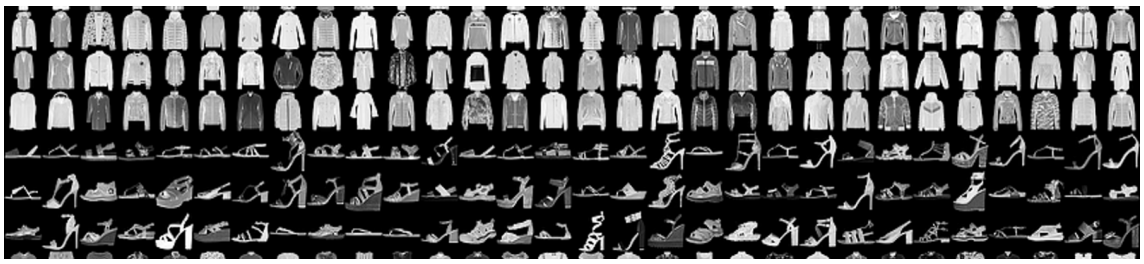

Zalando Clothing Classificaton

January 2, 2023

Authors:

Kristian Graven Hansen (krgh@itu.dk)

Lukas Sarka (lsar@itu.dk)

Mikkel Geisler (mgei@itu.dk)

1 Introduction

In this project, we will investigate different techniques for determining the type of clothing in an image using a dataset of 15,000 labeled images. The images are from the Zalando website and depict t-shirts/tops, trousers, pullovers, dresses, and shirts. We will examine three classification methods: feed-forward neural networks, decision trees, and random forests. The first two methods will be implemented using only Python standard libraries and the NumPy numerical library, while for the last method we will be using the preexisting implementation from Scikit-learn. In addition, we will perform exploratory data analysis and visualization of the dataset through a principal component analysis. The report will cover the implementation of the methods, as well as an interpretation and comparison of their performances.

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 in this project is a small subset of the original dataset consisting of 10,000 training and 5,000 test samples. Each sample is a 28x28 pixel gray-scale image, with an associated label indicating the type of clothing.

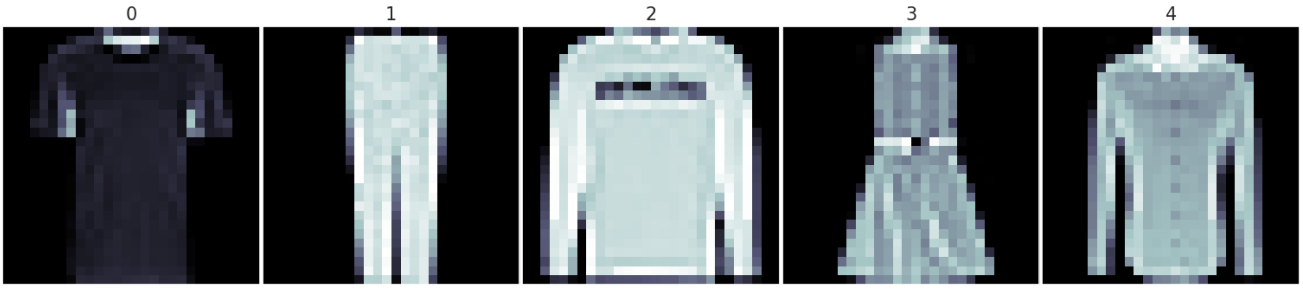

Figure 1: One sample from each class

Table 1 shows the mapping from class label to clothing type. The clothing types will be used instead of the labels in the report for better readability.

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

Table 1: Mapping class label to clothing type

2.2 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.

2.3 Preprocessing

The pixel values were in the range $[0, 255]$. This range was normalized to $[0, 1]$ by dividing each pixel by 255. One reason for scaling the data is to reduce the difficulty of training a neural network, as these models can struggle to learn from large values, due to large gradient values. Additionally, scaling the data can also slightly improve training time, as the models are able to more efficiently process smaller numbers.

