IT UNIVERSITY OF COPENHAGEN

Machine Learning Project

Autumn 2023

Course Code: BSMALEA1KU

Authors:
Caroline Sofie Skovby, *cssk@itu.dk*
Muna Hasan Mohamed, *mups@itu.dk*
Petr Boska Nylander, *pnyl@itu.dk*

1 Introduction

This project focuses on exploratory analysis and classification of the Fashion MNIST data set, which consists of labelled images of clothes. We are tasked to perform a dimensionality reduction (DR) of the data set using Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA). This reduced data set is used for classification with the Naïve Bayes (NB) classifier and Support Vector Machines SVM). Lastly, the original data set is classified using Convolutional Neural Networks (CNNs).

2 Exploratory Analysis

The data set is a npy file with 10.000 images in the train set and 5.000 images in the test set. Each image is stored as an array with 785 features. The first 784 features correspond to the pixels that make up the image, each being an integer between 0 and 255 that represents the colour of each pixel, with 0 denoting white and 255 black. The last feature is the label of the image. It is an integer corresponding to one of the five different types of clothing in the data set. The classes do not have an entirely equal distribution of samples in the training dataset, see Table 1. This distribution could have an influence on the performance of the classifiers.

Table 1: *Categories of clothes*

Types of clothing	T-shirt/Top	Trouser	Pullover	Dress	Shirt
Label	0	1	2	3	4
Train size	2033	1947	2011	2005	2014

Because of the many features it is difficult to get an understanding of the information in the data, since plotting each possible pair of features against each other would be unmanageable, time consuming and unlikely to capture any interesting connection across multiple features. Therefore, DR can improve the quality of the data set, by extracting the most relevant features and removing the ones that are uninteresting [21]. This makes the use of DR an essential step for this exploratory analysis and classification.

2.1 Principal Component Analysis

PCA is practical for exploratory data analysis because it can visualise data with many features in a two-dimensional scatterplot that maintains a lot of the information in the data. It does this by finding all the linear combinations of the p features that have the greatest variance among the samples. These are the Principal Components (PC), a new set of p features [7]. Instead of making scatterplots for all of these new features it is enough to only look at ones made with the first few PCs, as the PCs are ordered by how much variance they have which is what we are interested in.

PCA is an unsupervised learning method, so any groups and patterns that are formed in the scatterplot are based on the similarity of the features alone. Adding the labels as colours makes it easy to determine if there is any similarity among samples of the same type [7].

2.1.1 Method and Results

The implementation of the PCA is based on the TA session [4] and the lecture presentation [7]. We first standardize the dataset by *StandardTransform* from the *scikit-learn* library to have a $\mu = 0$, $\sigma = 1$, and to prevent domination of features with a large variance, which could lead to incorrect results [20, 9]. The data set is of the form,

$$\mathcal{X} = \{\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_N\}, \tag{1}$$

where $\mathcal{X}_i$ is the i^{th} sample, and N is total number of samples. Each sample $\mathcal{X}_i$ consists of p features, not including the label. Our objective is to find the direction of maximal variance of this matrix. We achieve that by finding the vector a that maximizes the variance of the projection,

$$a_1^T \mathcal{X} = a_{11}\mathcal{X}_1 + a_{12}\mathcal{X}_2 + \dots a_{1p}\mathcal{X}_p, \tag{2}$$

where a_1 is a unit vector. We now want to maximize $\text{Var}(a_1^T \mathcal{X})$, which according to the theorem for variance of linear transformation results in,

$$a_1^T S a_1, \tag{3}$$

Where, S is the covariance matrix.

By setting the constraint $a_1^T a_1 = 1$ and using the Lagrangian multipliers, we arrive at

$$S a_1 = \lambda_1 a_1 \tag{4}$$

The solution is the eigenvector and eigenvalue of matrix S. We calculate the eigenvalues and eigenvectors of S, by use of *np.linalg.eig()*, and then we sort the obtained eigenvectors and eigenvalues in a descending order. The PCs are obtained by projecting the eigenvectors a onto the matrix containing the data set $\mathcal{X}$, resulting in a matrix, where the first column is the first PC, the second column is the second PC and so on.

While the first and second PCs are considered the best, due to having the greatest variances, it does not guarantee that they will combine into the best visualization of the data. A good visualization will have samples of the same class close to each other, while having as little overlap between different classes as possible [7]. We therefore investigated scatterplots with the first five PCs to determine which combination of PCs is the most interesting one. We made a dataframe with those PCs, added the labels and then plotted scatterplots for each combination, see Figure 1. The scatterplots made with a combination of the

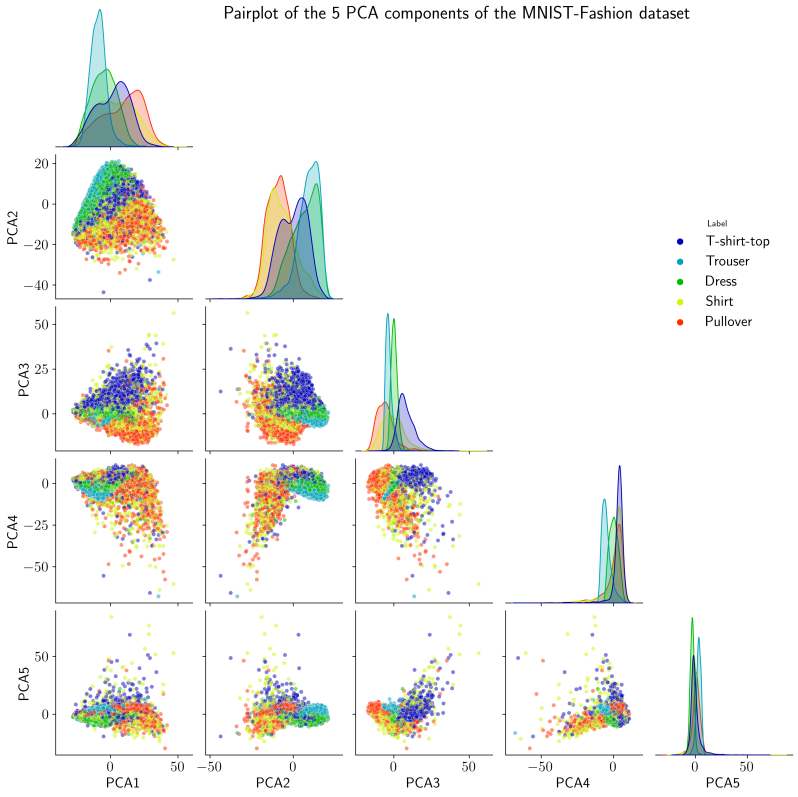


Figure 1: *Pair plot of the first five principal components.*

first three PCs have the most spread, and the rest are mostly clustered together with different classes

