrices, which implies that their columns are normalized. Therefore, solving for α_k and β_k gives us

$$\alpha_k = ||\alpha_k u_k||_2 = ||A_t v_k - \beta_{k-1} u_{k-1}||_2 \tag{10}$$

and

$$\beta_k = ||\beta_k v_{k+1}||_2 = ||A_t^T u_k - \alpha_k v_k||_2 \tag{11}$$

Using these equalities, we can now write the (pseudo)code for the double recursion that is Golub-Kahan-Lanczos upper bidiagonalization.[1]

Algorithm 1 Golub-Kahan Upper Bidiagonalization

procedure GOLUB-KAHAN UPPER BIDIAGONALIZATION(A_t)
  $\beta_0 \leftarrow 0$
  $v_1 \leftarrow$ arbitrary unit (2-norm) vector
  for k `in` $1..n$ **do**
    $u_k \leftarrow A_t v_k - \beta_{k-1} u_{k-1}$ ▷ $\beta_0 = 0$, so it does not matter that u_0 is undefined
    $\alpha_k \leftarrow \|u_k\|_2$
    $u_k \leftarrow u_k / \alpha_k$
    $v_{k+1} \leftarrow A_t^T u_k - \alpha_k v_k$
    $\beta_k \leftarrow \|v_{k+1}\|_2$
    $v_{k+1} \leftarrow v_{k+1} / \beta_k$
  return α, β, u, v

After performing Golub-Kahan-Lanczos upper bidiagonalization, we have obtained B, U_1 and V_1. The next step is to find the SVD of B. Since B is a real upper bidiagonal matrix, $B^T B$ and $B B^T$ are both real symmetric tridiagonal matrices. When B is defined as in (5), the explicit forms of $B^T B$ and $B B^T$ are given as

$$B^TB = \begin{bmatrix} \alpha_1^2 & \alpha_1\beta_1 & & & \\ \alpha_1\beta_1 & \alpha_2^2+\beta_1^2 & \alpha_2\beta_2 & & \\ & \alpha_2\beta_2 & \ddots & \ddots & \\ & & \ddots & \alpha_{n-1}^2+\beta_{n-2}^2 & \alpha_{n-1}\beta_{n-1} \\ & & & \alpha_{n-1}\beta_{n-1} & \alpha_n^2+\beta_{n-1}^2 \end{bmatrix} \tag{12}$$

and

$$BB^T = \begin{bmatrix} \alpha_1^2+\beta_1^2 & \alpha_2\beta_1 & & & \\ \alpha_2\beta_1 & \alpha_2^2+\beta_2^2 & \alpha_3\beta_2 & & \\ & \alpha_3\beta_2 & \ddots & \ddots & \\ & & \ddots & \alpha_{n-1}^2+\beta_{n-1}^2 & \alpha_n\beta_{n-1} \\ & & & \alpha_n\beta_{n-1} & \alpha_n^2 \end{bmatrix} \tag{13}$$

To compute the SVD of B, the eigenproblems of B^TB and BB^T need to be solved, as the solutions for these problems can be directly used to find the SVD of B. To solve the eigenproblems, an eigenproblem solver for real symmetric tridiagonal matrices from the SciPy library is used[3]. Once the solver has done its job, U_2, V_2 and Σ can be easily determined, according to corollary 2.3.1. Since $A_t = U_1U_2\Sigma(V_1V_2)^T$, calculating $U_t = U_1U_2$ and $V_t = V_1V_2$ finally yields the truncated SVD of A_t.

2.5 Dimensionality reduction

Having seen how to find principal components, all that is left to do is to use these principal components to map our feature vectors to a lower dimensional space. The principal components form a basis of this space and each feature vector will be written as a linear combination of this basis that best approximates the initial feature vector. That is, each feature vector will be projected onto the principal components, which are unit vectors. Let p be the number of principal components and n the number of m-dimensional feature vectors such that $p < m$. Each of these feature vectors is then mapped by a function $f : \mathbb{R}^m \to \mathbb{R}^p$, such that

$$f(\vec{x}) = (\vec{x} \cdot \vec{c_1}, \vec{x} \cdot \vec{c_2}, \ldots, \vec{x} \cdot \vec{c_p})^T \tag{14}$$

where c_1 is the first principal component, c_2 the second, et cetera. These reduced feature vectors can now be stored in a feature matrix of dimensions $\mathbb{R}^{n \times p}$. This concludes the section about PCA. Next up in the pipeline is the classification phase, of which the specifics will be explained in the following section.

3 Classification

Now that we have become acquainted with PCA, it is time to tackle classification. Remember that a classifier is a piece of software that is first trained on a training dataset (Figure 5), in order to later be able to classify feature vectors from a testing dataset (Figure 6).

The specific classifier that we will be using is a **Support Vector Machine (SVM)** from the scikit-learn package[11]. Mainly since it is known to be a fast algorithm that can be run in a reasonable amount of time on a personal computer, which is especially useful when working with large datasets.

Other classifiers include **neural networks**[9] and **decision trees**. The reason for not using them is that SVM has preferable properties, some of which will be mentioned later on on this section.

