IT UNIVERSITY OF COPENHAGEN

Project Title

Machine Learning (BSc DS) Fall 2024

András Fekete
afek@itu.dk

Sune Kjær Christiansen
sunc@itu.dk

January 5, 2025

Contents

Abstract

This study compared the performance of Support Vector Machines, Feed-Forward Neural Networks, and Decision Trees on a subset of the Fashion-MNIST dataset. All models achieved strong performance, with SVMs performing best due to their strength in handling high-dimensional data, followed closely by neural networks, and decision trees providing interpretability at slightly lower accuracy.

1 Introduction

1.1 Background

The Fashion-MNIST dataset is a widely used benchmark for evaluating machine learning models in image classification tasks. It offers a more challenging alternative to the traditional MNIST dataset, featuring images of clothing items rather than simple handwritten digits. Image classification remains a fundamental problem in machine learning, with applications ranging from retail to healthcare. By focusing on a subset of Fashion-MNIST, this project aims to explore the performance of different classifiers in tackling real-world challenges posed by visually similar categories.

1.2 Objectives

The primary objective of this project is to train and evaluate three different classification algorithms on the subset of Fashion-MNIST. These algorithms will be assessed on the basis of their ability to accurately classify the five clothing categories in the dataset. Key performance metrics such as accuracy, precision, recall, and computational efficiency will be analyzed to determine the strengths and limitations of each model. Two of the models will be implemented from scratch, highlighting the importance of understanding the inner workings of machine learning algorithms and gaining deeper insights into their design and functionality. This analysis aims to provide insights into the suitability of various classification approaches for image-based tasks, particularly in scenarios with limited data and closely related class features.

2 Exploratory Data Analysis

2.1 Data Overview

The dataset consists of 10,000 training images and 5,000 testing images. Each image is represented as a 28x28 pixel grayscale matrix, where each pixel's intensity ranges from 0 (black) to 255 (white). The dataset includes five clothing categories: t-shirt/top, trousers, pullover, dress, and shirt, with each image labeled accordingly.

Dataset	T-shirt	Trouser	Pullover	Dress	Shirt	Total
Training	2,033	1,947	2,001	2,005	2,014	10,000
Testing	1,000	1,000	1,000	1,000	1,000	5,000

Figure 1: Class distribution

The table above presents a balanced dataset, with each class containing nearly the same number of samples in both the training and testing sets. This balance simplifies model evaluation, as accuracy serves as a reliable and straightforward metric for overall performance.

The average pixel intensity across all images is approximately 77, with a standard deviation of 90, indicating a wide range of brightness levels. This suggests a diverse set of images in terms of visual complexity and contrast.

2.2 Preprocessing: Image Standardization

Each image undergoes standardization, referred to as *per-image standardization*, where pixel values are scaled to have a mean of 0 and a standard deviation of 1. Representing a set of images as a matrix where each row contains the intensity values of one image, the new value at position ij can be calculated as

$$x'_{ij} = \frac{x_{ij} - \mu_i}{\sigma_i}$$

where x_{ij} denotes the pixel value at the i-th row and j-th column. μ_i and σ_i represent the mean and standard deviation of the i-th row, respectively. This technique follows the principles described in TensorFlow's documentation on image preprocessing

This step minimizes variations in brightness and contrast across the dataset, enabling models

to focus on structural and textural features rather than absolute intensity levels, which are not factors for distinguishing between classes. (Two identical images, one brighter than the other, should ideally belong to the same class.) The pixel intensities are homogeneous in their physical meaning and scale, so variations in intensities across features (pixels) already capture meaningful patterns, such as edges and textures. Therefore, additional feature standardization (column-wise) would artificially enforce equal variance, which we think is unnecessary and could potentially distort these true patterns.

Although decision tree models are inherently less sensitive to feature scaling, standardization can indirectly benefit them when used in conjunction with dimensionality reduction techniques like Principal Component Analysis (PCA).

