CSL2050

Pattern Recognition and Machine Learning

Handwritten Digit Recognition

findings:

Git hub link: https://github.com/sujayv16/PRML_PROJECT
Colab file link: Handwritten Digit Recognition
Project page:
Presentation link:
https://www.canva.com/design/DAGC83lw9hE/5bd7AbscobKZ-rHgbEVYDg/edit
Spotlight video: **https://youtu.be/UzAvotuFquI?feature=shared**

Aim:

to understand traditional machine learning techniques comprehensively. This entails learning and applying these techniques to address a chosen task, conducting systematic performance evaluations, comparing different methods, and developing innovative ideas.

Algorithms used:

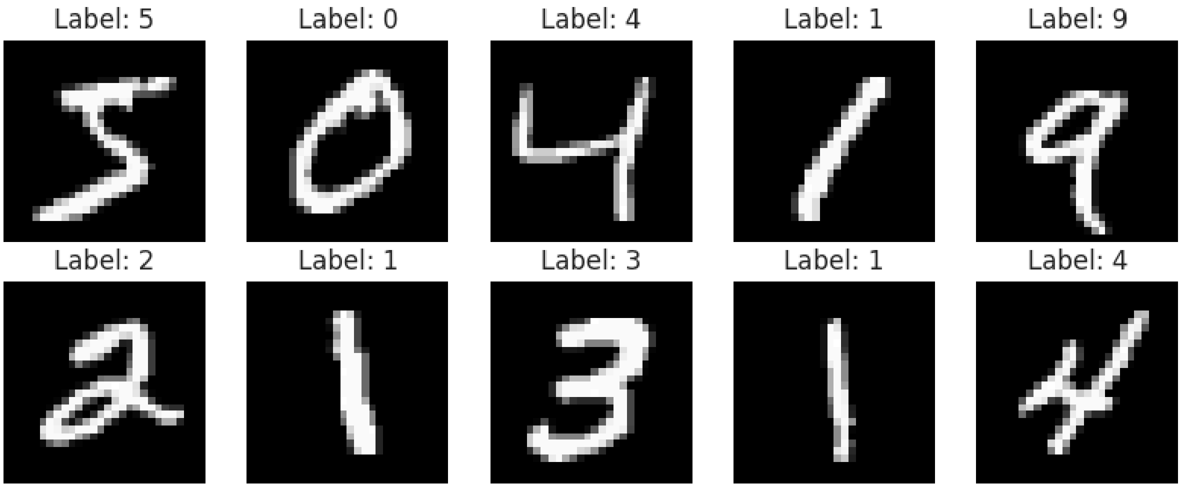
- **Knn Classifier**
- **Neural Network**
- **Convolutional Neural Network (CNN)**
- **Random Forest Classifier**
- **Decision Tree Classifier**
- **Classifier Using Multi-Layer Perceptrons (MLP)**
- **Support Vector Machine (SVM)**

About the data set:

The MNIST dataset, an abbreviation for the Modified National Institute of Standards and Technology dataset, comprises 60,000 training images and 10,000 testing images. Each image is a grayscale, small square of dimensions 28x28 pixels, depicting handwritten single digits ranging from 0 to 9. The primary objective is classification, where the task involves assigning a given handwritten digit image to one of ten classes representing integers from 0 to 9. Notably, the dataset has been extensively explored and is considered "solved," with top-performing models, notably deep learning convolutional neural networks (CNNs), achieving classification accuracies surpassing 99% on the test set, accompanied by error rates typically ranging between 0.4% and 0.2%. Originally derived from the NIST dataset, MNIST is a subset comprising images from Special Database 1 and Special Database 3, which were compiled from handwritten digits by high school scholars and workers from the United States Census Bureau, respectively. Despite its simplicity, MNIST remains pivotal in machine learning research and education, serving as a benchmark for evaluating novel algorithms and techniques in image classification.

Loading MNIST Dataset:

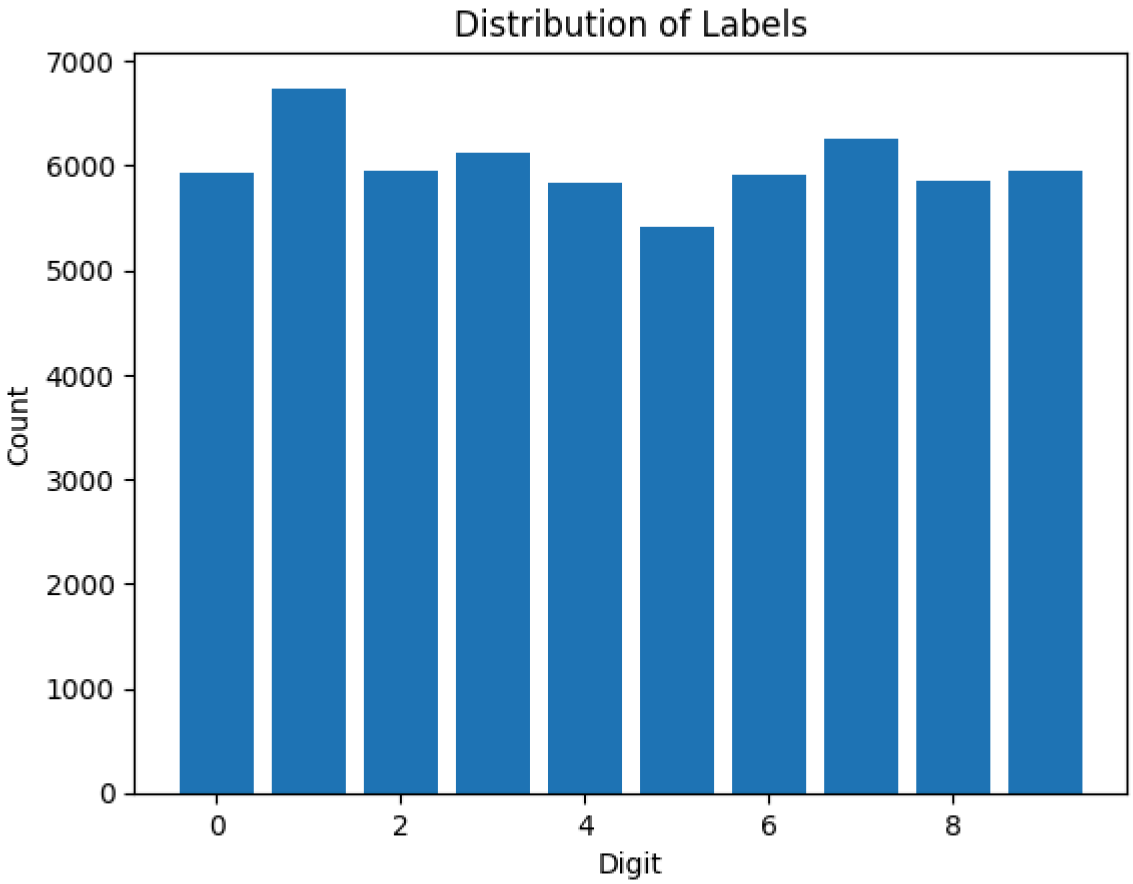
The MNIST dataset is loaded using the mnist.load_data() method. It includes handwritten numerals and labels for both training and testing sets.



Visualisation:

Matplotlib is used to display training set sample images. Each photograph is accompanied by its respective label.

The distribution of labels in the training set is depicted using a bar plot. This sheds light on the balance or imbalance of digit classes in the dataset.