3.1 Support Vector Machine

To explain how an SVM works, let us consider a dataset of entries belonging to two different classes. An SVM will aim to find a separating hyperplane, in the space that the data resides in, that best splits the data into these two classes. Even if we assume that our dataset in question can be perfectly split by class, there are multiple ways of choosing this hyperplane. Where SVMs differ from other classifiers such as neural networks[9], is that instead of having all data points influence optimality, only those data points that are close to the decision boundary are taken into account when choosing a decision boundary, a hyperplane in this case.

For convenience and to conform to the terminology as used in the field of classification, we shall introduce a couple of terms. We define the **margin** of a hyperplane as the minimal distance between the hyperplane and any data point. In the context of SVMs, we say that a separating hyperplane is **optimal** when its margin is maximized. Any data points that have a distance to the hyperplane that is equal to the margin are called **support vectors**.

In extreme cases, a training set can consist completely out of support vectors, though this is rarely the case. In most cases, the subset of training samples consisting of its support vectors is relatively very small. And the optimal hyperplane is uniquely determined by this support vector subset.

3.2 Finding the hyperplane

The aim of a SVM is to find the hyperplane that best splits the feature vectors into their respective classes, which in this context will be called the **optimal hyperplane** H. We shall attempt to derive a method for obtaining H.

For the sake of simplicity at this stage, we consider a dataset that has only two classes, which we shall denote by $+$ and $-$. For such a dataset, we can define a hyperplane that divides the feature vectors in the dataset into its classes, but is not necessarily optimal, as follows. We use a similar approach as [18], section 5.

Definition 3.1 (Dividing hyperplane)**.** *A dividing hyperplane H_d of a set of feature vectors belonging to two classes + and − is exactly the set of vectors $\vec{x}$ for which*

$$\vec{w} \cdot \vec{x} + b = 0 \tag{15}$$

, where b is a ***bias*** *constant and $\vec{w}$ is a vector orthogonal to H_d such that*

$$\begin{aligned} \vec{w} \cdot \vec{x}_+ + b &\geq 1 \\ \vec{w} \cdot \vec{x}_- + b &\leq -1 \end{aligned} \tag{16}$$

holds true for any two feature vectors $\vec{x}_+$ and $\vec{x}_-$ from the training dataset, belonging to the classes + and − respectively. We call $\vec{w}$ the ***weight vector*** *of H_d.*

For mathematical convenience, it helps to introduce for each feature vector x_i a constant y_i that allows us to reformulate (16) as a single (in)equality. By introducing

$$y_i = \begin{cases} 1, & \text{if } x_i \text{ belongs to class } + \\ -1, & \text{if } x_i \text{ belongs to class } - \end{cases}$$

(16) can be reformulated as

$$y_i(\vec{w} \cdot \vec{x_i} + b) \geq 1 \tag{17}$$

or equivalently as

$$y_i(\vec{w} \cdot \vec{x_i} + b) - 1 \geq 0 \tag{18}$$

for any feature vector x_i.

As mentioned earlier, the dividing hyperplane of interest is one with a margin that is maximal. The margin of a hyperplane is determined by its support vectors, the vectors with endpoints closest (Euclidean) to the hyperplane itself. All feature vectors $\vec{x_i}$ for which $\vec{y_i}(\vec{w} \cdot \vec{x}_i + b) - 1 = 0$ is true are called the support vectors of a dividing hyperplane H_d.

Picking two support vectors $\vec{x}_+$ and $\vec{x}_-$, belonging to class + and − respectively, the margin d itself is the distance between $\vec{x}_+$ and $\vec{x}_-$ in the direction of $\vec{w}$. That is,

$$d = (\vec{x}_+ - \vec{x}_-) \cdot \frac{\vec{w}}{||\vec{w}||} \tag{19}$$

and because of (18) this is equivalent to

$$d = \frac{2}{||\vec{w}||} \tag{20}$$

.

To find the optimal dividing hyperplane for our dataset, the margin d needs to be maximized. In other words, the aim is to find a $\vec{w}$ such that $||\vec{w}||$, or $\frac{1}{2}||\vec{w}||^2$ for convenience

later on, is minimized, with (18) as a constraint for each $\vec{x}_i$. In order to be able to solve this problem using the Lagrange multiplier method[2], we formulate it as follows

$$\begin{aligned} f(\vec{w}) &= \frac{1}{2}||\vec{w}||^2 \\ g_i(\vec{w}, b) &= y_i(\vec{x}_i \cdot \vec{w} + b) - 1 \geq 0 \end{aligned} \tag{21}$$

Writing it in this form reveals an advantage of SVMs over some other classifiers. Since f is quadratic, there are no other optima aside from the global minimum. Neural networks, for example, have a tendency to get stuck in local optima[9] - a problem that SVMs avoid completely by having no local optima (aside from the global optimum) to begin with.

