IT UNIVERSITY OF CPH

Clothing Classification

Exploring Different Methods for Determining Clothing Types

January 3rd, 2024

Christian Sneftrup Fleischer
chfl@itu.dk

Sander Engel Thilo
saet@itu.dk

Presented for the Course:
BSMALEA1KU Machine Learning,
3rd Semester of BSc, Data Science
IT University of Copenhagen
Denmark

Github Repository:
https://github.com/ChristianSneffeFleischer/MachineLearningExam

Abstract

This study focuses on developing an application that utilizes image classification methods to determine types for images of clothing items. Throughout our research, we reduced dimensions through Principal Component Analysis (PCA) and self-conducted Linear Discriminant Analysis (LDA), and discussed the method of usage for each of these. Specifically, analysis of how the first few principal components enlighten specific transformations of an image. Furthermore, we applied three different classification methods - Naive Bayes, K-Nearest Neighbors (KNN) and a Convolutional Neural Network (CNN) - to finally evaluate that classifying with CNN gives the most optimal results, as we obtained a final test accuracy of 0.8934. The choice of classifier is highly dependant on the specific requirements of the task, with the CNN offering superior performance at the cost of increased complexity and computational demands.

Contents

1 Introduction

This study investigates a series of clothing items, with the purpose of achieving the ability to classify the type of these items. This will be attempted by developing methods to reduce dimensions and classify through different machine learning methods. The aim is to provide an efficient and reliable tool to determine the type of clothing from a data set based on grayscale images.

The motivation for this project is to apply machine learning methods on a real world data set. The data set is extracted from Zalando.com [Xiao et al., 2017]; an e-commerce platform selling all forms of clothes. In general, the usage of machine learning for image classification can prove highly effective, making the exploration of the methods further enticing.

2 Visualization & Exploratory Data Analysis

2.1 Data Exploration

This project will be conducted from a data set consisting of 15.000 labelled images of clothing items. Each image is a grayscale 28x28 picture of either a t-shirt/top (Label: 0), trousers (Label: 1), a pullover (Label: 2), a dress (Label: 3), or a shirt (Label: 4) as seen with example in Figure 1.

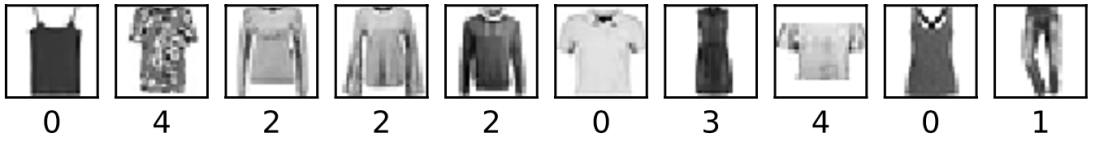


Figure 1: A Random Sample of 10 Images from the Training Data Set.

The data is divided into a training set with 10.000 images and a test set with 5000 images. The data set is balanced, with around 2000 observation of each class in the training data, and exactly 1000 observations of each class in the test data. The raw data is formatted in a single table, where each row is an image and all columns, except the last, represents the pixel values between 0 and 255 for each 28×28 pixel image (784 total pixel values). The 785^{th} and last column is the category of clothing and is represented by the labels (0-4) as aforementioned.

2.2 Dimensionality Reduction

Reduction of dimensions in the data set is necessary due to the large amount of the pixel value features. Generally, dimensionality reduction methods are suitable when working with larger data sets, as they often help to prevent overfitting, and only return features that are most predominant for the data set. We choose to implement Linear Discriminant Analysis (LDA) and Principal Component Analysis (PCA) to transform the data and its dimensions. Both of these methods are broadly used for exactly this, and work similarly, but have some key differences. Dimensionality reduction through LDA maximises the separation between classes by projecting the data onto a lower-dimensional subspace. PCA however, maximises the variance in the data by transforming the original data into a new set of uncorrelated variables called principal components. Moreover, it is important to note that LDA is a supervised method, whereas PCA is unsupervised.

2.2.1 Implementation

The two methods are implemented through the scikit-learn **sklearn.decomposition.PCA** function [Pedregosa et al., 2011] and a custom-made LDA class, respectively. LDA aims to perceive the optimal projection of the data that maximizes separation between classes. This separation is found by maximizing the distance between the mean of the classes and minimising the spread within each class.

The custom LDA achieves this through the following steps:

1. **Initialization**
   An LDA object is first initialized, where only the number of desired discriminant variables k is defined as a hyperparameter.

2. **Fitting Model**
   This function takes our training data and fits a discriminant subspace to it. This is done through firstly calculating the mean of all samples μ. Hereafter, we compute the within- and between-class scatter matrices S_W and S_B. The within-class scatter matrix S_W is computed by iterating through each class and calculating the difference between each data point in the class and the class mean. For each class, this matrix is then added into the total S_W. The between-class scatter matrix S_B is computed by taking the difference between the class means and the total mean of all data points, and then multiplying this value by the number of observations in the corresponding classes. These class specific values are then put into the total between-class scatter matrix S_B. Next, the inverse of S_W is computed to solve the generalized eigenvalue problem shown in equation 1, where v is the eigenvector, and λ is the eigenvalue.