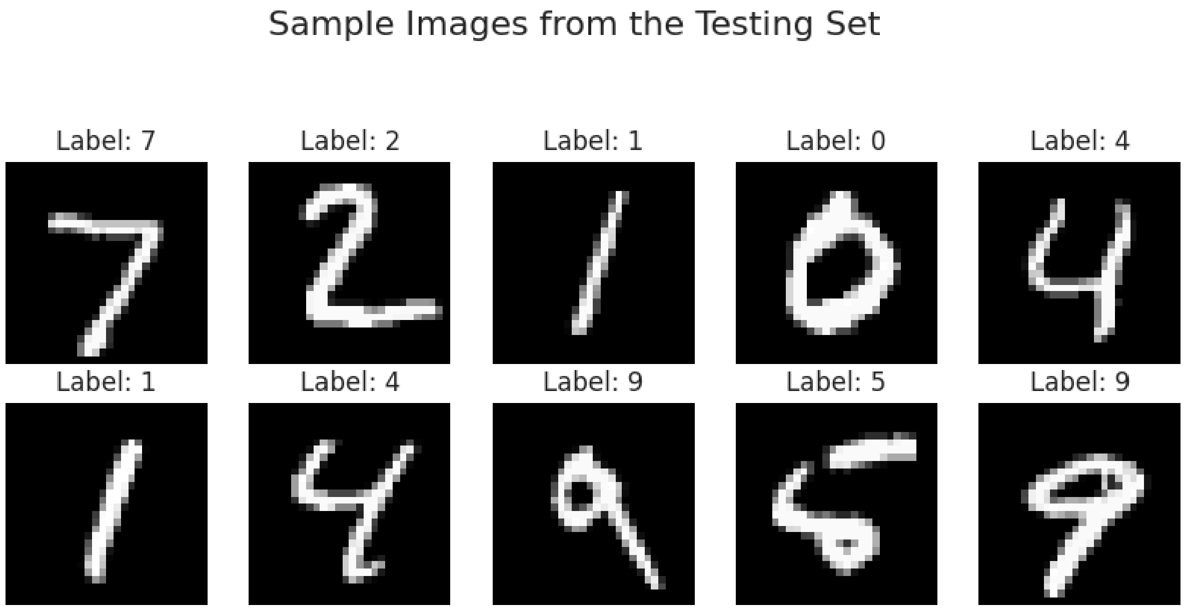


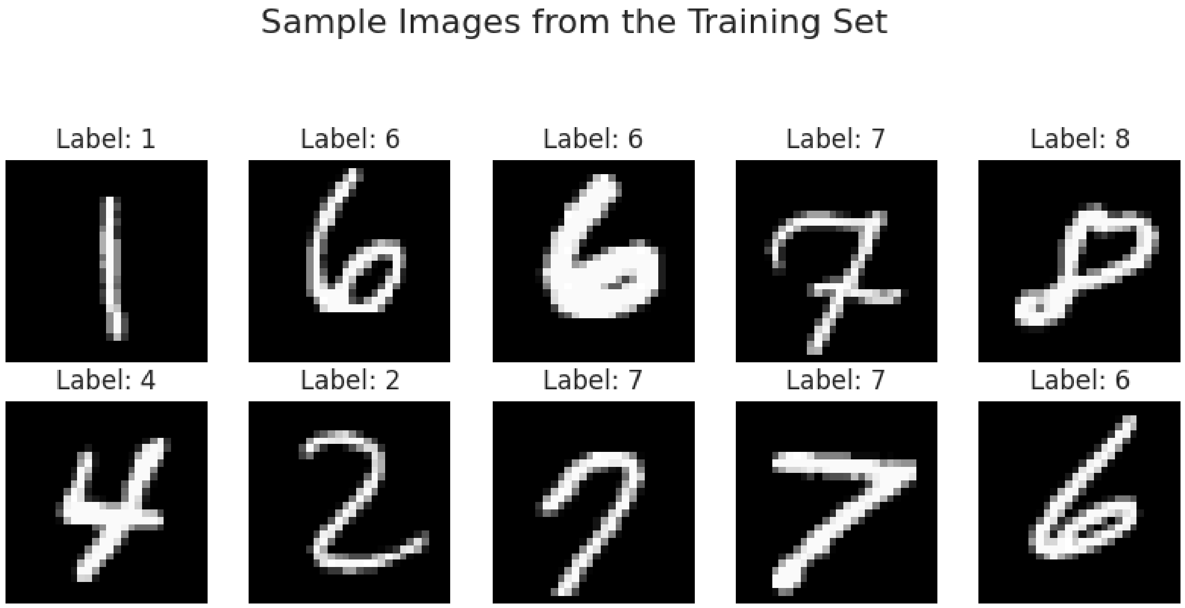
Data preprocessing:

A single channel (28x28x1) is created by reshaping the photos.

After being transformed to floats, pixel values are normalised to the interval [0, 1].
Using train_test_split() from scikit-learn, the training data is divided into training and validation sets.

Additional Visualization Sample photos from the training set, sample images from the testing set, and sample images following preprocessing are examples of additional visualisations.





Printing Dataset Details:

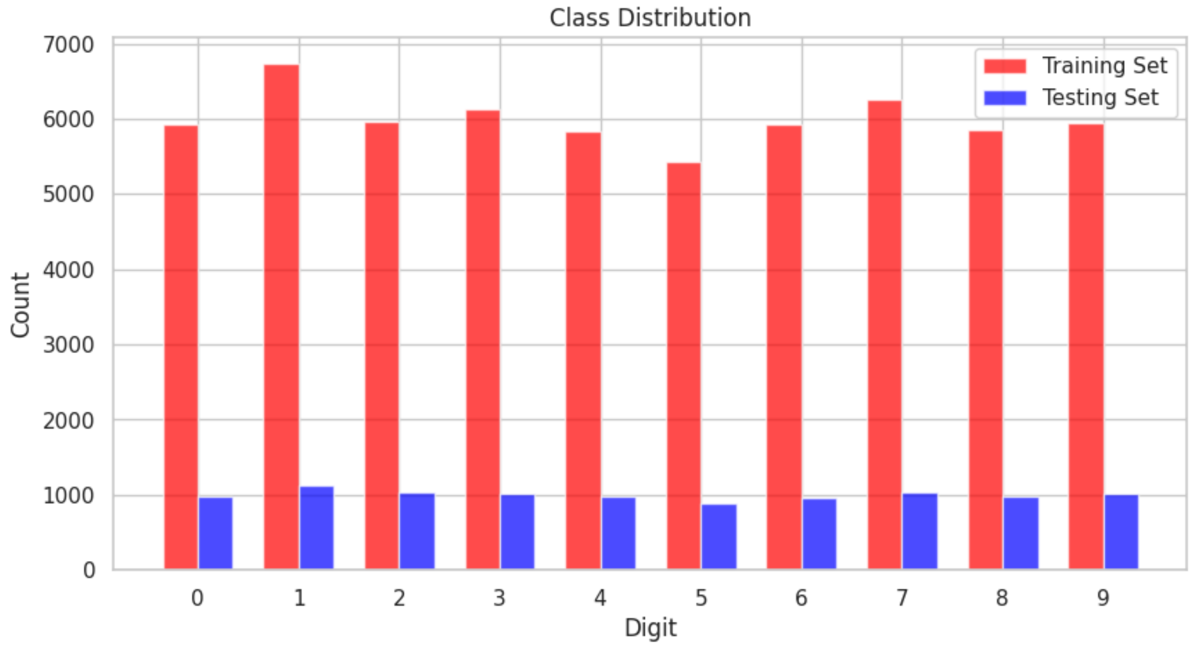
The number of samples in the training and testing sets, the shape of each image, and the total number of unique classes are among the basic details about the split dataset that are printed.