overlapping. The left-to-right diagonal shows that the fourth and fifth PCs have small variances for all of their classes, compared to the third and especially second and first PCs. It is clear that the first two PCs make the best scatterplot, by spreading the data and separating the classes more than the others. Given that the fourth and fifth PCs do not capture much variance, it can be assumed that neither will the remaining 779 PCs, so investigating the first two is enough to gain some insight into our data.

The scatterplot with the first two PCs, is shown in Figure 2. PCA has found a view that groups samples of the same class together, while maintaining separation between different classes, all without knowing the class of the samples. This is one of the benefits of PCA, as this confirms the presence of some feature similarity within the classes, which makes it possible to train a classifier for this data.

We see that Trousers, Dresses, T-shirt-tops and Pullovers are somewhat grouped separately, while Shirts are much more scattered around in the areas occupied by T-shirt-tops and Pullovers. This could be because shirts come in a broad range of styles that shares similarities with T-shirts/tops and pullovers.

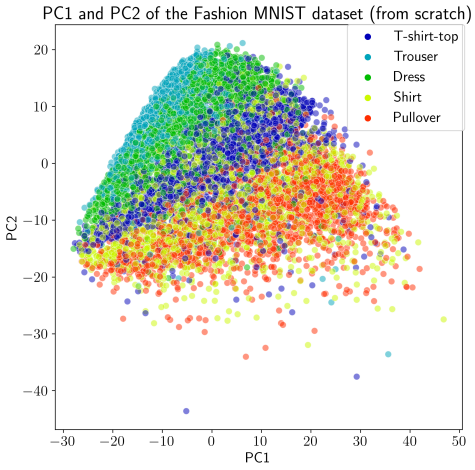


Figure 2: *The high dimensional data is projected down into two dimensions and forms a scatterplot with the first PC, PC1, and second PC, PC2, on the X and Y axis. The colour of the points represent the label of the image.*

It appears that the first two PCs can be used in some cases to determine what type of clothing is depicted on an image. It is therefore interesting to find out what features they represent. This can be learned by inspecting visualizations of the first two eigenvectors, see Figure 3.

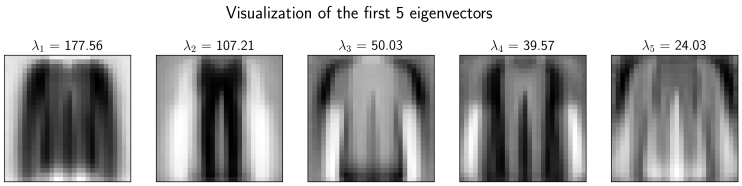


Figure 3: *Visualization of the first 5 eigenvectors with their corresponding eigenvalue. The eigenvectors were reshaped to be 28x28 arrays and plotted with matplotlib.pyplot.imshow() with the colour set to binary.*

An eigenvector shows the feature its corresponding PC measures. Linearly combining these eigenvectors with a sample's PC1 and PC2 values results in a close approximation of the original image. The colours in the eigenvector images shows where the eigenvectors subtract and add value to a pixel. Black pixels add a high value, white pixels subtract a high value, while grey pixels are in between and do not change

as much. The first eigenvector visualization has a pair of trousers and long sleeves in grey, which means that in a linear combination PC1 neither subtracts nor adds considerable value to the pixels in those areas. The black pixels are padding around the sleeves and trousers, which adds value to those pixels. The pixels in white do not contain a clothing item and subtracts pixel value there. The second eigenvector visualization will add a pair of trousers and subtract sleeves. Combining the first two images would result in a black shirt with grey sleeves.

In Figure 4, nine samples with various combinations of high and low PC1 and PC2 values were highlighted in the scatterplot, see Subfigure 4a, and presented as their original image, see Subfigure 4b. The top left observation has a high PC2 value which means the second eigenvector is highly present, which adds a pair of trousers and removes sleeves, and a low PC1 value which means that there is little presence of the first eigenvector, so it does not add sleeves and padding between the trouser legs. This should combine into an image of a pair of trousers and we can see in Subfigure 4b That this agrees with the sample's original image. The top middle observation has a slightly higher PC1 value than the trousers did, which adds some padding between the trouser legs and some short sleeves, but the high PC2 value removes those sleeves because of the white areas. That should add up to an image of a dress with no sleeves, as shown in Figure 4b.

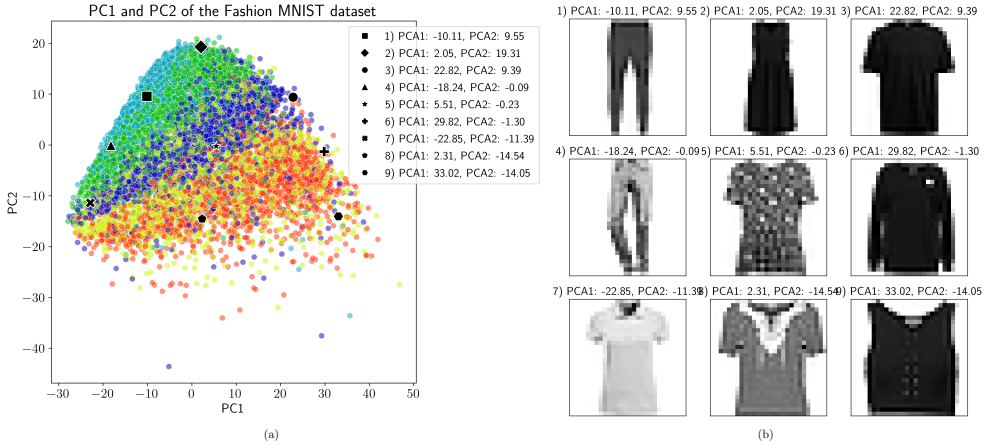


Figure 4: *Scatterplot of the first two PCs with 9 highlighted observations, left. Corresponding images of the observations, right*