2.3 Dimensionality Reduction

Principal Component Analysis (PCA) was used to reduce the dimensionality of the image data, enabling visualization and revealing dominant patterns while filtering out noise and redundancy. Since the data is standardized per image, we center each feature around 0 to remove any residual mean differences before applying PCA. By projecting the resulting 784-dimensional image data (28x28 pixels) onto two principal components, this visualization revealed distinct clusters for four of the five classes: t-shirt/top, dress, trouser, and pullover. However, the shirt class (yellow on the plot) demonstrated significant overlap with other categories, suggesting it may present challenges for classification due to its similarity to other classes.

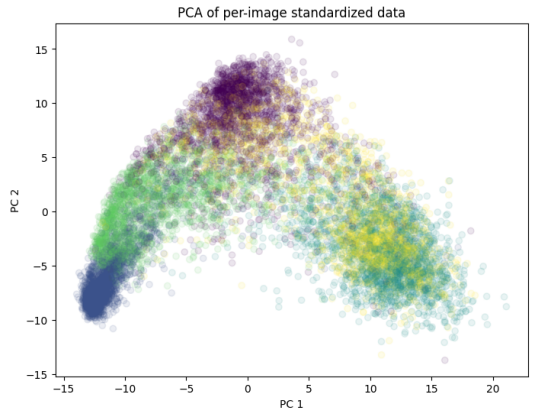

Figure 2: PCA Visualization.

After the 2D plot, we wanted to determine the optimal number of principal components to retain when transforming data for training—maximizing retained variance while minimizing dimensions. We used Parallel Analysis because it is regarded as one of the most rigorous approaches for selecting the optimal number of principal components. By comparing explained variance ratios from the actual data to those from randomly generated data, it identifies components that explain more variance than expected by chance. For each component, we generated 1000 random explained variance ratios using a normal distribution to simulate noise, ensuring that each row of the data was normally distributed and thus comparable to our per-image standardized data. At the end, we created a scree plot comparing the two lists of explained variance ratios (random and non-random from the data). We determined the number of principal components where the explained variance of our data fell below that of random noise (based on the 50th percentile) serving as a cutoff point. This analysis resulted in 65 principal components. This method is based on the approach proposed by Horn (1965)(3), which provides a reliable framework for determining the optimal number of components.

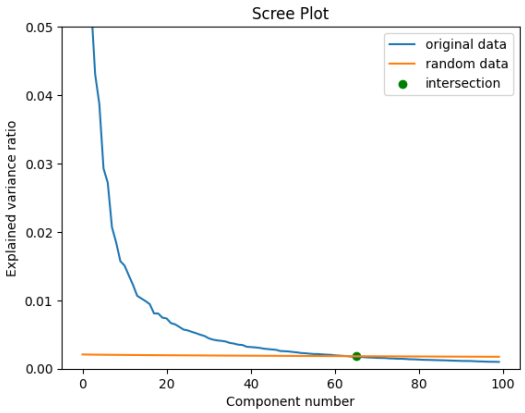

Figure 3: Scree Plot

3 Classification Methods

3.1 Introduction

As our approaches to this classification problem we used Decision Trees, Feed-Forward Neural Networks, and Support Vector Machines (SVMs).

Decision Trees(4), (5) offer an interpretable, rule-based approach that splits data based on feature thresholds, constructing a tree-like structure for decision-making. They are particularly effective for datasets with categorical features and provide insights into feature importance by measuring how much each feature reduces impurity or improves information gain during splits.

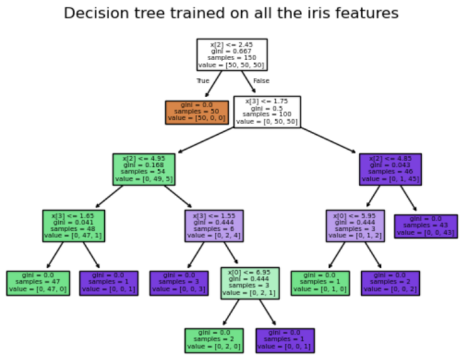

Figure 4: An example of a decision tree. Each internal node represents a splitting rule based on a feature, and each leaf node represents an outcome or prediction.