2.4 Class Distribution

The success of a machine learning model in predicting classes largely depends on the distribution of those classes within the training and test datasets. Both the datasets are balanced, with the test being fully balanced (1000 of each class). This is visualized in **Figure 2**.

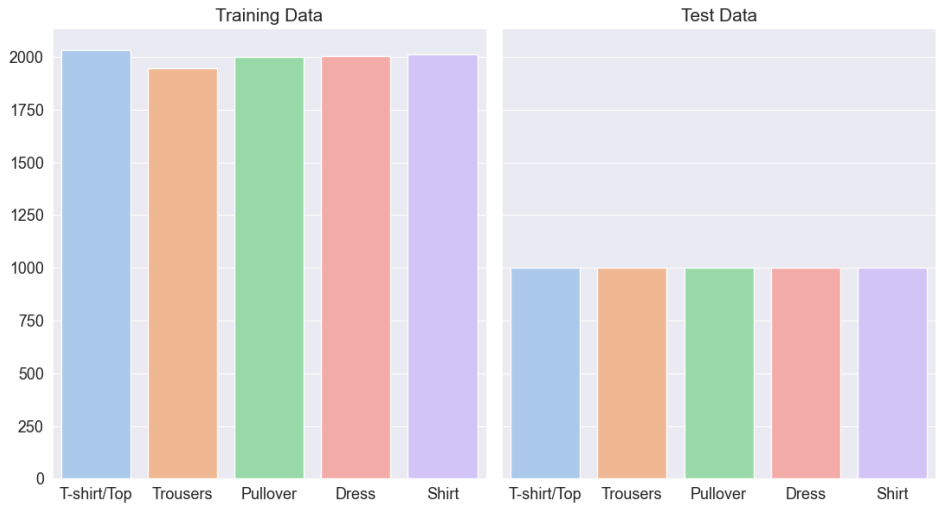

Figure 2: Distribution of clothing types

3 Exploratory Data Analysis

An effective way to understand and explore a dataset is through visualization of the feature distribution for each class. However, when the number of features is high, such as 728 in this case, this method may not be practical or informative. In these situations, alternative approaches for visualizing the dataset can be utilized to extract useful insights.

3.1 Principal Component Analysis

One way to gain insights from a large dataset that might be difficult to visualize otherwise is through the use of principal component analysis (PCA). PCA is a technique that reduces the number of features in a dataset while preserving the most important information. It does so by transforming the original variables into a new set of variables called principal components, which are uncorrelated and capture the maximum amount of variance in the data. In the following analysis, we will investigate the relationships between the first three principal components.