The classes are challenging to distinguish when they are in the area of the bottom left observation. Looking at the images in Figure 4b, we see that the colour of the pixels that comprise the shirt are similar to the background colour. Considering the eigenvector visualizations, it makes sense that neither will have a significant presence when almost all of the pixels in an image have values close to 0, because they consist of large white areas. Moreover, as the value for either PC increases, so does the contrast between the colour of the clothing item and the background. The reason why PC1 and PC2 are bad at separating some images is because they are too similar to the background for them to be properly distinguished.

Even though this scatterplot uses the two best principal components, PCA does not separate the classes as well as we would like for doing classification.

2.2 Linear Discriminant Analysis

An alternative to PCA is LDA, which is a supervised machine learning method that uses the class labels to find the greatest discrimination between classes [20]. It achieves this by finding the ideal linear transformation that maps the set of features to a lower dimensional subspace and maximizes the ratio of separation between the classes [12, 19]. This process is illustrated in Figure 5. To obtain the maximum ratio of separation for the data set, it is necessary to calculate the Between-Class-Matrix (S_B) and Within-Class-Matrix (S_W). The S_B describes the separation of the classes, which is the distance between a class

mean and the mean of all observations. The S_W describes as the distance between the class mean and the observations within the class [19]. The final step is projecting the features to the lower dimensional space which achieves this maximum separability [19].

Figure 5: *The Figure illustrates the basic principle behind LDA, which tries to maximize the between-class variance and minimize the within-class variance. Figure adapted from [1].*

2.2.1 Method and Results

The following description and implementation are based on the paper from [19, 2] and [6].

We start with a data set as with PCA, which is also standardized, see equation (1), but including the labels. Then we divide the data set into c classes, where $k = 1, \ldots, c$ using the labels,

$$X = [\omega_1, \omega_2, \ldots, \omega_c]\,. \tag{5}$$

The ω_k represents the subset of all samples belonging to the k^{th} class.

We then calculate the mean of each class μ_k and the mean of the entire dataset μ respectively,

$$\mu_k = \frac{1}{n_k} \sum_{x_i \in \omega_k}, \tag{6}$$

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i. \tag{7}$$

Then with (6) and (7) we calculate the S_B,

$$S_B = \sum_{k=1}^{c} n_k (\mu_k - \mu)(\mu_k - \mu)^T. \tag{8}$$

Where n_k is the number of samples in the k^{th} class.

We then calculated the S_W as,

$$S_W = \sum_{k=1}^{c} \sum_{i=1}^{n_k} (x_{ik} - \mu_k)(x_{ik} - \mu_k)^T. \tag{9}$$

The x_{ik} corresponds to the i^{th} sample of the k^{th} class.

To obtain the ratio of the maximum separability, we find the matrix W, that maximizes the following equation [6, 19, 2],

$$\arg\max_W = \frac{W^T S_B W}{W^T S_W W}. \tag{10}$$

The solution to equation (10) is [2, 19],

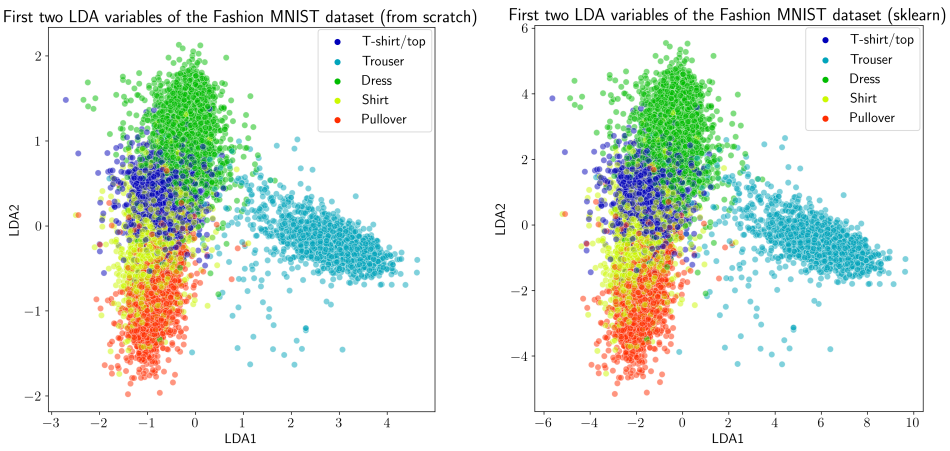
$$W = S_W^{-1} S_B. \tag{11}$$

We now calculate the eigenvalues and eigenvectors of W with *np.linalg.eig()*, sort them in a descending order based on the eigenvalues, and then project the data matrix $\mathcal{X}$ onto the eigenvectors V to obtain the projection of the data set onto the lower dimensional space,

$$Z = \mathcal{X}V. \tag{12}$$

The visualization of the LDA can be seen in Figure 6, where the Subfigure 6a contains the LDA implemented from scratch and the Subfigure 6b contains the LDA implemented by the standard library *scikit-learn*. Both implementations are identical, except the scales of the *scikit-learn* implementation are scaled approximately by a factor of two on both the $LDA1$ and $LDA2$ axes.

This visualization succeeds at showing a view that keeps samples of the same class close to each other while keeping the overlap of different classes relatively small. While it does not perfectly separate the classes, it does so far better than the PCA, which is why we prefer to use the reduced LDA version of the data set to train classification models. Shirts and T-shirt/tops are still overlapping but not as much as in the PCA. While PCA had an area where all classes where overlapping, LDA only has two classes overlapping at a time, making it better for classification. The PCA had Trousers placed close to Dresses while the LDA has Trousers almost completely separate from the other classes and is now closest to T-shirt/tops. This, and the overall difference of the patterns in the two scatterplots, would mean that LDA1 and LDA2 measure completely different features from what we saw with PC1 and PC2.