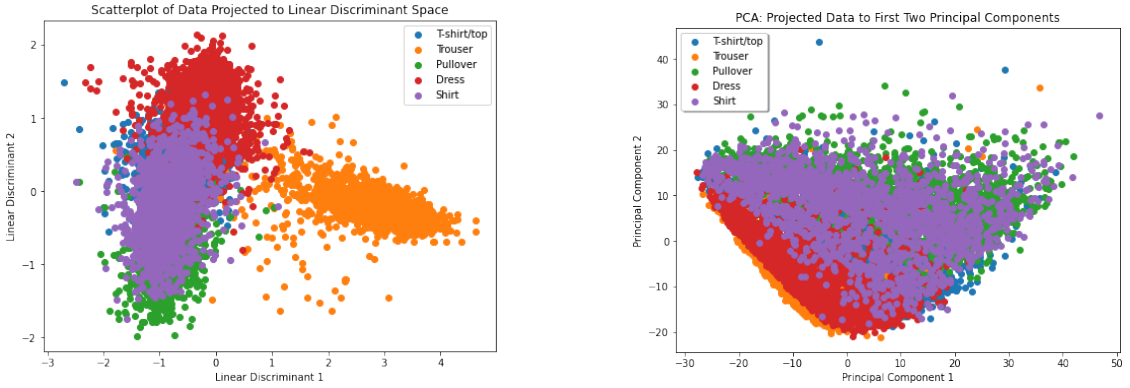
$$\mathrm{S}_W^{-1}\mathrm{S}_B\mathrm{v} = \lambda\mathrm{v} \tag{1}$$

   From this, the top k normalized eigenvectors are found from a sorted array with the eigenvectors in descending order. This whole process is solved using NumPy's **numpy.linalg** library and its specific techniques [Harris et al., 2020].

3. **Transforming Data**
   A transformation function projects the data onto the discriminant subspace, formed by the k normalized eigenvectors, learned in the last step. A projected data set can now be returned with k dimensions.

2.2.2 Visualization

The two methods are visualised in scatterplots in figure 2, where the data set is reduced to two features. In figure 2a the data set is reduced to two linear discriminant variables with the custom-made LDA, and in 2b it is reduced to two principal components with scikit-learn's PCA. Scaled training data is used in both plots, through the **sklearn.preprocessing.StandardScaler** function [Pedregosa et al., 2011]. This, combined with the two generally different approaches, determines why the plots have different parameters on the axes. The different scales of parameters, however, don't change the fact that the plots are still vastly comparable.



(a) Data projected to the first two linear discriminant variables in LDA model

(b) Data projected to the first two principal components in PCA model

Figure 2: Comparison of LDA and PCA Dimensionality Reduction Methods

From figure 2 a clear difference can be seen in how the first two linear discriminant variables and the first two principal components have the data projected onto them. The classes generally overlap, however the clothing type trouser is more distinctly spread in 2a than in 2b. This spread makes intuitive sense in terms of trousers distinguishing themselves from the other clothing types visually. Furthermore, this proves the maximization

in class separation that LDA aims to achieve, and it is clear that 2a maximises this better than 2b. The classes are not very well separated in 2b, which also indicates that these first two principal components do not capture important discriminatory information. Moreover, the spread of points in 2b is relatively similar in all the classes, which is another indicator, that more than two principal components are needed.

In this case, LDA can be considered a better dimensionality reduction technique than PCA when reducing to only two features. Even though there are still possible improvements with regards to the LDA technique, the two linear discriminant variables seems to separate the classes better and is generally a more meaningful reduction than PCA with two principal components. A lot of variance might be explained by additional principal components.

2.2.3 Exploring First Few Principal Components

In general, when looking into the first few principal components, it is interesting to consider how these directions correspond to a specific transformation of an image. When doing PCA, the principal components are vectors, hence describing a direction - specifically a direction of greatest variance. Every time a dimension is reduced, new basis vectors (the principal components) are found. The first k principal components therefore span a k-dimensional subspace, which is seen as the "best" k-dimensional view of data. Thus, the data is rotated and projected onto k dimensions. [James et al., 2021].

The first principal component accounts for the largest amount of variance in the data. With regards to image analysis and specific transformation of these, the first principal component will often capture the most dominant overall pattern, or maybe a superior feature present in the image. The second principal component is orthogonal and therefore uncorrelated to the first, and captures the next largest amount of variance. One could expect that it would most often capture expression, lighting or shadows in images. The third principal component is orthogonal to both the first and second PC's, and captures more specific object shapes. Subsequent principal components will capture less global variations and increasingly higher amounts of specific features will be enlightened. [Sartorius, 2021].

For the clothing data set, figure 3 shows the first few principal components and their impact on an example image, where a T-shirt/top (label 0) is plotted. The captured textures and features get more specific for each k principal component. The difference is clearer between PC2 and PC3 than between PC1 and PC2, which indicates that there are particular features in the image that are more well-differentiated by PC3. In this example image, it is clear already after the first three principal components, that the clothing item is a t-shirt/top, however if it was a pullover or a shirt, subsequent principal components might be needed to distinguish the type of the item, with regards to e.g. collar type.