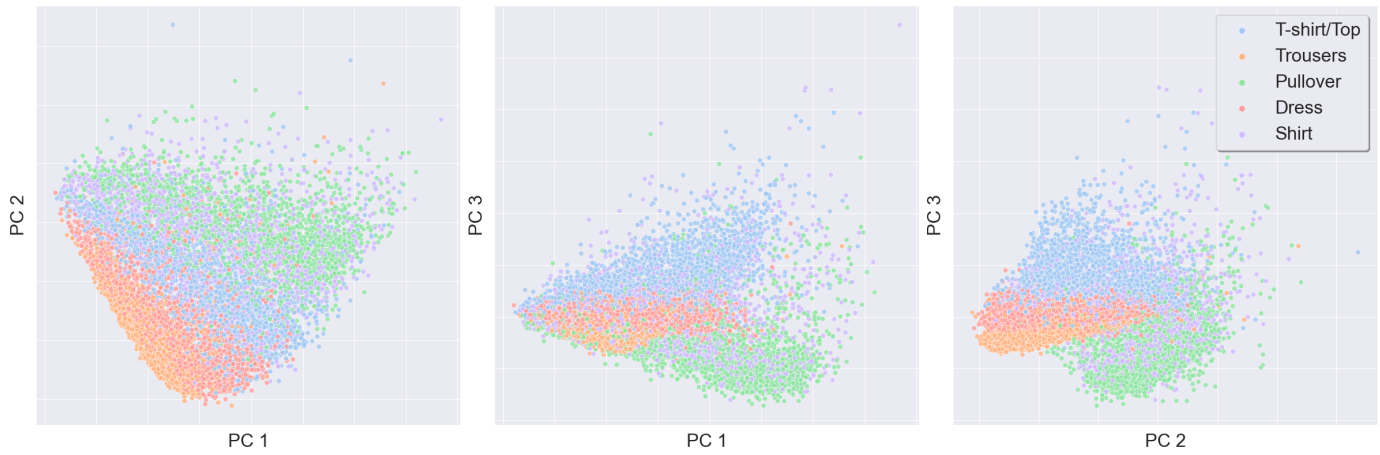

Figure 3: Visualizing relationships between the first 3 principal components

The first 3 principal components explain 42.7% variance, and it requires 61 principal components to explain above 90% variance. PCA plots can be useful for understanding how well the different classes are separated and can provide insight into the difficulty of accurately predicting the various clothing types. **Figure 3** demonstrates that there is significant overlap between many of the classes, particularly between Shirts and Pullovers. This overlap could potentially make it difficult for a classifier to distinguish between these two classes.

4 Decision Tree

In this section, we will discuss how the `DecisionTreeClassifier()` is implemented and how the hyperparameters are chosen, as well as the results of using this model.

A decision tree is a supervised machine learning model that can be used for classification or regression tasks. It works by building a tree-like structure, where at each internal node, the algorithm evaluates all the available features and selects the split that maximizes the purity of the resulting nodes. This process is repeated until a stopping condition is reached, such as a maximum depth or no further improvement in purity. The resulting leaf nodes are also known as decision regions, and the prediction for a sample is made by traversing the tree to find the correct decision region, and then predicting the most common class in that region. Decision trees are popular because they are easy to understand and interpret due to their tree-like structure, and they can handle different types of data and decision boundaries. However, during training, the algorithm only considers the best split for each internal node without considering the potential negative effects on future splits, which is known as a greedy approach, and can potentially lead to suboptimal trees for some datasets.

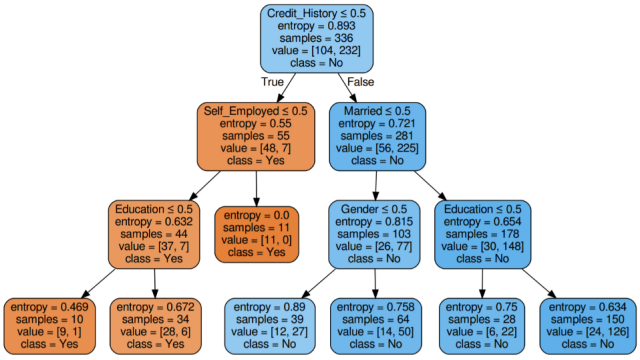

Figure 4: A Simple Decision Tree

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 represents the internal and leaf nodes that make up the decision tree. The `Node()` has the most responsibility of the two classes. 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 either the gini or entropy impurity measure. The term 'best split' refers to the process of selecting the feature and value, that will result in the highest purity gain. The best split of a given `Node()` can then be used to create two new nodes `self.left` and `self.right`.

4.1.2 DecisionTreeClassifier()

The `DecisionTreeClassifier()` is the classifier class. It has the `fit()` and `predict()` methods and two additional methods `get_depth()` and `get_n_leaves()`, that can be called after the model has been fitted, to get the depth and number of leaf nodes of the resulting decision tree. The `fit(X, y)` method takes the following two parameters, a feature matrix `(X)` and class labels `(y)`. This data is fed into the first node of the tree (the root). The purity of the node is then found and if a split can improve the purity and all split criteria (`max_depth` and `min_samples_split`) are fulfilled the node calls `_split()` on itself. This happens recursively and is how the tree-structure is created. The `predict(X)` method takes an array of samples, or a single sample, and predicts the corresponding label. It works by traversing down the tree until a leaf node is reached, and then predicting the most frequent class in that node. The constructor takes the following four parameters:

- `max_depth` - (An integer - the maximum depth to which the tree can grow)

- `min_samples_split` - (An integer - minimum number of samples in a node for it to split)
- `criterion` - (A string - impurity measure ('gini' or 'entropy'))
- `random_state` - (An integer - set the random seed for reproducible results)

4.2 Correctness

To validate the correctness of our implementation a thorough comparison with the Scikit-learn version was made. First, a comparison was made between the accuracy scores of 200 Scikit-learn models and 200 of our models using the `load_digits()` dataset from `Scikit-learn.datasets`, with a training and test size of 80% and 20% respectively.