In the form of a Lagrangian, it becomes

$$\begin{aligned} L(\vec{w}, b, \lambda) &= f(\vec{w}) - \sum_i \lambda_i g_i(\vec{w}, b) \\ &= \frac{1}{2}||\vec{w}||^2 - \sum_i \lambda_i (y_i(\vec{w} \cdot \vec{x}_i + b) - 1) \end{aligned} \tag{22}$$

with a gradient

$$\nabla L(\vec{w}, b, \lambda) = \nabla f(\vec{w}) - \sum_i \lambda_i \nabla g_i(\vec{w}, b) = 0 \tag{23}$$

, giving us

$$\begin{aligned} \frac{\partial}{\partial \vec{w}} L(\vec{w}, b, \lambda) &= \vec{w} - \sum_i \lambda_i y_i \vec{x}_i = 0 \\ \frac{\partial}{\partial b} L(\vec{w}, b, \lambda) &= -\sum_i \lambda_i y_i = 0 \\ \frac{\partial}{\partial \lambda_i} L(\vec{w}, b, \lambda) &= y_i(\vec{w} \cdot \vec{x}_i + b) - 1 = 0 \end{aligned} \tag{24}$$

Applying (24) to (22) results in

$$\begin{aligned} L(\vec{w}, b, \lambda) &= \frac{1}{2}(\sum_i \lambda_i y_i \vec{x}_i) \cdot (\sum_j \lambda_j y_j \vec{x}_j) - (\sum_i \lambda_i y_i \vec{x}_i) \cdot (\sum_j \lambda_j y_j \vec{x}_j) - \sum_i \lambda_i y_i b + \sum \lambda_i \\ &= -\frac{1}{2}(\sum_i \lambda_i y_i \vec{x}_i) \cdot (\sum_j \lambda_j y_j \vec{x}_j) + \sum \lambda_i \\ &= \sum_i \lambda_i - \frac{1}{2} \sum_i \sum_j \lambda_i \lambda_j y_i y_j (\vec{x}_i \cdot \vec{x}_j) \end{aligned} \tag{25}$$

By the Kuhn-Tucker theorem[18], the problem of maximizing $L(\vec{w}, b, \lambda)$ such that $\sum_i \lambda_i y_i = 0$ and $\lambda_i \geq 0$ shares the same solution with our initial problem. This is great news, as it turns out that are now able to solve our initial problem through calculation of the dot products $\vec{x}_i \cdot \vec{x}_j$. This makes clear why we went through the whole process of reformulation our initial problem. All that is left to do is to find λ_i. How to do so can be found in [18].

3.3 Extension to multiclass problems

Unlike the previous example, not every classification problem is so simple that it can be solved with a binary SVM. In fact, for most classification problems one has to deal with way more than 2 classes. Luckily, binary SVMs as explained thus far can be extended to multiclass SVMs to train and classify datasets with $k > 2$ classes.

One way of doing so is called the one versus all strategy. This involves training k binary SVMs, pitting each class against all the other classes, as the name suggests. During the testing phase, an unclassified feature vector would be tested by these k binary SVMs and be classified as belonging or not belonging to each of the k classes. What could be a drawback of this approach are that a testing feature vector can be classified as belonging to multiple classes, or none. For classification problems where the classes do not need to be mutually exclusive, this does not pose a problem. When classes are constrained to be mutually exclusive however, using this approach is simply not an option because of this property.

A multiclass SVM that avoids this problem is one built around the one versus one strategy, which trains $\frac{k(k-1)}{2}$ binary SVMs - one for each pair of classes. When classifying a new feature vector, running it through the $\frac{k(k-1)}{2}$ binary SVMs yields $\frac{k(k-1)}{2}$ classification outcomes, from which the majority vote is taken as the actual classification. Immediately, it seems alarming that such a big number of SVMs is to be trained. In practice however, the number of classes is almost always very small compared to the number of feature vectors and the number of dimensions, so more often than not, this does not pose a problem.

3.4 Kernels

Naturally, data sets that do not allow to be split in such a way exist. To show an example; imagine that we are classifying two bird species. Both species share a strikingly similar appearance, but one trait gives them away immediately. The first species has horizontal stripes and the other vertical stripes, and no, we are not rotating birds by 90 degrees. Now, one bird of the first species could have a mutation, that only affects the orientation of the stripes - they are now vertical. Though from the inside, it is still a bird of the 'horizontal species'. A classifier would likely not be able to tell the difference between this bird and the birds of the 'vertical species'.

A great thing about SVMs is that, if such a thing is the case in the training dataset, so-called **kernels** can be used to transform the feature space such that the dataset becomes linearly separable. The general idea is that the data gets mapped by a function ϕ to a higher

dimensional space in order to gain linear separability. By using such a transformation, our function to optimize (25) would become

$$L(\vec{w}, b, \lambda) = \sum_i \lambda_i - \frac{1}{2} \sum_i \sum_j \lambda_i \lambda_j y_i y_j (\phi(\vec{x}_i) \cdot \phi(\vec{x}_j)) \tag{26}$$