Feed-Forward Neural Networks consist of layers of interconnected nodes, where each connection has an associated weight. Input data passes through these layers, and at each node, a weighted sum of inputs is computed and transformed by an activation function. The output is propagated forward through the network to generate predictions. Weights are adjusted using backpropagation, which minimizes prediction errors by iteratively updating weights based on gradients, enabling the network to learn complex patterns and relationships in the data.

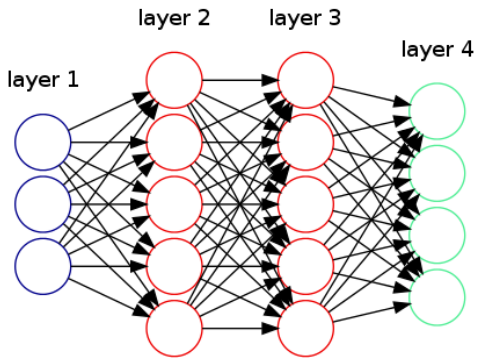

Figure 5: Feed-Forward Neural Network

Support Vector Machines (SVMs) classify data by finding a hyperplane, a boundary that separates different classes. They aim to maximize the margin—the distance between the hyperplane and the closest data points—making them effective for datasets with clear decision boundaries.

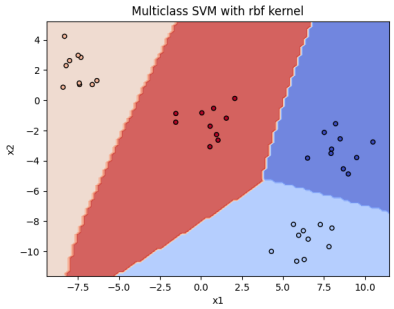

Figure 6: Support Vector Machine

3.2 Decision Tree

3.2.1 Implementation From Scratch

This implementation constructs a decision tree classifier using Python and NumPy for optimized computation.

Big Picture: The decision tree is built recursively using the _build_tree method. At each node, the algorithm evaluates all features and thresholds to find the best split based on impurity reduction. If a stopping criterion is met, such as reaching maximum depth or insufficient samples, the node becomes a leaf, predicting the majority class. Otherwise, the data is split into left and right subsets, and the recursion continues for each subset.

Gini Impurity: (5) Measures the probability of misclassification within a node. It is defined as:

$$Gini(S) = 1 - \sum_{i=1}^{C} p_i^2$$

where C is the number of classes, and p_i is the proportion of samples belonging to class i.

Information Gain:(5) Quantifies the reduction in impurity achieved by a split. It is calculated as:

$$IG(S, S_l, S_r) = G(S) - \left(\frac{|S_l|}{|S|} G(S_l) + \frac{|S_r|}{|S|} G(S_r) \right)$$

where S is the set of samples before the split, S_l is the subset of samples resulting from the split that go to the left child node, S_r is the subset of samples

resulting from the split that go to the right child node, $G(S)$ represents the impurity measure (in our case Gini impurity) of the set S, and $|S|$, $|S_l|$, and $|S_r|$ denote the number of samples in sets S, S_l, and S_r, respectively.

Splitting Data: Each feature is tested with different thresholds to determine splits that maximize information gain. Thresholds are derived dynamically based on sorted feature values, ensuring computational efficiency.

Optimization: NumPy operations are used extensively to avoid loops, enabling fast computation. Dynamic updates of class counts and impurities leverage vectorized operations, making the algorithm scalable to larger datasets.

3.2.2 Reference Implementation

We developed a reference implementation using the sklearn module to validate the accuracy and correctness of our custom decision tree classifier. By utilizing the DecisionTreeClassifier class from sklearn, we conducted direct performance comparisons between the two implementations. In Section 5, we present a detailed evaluation using test data to compare the performance metrics of both the reference implementation and our custom classifier.

3.3 Feed-Forward Neural Network

3.3.1 Implementation From Scratch

This implementation defines a feedforward neural network with multiple hidden layers, customizable learning rates, and dropout regularization.