(a) *LDA implemented from scratch* (b) *LDA implemented with scikit-learn*

Figure 6: *Figure depicts both implementations of the LDA with our implementation and with the standard library scikit-learn*

3 Classification

There are many different machine learning models that can build a classifier, and their performance depends on the data's size, distribution and dimension. Trying out different models and comparing the results is useful to verify the performance of the models and to gain further understanding of the data.

3.1 Naïve Bayes

NB is a generative classification model that tries to approximate Bayes' classifier. It uses the prior probabilities π_k and density functions $f_k(x)$ of each class with the highest posterior probability and is of the form [9],

$$Pr(Y = k|X = x) = \frac{\pi_k f_{k1}(x_1) \cdot f_{k2}(x_2) \cdots f_{kp}(x_p)}{\sum_{l=1}^{K} \pi_l f_{l1}(x_1) \cdot f_{l2}(x_2) \cdots f_{lp}(x_p)}. \tag{13}$$

The density functions are unknown and needs to be estimated from the data. NB does that by assuming independence of predictors within each class [9],

$$f_k(x) = f_{k1}(x_1) \cdot f_{k2}(x_2) \cdots f_{kp}(x_p). \tag{14}$$

To estimate a density function, NB uses a non-parametric estimate of f_{kj},

$$\hat{f}(x) = \frac{\text{No. of observations in bin containing } \quad x}{nh}. \tag{15}$$

Where n is the number of observations in a class and h is the bin width of the histogram f_{kj} is estimating. This results in a histogram for every combination of classes and features. Applying the Naive Bayes assumption to Bayes' theorem gives an equation that will give the posterior probability estimate of an observation being in the k'th class [9].

3.1.1 Method and Results

To implement the NB, we made the following functions. First, Algorithm 1 calculates the likelihood of an observation belonging to class k. Where each class contains two variables (LDA1 and LDA2), a histogram is calculated for each variable. The number of bins is a hyperparameter and is computed by the Freedman-Diaconus rule (FDR) [22]. We decided to compute the number of bins with this method, instead of doing a cross-validation of a different number of bins for each class, since the distribution of each variable is slightly different, and searching for the optimal number of bins many times with cross-validation is computationally heavy. After the histograms are constructed, the minimum and maximum value of the histogram is found. The function returns zero if a new sample falls outside of the interval of the histogram. If a sample falls within that interval, the function returns the likelihood of the sample belonging to class k.

Algorithm 1:

function CalculateClassLikelihood (ClassDataframe, Label_1, Label_2, x_1, x_2);
Input : Dataframe, consisting of a observations within a class k, String Label1, String Label2, observation1, observation2
Output: Likelihood of a observation given a class
for $i \in \{1, 2\}$ **do**
  Calculate IQR_i for Label_i
  Calculate BinWidth_i

$$\text{BinWidth}_i = 2\frac{\text{IQR}_i(\text{ClassDataframe})}{\sqrt[3]{n}} \tag{16}$$

  Where n is length of the ClassDataframe
  Calculate the number of bins, b_i for Label_i
  Divide the ClassDataframe into the segment_i of discrete intervals, where the number of bins is b_i
  The segment_i *consists of the discrete intervals, where each interval has k elements, i.e number of observations containing* x_i
  Find $\min_i$ and $\max_i$ value of the first and last bin respectively in the segment_i
if $x_i > \max_i$ or $x_i < \min_i$ **then**
  return 0;
else
  return

$$\pi_k \prod_{i=1}^{2} \left(\frac{\text{segment}_i(x_i)}{n \cdot \text{width}_i} \right) \tag{17}$$

  Where, π_k *is the prior of the class*;

Algorithms 2 and 3 are then used to calculate the posterior probability and classify the observation to the class with highest probability, respectively. When calculating class likelihoods, the implementation Algorithm 3 lead to division by zero, when the *Sum of all likelihoods* were zero. To prevent this, we have introduced an *epsilon* variable, which is a smallest possible float number that prevents this. In the Algortihm 3, if the posterior probability is zero, the classifier classifies to a class with a highest prior, 'T-shirt/top'. The example of the use of all three algorithms can be seen in the Listing 1.

Algorithm 2:

Classify (ClassProbability);
Input : Recieves probability of an item belonging to a particular class
Output: Returns the label of class, with a highest index
Finds the max index in ClassProbability
LabelsOfClasses = ['T-shirt/top','Trouser', 'Pullover', 'Dress', 'Shirt']
if IndexMax = 0 **then**
    return 'T-shirt/top';
else
    return the class at IndexMax in LabelsOfClasses;

Algorithm 3:

ClassProbabilities ($\text{Label}_1, \text{Label}_2, x_1, x_2$, ListOfClasses);
Input : String Label1, String Label2, observation1, observation, list of classes, which are in a form of a Dataframe
Output: Returns the posterior probability
Initialize an empty list called likelihoods
for $i \in \{1, 2, 3, 4, 5\}$ **do**
    likelihood = **CalculateClassLikelihoods**(i, $\text{Label}_1, \text{Label}_2, x_1, x_2$)
    likelihoods append linelihood
Sum of all likelihoods = $\sum_i^n \text{likelihoods}_i$
Initialize an empty list called Probabilities
for $j \in$ Likelihoods **do**
    if Sum of All Likelihoods == 0 **then**
        epsilon = smallest possible float
        probability = $\frac{\text{j}}{\text{epsilon}}$
        Append probability to the list Probabilities
    else
        probability = $\frac{\text{j}}{\text{Sum of All likelihoods}}$
        Append probability to the list Probabilities
return **Classifty(Probabilities)**

