Zalando Clothing Classificaton

December 31, 2022

Authors:

Kristian Graven Hansen (krgh@itu.dk)

Lukas Sarka (lsar@itu.dk)

Mikkel Geisler (mgei@itu.dk)

1 Introduction

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets

2 Data and Preprocessing

2.1 Dataset

The Fashion-MNIST is a dataset of Zalando article images, consisting of a training set of 60,000 samples and 10,000 test samples. The dataset used here is a small portion of the original dataset, 10,000 training and 5,000 test samples. Each sample is a 28x28 pixels gray-scale image with a label indicating the type of clothing item associated with each.

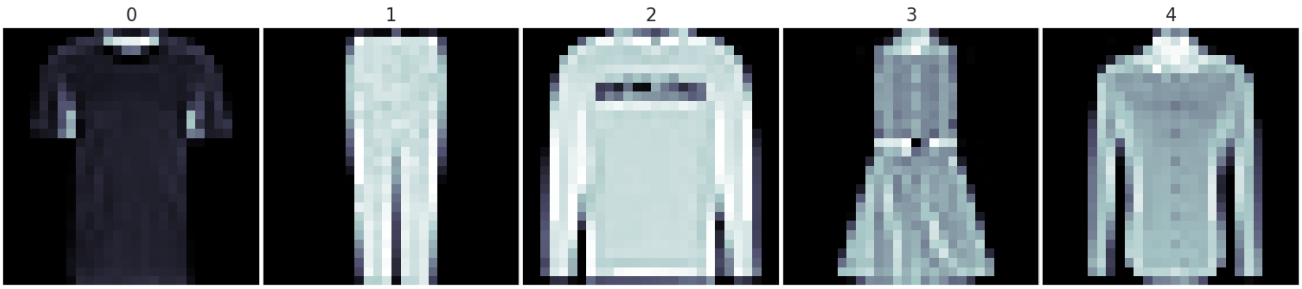

Figure 1: One sample from each class

2.2 Naming Conventions

The exploration of samples within each class gave rise to more appropriate names. Class 0 became T-shirt etc. Below is the naming conventions, which will be used throughout the report for better readability.

Label	**0**	**1**	**2**	**3**	**4**
Type of clothing	T-shirt/Top	Trousers	Pullover	Dress	Shirt

Table 1: Mapping from class-labels to clothing type

2.3 Data Cleaning

The dataset provided was already in a cleaned state, which was verified by checking for missing values, and checking that the pixel values were no greater than 255 and no smaller than 0.

2.4 Preprocessing

The pixels values were in the range $[0, 255]$. This range was normalized to $[0, 1]$ by dividing each pixel by 255. This was done to improve training time as working with small number is slightly faster computationally, and because neural networks prefers to work with small numbers.

2.5 Class Distribution

Whether or not a machine learning model can learn to predict classes well depends to a high degree on how those classes are distributed within our training and training dataset. Both the datasets are extremely balanced, with the test being fully balanced (1000 of each class). This is illustrated on the plots below

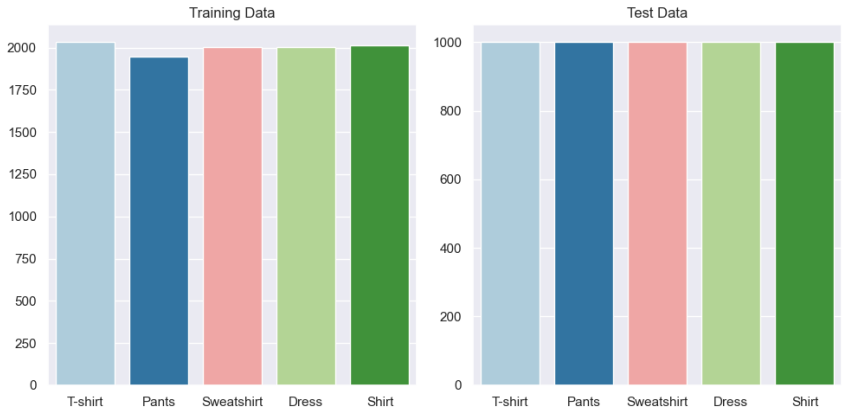

Figure 2: Distribution of clothing items in our training and test dataset

3 Exploratory Data Analysis

One simple yet useful way to explore a dataset is to visualize the feature distribution for each class. In our situation this is simply not possible and not very insightful as we have too many features. To be precise $728 = 28 \times 28$ features/pixels. This doesn't mean the dataset can't be visualized, which we will discuss in the next section.