Network Architecture The neural network is initialized with a given number of layers and neurons per layer. The weights(6) ($W^{[l]}$) are initialized using He initialization, which scales the weights based on the size of the previous layer:

$$W^{[l]} \sim \mathcal{N}\left(0, \frac{2}{n_{l-1}}\right) \tag{1}$$

and biases(6) ($b^{[l]}$) are initialized to zero:

$$b^{[l]} = 0 \tag{2}$$

This initialization improves convergence by maintaining appropriate variance in activations.

Forward Propagation(7) computes outputs layer by layer, starting from the input data. Each layer computes its linear combination (z) and applies an activation function:

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]} \tag{3}$$

$$a^{[l]} = f(z^{[l]}) \tag{4}$$

where $f(\cdot)$ is the ReLU activation function:

$$f(x) = \max(0, x) \tag{5}$$

Dropout is a regularization technique applied during training to randomly deactivate neurons with probability p. This helps prevent overfitting by forcing the network to learn redundant and distributed representations. During inference, dropout is not applied, but the activations are scaled by $\frac{1}{1-p}$ to maintain the expected output magnitude, ensuring consistency with training.

The final layer applies the softmax activation function to produce class probabilities:

$$\hat{y} = \text{softmax}(z^{[L]}) \tag{6}$$

Loss Function(8) The network uses the cross-entropy loss function to measure prediction error:

$$L = -\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{C} y_{ij} \log(\hat{y}_{ij}) \tag{7}$$

where y_{ij} is the one-hot encoded true label, and $\hat{y}_{ij}$ is the predicted probability.

Backward Propagation(9) Gradients are computed using backpropagation. For the output layer, the gradient of the loss with respect to the logits is:

$$\delta^{[L]} = \hat{y} - y \tag{8}$$

For hidden layers, gradients propagate backward using the chain rule:

$$\delta^{[l]} = (W^{[l+1]})^T \delta^{[l+1]} \cdot f'(z^{[l]}) \tag{9}$$

where $f'(\cdot)$ is the derivative of the ReLU activation:

$$f'(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{10}$$

The gradients for weights and biases are computed as:

$$\frac{\partial L}{\partial W^{[l]}} = \frac{1}{m} \delta^{[l]} (a^{[l-1]})^T \tag{11}$$

$$\frac{\partial L}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} \delta_i^{[l]} \tag{12}$$

Weights and biases are updated using gradient descent:

$$W^{[l]} := W^{[l]} - \alpha \frac{\partial L}{\partial W^{[l]}} \tag{13}$$

$$b^{[l]} := b^{[l]} - \alpha \frac{\partial L}{\partial b^{[l]}} \tag{14}$$

where α is the learning rate.

Training and Evaluation The training process uses mini-batch gradient descent, iterating through randomized batches of data to reduce computation cost. The network monitors performance using the loss function and periodically evaluates the accuracy of predictions:

$$\text{Accuracy} = \frac{1}{m} \sum_{i=1}^{m} 1(\hat{y}_i = y_i) \tag{15}$$

3.3.2 Reference Implementation

Just as we implemented a reference model using a well-known module for the decision tree classifier, we also developed a reference model for our neural network. We chose to construct a neural network using the Tensorflow module. This ensures we can evaluate the correctness and performance of our implementation. The comparison results will be presented in Section 5.

3.4 Support Vector Machine (SVM)

3.4.1 Why SVM?

Support Vector Machine (SVM) was chosen as our third model because its strengths align well with the requirements of our classification task. Each image in our dataset consists of 784 features (28x28 pixels), making it high-dimensional. SVM is particularly effective at handling data with many features, ensuring that it can separate classes effectively in such a feature-rich space.

Another key strength of SVM is its robustness to outliers. The decision boundary of an SVM is influenced only by the support vectors—critical data points near the margin—making the model largely unaffected by data points far from the margin. This ensures that mislabeled or noisy images do not significantly degrade performance.