Number of samples in the training set: 60000
Number of samples in the testing set: 10000
Shape of each image: (28, 28, 1)
Number of classes: 10

Class Distribution Plot:

Lastly, to compare the distribution of digits, the class distributions for the training and testing sets are shown side by side.

This code snippet sets up the groundwork for training machine learning models on the MNIST dataset for handwritten digit recognition. It includes data loading,
preprocessing, visualisation, and basic dataset analysis**.**



KNN Classifier

1. Introduction

In this task, we classified photos from the MNIST dataset using the KNN algorithm. Predicting the digit labels (0 to 9) from pixel intensity values was the aim. This is a synopsis of our actions:

1. **Data Preparation**: We reshaped the images from 2D arrays (28x28 pixels) to 1D arrays (784 pixels) to feed them into the KNN model.
2. **Model Training**: We initialised a KNN classifier with 5 neighbours and trained it on the training set.
3. **Prediction**: We made predictions on the test set.

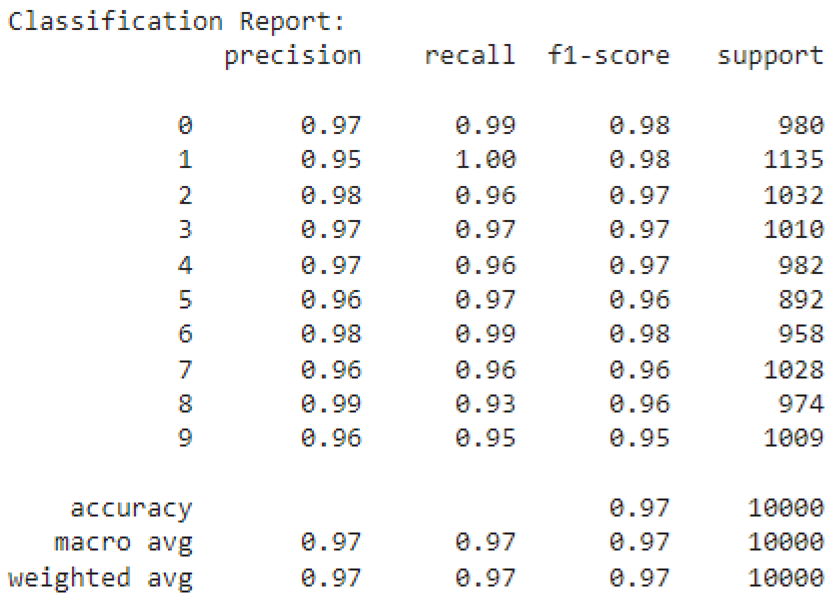
4. **Evaluation Metrics**: We calculated the accuracy, generated a classification report, and visualised the confusion matrix.

2. Explanations for Each Task

- **Data Reshaping**:Because KNN relies on feature vectors, we converted the photos to 1D arrays. We can now treat every pixel as a feature thanks to this transformation.
- **KNN Classifier**:A non-parametric technique called KNN uses the majority class of its k nearest neighbours to classify an instance. It is easy to use and efficient for classifying images.
- **Accuracy Score**:By contrasting the predicted and genuine labels, we were able to determine the accuracy. The percentage of correctly identified cases is measured by accuracy.