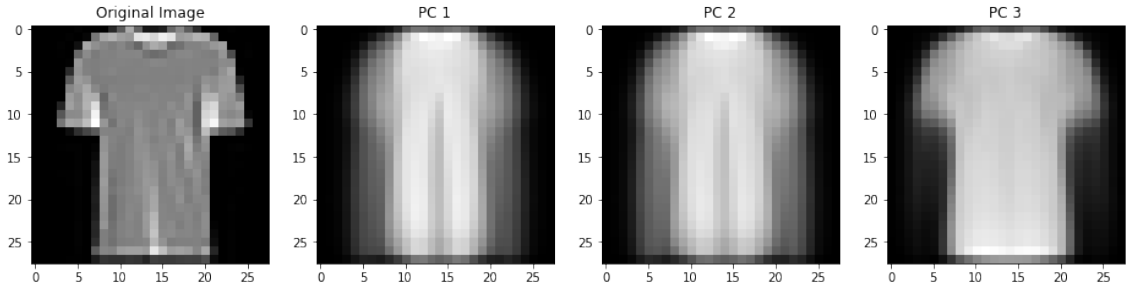


Figure 3: Example of First Three Principal Components and What They Capture in an Example Image

3 Classification

Successionally, we want to identify the optimal method to classify the data set into the five clothing types. Using the explained methods for dimensionality reduction combined with three different types of classification models, we aim to find the best possible method. Visualisations and performance metrics offers insight into exactly this.

When testing the classifiers in this project, we consider test accuracy the most important performance metric. This choice is made on the basis that the training data is balanced and multi-class, making test accuracy a single, reliable metric which provides a strong indication of how well the classifier performs over all classes. Furthermore, the primary objective of the project is to maximize the overall frequency of correct predictions, which accuracy is the direct measure of.

3.1 Naive Bayes Classifier

3.1.1 Classifier Overview

The first method implemented in the attempt to classify the data is a Naive Bayes classifier. Naive Bayes is a classification model based on Bayes' theorem (equation 2), which describes a conditional probability; the probability of one event happening given that another event is true.

$$\mathrm{P}(A|B) = \frac{\mathrm{P}(B|A)\mathrm{P}(A)}{\mathrm{P}(B)} \tag{2}$$

In the context of classification, Naive Bayes takes advantage of Bayes' theorem to assign data points to classes, based on knowledge about the underlying distributions of data for each class. The classifier assumes that when given the class variable, any given feature value is independent of the value of any other feature, giving it the 'Naive' aspect.

Naive Bayes classifiers are often modelled as parametric, making assumptions about the data distribution, e.g. by assuming a Gaussian distribution. Contrary, non-parametric implementations of the classifier make fewer assumptions about the distribution and can be more flexible in capturing the patterns in the data. An example of a non-parametric method is a kernel density estimate, where summing kernels placed on each data point obtains a smoothed estimate of the underlying data distribution.

The Naive Bayes classifier implemented in this study is an example of a non-parametric model which uses a kernel density estimate. For each class, the model estimates the univariate feature distributions following the formula giving in equation 3, where n is the amount of observations in the class and h is the kernel bandwidth.

$$\hat{f}(x) = \frac{\text{No. of observations in } [x-h; x+h]}{2nh} \tag{3}$$

3.1.2 Implementation

Implementation of the Naive Bayes classifier consists of training a non-parametric model focused on leveraging the distribution of training data for predictions. Because of this, when training, the model only needs to store the training data according to class and compute the prior class probabilities $\mathrm{P}(Y = y)$ for each class y.

In the model presented for this project, the first two linear discriminant variables from the LDA dimensionality reduction method are used as features X_1 and X_2. From these, the classifier predicts the class of an observation x by the following steps:

1. **Estimate Univariate Feature Distributions**
   For each class y, the model estimates the univariate feature distributions $f(x_1|y)$ and $f(x_2|y)$. This is done by determining the frequency of observations withing the range $[x-h; x+h]$, and normalizing this frequency with respect to the total number of class observations n and the bandwidth h (see equation 3). The bandwidth h then acts as an adjustable hyperparameter to fine-tune model performance.

2. **Calculate Joint Feature Distributions**
   Under the Naive Bayes assumption of feature independence, the joint distribution of features for each class is computed as the product of their individual univariate feature distributions, i.e., $f(x_1, x_2|y) = f(x_1|y)f(x_2|y)$.

3. **Calculate Posterior Probabilities**
   Using Bayes' theorem (equation 2), the model calculates the posterior probabilities for each class y. This involves multiplying the joint feature distributions by their respective prior probabilities to obtain the unnormalized posterior probabilities. These are then normalized by dividing each by the sum of unnormalized posterior probabilities across all classes for that observation.

4. **Make Prediction**
   The model predicts the class y of observation x by selecting the class with the highest normalized posterior probability.

3.1.3 Results

The Naive Bayes classifier has one hyperparameter: h, the kernel density bandwidth. This parameter decides how broad a range of neighboring observations should be considered, and therefore directly influences the model's behavior, specifically in relation to the bias-variance tradeoff and the risk of under- or overfitting to the training data. Choosing a high bandwidth will lead to smoother, broader kernel functions, which results in higher bias and risks oversimplifying the data, potentially leading to an underfit model. Contrary, a lower bandwidth will produce narrower, more peaked kernel functions. This can lead to lower bias, but higher variance, as the predictions become more sensitive to the specific training data and may cause overfitting.