Additionally, SVM's use of the RBF(1) (Radial Basis Function) kernel allows it to model complex, non-linear decision boundaries. This is especially important for classifying clothing images, where features of different classes (e.g., shirts and t-shirts) may overlap significantly in their pixel representations. The RBF kernel adapts to these non-linear relationships, enabling the model to capture subtle distinctions between classes effectively.

Overall, SVM's ability to handle high-dimensional data, robustness to outliers, and flexibility with non-linear relationships is the reason i chose SVM.

3.4.2 Parameter γ: Kernel Flexibility(1)

The RBF kernel, also known as the Gaussian kernel, is defined as:

$$\gamma = \frac{1}{2\sigma^2}$$

$$K(x, z) = \exp\left(-\gamma \|x - z\|^2\right)$$

Here, γ controls how far the influence of a single training point extends in feature space. A small γ value corresponds to a broader kernel function, where individual training points have a wide influence. This results in a smoother decision boundary but risks underfitting the data, as the model may fail to capture important details.

In contrast, a large γ value narrows the kernel's influence to a localized region around each training point. This increases the model's flexibility, enabling it to capture finer patterns in the data, but it also risks overfitting, as the decision boundary may become overly complex and sensitive to noise.

Selecting the appropriate γ value is critical for balancing these trade-offs. This is typically achieved through cross-validation to evaluate performance over a range of γ values.

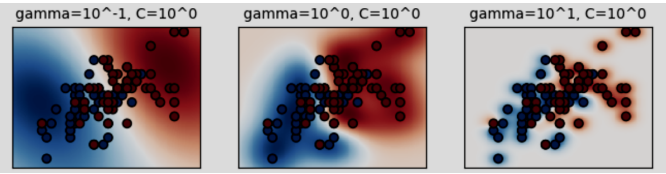

Figure 7: The effect of gamma

3.4.3 Parameter C: Regularization(2)

The parameter C determines the trade-off between achieving a large margin and minimizing classifica-

tion errors on the training data. A high C value places greater emphasis on minimizing classification errors, leading the model to fit the training data more tightly. While this reduces training error, it increases the risk of overfitting, as the model may capture noise and fail to generalize to unseen data.

Conversely, a small C value allows the model to tolerate more misclassifications in the training data, prioritizing a larger margin. This encourages the model to focus on the broader structure of the data rather than individual points, improving generalization but potentially increasing training error.

Choosing the right C value involves finding a balance between underfitting and overfitting, which can be achieved through cross-validation over a range of C values.

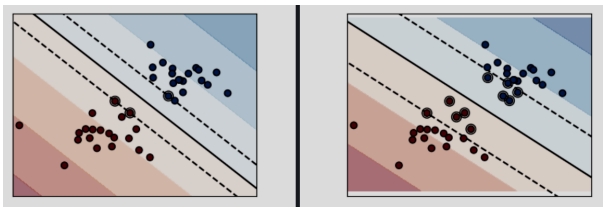

Figure 8: left = high C, rigth = low C

4 Hyperparameter Tuning

To identify the optimal hyperparameters for each of our models—Decision Tree, Feedforward Neural Network, and SVM—we defined a search space for each and employed scikit-learn's `GridSearchCV` with stratified k-fold cross-validation aiming to maximize accuracy. Stratification ensured that class proportions were maintained across folds, thus providing a more reliable estimate of model performance.

4.1 Decision Tree

We tuned two hyperparameters:

- **max_depth:** controlled by $[3, 5, 7, 9, 11]$. Excessively deep trees can overfit the data, so we limited the search to these values.
- **min_samples_split:** tested values $[2, 3, 4, 5]$. This parameter specifies the minimum number of samples required to split an internal node, helping control overfitting.

4.2 Feed-forward Neural Network

We explored three hyperparameters:

- **Layers (network architecture):**
  - $[input\_layer, 128, output\_layer]$
  - $[input\_layer, 256, 128, output\_layer]]$
  - $[input\_layer, 512, 256, 128, output\_layer]]$
- **Dropout:** $[0, 0.2, 0.3, 0.5]$
- **Learning rate:** $[0.001, 0.01, 0.1]$

We implemented a custom grid search to systematically evaluate every combination of these hyperparameters. Each configuration was trained and validated using the same, and the best-performing set of hyperparameters was selected.