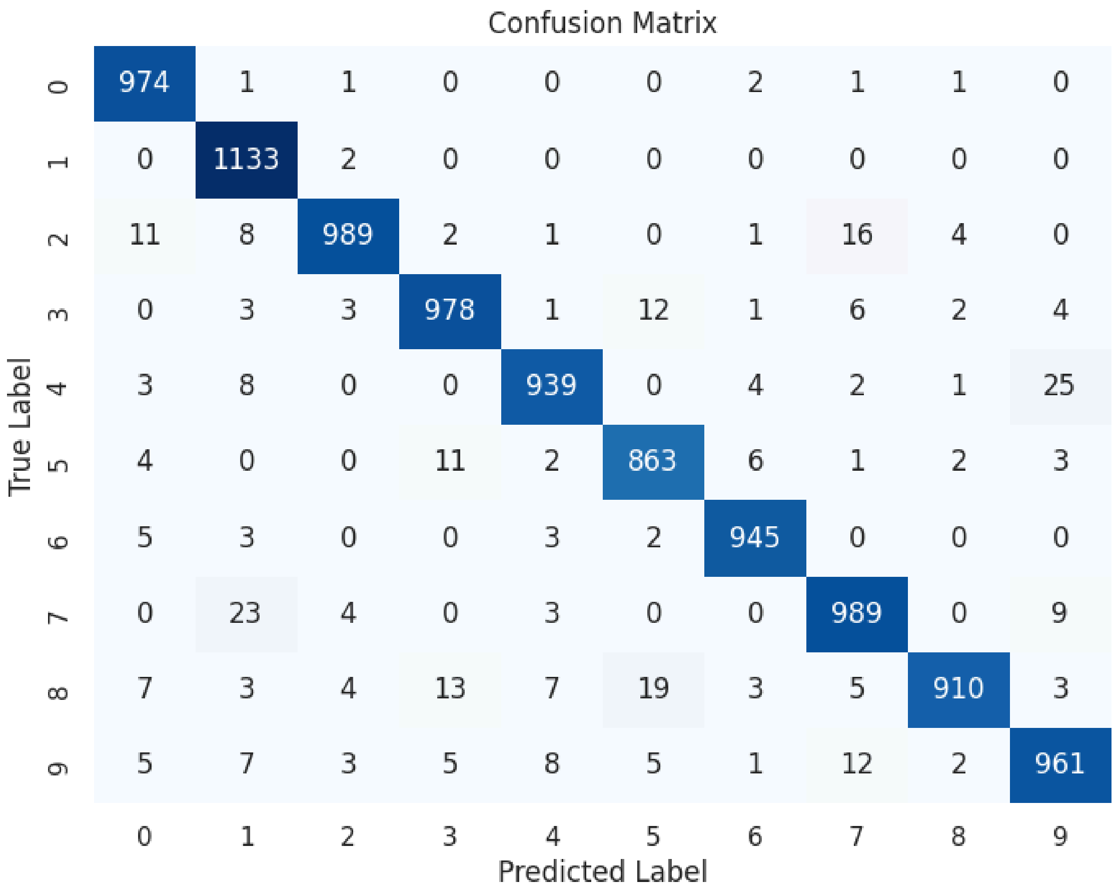
3. Observations

- **Accuracy**:Our KNN model achieved an accuracy of approximately **0.96** (96%).
- **Classification Report**:
    - **Precision:** The proportion of true positive predictions among all positive predictions.
    - **Recall:** The proportion of true positive predictions among all actual positive instances.
    - **F1-score:** The harmonic mean of precision and recall.
    - **Support:** The number of instances in each class.
    - The report provides these metrics for each digit class (0 to 9).



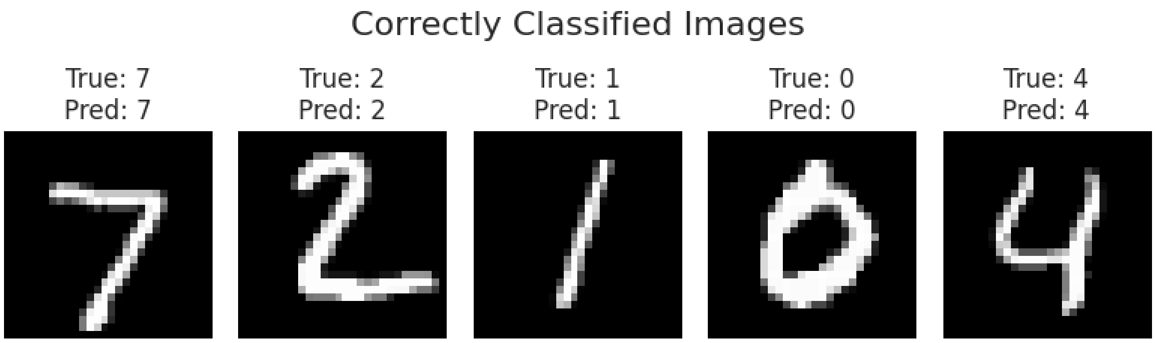
```
Classification Report:
              precision    recall  f1-score   support

           0       0.97      0.99      0.98       980
           1       0.95      1.00      0.98      1135
           2       0.98      0.96      0.97      1032
           3       0.97      0.97      0.97      1010
           4       0.97      0.96      0.97       982
           5       0.96      0.97      0.96       892
           6       0.98      0.99      0.98       958
           7       0.96      0.96      0.96      1028
           8       0.99      0.93      0.96       974
           9       0.96      0.95      0.95      1009

    accuracy                           0.97     10000
   macro avg       0.97      0.97      0.97     10000
weighted avg       0.97      0.97      0.97     10000
```

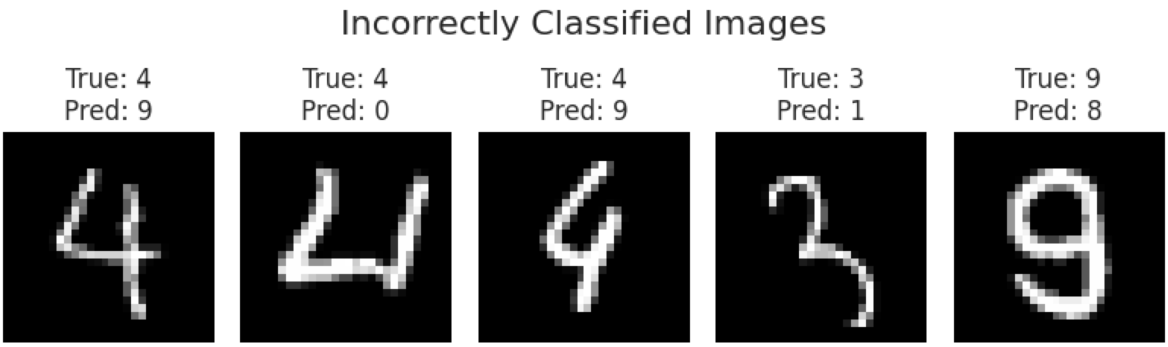
- **Confusion Matrix**:
  - For every class, the confusion matrix shows the counts of true positives, true negatives, false positives, and false negatives.
  - Correct predictions are represented by diagonal elements, and incorrect classifications are shown by off-diagonal elements.