This could be costly if ϕ is computationally heavy. However, since only $\phi(\vec{x}_i) \cdot \phi(\vec{x}_j)$ is needed to find a solution for our SVM, it suffices to have a **kernel function** K such that $K(\vec{x}_i, \vec{x}_j) = \phi(\vec{x}_i) \cdot \phi(\vec{x}_j)$. This allows us to bypass any direct computations of ϕ and even rids us of the need to know ϕ explicitly. The function to optimize then becomes

$$L(\vec{w}, b, \lambda) = \sum_i \lambda_i - \frac{1}{2} \sum_i \sum_j \lambda_i \lambda_j y_i y_j K(\vec{x}_i, \vec{x}_j) \tag{27}$$

A popular kernel function is the polynomial kernel, which is defined as

$$K(\vec{x}_i, \vec{x}_j, \gamma, r, d) = (\gamma(\vec{x}_i \cdot \vec{x}_j) + r)^d \tag{28}$$

Another popular kernel function is rbf, given as

$$K(\vec{x}_i, \vec{x}_j, \gamma) = \exp(-\gamma ||\vec{x}_i - \vec{x}_j||^2) \tag{29}$$

where $\gamma > 0$. Lastly, there is the sigmoid kernel function, which is defined as follows

$$K(\vec{x}_i, \vec{x}_j, \gamma, r) = \tanh(\gamma(\vec{x}_i \cdot \vec{x}_j) + r) \tag{30}$$

3.5 Overfitting

Even if a dataset is not linearly separable, an SVM can still be used on this dataset by making clever use of kernels. However, in some cases it is hard to reach linear separability even with the aid of kernel functions. And if one does succeed in making it linearly separable, the SVM is at high risk of **overfitting**. Overfitting is a phenomenon where a classifier is trained to the point that it suffers from not being generalized enough. Figure 7 shows an example where it is preferred to find a decision boundary that does not separate the feature vectors perfectly.

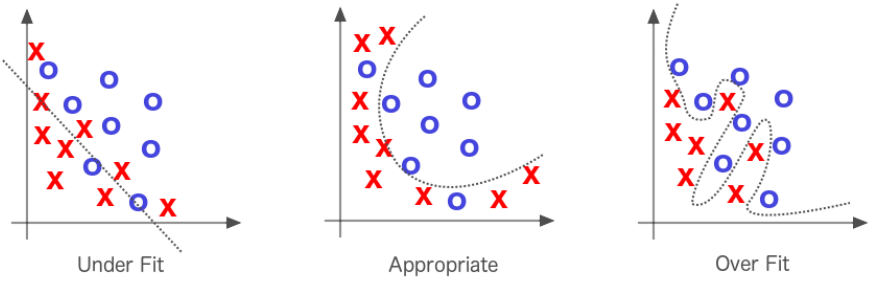

Figure 7: Underfitting and overfitting of a classifier on a training dataset

If overfitting behaviour is caused by two classes not having a clear boundary, but rather a boundary streak where a mix of feature vectors of both classes is prevalent, **soft margins** could offer a solution.

The SVMs as we have discussed so far are all hard margin SVMs, that do not allow for decision hyperplanes that do not split the feature vectors perfectly by class. By to some extent allowing outliers when finding a decision hyperplane, soft margin SVMs seem to have a natural defence mechanism against overfitting.

4 Preprocessing

After becoming familiar with PCA and SVMs, it is time for preprocessing to join them on the battlefield that is called classification. It might not come as a surprise that the class of preprocessing methods is huge. In fact, any function that takes a vector to any other vector can be considered a preprocessing method for PCA, but not all of them are born equally useful. This section will introduce a number of promising candidates, that will be tested on their effects on classification accuracy in combination with PCA.

Although our input at this stage consists of feature vectors containing color values, we will be considering them two-dimensional images, or matrices. In this section, images are not to be confused with the mathematical notion of an image of a function. During the feature vector extraction step, the dimensions of each input image are stored, so they can be easily converted back into images at this step. This view will help with our understanding of the image preprocessing methods.

4.1 Gaussian blur

Blurring is a common type of image processing that knows many forms, with one of the most popular type of blur being Gaussian blurring. Intuitively speaking, a blur is a procedure

that alters the input image in such a way that the color value of each pixel of the image becomes more similar to the color values of its neighbouring pixels.

Before we can define our blurs, we need to introduce two functions s_{up}, $s_{\text{left}} : \mathbb{R}^{a\times b} \to \mathbb{R}^{a\times b}$ such that for any matrix $C = (c_{i,j}) \in \mathbb{R}^{a\times b}$ the functions are defined as

$$s_{\text{up}}((c_{i,j})) = \begin{bmatrix} c_{1,2} & c_{2,2} & \dots & c_{a-1,2} & c_{a,2} \\ c_{1,3} & c_{2,3} & \dots & c_{a-1,3} & c_{a,3} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{1,b} & c_{2,b} & \dots & \ddots & c_{a,b} \\ c_{1,1} & c_{2,1} & \dots & c_{a-1,1} & c_{a,1} \end{bmatrix} \tag{31}$$

and

$$s_{\text{left}}((c_{i,j})) = \begin{bmatrix} c_{2,1} & c_{3,1} & \dots & c_{a,1} & c_{1,1} \\ c_{2,2} & c_{3,2} & \dots & c_{a,2} & c_{1,2} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{2,b-1} & c_{3,b-1} & \dots & \ddots & c_{1,b-1} \\ c_{2,b} & c_{3,b} & \dots & c_{a,b} & c_{1,b} \end{bmatrix} \tag{32}$$

Explained in words, s_{up} shifts a whole $a \times b$ matrix one row up, moving the uppermost row to the bottom of the output matrix. s_{left} shifts it one column to the left, adding the leftmost column to the right. Naturally, the inverse of these functions can be described as a shift in the opposite direction.