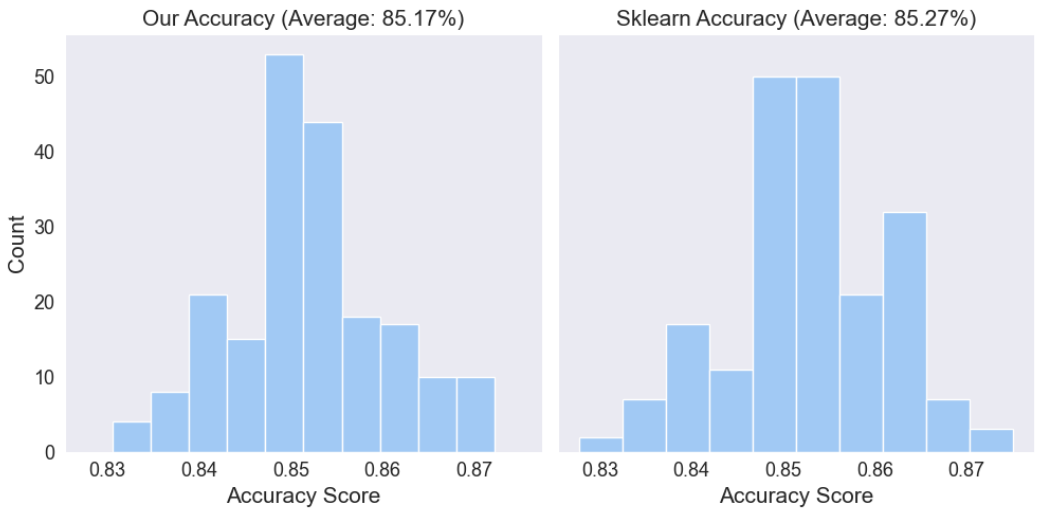

Figure 5: Accuracy distribution from 200 models

Each decision tree used in **Figure 5** is initialized with the default parameters of both the Scikit-learn and our model. This means `max_depth=None` and `min_samples_split=2`.
A decision tree has randomness involved in the training process. This is because if two splits yield the same purity gain, one of them is chosen at random. Therefore, the models might perform slightly different based on a single evaluation but very similar on average as seen in **Figure 5**.

The depth and number of leaves of the resulting trees were also compared and it was found that the decision trees in both implementations had the same depth and number of leaf nodes.

Finally, a comparison of a few of the actual splits (best feature and cutoff) was conducted to verify the accuracy of the implementations and ensure that everything was working as expected. It was found that the two implementations were in agreement on these splits.

4.3 Hyperparameters

A decision tree is extremely prone to overfitting to the training data. A decision tree is highly susceptible to overfitting to the training data, particularly if there is no control over the maximum depth or other measures to limit tree growth. This can lead to a very high training accuracy, but

it might struggle with unseen data. Another way to solve this is using post-pruning where the tree is first grown fully, and then afterwards its size is reduced. Post-pruning was not implemented, instead we focused on identifying an optimal combination of `max_depth`, `min_samples_split`, and `criterion` (gini or entropy). This was done using 5-fold cross-validation with the Scikit-learn class `GridSearchCV()`. The following parameter grid was specified and passed to the `GridSearchCV()` instance.

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

This led to the following combination of hyperparameters:

```
max_depth=9   min_samples_split=4   criterion=gini
```

These parameters were selected due to their ability to effectively prevent overfitting, by achieving the highest validation accuracy score. Due to the balanced dataset and the scope of this project, it was decided to find the parameters that achieved the highest accuracy score rather than other metrics such as precision or recall.