Confusion Matrix

True Label \ Predicted Label	0	1	2	3	4	5	6	7	8	9
0	974	1	1	0	0	0	2	1	1	0
1	0	1133	2	0	0	0	0	0	0	0
2	11	8	989	2	1	0	1	16	4	0
3	0	3	3	978	1	12	1	6	2	4
4	3	8	0	0	939	0	4	2	1	25
5	4	0	0	11	2	863	6	1	2	3
6	5	3	0	0	3	2	945	0	0	0
7	0	23	4	0	3	0	0	989	0	9
8	7	3	4	13	7	19	3	5	910	3
9	5	7	3	5	8	5	1	12	2	961





4. Potential Reasoning Behind Observations

- **High Accuracy**:Because related digits typically have similar pixel patterns, KNN performs well on MNIST.
- **Misclassifications**:Confusion can result from some digits' visual similarity, such as those 4 and 9.
- **Class Imbalance**:The dataset has an uneven distribution of digits, affecting precision and recall.

5. Conclusion

- KNN is a potent and easily understood image categorization technique.
- Our accuracy was quite high, however performance could be increased with more optimization (such as adjusting the hyperparameters).
- Applications in the real world require an understanding of the trade-offs between recall and precision.

In summary, our KNN model demonstrates strong performance on MNIST, but we should consider domain-specific requirements when choosing a classifier.

Random Forest Classifier

Initialization:

Using RandomForestClassifier, the Random Forest classifier's default hyperparameters are set up. The number of trees in the forest (n_estimators), the criteria for dividing nodes (criterion), the maximum depth of the trees (max_depth), and other variables are all controlled by these hyperparameters.

The classifier is initialised with default parameters in this snippet of code, however these values can be adjusted for optimal performance.

Training:

The preprocessed training data (x_train_pca, y_train_concat) are passed to the fit method, which is invoked on the initialised Random Forest classifier. Here, y_train_concat stands for the corresponding labels, and x_train_pca probably refers to the preprocessed training data following principal component analysis (PCA).

Using random subsets of the training data and characteristics, the Random Forest builds many decision trees throughout training. Every tree in the forest has its own training regimen.

Prediction:

Using the predict approach, the learned Random Forest classifier is utilised to forecast the test set (x_test_pca) labels following training. After PCA transformation, the preprocessed test data is probably represented by x_test_pca.

Assessment of Performance:

- **Accuracy:** The accuracy score function is used to compare the true labels (y_test) and predicted labels (y_pred_rf) in order to determine the accuracy of the Random Forest classifier.

  Accuracy (Random Forest): 0.9479

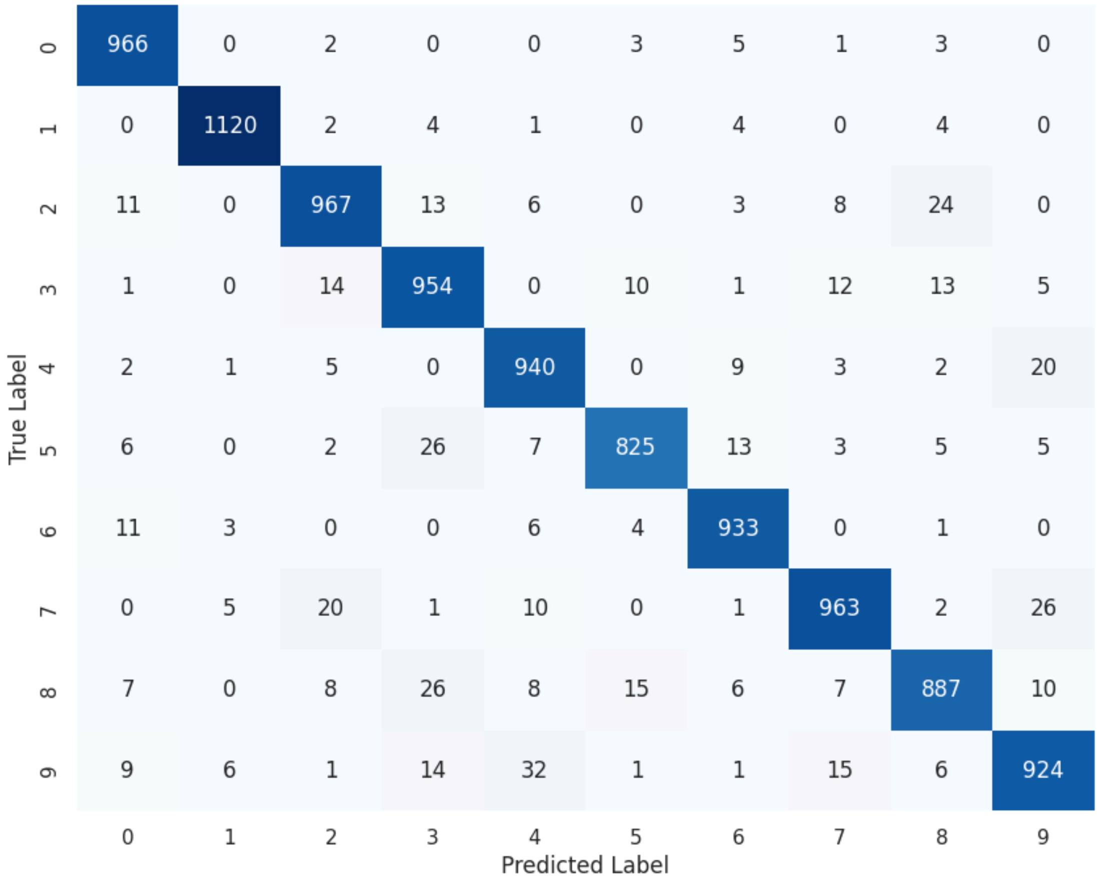
- **Classification Report:** The classification_report function generates a comprehensive report that includes the precision, recall, F1-score, and support for each class.