When using a blur, one can select a range $k \in \mathbb{Z}_{>0}$, which is a distance in pixels (the distance to a diagonal neighbour counts as 1 pixel). A pixel in the image to be blurred will be affected by any pixel that is within k pixels away.

The various types of blurs are characterized by their unique weight function $W : \mathbb{Z}\times\mathbb{Z} \to \mathbb{R}$, which is called a **kernel** (not to be confused with SVM kernels), such that

$$\sum_{x=-k}^{k}\sum_{y=-k}^{k} W(x,y) = 1 \tag{33}$$

If $B \in \mathbb{R}^{a\times b}$ is an image, the blurred image B_{blurred} would be defined as

$$B_{\text{blurred}} = \sum_{x=-k}^{k}\sum_{y=-k}^{k} W(x,y)(s_{\text{left}}^{x} \circ s_{\text{up}}^{y}(B)) \tag{34}$$

where the notation s_{left}^{x} stands for the x times repeated application of the function s_{left}.

One of the simplest forms of blurring is called **box blurring**. This type of blur has a kernel defined as

$$W_{\text{box blur}}(x,y) = \frac{1}{(k+2)^2} \tag{35}$$

which is not at all dependent on the distance at all.

Gaussian blurring is different from box blurring in the sense that it has a preference for pixels to have a bigger effect on nearby pixels the nearer they are. In fact, Gaussian blur uses the two-dimensional Gaussian function as a kernel. Instead of choosing a range k, the value of k is dependent on the user's choice of the standard deviation σ. If needed, the kernel is normalized such that (33) is (almost) true for this kernel.

Definition 4.1. *The **one-dimensional Gaussian function** is*

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \tag{36}$$

*, and the **two-dimensional Gaussian function** is the product of two one-dimensional Gaussian functions*

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{37}$$

, where x is the horizontal distance to the origin, y the vertical distance and σ the standard deviation of the Gaussian distribution.

[16]

A blur like this, that takes distance into account when it comes to influencing strength of a nearby pixel, has a greater tendency to retain details of the image, while still blurring it.

4.1.1 Expected results

We expect Gaussian blur to have a positive effect on classification accuracy for the MNIST dataset. To name an example, taking a look at the MNIST digits shown in Figure 5, shows that not all zeroes in this subset form closed circles, and some might not be neatly closed. By applying Gaussian blur, these imperfections get 'softened up', making the zeroes look more alike and making the circle shape they have in common easier for PCA to pick up on.

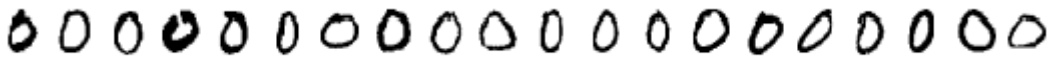

Figure 8: A few entries from the MNIST handwritten digits database: raw images

Figure 9: A few entries from the MNIST handwritten digits database: blurred with Gaussian blur with a range of 10

4.2 Bilateral filtering

With Gaussian blur, it is almost inevitable that edge information gets lost. One way to modify this process to retain edge information is to have the kernel depend on not only the distance to other pixels, but also on differences in color value.

Bilateral filtering comes with two additional parameters, σ_{color} and σ_{space}. The first parameter, σ_{color}, determines the size of the range of color values that can influence pixels with color values within that color range, regardless of distance. In practice, a higher σ_{color} value would make similar colors in an image even more similar, leading to solid blocks of identical colors. The second parameter, σ_{space}, determines the size of the area over which influence by color value (as a result of σ_{color}) is effective.

This makes bilateral filtering an excellent tool for ridding images of noise, while retaining edge information. However, it is very slow compared to blurs such as Gaussian blur and box blur.

4.2.1 Expected results

Bilateral filtering will not be able to fill any gaps like Gaussian blur does, because of its edge-retaining behaviour. However, it is capable of denoising an image while leaving its edges intact. On the MNIST dataset, this property of bilateral filtering will likely not shine, since the edges of the handdrawn digits are quite soft. One thing that could turn the tables for bilateral filtering is how it tries to equalize colors that are similar. Things like pen streaks that were written with low pressure might become more prevalent, revealing parts of images that might hold valuable information.

5 Results

To test for classification accuracy, the MNIST database of handwritten digits[10] has been used. This dataset has seen extensive use in academic research, and thus allows for comparisons to other work.

5.1 Experiment setup

For our experiment, we first randomly pick two subsets from both the MNIST training and testing data and extract feature vectors, as in Figure 1. While we will be running the input data through our classification pipeline multiple times, these subsets will remain constant. The things that will be changing in-between those runs are whether certain preprocessing methods are applied or not. This includes PCA, Gaussian blur and bilateral filtering. At the end of each run, the percentage of correctly classified entries is returned; the classifier accuracy.