The Listing 1 takes in the variables x_1 and x_2 on lines 12,13 as an input variables and returns the list of predicted labels.

```
1  # Separate the dataset into classes
2  bayes_class0 = Bayes_LDA_df[Bayes_LDA_df['Label'] == 'T-shirt/top']
3  bayes_class1 = Bayes_LDA_df[Bayes_LDA_df['Label'] == 'Trouser']
4  bayes_class2 = Bayes_LDA_df[Bayes_LDA_df['Label'] == 'Pullover']
5  bayes_class3 = Bayes_LDA_df[Bayes_LDA_df['Label'] == 'Dress']
6  bayes_class4 = Bayes_LDA_df[Bayes_LDA_df['Label'] == 'Shirt']
7
8  # add the classes into a list
9  list_of_classes = [bayes_class0, bayes_class1, bayes_class2, bayes_class3, bayes_class4]
10
11 # create variables containing LDA1 and LDA2
12 x_1 = Bayes_LDA_df['LDA1']
13 x_2 = Bayes_LDA_df['LDA2']
14
15 # create list, where correct c
16 train_prob = []
17 for i in range(len(x_1)):
18         probs = class_prob('LDA1', 'LDA2', x_1[i], x_2[i], list_of_classes)
19         train_prob.append(probs)
```

Listing 1: *Example of prediction of the Naïve Bayes*

The results we obtained from using our Naïve Bayes classifier on the train and test sets are shown in Figure 7 as confusion matrices.

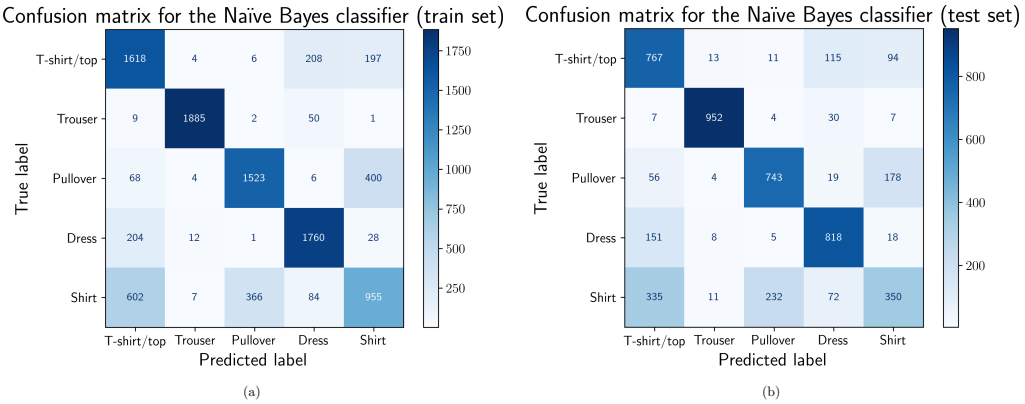


Figure 7: *Confusion matrices for the Naïve Bayes model using the train set (a) and the test set (b). The rows are the number of true labels in a class and the columns are the number of predictions for a class made by the model. True predictions are on the left-to-right diagonal.*

We can see by the colours that the train and test sets perform very similarly which means the model has neither overfitted nor underfitted too much on the train data. The biggest outlier is the Shirt class, where in the test set in Subfigure 7b, the colour of the true positives is lighter than with the train set in the Subfigure 7a. The model does a good job at correctly classifying most of the classes, with all the highest numbers being on the diagonals. There are some classes it is better at classifying than others. There are very few misses and wrong predictions with Trousers, which makes sense as we saw in Figure 6 that trousers are the most distinctive class, so they would rarely be mistaken for another class. The worst performing class is the Shirt class. We can see that for both the train and test set the model classifies a lot of shirts as T-shirt/tops and Pullovers and the other way around. As previously mentioned we saw in Figure 6, that the shirt samples were overlapping a lot with T-shirt/tops and Pullovers, so this performance is limited, since the model cannot be expected to perfectly distinguish between different classes that have the same features.

Table 2: *Naive Bayes Model Performance on Different Classes.*

Class	Precision	Recall	F1-Measure	Class Size
T-shirt/top	0.58	0.73	0.64	1000
Trouser	0.96	0.95	0.95	1000
Pullover	0.75	0.71	0.73	1000
Dress	0.78	0.81	0.80	1000
Shirt	0.52	0.39	0.45	1000

The large number of Shirts that get misclassified as a T-shirt/top could be influenced by the priors. The distribution of class sizes in the training data differs from the one in the test data, see Table 2. The train data class size determined the priors in our model, making it more likely to classify into the big classes. There were more train samples for T-shirt/top than Shirt samples, and in the test set there was an equal distribution among all classes. This, combined with the overlap of features for the two classes, makes the model more likely to classify as T-shirt/top, resulting in them having a low precision at 0.58 and Shirts having a low recall at 0.39. Shirt was the second biggest class, causing a similar effect to the Pullover class. The priors were similar though, and this influence should be small.

3.2 Support Vector Machines

SVM is a supervised learning method that separates a labeled data set by the hyperplane with the broadest possible margin between points of each class. Maximizing this margin achieves the best class separability. SVM is of the form [9, 23, 24],

$$y(x) = \sum_{i}^{n} \alpha_i y_i (x_i \cdot x) + b. \tag{18}$$

For classes that are linearly separable, SVM allows no samples to lie on the wrong side of the margin by the constraint $\alpha_i \geq 0$, this is also called *Hard Margin* [23, 24]. For classes that are not linearly separable, the samples may lie on or within the margin of the opposite class. The extent to which observations are allowed to be on the opposite side is controlled by the hyperparameter C, with the constraint $0 \leq \alpha_i \leq C$. This parameter compromises between penalizing the sample being on the wrong side of the margin and maximizing the margin width [23, 24]. This is called *Soft Margin*. Higher values for C means less tolerance and lower values for C means bigger tolerance. All samples on or within the margin have $\alpha > 0$ and are the models support vectors [23, 24].

3.2.1 Implementation and Results of SVM

The implementation of SVM has been based on [16].