4.3 SVM

For our SVM, we tuned:

- **gamma:** $[0.001, 0.01, 0.1, 1.0, 10.0]$
- **C:** $[0.001, 0.01, 0.1, 1.0, 10.0,$ 'scale', 'auto'$]$

Again, we used `GridSearchCV` with stratified k-fold cross-validation, mirroring the approach taken for the Decision Tree.

4.4 Final Chosen Hyperparameters

Based on our hyperparameter tuning procedure, the following configurations yielded the best performance for each model:

- **Decision Tree:**
  $max\_depth = 7$, $min\_samples\_split = 5$
- **Feedforward Neural Network:**
  $layers = [65, 512, 256, 128, 5]$,
  $dropout = 0.2$, $learning\_rate = 0.1$
- **SVM:**
  $C = 10$, $gamma =$ `'scale'`
  scale $= \frac{1}{n_{Features} \cdot \sigma^2}$

These final models serve as the basis for our evaluations and analyses.

5 Results

5.1 Model correctness evaluation

5.1.1 Decision Tree Implementation Evaluation

To validate the correctness of our custom Decision Tree implementation, we compared its accuracy against scikit-learn's reference implementation. Our model achieved an accuracy of **77.62% ± 0.57%**, whereas the scikit-learn Decision Tree attained **77.36% ± 0.57%**. Because this slight difference lies within the standard deviation, we conclude that our implementation performs comparably to scikit-learn's.

While a decision tree generally does not introduce randomness once parameters are fixed, minor discrepancies can arise from implementation details. For instance, our method uses information gain for splitting decisions, while scikit-learn defaults to the Gini impurity measure. Additionally, variations in how boundary conditions (e.g., $\geq$ threshold) are handled can lead to small differences in performance. Despite these nuances, our results confirm that our Decision Tree implementation is effectively equivalent to scikit-learn's in terms of accuracy.

5.1.2 Feed-Forward Neural Network Implementation Evaluation

Our custom Feed-Forward neural network achieved an accuracy of **86.41% ± 0.47%**, where as the TensorFlow reference model attained **78.75% ± 0.57%**. A notable distinction lies in how we implement gradient descent. Specifically, our approach uses mini-batch gradient descent, updating the parameters after each mini-batch, while the TensorFlow model employs stochastic gradient descent, updating weights and biases after every single sample. Although stochastic gradient descent typically converges more quickly, it can be more susceptible to overfitting compared to mini-batch methods.

We also tested the reference model with the same hyperparameters optimized for our mini-batch approach. However, the learning rate—which works well for mini-batch gradient descent—appears too high for purely stochastic updates, likely contributing to the reference model's reduced accuracy. In light of this, and given that our model performs well, we conclude that our implementation is on par with TensorFlow in terms of reliability.

5.2 Presenting Metrics

5.2.1 Decision Tree Performance Evaluation

Our Decision Tree model achieved the following metrics:

- **Accuracy:** 77.62% ± 0.57%
- **Recall:** 77.62% ± 0.57%
- **Precision:** 77.64% ± 0.58%
- **F1 Score:** 77.58% ± 0.58%

Although the overall accuracy is lower than that of our neural network, the Decision Tree still performs reasonably well for most classes.