Whenever PCA is enabled for a run, it will repeat the run 75 times with different amounts of principal components, ranging from 2 to 150, with increments of 2. At the end of this

series of runs, the pipeline will return an array of classifier accuracies corresponding to each run. In addition, it will return an array of singular values corresponding to the principal components used during the last run.

Before we can run tests, there are a number of parameters that need to be carefully selected. Of course, the sizes of the training dataset and the testing dataset in relation to each other are important. Having too small a training dataset in proportion to the testing dataset will likely lead to overfitting. Just as in the real world, a small sample size is rarely representative of the general case. Have too small a testing set would increase the influence that any outliers in the testing set have on the classifier accuracy. A rule of thumb for a good split is to reserve between 10% and 25% of the entire dataset as testing data[8]. In our experiments, we will be using a ratio of 80% training and 20% testing data, with 3000 training and 750 testing samples.

As for parameters of the preprocessing methods themselves, Gaussian blur has a range of $k = 5$, with the exception of what is denoted as 'Gaussian blur (3)'; this one has a range of $k = 3$. Bilateral filter has parameters $\sigma_{\text{color}} = \sigma_{\text{space}} = 75$.

We will be using the GaussianBlur and bilateralFilter methods from the OpenCV library[19].

5.2 Experiment results and discussion

In Figure 10 the resulting accuracy scores, on the same input dataset and using various combinations of preprocessing methods, are shown. At first glance, it seems like runs that included PCA had a hard time catching up to the runs without PCA, with a few exceptions. However, it is quite a good result when we take into account that for these runs, the feature vectors have undergone dimensionality reduction. All of them are falling within the 86% 90% accuracy range when at least 40 principal components are used, similar to the runs without PCA.

Contrary to our expectations, bilateral filtering without PCA achieved a high accuracy score. The idea of faint pen streaks becoming more prevalent due to bilateral filtering could be the underlying cause.

What is also unexpected is the relatively poor performance of any run that had Gaussian blur with range $k = 5$ enabled. Causes could be that by having a blur that can reach 5 pixels, which is very far in a 28×28 MNIST image, a big portion of the information is getting lost, making the images look muddy and more alike those of other labels. This is further supported by how well PCA and Gaussian (3) perform together. After all Gaussian (3) has a way shorter reach than the Gaussian blur with range $k = 5$.

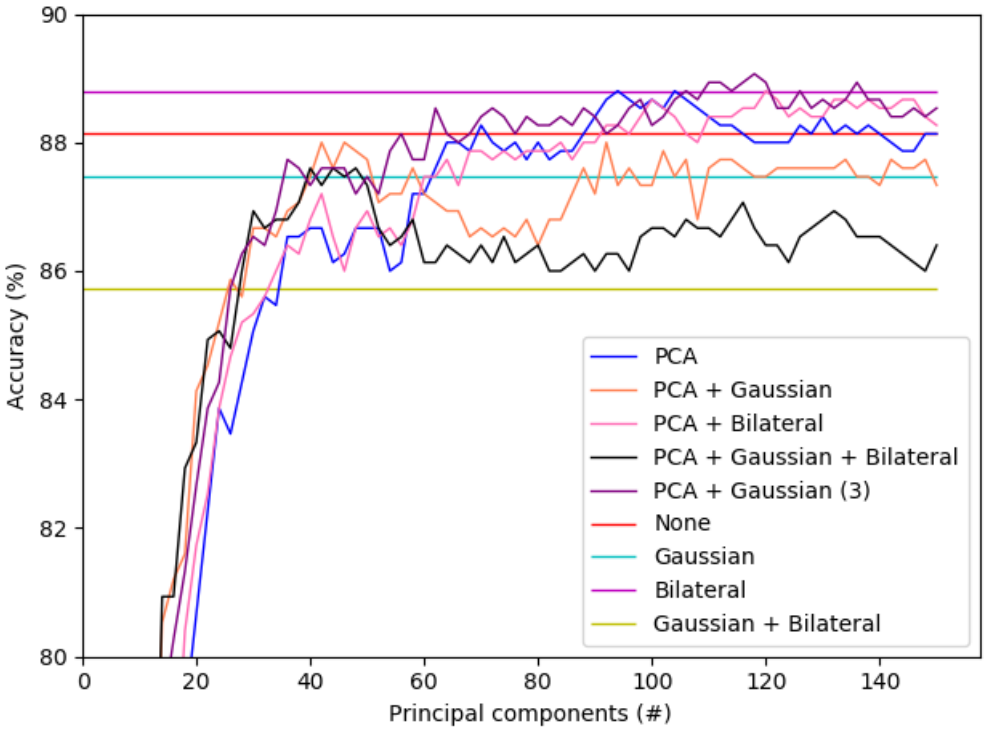

Figure 10: Classification accuracy of a SVM on 3000 training images and 750 testing images of the MNIST handwritten digit dataset, with different combinations of preprocessing methods.