For implementing the SVM model, we used the version from *scikit-learn*. The method *One-vs-Rest* (OVR) was used since SVM was initially made for binary classification. The OVR method creates c binary classifiers, where c is the number of classes. Each classifier is trained to discriminate one specific class as the target class and grouping together the remaining classes [10].

Scikit-learn uses various hyperparameters in its SVM implementation: kernels, and parameters C and γ. Kernels are used for the decision boundary. When separation is linear, then the linear kernel is optimal. However, whenever the decision boundaries are not linearly separable, the kernels, such as polynomial or radial basis function (RBF), are used to enlarge the feature space, mapping the data set into higher dimensions [9, 14]. The γ decides importance of individual observations, low values of γ cause the decision bounadies to be close to linear, converlsly higher γs lead to more complex boundies [3, 14].

To find the optimal hyperparameters, we used the *scikit-learn* function *GridSearch* on the parameters that have been randomly selected, see Table 3a, which performs an exhaustive search for optimal parameter search on 75 model candidates. For each, it performs five cross-validation folds, resulting in 375 fits [15, 13]. The most optimal set of hyperparameters in Table 3a are, RBF kernel, $C = 10$ and $\gamma = 0.01$.

We can see on Figure 8 that our SVM model performs similarly to our Naïve Bayes model, which is quite well. Shirts are again the most difficult class to classify, affecting the performance for T-shirt/tops and Pullovers as well.

Table 3: *SVM Model Parameters and Results*

(a) *Parameters for the grid search for the SVM.*

Kernel	C	γ
Linear	0.001	0.0001
RFB	0.01	0.001
γ	0.1	0.01
-	1	0.1
-	10	1
RBF	**10**	**0.01**

Note: These are the most optimal parameters.

(b) *SVM Model Performance on Different Classes.*

Class	Precision	Recall	F1-Measure	Class Size
T-shirt/top	0.59	0.74	0.65	1000
Trouser	0.97	0.95	0.96	1000
Pullover	0.71	0.76	0.73	1000
Dress	0.80	0.79	0.79	1000
Shirt	0.54	0.39	0.46	1000

Looking at Table 3b and comparing it to the performance of our Naïve Bayes model in Table 2, we can

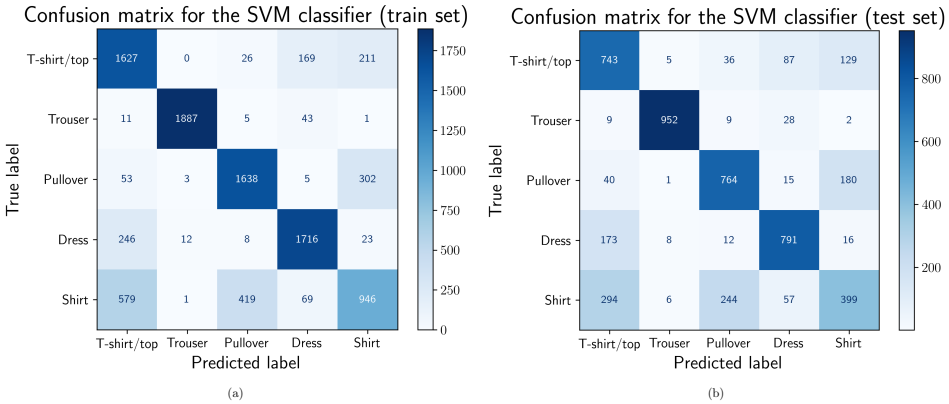


Figure 8: *Confusion matrices for the Support Vector Machines model using the train set (a) and the test set (b). The rows are the number of true labels in a class and the columns are the number of predictions for a class made by the model. True predictions are on the left-to-right diagonal.*

see that the two have near identical F1-measures for each class. From this we assume that the wrong predictions are caused by some of the same reasons.

Figure 9 shows the decision boundary for the SVM model. The decision boundary is nonlinear. It separates Trouser rather well, with few miss classifications. While Pullover, T-shirt/Top and Dresses are fairly well separated with the boundary, the Shirt overlaps to all the three regions. The overlap is most pronounced in the T-shirt/top and Pullover and less in the Dress region. The decision boundaries are not overfitting to the overlap of the classes which is generally better.

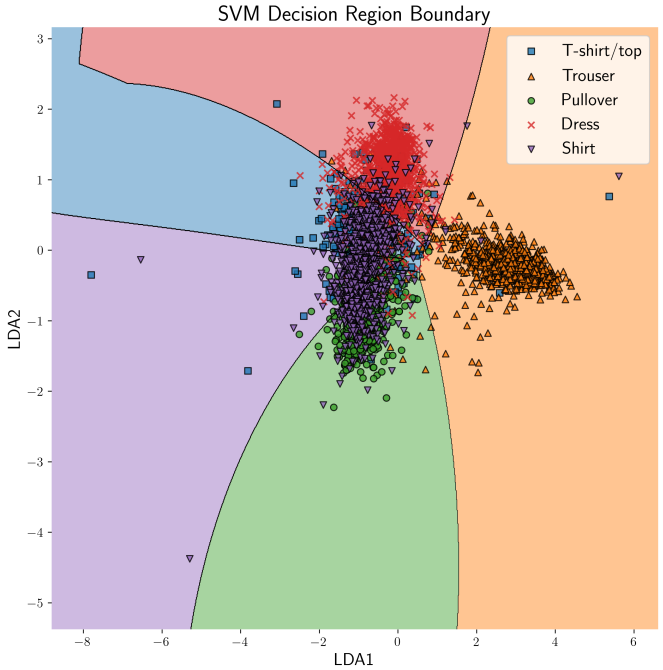


Figure 9: *Decision boundary for SVM with* ***kernel: RBF***, $\boldsymbol{C = 10}$ *and* $\gamma = \boldsymbol{0.01}$

3.3 Convolutional Neural Networks

Convolutional Neural Networks are deep learning models well-suited for grid-structured data and are known to perform well in image recognition tasks. We used Figure 10 as a guideline from our lecture