4.4 Results

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

The results in **Table 2** are from an instance of our implementation of the `DecisionTreeClassifier()` with the hyperparameters specified in section **4.3**. To reproduce these exact results it should additionally be initialized with `random_state=42`.

The `DecisionTreeClassifier()` achieved a test accuracy of 79.32% and training accuracy of 89.83%. The model emphasizes the difficulties in correctly classifying shirts, as seen by the Precision, Recall and F1-score. This fits well with the feature overlap seen in **Figure 3**. On the other hand, the model performs well at classifying the other classes, especially trousers.

5 Feed-Forward Neural Network

In this section, we will discuss how the `NeuralNetworkClassifier()` is implemented and how the hyperparameters are chosen, as well as the results of using this model.

A feed-forward neural network is a type of artificial neural network that consists of an input layer, one or more hidden layers, and an output layer. Each layer consists of a set of neurons, which are

connected to the neurons in the next layer through weights. The input layer receives input data and passes it through the network to the output layer, where the output is produced.

The hidden layers process the input data and transform it into a form that can be used by the output layer to produce a prediction. The weights of the connections between neurons are adjusted during the training process to minimize the difference between the predicted output and the actual output, a process known as backpropagation.

Feed-forward neural networks can be used for a wide range of tasks, including image and speech recognition, natural language processing, and predicting outcomes in complex systems. They are a popular choice for machine learning tasks due to their ability to learn and generalize from data. Typically neural networks require significantly more training-data than the traditional machine learning methods to produce desirable results.

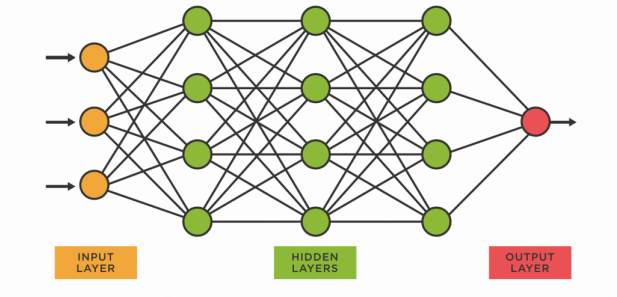

Figure 6: A Simple Neural Network Architecture

5.1 Implementation

The implementation of the Feed-Forward Neural Network was done in Python using two classes.

1. `DenseLayer()`
2. `NeuralNetworkClassifier()`

5.1.1 DenseLayer()

The `DenseLayer()` represents the hidden-layers and the output layer that make up the network. It should be initialized with a `layer_size` and an `activation`. The `layer_size` represents the number of neurons in the layer, for example if you have a multiclass classification with 3 classes the output layer should have `layer_size=3`. The `activation` parameter is the activation function to be applied to the output of the layer. The current implemented activation-functions are Sigmoid, ReLU and Softmax, where Softmax can only be applied to the output layer.

The `DenseLayer()` has two primary methods `forward()` and `backward()`. The `forward(x, weights, bias)` method performs a forward pass through the layer, given the input data (`x`), weights (`weights`), and biases (`bias`). It calculates the dot product of the inputs and weights, adds the bias, and applies

the activation function to the result to produce the output of the layer. It returns the pre-activated output of the layer, and the activation of the layer. The `backward(dA_curr, W_curr, Z_curr, A_prev)` method performs the backward pass through the layer, given the gradient of the cost with respect to the current layer's activations `(dA_curr)`, the current layer's weights `(W_curr)`, the current layer's pre-activation values `(Z_curr)`, and the previous layer's activations (`A_prev`). It calculates the gradients of the cost with respect to the weights (`dW`), biases (`db)`, and previous layer's activations (`dA`) using the chain rule. The return values of both the `forward()` and the `backward()` method are needed for backpropagation which happens in the `NeuralNetworkClassifier()`.