3.1 Principal Component Analysis

As mentioned before due to our large dataset it is hard to visualize feature distributions, this is where principal component analysis or PCA can be used instead. Principal Component Analysis (PCA) is a dimensionality reduction technique used to reduce the number of features in a dataset, while still preserving the most important information. It does this by transforming variables into a new set of variables, called principal components, which are uncorrelated from each other and explain the maximum amount of variance in the data. Below we explore the relationships between the first 3 principal components. These 3 principal components represent the direction of most variance in the original data

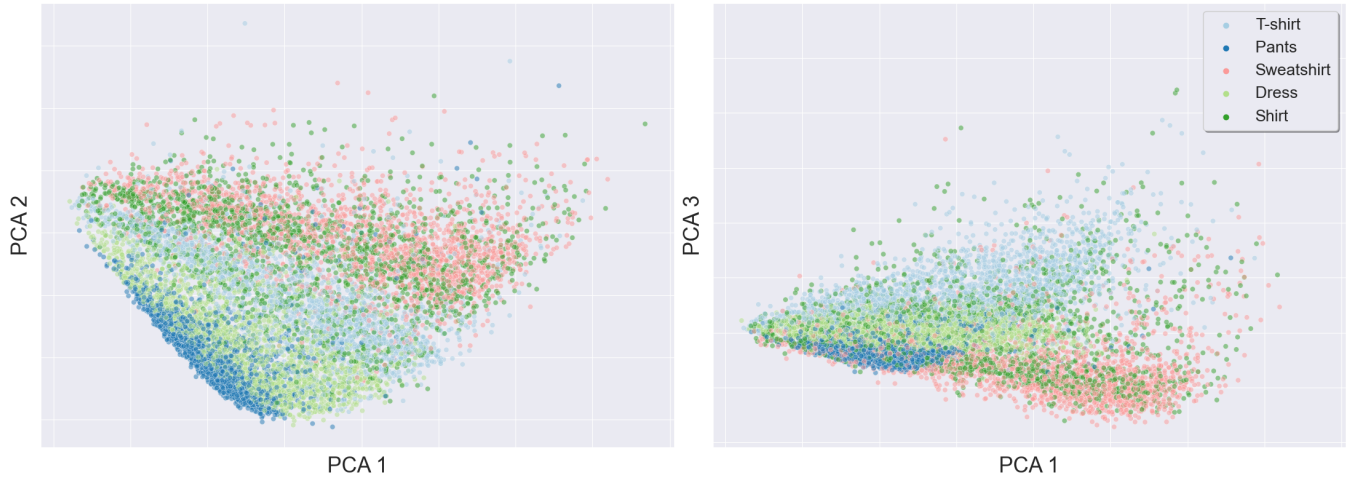

Figure 3: Visualizing some relationships between first 3 PCA's

4 Decision Tree

In this section we will discuss the implementation, choice of hyperparameters and results of the `DecisionTreeClassifier()`. A decision tree is a supervised machine learning model that can be used for regressing or classification. It works by building a tree-like structure, where at each internal node, the algorithm considers all the available features and chooses the split that maximizes the purity of the resulting splits. This process is repeated until a certain stop condition is reached, such as a maximum depth or no further improvement in purity. Decision tree classifiers are popular because they are easy to understand and interpret due to their tree-like structure, and they can handle different data types and types of decision boundaries.

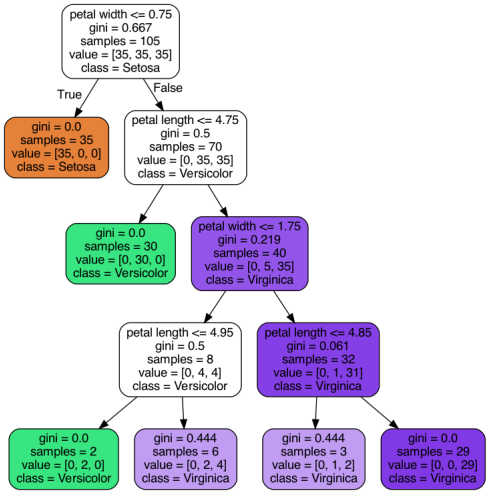

Figure 4: Simple Decision Tree

4.1 Implementation

As this project is focused on classification rather than regression a classification tree was implemented. It was done in Python using two classes:

1. `Node()`

2. DecisionTreeClassifier()

4.1.1 Node

The Node() class has the most responsibility of the two. It holds the data and passes it down the tree constantly splitting itself into more nodes with the `_split()` method. This is done by finding the best split with the `_get_best_split()` method based on the impurity measure specified. The best split refers to the feature and cutoff value that leads to the highest gain in purity in the two resulting nodes. After finding the best split, it can then create two new nodes, `self.left` and `self.right` if all split criteria are fulfilled. The two criteria implemented are `max_depth` and `min_samples_split` and are discussed further in the DecisionTreeClassifier() implementation. This splitting process is what happens recursively in the `fit()` method in the DecisionTreeClassifier(), and is how the tree-structure is build.

4.1.2 DecisionTreeClassifier

The DecisionTreeClassifier() class is the classifier interface which has the `fit()` and `predict()` methods and some 'less' important methods `get_depth()` and `get_n_leaves()`, that can be called after the model has been fitted. The `fit()` method as mentioned earlier simply starts with a root Node() and calls `_split()` recursively. The `predict()` method works by using the sample input to traverse down the tree until a leaf is reached, and predicting the most common class.
Our implementation allow for specification of the following 4 parameters.