The choice of bandwidth is decided through testing a range of 50 different h-values from 0.01 to 1, with the objective of maximizing accuracy when predicting test data. The results of this process is that two h-values produce the same test accuracy of 0.7292, namely $h = 0.15$ and $h = 0.35$. Figure 4 shows the test accuracies for all bandwidth values in the range.

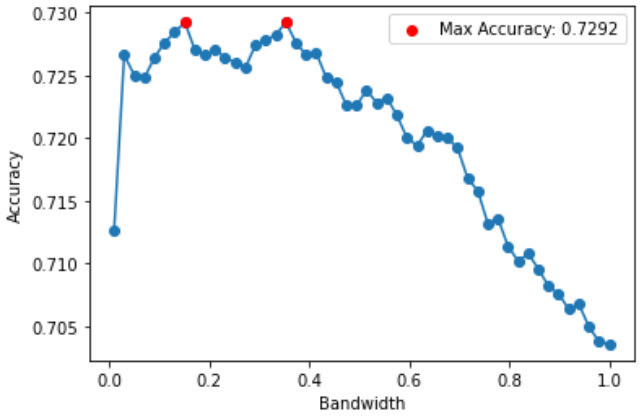


Figure 4: Naive Bayes Accuracy vs Bandwidth h.

As seen in table 4, the remaining performance metrics for the two bandwidth values are also extremely similar.

	$h = 0.15$				$h = 0.35$			
	precision	recall	f1-score	support	precision	recall	f1-score	support
t-shirt/top	0.59	0.74	0.66	1000	0.59	0.77	0.66	1000
trouser	0.98	0.95	0.96	1000	0.98	0.95	0.96	1000
pullover	0.73	0.76	0.74	1000	0.72	0.78	0.75	1000
dress	0.81	0.81	0.81	1000	0.81	0.80	0.80	1000
shirt	0.53	0.38	0.44	1000	0.54	0.35	0.43	1000
macro avg	0.73	0.73	0.72	5000	0.73	0.73	0.72	5000
weighted avg	0.73	0.73	0.72	5000	0.73	0.73	0.72	5000
accuracy	0.7292				0.7292			

Table 1: Naive Bayes Performance Metrics, $h = 0.15$ and $h = 0.35$

Further interpretation of the results show that both models perform particularly well when classifying trousers. This is most likely due to the relatively small overlap with other classes in the linear discriminant feature space (figure 2a), making the differentiation easier for the model. In contrast, the shirt class has the largest overlap with other classes, leading both models to struggle with predicting this class correctly.

Perhaps most notably, the macro and weighted average statistics are exactly the same for both models, indicating that the performance across classes when considered as a whole is balanced. Thus, the models achieve a similar bias-variance tradeoff, making both bandwidth settings viable choices.

The test accuracy of 0.7292 is relatively modest. However, this is to be somewhat expected when considering the specific model used to predict the data. While the Naive Bayes is easy to implement, its simplicity often comes at the costs of performance. Relying purely on feature distributions may not be enough to obtain high frequencies of correct predictions, as the data in some cases contain intricate patterns which are not modelled effectively by the classifier. Furthermore, while the input features are LDA discriminant variables, which maximizes separability between classes and reduces correlation, one cannot assure complete independence of features; the fundamental assumption of any Naive Bayes classifier.

These limitations make is sensible to explore other, more complex, models to improve classification performance.

3.2 K-Nearest Neighbours Classifier (KNN)

3.2.1 Classifier Overview

Apart from the Naive Bayes classifier, we decided upon two other classification methods that might perform better when predicting clothing types from the data. The first of these is the K-nearest neighbors (KNN) model, which, like the Naive Bayes model, predicts classes based on estimates of posterior class probabilities. However, contrary to Naive Bayes, KNN approximates the posterior probabilities by counting the frequency of each class y from the k closest neighbours to the observation in question. Specifically, the estimation follows equation 4, where $\mathcal{N}_0$ represents the k closest points to the observation x, and y_i is the class of a data point i in $\mathcal{N}_0$. [James et al., 2021].

$$\mathrm{P}\left(Y = y \mid X = x\right) = \frac{1}{k} \sum_{i \in \mathcal{N}_0} I\left(y_i = y\right). \tag{4}$$

The class with the highest posterior probability is then chosen as the prediction for observation x.

3.2.2 Implementation

Implementation of the KNN model is done using the **sklearn.neighbors.KNeighborsClassifier** model from scikit-learn, as it is broadly used and implements the KNN classification method efficiently [Pedregosa et al., 2011]. The model is further optimized by exploring the use of different feature selection and extraction techniques for the training data, as well as varying the choice of the hyperparameter k.

Firstly, the KNN classifier is trained on the whole data set with all 784 features, i.e., using no dimensionality reduction. Secondly, we train on the data set reduced with our custom-made LDA. As the maximum amount of derived linear discriminant variables is the number of classes minus one (equalling 4 in our case) our options are to project the data onto the first two, three and four discriminant variables. These three LDA reductions are denoted LDA2, LDA3, and LDA4, respectively. Furthermore, we train the classifier on the data set reduced with PCA. We choose to set the percentage of variation that needs to be explained from our principal components to 95%, as this achieves to simplify the data while still maintaining the most important patterns.