Figure 11 shows the singular values that correspond to the principal components that were computed during the runs. Here we can see two clear groups of preprocessing method combinations that show different behaviour. PCA and PCA in combination with bilateral filter keep picking up relatively significant principal components, even for higher principal component numbers. For all preprocessing combinations containing Gaussian blur (with a range of $k = 5$), it seems to die down quickly. Initially, they find principal components corresponding to higher singular values than the rest, however. Its cause could be that Gaussian blur still creates some sort of similarity between images belonging to the same class, by filling holes for examples. However, it still tosses information all over the place because of its relatively high range, so after PCA picking up on the things that it has made more similar, there is little interesting structure left behind.

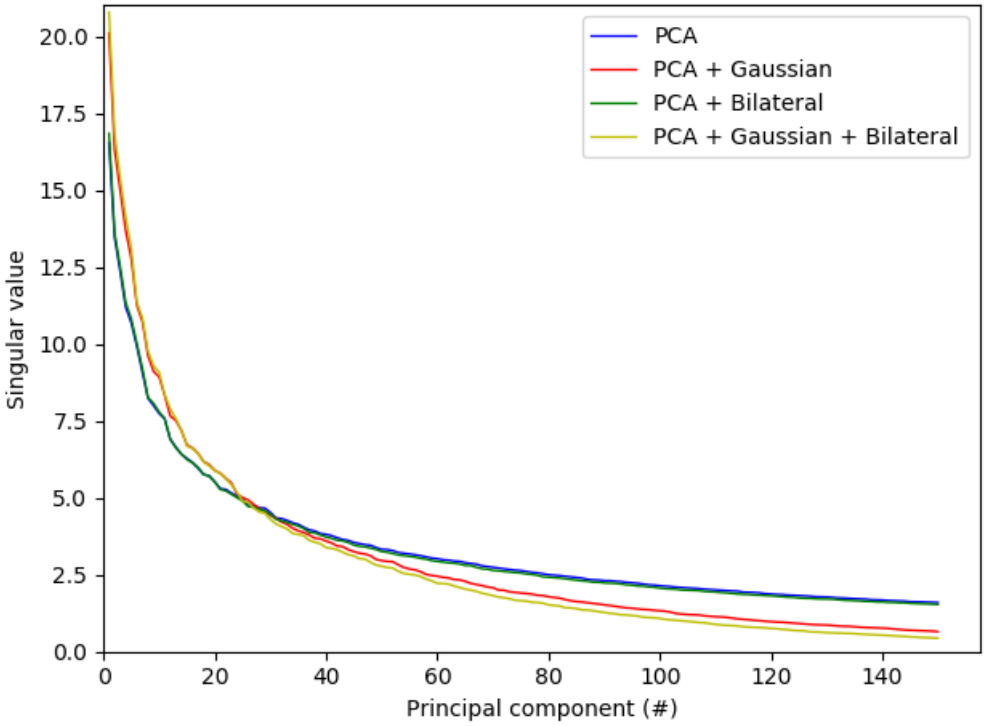

Figure 11: Singular values corresponding to each principal component as found when performing PCA on 3000 training images and 750 testing images of the MNIST handwritten digit dataset, with various combinations of preprocessing steps.

6 Conclusion

All in all we can say that using PCA as a dimensionality reduction method, and possibly also doubling it as a denoiser, is a viable strategy when it comes to classification problems. Even when using small numbers of principal components, it produces favorable results whilst keeping dimensionality low. Using Gaussian blurring or bilateral filtering in combination with PCA did not achieve any outstanding results when used on the MNIST handwritten digits database, however.

7 Future research

Unfortunately, there is only so much that one can fit into a bachelor's thesis, forcing one to leave quite a number of interesting research topics untouched. An obvious example is that a similar kind of research can be done using different classifiers or preprocessing methods. The classifier and preprocessing methods used in this thesis are not too computationally heavy. Fewer technical limitations would clear the way for more possible combinations, likely with higher classification accuracy. This is not to say that the results found in this thesis are of no value, as technical limitations are very much present in the real world. A second example includes the use of another feature extraction called **Independent Component Analysis (ICA)**[15]. While PCA tries to find correlation in the input features through maximizing variance, ICA tries to find statistical independence. When performing PCA on a set of images of human faces, the first principal component would likely look like an average face. Performing ICA, on the other hand, would give as its independent components the parts of the face that can vary from face to face independently, such as hair, eyes and noses. Having a certain hair color is most likely (nearly) uncorrelated with having a certain nose shape, especially considering the amount of people that dye their hair nowadays. ICA might be interesting to use in combination with decision trees, as these classifiers are built to simulate a web of choices that ends in a classification. Each of those choices could be related to one such independent component.

References

[1] Zhaojun Bai et al. *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*. URL: http://www.netlib.org/utk/people/JackDongarra/etemplates/book.html.

[2] Dimitri P. Bertsekas. *Constrained Optimization and Lagrange Multiplier Methods*. Academic Press, 2014.

[3] The SciPy Community. *SciPy*. URL: https://scipy.org/.

[4] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, 2000. ISBN: 9780521780193.