(i) `max_depth` – (specify maximum depth to which the tree can grow)
(ii) `min_samples_split` – (specify minimum number of samples in a node before it can be split)
(iii) `criterion` – (split based on either gini or entropy)
(iv) `random_state` – (set random seed for reproducible results)

4.2 Asserting Correctness

To validate the correctness of our implementation a thorough comparison with the sklearn version was done. First a comparison of 100 sklearn models and 100 of our models accuracy score was done using the `load_digits()` dataset from `sklearn.datasets` This can be seen in Figure 5.

Each decision tree used in Figure 5 is fitted with the default parameters of both the sklearn and our model. This means that the `max_depth` is not restricted and the `min_samples_split` is set to 2. A decision tree has randomness involved in the training process. This is because if two splits yield the same gain one of them will be chosen at random. This means the models might perform slightly different in a single test but very similar on average.

The resulting tree depths and number of leaves were also compared and showed that the decision trees in both implementations grow to the same depth and with the same number of leaf nodes.

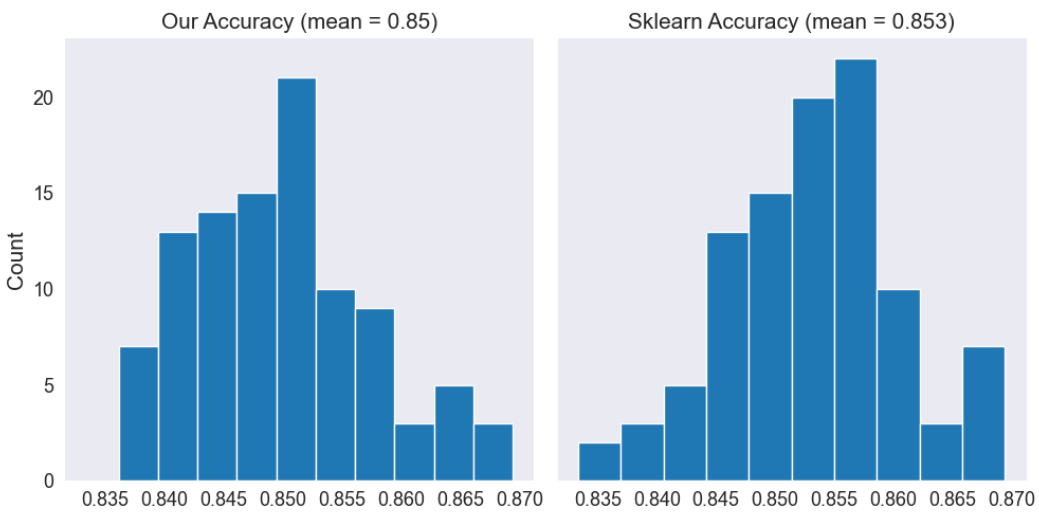

Figure 5: Accuracy distribution from 100 models

Lastly a comparison of **some** of the actual splits (best feature and cutoff) were done to assert correctness and as a sanity check, and the two implementations also agreed on this.

4.3 Hyperparameters

A decision tree is very prone to overfitting to the training data. This is especially the case if the max depth or other early stopping to the growth is not controlled. This can lead to a very high training accuracy but might struggle with unseen data. Another way to control this is using post-pruning where the tree is first grown fully, and then afterwards its size is reduced. This was not implemented in our model so what was tried instead was to find the optimal `max_depth`, to reduce overfitting. Along with finding an optimal `max_depth`, a good combination of `max_depth`, `min_samples_split` and `criterion` (gini or entropy) was found. This was done `GridSearchCV()` from sklearn, where a range of values for each of the mentioned parameters was searched for. `GridSearchCV()` uses cross-validation, in this case 5-fold, on the training data, and tests all combinations specified in the following parameter grid:

- `max_depth`: [None, 5, 7, 9, 11, 12]
- `min_samples_split`: [2, 4, 8, 10]
- `criterion`: [gini, entropy]

This gave the best combination as `max_depth=9`, `min_samples_split=4`, `criterion=gini`. This combination was then used to train our own implementation and the results will be reported in the next section

4.4 Results

The results reported is from our own implementation of the `DecisionTreeClassifier()` with the parameters specified earlier. To reproduce these results the `random_state` parameter should be set to 42.

Evaluation Metric	Accuracy Score
Training Accuracy	89.83%
Test Accuracy	79.32%

Class	Precision	Recall	F1-Score
T-shirt/Top	0.76	0.75	0.75
Trousers	0.97	0.92	0.95
Pullover	0.80	0.83	0.81
Dress	0.82	0.88	0.85
Shirt	0.61	0.60	0.60

Table 2: Decision Tree Performance Evaluation

4.4.1 Discussing Results