To validate the different techniques, we run cross validation through **sklearn.model_selection.cross_val_score**, where we set the amount of folds to the default five [Pedregosa et al., 2011]. This function provides a list of the accuracy scores for each fold, and from this a mean accuracy score can be calculated. This method is run on the five differently trained data sets, where we cross validate for every neighbor in a range from 1 to 101, while only checking uneven values of k to avoid ties. The upper limit of $k = 101$

is chosen following the rule-of-thump of checking values of k around the square root of the size of the training data (10.000 in this case) [Band, 2020]. With regards to feature scaling, we once again use the **sklearn.preprocessing.StandardScaler** function [Pedregosa et al., 2011]. However, it is now used in a combination with the library **sklearn.pipeline.Pipeline** [Pedregosa et al., 2011]. This combination ensures that the scaling happens for each fold, so that we do not use pre-scaled data to cross validate on. In figure 5, the different techniques are plotted for all the different neighbors and their mean accuracy score from cross validation. Only the best performing LDA is plotted.

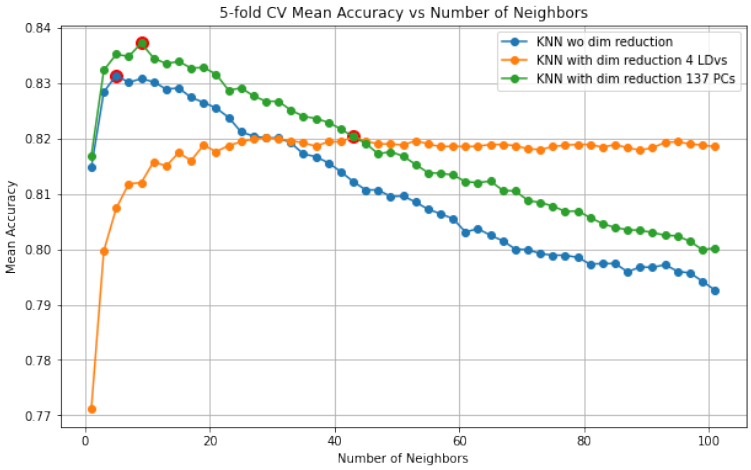


Figure 5: Different Dimensionality Reduction Techniques for KNN and Their Performance

3.2.3 Results

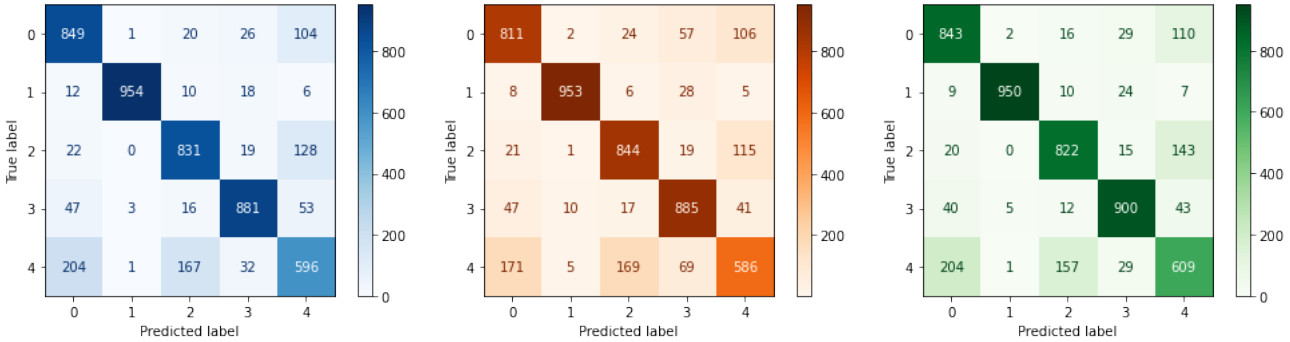
After training the different models on the training data and finding the optimal k for each technique, we use the models to predict on the test data. The results are very similar to the ones gotten with the training data, which indicates a good classification performance. The results are shown in table 2.

Dim. Red. Technique	Num. Features	k	Mean Accuracy
None	784	5	0.8222
LDA2	2	61	0.734
LDA3	3	87	0.7798
LDA4	4	43	0.8158
PCA 95%	137	9	**0.8248**

Table 2: Mean Accuracy Scores for KNN Classifier

The models trained on data with no dimensionality reduction and dimensionality reduction through PCA perform better than all models trained on LDA-projected data. The PCA model does however perform slightly better than the model trained without reduced dimensions, having a test accuracy of 0.8248. This shows that the KNN model predicts the clothing types more accurately than the previously presented Naive Bayes model. However, it should be noted that we only tested the Naive Bayes classifier on the data set projected onto the first two linear discriminant variables (LDA2), and when looking at the result for LDA2 in the KNN model, it performs almost the same as Naive Bayes. This indicates that the difference in performance between the Naive Bayes and KNN classifiers is rather insignificant when trained on the same data, while also showing the effectiveness of choosing the best dimensionality reduction technique.

Lastly, the different techniques are plotted below as confusion matrices in figure 6. With this visualization, we get a clearer view on the specific accuracy scores of the KNN models. Again, only the best performing LDA (LDA 4) is plotted.