5.1.2 NeuralNetworkClassifier()

The `NeuralNetworkClassifier()` is the classifier class. It's primary methods are `fit()` and `predict()`. Additionally, it includes a lot of helper methods that all have different responsibilities. These include initializing and updating weights and biases based on the layer sizes and input data dimensions specified. The constructor take the following four parameters:

- `layers` - (A list of DenseLayer objects - the layers in the network)
- `learning_rate` - (A float - how fast the weights update after each mini-batch)
- `epochs` - (An integer - the number of epochs to train the neural network for)
- `random_state` - (An integer - set the random seed for reproducible results)

The `fit(X, y, batch_size, validation_size)` method is used to train the model, and takes four parameters: The training data (`X`) and labels (`y`), the `batch_size` (an integer indicating the number of samples to use in each mini-batch during training). Lastly `validation_size` (a float indicating the proportion of the training data to use for validation). To keep it short the `fit()` method does the following: For each mini-batch, the model makes predictions using the `_forward()` method, computes the loss using the `delta_cross_entropy()` function. It then performs backpropagation with the `_backward()` method and finally updates weights and biases using the gradients calculated with the `_backward()` method. This process is performed as many times as specified with by the `epoch` parameter in the constructor.

5.2 Correctness

For the scope of this project the correctness of our implementation was done primarily visually. The training accuracy and training loss was compared to a similar model from the TensorFlow library. The models used have two ReLU activated hidden layers with 128 and 64 neurons respectively, a Softmax activated output layer with 5 neurons (one for each clothing type), a learning rate of 0.01. For reproduction of these exact graphs `random_state=42`, and `tf.random.set_seed(42)` should be used.

Figure 7: Comparing Accuracy and Loss with TensorFlow

The trend in accuracy and loss for our model, as shown in **Figure 7**, is comparable to that of TensorFlow's model. However, TensorFlow's model exhibits a smoother trend in the curves, due to its better optimization. Based on the results displayed in **Figure 7**, our model can be considered to have been implemented correctly.

5.3 Hyperparameters

Neural networks are excellent at learning and generalising from data. This however can often lead to overfitting, when the number of epochs or learning rate is too high. We restricted ourselves to just look at the number of epochs while keeping all the other fixed, meaning, using the same hidden layers as in section **5.2** and the following parameters `learning_rate=0.01`, `batch_size=32`, `random_state=42`. It was optimal to choose a number of epochs that resulted in a high validation accuracy and low validation loss. **Figure 8** shows the models performance during training and was used to choose 70 as the number of epochs. These parameters were then used to train an instance of the `NeuralNetworkClassifier()` and the results are reported in the following section.

Figure 8: Performance during training

Evaluation Metric	Accuracy Score
Training Accuracy	94.29%
Test Accuracy	85.16%

Class	Precision	Recall	F1-Score
T-shirt/Top	0.81	0.80	0.81
Trousers	0.97	0.97	0.97
Pullover	0.86	0.87	0.86
Dress	0.90	0.89	0.90
Shirt	0.72	0.73	0.72

Table 3: Neural Network Performance

5.4 Results

The `NeuralNetworkClassifier()` achieved a test accuracy of 85.16% and training accuracy of 94.29%. Again, our model emphasizes the difficulties in correctly classifying shirts, as seen by the Precision, Recall and F1-score in **Table 3**. The model performs well on correctly classifying Trousers and Dresses, with an achieved F1-score of 0.97 and 0.9.

6 Random Forest

The last classifier used in this project is the `RandomForestClassifier()` from `sklearn.ensemble`. A random forest is a type of ensemble learning method. An ensemble method is a machine learning technique that combines the predictions of multiple models to make more accurate predictions than any individual model. In a random forest, a set of decision trees are typically trained on a subset of the original dataset. The predicted class of an input sample is a vote by the trees in the forest, weighted by their probability estimates. That is, the predicted class is the one with highest mean probability estimate across the trees. Therefore, a random forest is not as prone to overfitting as a single decision tree, since this voting process helps to reduce the variance of the model, and the trees are trained on different parts of the dataset.