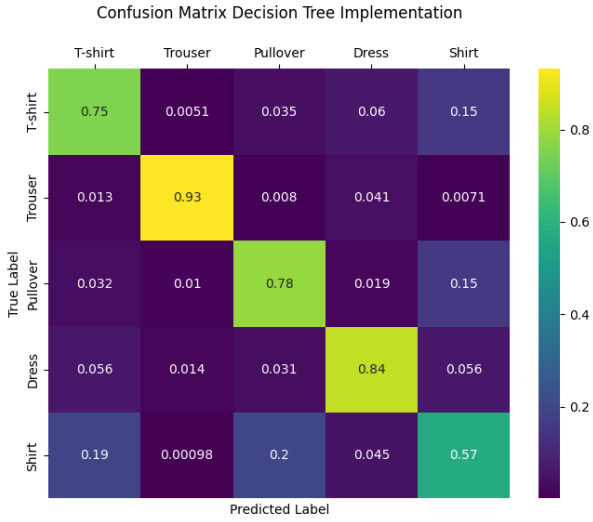

Figure 9

- **Trouser** shows the highest correct classification rate at 0.93, indicating that the model consistently identifies Trousers accurately.
- **Shirt** has the greatest confusion, with only 0.57 on the diagonal. Notably, 19% of shirts are misclassified as T-shirts and 20% as Pullovers.
- **T-shirt** also shows moderate confusion, with 15% of T-shirts being predicted as Shirts.
- **Dress** and **Pullover** are relatively well-classified, with 0.84 and 0.78 on the diagonal, respectively.

5.2.2 Feedforward neural network preformance Evaluation

To assess our feedforward neural network, we evaluated several standard classification metrics:

- **Accuracy:** 86.41% ± 0.47%
- **Recall:** 86.41% ± 0.47%
- **Precision:** 86.36% ± 0.48%
- **F1 Score:** 86.33% ± 0.48%

Overall, the model demonstrates solid performance, with balanced precision and recall leading to a similarly high F1 score.

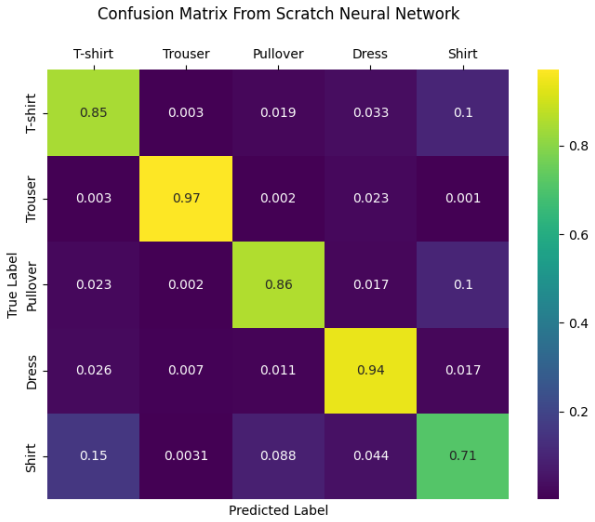

Figure 10

- **Trouser** is the most accurately classified category, with a diagonal value of 0.97.
- **Shirt** shows the lowest diagonal value (0.71), indicating greater confusion, particularly with T-shirts.
- Misclassifications among **T-shirt**, **Pullover**, and **Shirt** suggest similarities that can cause the network to confuse these classes.

5.2.3 SVM preformance Evaluation

The SVM classifier achieved the following metrics:

- **Accuracy:** 87.22% ± 0.48%
- **Recall:** 87.22% ± 0.48%
- **Precision:** 87.26% ± 0.48%
- **F1 Score:** 87.21% ± 0.48%

These results indicate that the SVM model provides strong overall performance, with balanced precision and recall leading to a similarly high F1 score.