(a) No DR Model (b) LDA4 Model (c) PCA Model

Figure 6: Confusion Matrices for 3 best performing models

It is clear from the matrices that the shirt class is predicted least accurately across all three models. The model also often mistakes a T-shirt/top for a pullover, or vice versa, which makes intuitive sense considering the shape of the specific items. Nevertheless, the PCA model was best at predicting these shirts correctly. As it also has the highest mean accuracy score of all the models, it seems that PCA is the best choice as a dimensionality reduction method for our KNN classifier.

Following the choice of using PCA-reduced data and $k = 9$ when training the KNN model, we can further analyse the test results as seen in table 3.

	precision	recall	f1-score	support
t-shirt/top	0.76	0.84	0.80	1000
trouser	0.99	0.95	0.97	1000
pullover	0.81	0.82	0.82	1000
dress	0.90	0.90	0.90	1000
shirt	0.67	0.61	0.64	1000
macro avg	0.83	0.82	0.82	5000
weighted avg	0.83	0.82	0.82	5000
accuracy	0.8248			

Table 3: KNN Performance Metrics, PCA Training Data and $k = 9$

Like the Naive Bayes model, the KNN classifier performs best when predicting the trouser class, while classifications of shirts produce the worst results. Although it is harder to deduce from the principal components in figure 2b, when compared to the linear discriminant variables in figure 2a, it still seems apparent that the difference in performance across these classes stems from the overlap with other classes. The observations belonging to the trouser class also appear more collected in the lower left corner of the plot, making predicting this class easier, while the shirt observations are more spread, leading to worse results. Thus, the underlying patterns in the raw data seem to have a significant effect on the classification results, whether projected onto linear discriminant variables or principal components. To fully capture these data patterns, we will lastly explore Convolutional Neural Networks, a more advanced classification method which uses no dimensionality reduction techniques.

3.3 Convolutional Neural Network (CNN)

3.3.1 Classifier Overview

For the final classification method of this project, we choose to implement a Convolutional Neural Network (CNN). CNN's are a class of deep learning models which attempt to classify images by finding and interpreting patterns in the input data, similarly to how human brains use pattern recognition to identify objects in real life. This type of classifier has shown to be extraordinarily effective in predicting image classes. CNN

classification models work by passing the input data through multiple layers, continuously transforming it until a final output class is produced. These layers usually consist of three different types: convolutional layers, pooling layers, and fully connected layers [James et al., 2021].

- **Convolutional Layers**
  The input images are first passed through what is called convolutional layers. Each convolutional layer comprises a number of learnable filters, which are designed to capture specific features in the image, such as edges or textures. The filters work by applying a convolution to the image matrix, a process which involves performing element-wise multiplication of the elements in the matrix and then summing the obtained products.

- **Pooling Layers**
  Following a convolutional layer, it is common practice to pass the convoluted image through a pooling layer. Pooling layers shrink the image's size, a technique which reduces the computational demands of the network and can help to avoid overfitting. This is done by splitting the image into individual blocks (often of 2×2 pixels) and extracting a single value from each block to obtain a smaller version of the image. Methods of extracting values from the blocks include max pooling, where the maximum value is chosen, and average pooling, where an average value is computed instead.

- **Fully Connected Layers**
  Finally, the data is flattened to one-dimensional arrays and passed to a series of fully connected layers, which decides the class of the image based on the patterns identified by the convolutional and pooling layers. Each fully connected layer contains a number of neurons, all of which are connected to every neuron in the previous layer. Through neuron biases, connection weights, and activation functions, the fully connected layers map the extracted features to a predicted class.

When training the model, the internal parameters of the model are iteratively adjusted over a number of epochs (repetitions), through a process known as backpropagation. Backpropagation uses the gradient descent algorithm to iteratively update the biases, weights, and convolutional filters to minimize the classification error. By exposing the classifier to a large amount of training data, it gradually refines its ability to identify and classify complex image patterns, thereby improving its rate of correct predictions [Kostadinov, 2019].

3.3.2 Implementation

For this project, the CNN model is implemented using the **PyTorch** deep learning framework [Paszke et al., 2019]. The architecture of the model is based off a modified version of the LeNet architecture [Bangar, 2020]. Although the final model retains the simplicity and most of the features found in the original LeNet version, we have performed rigorous trial and error testing of different modifications to build a model which is best fit to classify our data.

Before data is actually used in the classifier for training and prediction, it is passed through a simple preparation step. This step consists of first scaling the pixel values to lay between 0 and 1. This improves the efficiency of the model's training process by ensuring that network weights do not grow too large. Next, 20% of the training data is extracted into validation data, which is used when training the model to improve prediction performance. For the last part of preparation, we transform training, validation, and test data into tensor data sets for efficient processing with **PyTorch** and put them into **DataLoader** objects: a data structure which allows the model to process the data in batches. After being prepared, the data can be passed as input to the classifier, either when training or predicting. The model architecture design is split into three blocks as follows:

1. **First Convolutional Block**
   The first of these blocks is what we choose to call the first convolutional block. It is comprised of a convolutional layer, a batch normalization function, a sigmoid activation function, and finally, a pooling layer. As our image data is in grayscale, the convolutional layer takes one input channel. It then applies the convolution with a kernel size of 5×5 pixels, a stride of one pixel and a padding of two pixels on each side of the image. This returns six output channels, known as feature maps, which are then passed through batch normalization (only when training) and the sigmoid activation function. Batch

normalization further stabilizes the training process to improve performance and reduces the number of epochs required to train the model, while the sigmoid activation introduces non-linearity to the model, enabling the model to learn complex patterns in the data. Lastly, the data is pooled using a pooling layer with a kernel of size 2×2 and a stride of two pixels.