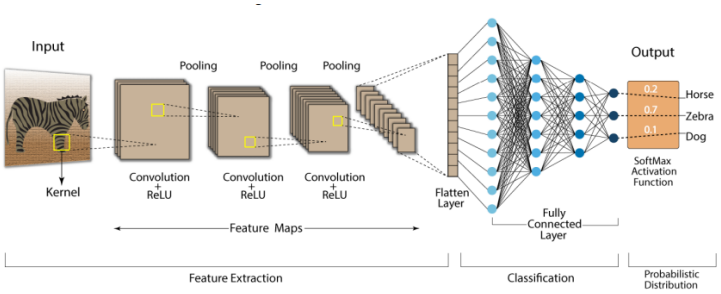


Figure 10: *Overview of Convolutional Neural Networks (CNNs) architecture, image from [5]*

notes, which illustrates how a CNN classifies an image of a zebra. The structure of a CNN can roughly be divided into two segments: an upstream feature extractor and a downstream classifier. These segments are further divided into different layers [9, 11].

The feature extractor starts with an input layer that receives the input data and processes it through a convolution filter (kernel). The output of a filter forms a feature map. A filter contains weights that are trained and updated as data is processed through the different layers. In the first layer, the kernel tries to detect low-level features from the image. During the convolution process, the filter is applied to a portion of the original image, with each weight in the filter being multiplied by the corresponding element of the image. The outcome is a single value that sums up that portion of the image. This process is repeated by sliding the filter across the image until a convolved layer is obtained [9, 11].

In a CNN, each image is structured with spatial dimensions set by the input image size and a number of channels, which relate to the depth. For the first input layer, the number of channels corresponds to the color channels in the image. In deeper layers, the depth is defined by the number of filters used in each convolutional layer [11, 9].

The convolutional layers are made of a large number of convolution filters. These filters detect different signature features in the image, such as shapes, or textures. An activation function is applied to extract more complex patterns. Pooling layers are added between convolutional layers to reduce the image into a smaller summary image, which downsizes the input for the following layers. This reduction lowers the amount of computing work done during training. After this process has been repeated some given number of times a flatten operation transforms the convolutional layer(s) into a 1-dimensional vector [11, 9].

The classifier segment uses fully connected (dense) layers to classify the features into class probabilities. If there are more than two classification tasks an activation function is typically applied and outputs a probabilistic distribution over the classes [11].

Overfitting is an inevitable issue with deep learning models like CNN, especially when the training data set is not large or diverse enough. One way to reduce overfitting is by adjusting the learning rate. When it is too high it will miss the optimal solution, and when it is too low the model will memorize the training data and potentially overfit to it. Batch normalization is one way to optimize the learning rate [8].

Another way to prevent overfitting is adding dropout layers where random neurons are temporarily removed during training, to make the remaining neurons update their weights which makes the network to be better generalized [9].

L2 Regularization is another approach to prevent overfitting, by penalizing larger weights within the network, and encouraging them to be smaller and more regularized weight values [18].

3.3.1 Method and Results

In this project, *APIs*, *Keras*, and *TensorFlow* integration were primarily used as tools for accessing and processing data. The Sequential model API was employed to construct the CNN architecture [17]. The architecture of the Base Model is presented in Listing 2.

Please note that the figures from the Jupyter notebook might look slightly different, as we did not use a fixed seed.

```
# Input_shape=(28, 28, 1), 28,28 refers to the height and width of the images in pixels. 1
    indicates, that each image has only one colour channel, and is greyscale

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D((2, 2), strides=None, padding='valid'))

# Second convolutional block. As we move into more blocks,
#layers to learn more complex features
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D((2, 2), strides=None, padding='valid'))

# Third convolutional block
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.MaxPooling2D((2, 2), strides=None, padding='valid'))

# Flatten and dense block.
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dropout(0.5))  # Increased dropout rate for dense layer
model.add(layers.Dense(5, activation='softmax'))  # We have 5 classes
```

Listing 2: *The architecture of the Base Model*

We trained the model following the guidelines provided on the *TensorFlow* website and selected functions as recommended by the site [18].

The *model.compile()* function was employed to set up the training configuration. It takes the loss function, the optimizer algorithm, and the metrics that the model will evaluate during in training and testing phases.

Since there are 5 labels in the data, the *Categorical Cross-Entropy* was chosen as the loss function for the model.

The *EarlyStopping* function from *TensorFlow* was used to monitor the validation loss. The training process of the model is stopped, when the validation loss is not improved for specific number of epochs (the patience parameter). The best weights from those epochs are restored.

The *model.fit* function trains the neural network on the training data in batches of 250 samples, and it also evaluates validation data set for its performance.

To analyse the impact of L2 regularization on reducing overfitting, we trained an expanded model featuring a Flatten layer followed by 4 dense layers, each having L2 regularization to prevent large weight values. Additionally, a reduced learning rate was applied. This model was compiled and afterwards trained using the *compile_and_fit function*.

The model's summary of the Base Model shows that the majority of parameters are trainable (60,549 out of 60,869). The non-trainable parameters indicate that the model includes layers with parameters that are not updated during training. Figure 11 compares the loss function of the two models. The L2 model has a smooth decay in both tests, with both lines closely following each other while the Base Model exhibits a gap between the two data sets which means it has problems with generalizing to new data.

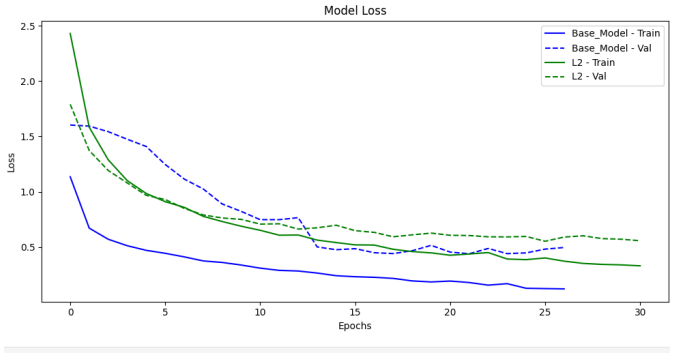


Figure 11: *Model Loss*

The Base Model gets a lower test loss (0.49) than the L2 Model (0.62), see Table 4b, but the L2 Model achieves a higher test accuracy, suggesting better performance in classifying the test data accurately. Looking at Table 4a we see that the precision, recall and F1 scores are all very similar and no model is clearly outperforming the other. We expected the L2 model to perform better then the Base Model but the two models performed almost identically. Since our focus was primarily on reducing overfitting, it might be worthwhile to "relax" our model to potentially improve its performance. For instance, when implementing Batch Normalization, it leads to better results without including Dropout [8].