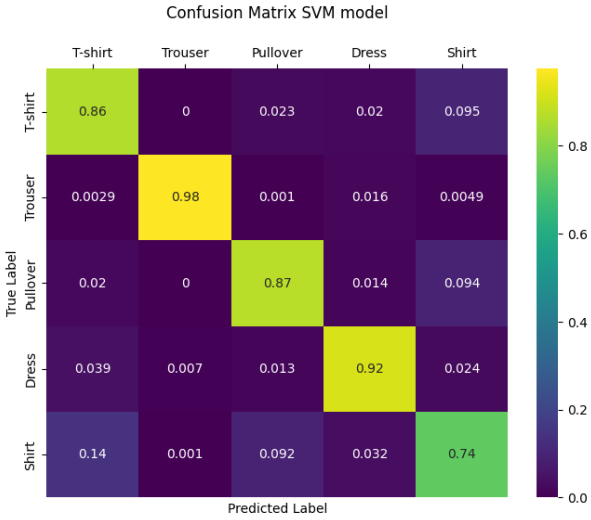

Figure 11

- **Trouser** (0.98 on the diagonal) is classified most accurately, with minimal misclassification into other categories.
- **Pullover** and **Dress** each exhibit fairly high correct classification rates (0.87 and 0.92, respectively).
- **T-shirt** has slightly more confusion compared to Trouser and Dress; 9.5% of T-shirts are misclassified as Shirts.
- **Shirt** has a diagonal score of 0.74, which is lower than other classes, indicating the model sometimes confuses Shirts with T-shirts or Pullovers.

6 Interpretation of Results

The performance of the models highlights key insights about the dataset and classification challenges posed by specific classes.

Trousers were the easiest class to distinguish across all models due to their distinct vertical structure, reducing overlap with other categories and simplifying classification. Conversely, shirts were

the hardest to classify, likely due to their similarity to t-shirts and pullovers, causing ambiguity for all models.

The remaining classes—t-shirts, pullovers, and dresses—had comparable predictability, reflecting that their similar shapes and features were captured consistently by all classifiers.

The SVM achieved the highest performance, likely due to its ability to handle high-dimensional data and create non-linear decision boundaries with kernel functions. Neural Networks closely followed, demonstrating their ability to model complex patterns but showing potential for improvement with larger datasets. Decision Trees, while ranking third, remained interpretable and efficient, making them suitable for tasks prioritizing transparency.

Performance differences may have been influenced by the limited hyperparameter tuning. A broader search space could potentially improve results, especially for neural networks and decision trees.

Despite these differences, all models achieved accuracy close to or above 80%, suggesting that the dataset's distinct classes and effective preprocessing minimized noise and redundancy. These results emphasize that classifier selection should balance accuracy, interpretability, and computational efficiency, depending on application requirements.

7 Conclusion

The results highlight the importance of preprocessing, including dimensionality reduction and standardization, in simplifying the learning problem and improving classifier performance. Despite limited hyperparameter tuning, the models demonstrated high accuracy, emphasizing the suitability of the dataset for classification tasks.

Future work can focus on refining the dimensionality reduction process by using stricter percentiles (e.g., 90th or 95th) in parallel analysis to better capture variance and reduce noise. Additionally, adopting more systematic and data-driven methods for hyperparameter selection could improve efficiency and accuracy. Neural network performance may also benefit from experimenting with alternative activation functions, such as leaky ReLU, to enhance learning capabilities. These improvements can further optimize classifier performance and generalizability across datasets.

References

[1] P. Zahadat, "Lecture 17, slide 40: Artificial neural networks." Lecture slides, Introduction to Machine Learning, 2024.

[2] P. Zahadat, "Lecture 18, slide 33-39: Artificial neural networks." Lecture slides, Introduction to Machine Learning, 2024.

[3] E. Dobriban and A. B. Owen, "Deterministic parallel analysis: An improved method for selecting factors and principal components," *arXiv preprint arXiv:1711.04155*, 2017.

[4] G. James, D. Witten, T. Hastie, R. Tibshirani, and J. Taylor, *An Introduction to Statistical Learning with Applications in Python*. New York, NY: Springer, 2023.

[5] P. Zahadat, "Lecture 14: Decision trees." Lecture slides, Introduction to Machine Learning, 2024.

[6] P. Zahadat, "Lecture 21, slide 36: Artificial neural networks." Lecture slides, Introduction to Machine Learning, 2024.

[7] P. Zahadat, "Lecture 20, slide 23-33: Artificial neural networks." Lecture slides, Introduction to Machine Learning, 2024.

[8] P. Zahadat, "Lecture 21, slide 9-10: Artificial neural networks." Lecture slides, Introduction to Machine Learning, 2024.

[9] P. Zahadat, "Lecture 20, slide 42: Artificial neural networks." Lecture slides, Introduction to Machine Learning, 2024.