2. **Second Convolutional Block**
   Following the first convolutional block, we drop some of the neurons before passing to the next block when training, with a probability determined by the hyperparameter *dropout rate*. This is done to ensure that the model does not rely too heavily on a certain set of features, helping to prevent overfitting. Otherwise, the second convolutional block is very alike to the first convolutional block, although it differs in the amount of the input and output channels. As it is passed data from the first convolutional block, the convolutional layer takes six channels as input, and it outputs 16 channels. While the convolutional layer in the first block applies two pixels of padding on each side of the image, this layer uses none. The batch normalization, activation and pooling functionality is exactly the same as those used previously.

3. **Fully Connected Block**
   Finally, the image data is passed to the fully connected block. This consists of three fully connected layers, designed to decide the label of each input image based on the features extracted in the convolutional blocks. Before each layer, a dropout is applied again at the same rate as previously. The data is then flattened to a one-dimensional vector. In the second convolutional block, each of the 16 feature maps are reduced to 5×5 pixels by pooling, leading the first flattened layer to have $16 \times 5 \times 5 = 400$ neurons. After the dropout, batch normalization, and activation operations, each neuron in the first fully connected layer is connected to 120 neurons in the second fully connected layer. The whole process is then repeated for this layer, which connects to 84 neurons in the last layer. The last layer predicts the class for each image, and thus returns only five neurons, one for each class. The values of these neurons are obtained by converting the raw output scores, called logits, into floating point values between 0 and 1 using a softmax function. The output values represent the probability of a data point being in each of the five classes, and thus sum to 1. The greatest output probability determines the predicted class for each image.

We train the model using back propagation with a gradient descent algorithm. For each epoch, the back propagation adjusts the internal parameters of the model to improve the accuracy of predictions. To do this, we use the **torch.optim.Adam** optimizer [Paszke et al., 2019]. The optimizer adjusts the learning rate of each weight in the network individually, making the optimization more effective and less sensitive the given hyperparameters. Additionally, we use the **torch.nn.CrossEntropyLoss** criterion loss function, which employs the softmax function for computing class probabilities and calculates the cross-entropy loss between these probabilities and the actual class labels [Paszke et al., 2019].

The CNN classifier has two key hyperparameters. The previously mentioned *dropout rate* decides the rate at which neurons are dropped at each layer. The choice of this hyperparameter is crucial, as a too high rate might lead to the model not learning enough, while a very low value can lead to overfitting. The second hyperparameter is the *learning rate*, which determines the step size at each iteration while moving toward a minimum of the loss function, effectively controlling how much the weights are adjusted during training. A high learning rate can cause the model to converge quickly, but might overshoot the minimum loss, while a low learning rate can cause a long learning process that risks getting stuck at a local minima. To combat the cons of choosing a too high learning rate, we employ a schedular function from **torch.optim.lr_scheduler.ReduceLROnPlateau**, which reduces the learning rate by 90% when the validation loss has not improved over a predetermined amount of epochs [Paszke et al., 2019].

Finally, to optimize training efficiency and prevent overfitting, we have incorporated an epoch evaluation algorithm in the CNN model which is run after each epoch. This monitors the model's performance and stops the training process if it sees no improvement in the validation loss over a predetermined amount of epochs, decided by a hyperparameter *patience*. This hyperparameter also decides when the scheduler should reduce the learning rate, which is done when half of the patiente has passed without improvement. The epoch evaluation algorithm also saves the version of the trained model with the lowest validation loss, ensuring that any deprecating training results are discarded.

3.3.3 Results

Choosing the optimal values for the hyperparameters *learning rate* and *dropout rate* is crucial for the performance of the model. Thus, it is in our best interest to test a range of different values for each to maximize the amount of correct predictions. This is done through rounds of cross validation, where, for each combination of specific learning rate and dropout rate values, the training data is split into five folds using **sklearn.model_selection.KFold** [Pedregosa et al., 2011]. Five new instances of the CNN model are then trained, each using a new fold as validation data, and the average validation statistics are returned. Using this method, we test eight different learning rates ranging from 0.0001 to 0.3, and nine different dropout rates ranging from 0.0001 to 0.5, for a total of 72 different combinations. We find the best combination to be a learning rate of 0.0025 and a dropout rate of 0.2, which produces a loss of 0.3109 and an accuracy of 0.8934 when predicting test data. Thus, we get the performance metric from the best CNN model as seen in table 4.

	precision	recall	f1-score	support
t-shirt/top	0.87	0.87	0.87	1000
trouser	0.99	0.97	0.98	1000
pullover	0.89	0.91	0.90	1000
dress	0.92	0.94	0.93	1000
shirt	0.80	0.77	0.78	1000
macro avg	0.89	0.89	0.89	5000
weighted avg	0.89	0.89	0.89	5000
accuracy	0.8934			

Table 4: CNN Performance Metrics,
Learning Rate = 0.0025, Dropout Rate = 0.2

Most notable is the accuracy, which shows a significant improvement when compared to the Naive Bayes and KNN classifiers with test accuracy scores of 0.7292 and 0.8248, respectively. The superior performance demonstrated by the CNN model can likely be attributed to multiple factors. Contrary to the two other classification models, which require manual feature engineering through PCA or LDA methods, CNN uses its convolutional layers to automatically extracts features from the images. This, combined with its deeper, hierarchical structure, allows the model to better identify and process more complex patterns in the image data. Furthermore, CNN models are more resilient to image variations, such as shifts or distortions, making them highly applicable to classify real-world data like that used for this project.