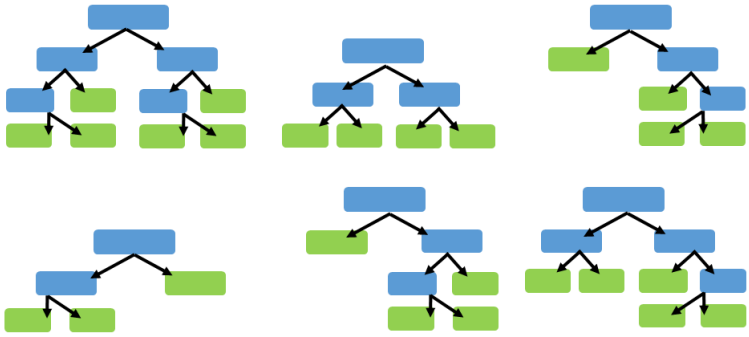

Figure 9: A Random Forest Illustration

6.1 Hyperparameters

The hyperparameters were chosen similarly to the `DecisionTreeClassifier()` with a grid search. The parameter grid used was the following:

- `max_depth`: [None, 10, 15, 20, 21, 22, 23]
- `min_samples_split`: [2, 4, 8]
- `criterion`: ['gini', 'entropy']

Which gave the best combination as:

```
max_depth=21   min_samples_split=4   criterion=gini
```

These values were selected due to their ability to effectively prevent overfitting and improve the accuracy of the model, just like the decision tree parameters. Then an instance of the `RandomForestClassifier()` was trained, while keeping all other parameters at their default values. This meant that the result random forest had 100 trees. Additionally `random_state=42` was used again, and can be used to reproduce the exact results reported in the next section.

6.2 Results

Evaluation Metric	Accuracy Score
Training Accuracy	99.79%
Test Accuracy	84.88%

Class	Precision	Recall	F1-Score
T-shirt/Top	0.81	0.85	0.82
Trousers	0.99	0.95	0.97
Pullover	0.82	0.89	0.85
Dress	0.89	0.93	0.91
Shirt	0.73	0.63	0.68

Table 4: Random Forest Performance

The `RandomForestClassifier()` achieved a test accuracy of 84.88% and training accuracy of 99.97%. The random forest classifier also performs very well at correctly classifying Trouser and Dresses with a F1-score of 0.97 and 0.91 respectively as seen in **Table 4**. This classifier also struggles at classifying Shirts.

7 Discussion of Results

Although all three models would be a decent choice at correctly classifying the images with a reasonable accuracy, it is clear that the feed-forward neural network and random forest classifiers are more suitable for this task. The same pattern persists for all our classifiers, which is that they have an easier time correctly classifying Trousers and Dresses, and they all show difficulty correctly classifying Shirts.

The worst performing classifier is the `DecisionTreeClassifier()`, as it is too simple for this problem, achieving a test accuracy of *only* $\sim 80\%$.
The `NeuralNetworkClassifier()` and the `RandomForestClassifier()` both perform better than the `DecisionTreeClassifier()`, with a similar test accuracy of $\sim 85\%$ overall.
It is worth mentioning that the neural network outperforms the random forest classifier by a few tenths.

To achieve higher accuracy for this particular problem, a model that can detect more complex patterns in data is favourable. This is shown from the accuracy achieved from using the `RandomForestClassifier()`, which is a lot more complex than a single decision tree, since it consists of (in our case) 100 individual decision trees, trained on different subsets of the data, which lead to $\sim 5\%$ performance than the

decision tree.

Since our problem is image classification and neural networks excel at these tasks, it would be reasonable to assume that the `NeuralNetworkClassifier()` would outperform the other classifiers. As mentioned earlier our neural network is a few tenths better than the random forest in terms of accuracy, which could indicate that the network is too simple for this task. To increase the model complexity, it would be optimal to use a different network architecture, with i.e more layers. The use of filters, pooling and convolutions could retain the information in the images which is lost in our network. It is well known that neural networks need a large training dataset for the model to perform well. As the training data in our dataset is one sixth the size of the original Fashion-MNIST dataset, this could be a limiting factor to the accuracy of our model.