Table 4: *Summary of Classification Metrics and Model Performance*

(a) *Combined Classification Metrics*

Model	**Class**	**Precision**	**Recall**	**F1 Score**
L2 Model	T-shirt/top	0.83	0.79	0.81
	Trouser	0.98	0.96	0.97
	Pullover	0.85	0.86	0.85
	Dress	0.92	0.89	0.90
	Shirt	0.69	0.75	0.72
Base Model	T-shirt/top	0.84	0.78	0.81
	Trouser	0.95	0.97	0.96
	Pullover	0.83	0.78	0.80
	Dress	0.83	0.92	0.87
	Shirt	0.71	0.72	0.71

(b) *Loss & Accuracy*

Model	**Test Loss**	**Test Accuracy**
L2 Model	0.61	0.85
Base Model	0.49	0.83

4 Conclusion

The CNN clearly outperforms the other two models. The Shirt class caused trouble for the Naïve Bayes and SVM which the CNN avoided. This was expected as CNN uses the spatial information of the image while the other models only see the images as a string of numbers. We were surprised by how similarly the Naïve Bayes and SVM performed as SVMs usually do not perform well with large data sets, but a size of 10.000 samples was apparently not too much. We believe that some of the limitations in performance could come from the data. Shirts, T-shirs, Tops and Pullovers all have definitions with a lot of overlap and it would also be difficult for a person to correctly classify those images. On the other hand, SVM and NB provide all the necessary information about the model and why the model has made particular decisions, ie. probabilities and SVM boundaries. Comparing the metrics, CNN is the preferred model for most correct classification; however, if information on how the model classify is required, the less black box models like SVM and NB would be preferred.

Bibliography

[1] Al-Shiha, A. (2018). *Biometric face recognition using multilinear projection and artificial intelligence*. PhD thesis.

[2] Alpaydin, E. (2020). *Introduction to machine learning*. MIT press.

[3] Ben-Hur, A. & Weston, J. (2010). A user's guide to support vector machines. *Data mining techniques for the life sciences*, (pp. 223–239).

[4] Cornish, C. K. (2023). Principal component analysis. Excercise Session 21, IT University of Copenhagen, 17/11/2023.

[5] Developers˙Breach (2023). Convolution neural network - deep learning. https://developersbreach.com/convolution-neural-network-deep-learning/. Accessed: 15/12/23.

[6] Ghodsi, A. (2015). Pca, fisher's discriminant analysis (fda), university of waterloo. https://youtu.be/hGKt0yy9q_E. Accessed 18/12/23.

[7] Graversen, T. (2023). Principal component analysis. Lecture 20, IT University of Copenhagen, 14/11/2023.

[8] Ioffe, S. & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning* (pp. 448–456).: pmlr.

[9] James, G., Witten, D., Hastie, T., Tibshirani, R., et al. (2013). *An introduction to statistical learning*, volume 112. Springer.

[10] Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. MIT press.

[11] OpenCV, L. (no date). Understanding convolutional neural network (cnn): A complete guide. https://learnopencv.com/understanding-convolutional-neural-networks-cnn/. Accessed: 15/12/23.

[12] Park, C. H. & Park, H. (2008). A comparison of generalized linear discriminant analysis algorithms. *Pattern Recognition*, 41(3), 1083–1097. Part Special issue: Feature Generation and Machine Learning for Robust Multimodal Biometrics.

[13] scikit-learn (2023). sklearn.model˙selection.gridsearchcv. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html#sklearn.model_selection.GridSearchCV. Accessed: 2023-12-10.

[14] scikit-learn (2023a). 1.4. support vector machines. https://scikit-learn.org/stable/modules/svm.html#svm-kernels. Accessed: 2023-12-10.

[15] scikit-learn (2023b). 3.2. tuning the hyper-parameters of an estimator. https://scikit-learn.org/stable/modules/grid_search.html. Accessed: 2023-12-10.

[16] Solutions, V. B. (2020). Svm hyperparameter tuning using gridsearchcv. https://www.vebuso.com/2020/03/svm-hyperparameter-tuning-using-gridsearchcv/. Accessed: 2023-12-10.

[17] TensorFlow (2023a). Convolutional neural network (cnn). https://www.tensorflow.org/tutorials/images/cnn. Accessed: 15/12/23.

[18] TensorFlow (2023b). Overfit and underfit. https://www.tensorflow.org/tutorials/keras/overfit_and_underfit. Accessed: 15/12/23.

[19] Tharwat, A., Gaber, T., Ibrahim, A., & Hassanien, A. E. (2017). Linear discriminant analysis: A detailed tutorial. *Ai communications*, 30(2), 169–190.

[20] Tsourakis, N. (2022). *Machine Learning Techniques for Text*. Birmingham, England: Packt Publishing.

[21] Velliangiri, S., Alagumuthukrishnan, S., et al. (2019). A review of dimensionality reduction techniques for efficient computation. *Procedia Computer Science*, 165, 104–111.

[22] Wikipedia (2023). Freedman–diaconis rule. https://en.wikipedia.org/wiki/Freedman%E2%80%93Diaconis_rule. [Online; accessed 20-December-2023].

[23] Zahadat, P. (2023a). Lecture 14: Support vector machines (svm). Lecture 14, IT University of Copenhagen, 13/10/2023.

[24] Zahadat, P. (2023b). Lecture 15: Support vector machines (svm) - cont. Lecture 15, IT University of Copenhagen, 13/10/2023.