Although this classifier does not use any manual feature extraction, we once again observe the tendency to classify trousers with almost no mistakes, while having the most issues when classifying shirts. This confirms that there are patterns in the raw data regarding these two types of clothing, most likely referring to its shape, which significantly influences the ability to distinguish them from other classes. However, the model still demonstrates a notable improvement in both precision and recall for the shirt class, again indicating that its ability to handle more complex data patterns and overlapping features is enhanced when compared to the other classifiers.

The macro and weighted averages are closely aligned with the overall test accuracy, showing that the model is highly consistent across all classes. This suggests that it is well-tuned and balanced, while managing the bias-variance tradeoff across the various types of clothing.

4 Discussion & Conclusion

After a thorough walk-through of the three classification models, we gather all the impressions and results, and finally evaluate which model is optimal for the classification problem of predicting clothing types of real-world data.

As test accuracy is considered the most important performance metric, we have in table 5 collected the test accuracy scores for our best performing model for each classifier type.

Classification Model	**Num. Features**	**Test Accuracy**
Naive Bayes	2 (LDA)	0.7292
KNN	137 (PCA)	0.8248
CNN	784	**0.8934**

Table 5: Test Accuracy Scores for the Three Classification models

These results once again show that the CNN classifier generally performs the best. This is to be expected when observing how the models differ in complexity; ranging from simple probability calculations in the Naive Bayes classifier to the multi-layered hierarchical structure of the CNN model, the correlation between algorithm intricacy and performance is apparent. As CNN processes raw data, its computational demands are greatly increased when compared to the other models. Therefore, one should consider whether the improved performance of the neural networks justifies its increased complexity and computational cost when choosing a classification model. As an example, if the implementation and computational costs of a CNN model shows to be a limiting factor, the KNN model can be used as a simpler alternative, as its performance is relatively close to that of the CNN while being significantly more computationally efficient.

The CNN classifier demonstrates a further advantage in the context of generalizability and real-world application, as its capability to process raw image data and learn its patterns enables it to adapt to unpredictable, real data. In scenarios where detailed image analysis is required, such as fashion retail or medical imaging, it would be a highly suitable method. Considering the data set used in this project, one could imagine a clothing retailer implementing a neural network model to ease the task of categorizing items. However, if the complexity of the data is lower, and quick prototyping is prioritized, Naive Bayes and KNN can offer simpler alternatives.

While the CNN model has demonstrated good performance in this study, much can be done to further improve its efficiency and address computational demands. While the architecture of our model has proven effective, it is also relatively simple. This suggests that further modifications, such as increasing the amount of layers or neurons, might lead to even better performance. Improvement might also be gained by exploring different techniques which were not touched upon in this project, such as transfer learning [Donges, 2022] or data augmentation [Shah, 2022]. For Naive Bayes and KNN, future improvements could focus on fine-tuning the feature selection process to achieve higher accuracy. Finally, one could also attempt to combine the strength of the presented classifiers in a hybrid model, aiming to balance performance with practicality and computational efficiency.

The research of this study has shed light upon the possibility of classifying clothing items through algorithmic model-building in image classification. Through usage of three different models that classify the item images, we conclude that classifying the data with CNN models outperforms Naive Bayes- and KNN-classification. While always striving for better performance, we consider the results obtained by the models, especially regarding CNN, highly satisfactory. Through relatively simple implementations, we were able to create accurate predictors of clothing categories. The potential for this technological approach is vast, and experimenting with more complex methods could result in even better performing models.

5 References

[Band, 2020] Band, A. (2020). How to find the optimal value of k in knn. Accessed 23-12-2023.

[Bangar, 2020] Bangar, S. (2020). Lenet-5 architecture explained. Accessed 20-12-2023.

[Donges, 2022] Donges, N. (2022). What is transfer learning? Accessed 02-01-2024.

[Harris et al., 2020] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825):357–362.

[James et al., 2021] James, G., Witten, D., Hastie, T., and Tibshirani, R. (2021). *An Introduction to Statistical Learning*. Springer.

[Kostadinov, 2019] Kostadinov, S. (2019). Understanding backpropagation algorithm. Accessed 24-12-2023.

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

[Pedregosa et al., 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[Sartorius, 2021] Sartorius (2021). What is principal component analysis (pca) and how it is used. Accessed on 25-12-2023.

[Shah, 2022] Shah, D. (2022). The essential guide to data augmentation in deep learning. Accessed 02-01-2024.

[Xiao et al., 2017] Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for bench- marking machine learning algorithms.