[5] A. M. Davie and A. J. Stothers. "Improved bound for complexity of matrix multiplication". In: *Royal Society of Edinburgh* 143.2 (2013), pp. 351–369. URL: https://www.cambridge.org/core/journals/proceedings-of-the-royal-society-of-edinburgh-section-a-mathematics/article/improved-bound-for-complexity-of-matrix-multiplication/998F772AF916572803EBA9C1AD7B4FC1.

[6] Kim Esbensen and Paul Geladi. "Principal component analysis". In: *Chemometrics and Intelligent Laboratory Systems* 2.1-3 (1987), pp. 37–52. URL: https://www.sciencedirect.com/science/article/pii/0169743987800849#!.

[7] Gene H. Golub. *Matrix Computations*. JHU Press, 2012.

[8] Isabelle Guyon. "A scaling law for the validation-set training-set size ratio". In: (). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.33.1337&rep=rep1&type=pdf.

[9] Ben Kröse et al. *An introduction to Neural Networks*. University of Amsterdam, 1993.

[10] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. *The MNIST database of handwritten digits*. URL: http://yann.lecun.com/exdb/mnist/.

[11] *Linear Support Vector Machine*. URL: http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html. (accessed: 01.09.2016).

[12] Ilias G. Maglogiannis. *Emerging Artificial Intelligence Applications in Computer Engineering*. IOS Press, 2007. ISBN: 9781586037802.

[13] D. D. Muresan and T. W. Parks. "Adaptive principal components and image denoising". In: *Proceedings 2003 International Conference on Image Processing* 1 (2003), pp. I-101-4. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1246908&isnumber=27937.

[14] Sylvain Paris et al. *scikit-learn Support Vector Machines documentation*. URL: https://people.csail.mit.edu/sparis/bf_course/course_notes.pdf.

[15] PierreComon. "Independent component analysis, A new concept?" In: *Signal Processing* 36.3 (1994), pp. 287–314. URL: https://www.sciencedirect.com/science/article/pii/0165168494900299.

[16] Linda Shapiro and George Stockman. *Computer vision*. 2000.

[17] Jonathon Shlens. "A Tutorial on Principal Component Analysis". In: (2014). URL: https://arxiv.org/pdf/1404.1100.pdf.

[18] Baxter Tyson Smith. "Lagrange Multipliers Tutorial in the Context of Support Vector Machines". In: (2004). URL: https://www.engr.mun.ca/~baxter/Publications/LagrangeForSVMs.pdf.

[19] OpenCV Team. *OpenCV library*. URL: https://opencv.org/.

[20] *Truncated SVD and its Applications*. URL: http://langvillea.people.cofc.edu/DISSECTION-LAB/Emmie'sLSI-SVDModule/p5module.html.

Appendix A Algorithms

Algorithm Golub-Kahan Upper Bidiagonalization

procedure GOLUB-KAHAN UPPER BIDIAGONALIZATION(A_t)
  $\beta_0 \leftarrow 0$
  $v_1 \leftarrow$ arbitrary unit (2-norm) vector
  for k `in` $1..n$ **do**
    $u_k \leftarrow A_t v_k - \beta_{k-1} u_{k-1}$ ▷ $\beta_0 = 0$, so it does not matter that u_0 is undefined
    $\alpha_k \leftarrow \|u_k\|_2$
    $u_k \leftarrow u_k / \alpha_k$
    $v_{k+1} \leftarrow A_t^T u_k - \alpha_k v_k$
    $\beta_k \leftarrow \|v_{k+1}\|_2$
    $v_{k+1} \leftarrow v_{k+1} / \beta_k$
  return α, β, u, v

Algorithm Principal Component Analysis

procedure PCA(A, t, threshold)
  $A_t \leftarrow$ TRUNCATE(A, t) ▷ Forms truncated matrix A_t of given size t
  $u, s, vt \leftarrow \text{SVD}(A_t)$
  $s \leftarrow s[(\geq \text{threshold})]$ ▷ Truncate s such that all singular values are $\geq$ threshold
  $vt \leftarrow vt[: s.length]$ ▷ Remove columns $vt_{\text{s.length}} \ldots vt_{\text{vt.length}}$ from vt
  return s, vt ▷ We use the columns of vt as principal components

Algorithm SVD

procedure SVD(A_t)
  α, β, u_1, $v_1 \leftarrow$ GOLUB-KAHAN UPPER BIDIAGONALIZATION(A_t)
  $\gamma_1 \leftarrow \alpha_1^2$ ▷ Start computing $B^T B$. γ is the diagonal of $B^T B$
  for k `in` $1..t-1$ **do**
    $\gamma_{i+1} \leftarrow \alpha_{i+1}^2 + \beta_i^2$
    $\delta_i \leftarrow \alpha_i \beta_i$ ▷ δ is the off-diagonal of $B^T B$
  Let $B^T B$ be the matrix with γ as its diagonal, δ as its off-diagonal
  $u_2, s^2, v_2 \leftarrow$ EIGENDECOMPOSITION($B^T B$)
  $u \leftarrow u_1 u_2$
  $vt \leftarrow$ TRANSPOSE($v_1 v_2$)
  return u, s, vt