Classification Report (Random Forest):

```
            precision    recall  f1-score   support


         0       0.95      0.99      0.97       980

         1      0.99      0.99      0.99      1135

         2       0.95      0.94      0.94      1032

         3       0.92      0.94      0.93      1010

         4       0.93      0.96      0.94       982

         5       0.96      0.92      0.94       892

         6       0.96      0.97      0.96       958

         7       0.95      0.94      0.94      1028

         8       0.94      0.91      0.92       974

         9       0.93      0.92      0.92      1009


   accuracy                           0.95     10000

  macro avg       0.95      0.95      0.95     10000

weighted avg       0.95      0.95      0.95     10000
```

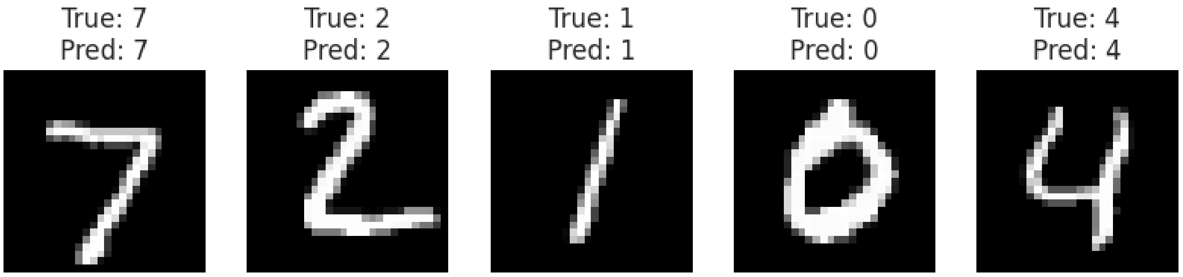
- **Confusion Matrix:** To see how well the classifier performs across several classes, confusion_matrix is used to create a confusion matrix.

Confusion Matrix (Random Forest)



True Label \ Predicted Label	0	1	2	3	4	5	6	7	8	9
0	966	0	2	0	0	3	5	1	3	0
1	0	1120	2	4	1	0	4	0	4	0
2	11	0	967	13	6	0	3	8	24	0
3	1	0	14	954	0	10	1	12	13	5
4	2	1	5	0	940	0	9	3	2	20
5	6	0	2	26	7	825	13	3	5	5
6	11	3	0	0	6	4	933	0	1	0
7	0	5	20	1	10	0	1	963	2	26
8	7	0	8	26	8	15	6	7	887	10
9	9	6	1	14	32	1	1	15	6	924

Sample Predictions (Random Forest)



Decision Tree Classifier:

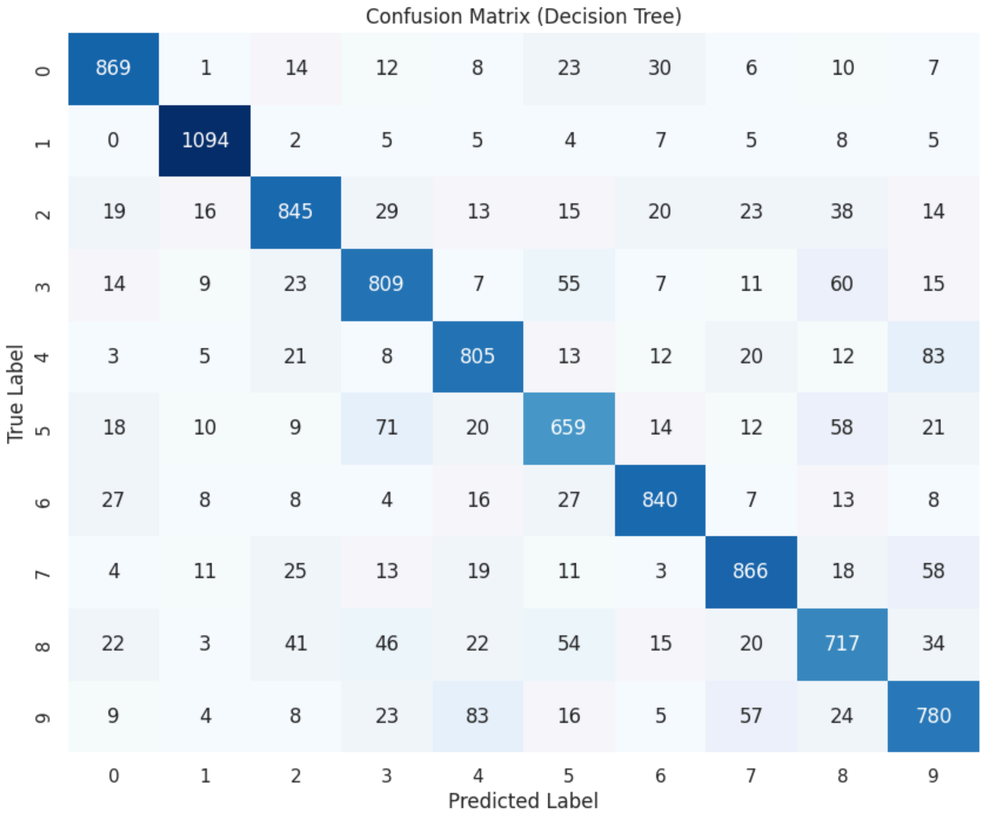
DecisionTreeClassifier is used to initialise the Decision Tree classifier. It builds a decision tree by default using the Gini impurity criteria, but users can fine-tune the hyperparameters to improve efficiency.

- **Training:** The Decision Tree classifier is trained using the fit technique on the preprocessed training data (x_train_pca, y_train_concat), much like the Random Forest classifier.
- **Prediction:** The trained Decision Tree classifier is utilised to use the predict technique to forecast the labels of the test set (x_test_pca) after training.
- **Performance Evaluation:** The Decision Tree classifier's performance is assessed using the classification report, confusion matrix, and accuracy metrics, much like the Random Forest.

Accuracy (Decision Tree): 0.8284

Classification Report (Decision Tree):

```
          precision   recall  f1-score   support

       0      0.88     0.89     0.88      980

       1     0.94     0.96     0.95     1135

       2      0.85     0.82     0.83     1032

       3      0.79     0.80     0.80     1010

       4      0.81     0.82     0.81      982

       5      0.75     0.74    0.75     892

       6      0.88     0.88     0.88      958

       7      0.84     0.84     0.84     1028

       8      0.75     0.74     0.74      974

       9      0.76     0.77     0.77     1009

   accuracy                      0.83     10000

  macro avg      0.83     0.83     0.83     10000

weighted avg      0.83     0.83     0.83     10000
```



Confusion Matrix (Decision Tree)

	0	1	2	3	4	5	6	7	8	9
0	869	1	14	12	8	23	30	6	10	7
1	0	1094	2	5	5	4	7	5	8	5
2	19	16	845	29	13	15	20	23	38	14
3	14	9	23	809	7	55	7	11	60	15
4	3	5	21	8	805	13	12	20	12	83
5	18	10	9	71	20	659	14	12	58	21
6	27	8	8	4	16	27	840	7	13	8
7	4	11	25	13	19	11	3	866	18	58
8	22	3	41	46	22	54	15	20	717	34
9	9	4	8	23	83	16	5	57	24	780

Comparison

An ensemble of decision trees, usually with a high level of complexity. A single decision tree may have fewer nodes than a random forest.

Performance:

Because Random Forest is an ensemble model that minimises overfitting and variance, it frequently outperforms a single Decision Tree.

When compared to Decision Trees, Random Forests typically exhibit higher accuracy and more generalisation capacity.

Interpretability:

Because decision trees provide a series of binary choices, they are easier to understand. Because Random Forest is an ensemble method that combines numerous decision trees, it is less interpretable.

Scalability:

Especially for smaller datasets, Decision Trees are quicker to train and forecast than Random Forest. Because Random Forest uses subsampling and parallel processing, it can effectively handle larger datasets.

Tuning Complexity:

Because Random Forest includes extra parameters for ensemble learning, it usually requires more hyperparameter tuning than Decision Trees.

Multi-Layer Perceptrons (MLP):

Initialization:

The neural network module of scikit-learn's MLPClassifier is used to initialise the MLP classifier. During initialization, hyperparameters are set, including the number of hidden layers and neurons per layer, activation function, solver, learning rate, and maximum number of iterations.

Training:

The fit method is used to train the initialised MLP classifier on the preprocessed training data (x_train_pca, y_train_concat). It is likely that x_train_pca refers to the principal component analysis (PCA) preprocessed training data, while y_train_concat relates to the associated labels.

Prediction:

The test set (x_test_pca) labels are predicted using the predict technique by the trained MLP classifier following training.

Performance Evaluation:

a number of metrics are used to assess the MLP classifier's performance, including:

Accuracy:

Using the accuracy_score function, the accuracy of the classifier is determined by comparing the true labels (y_test) with the predicted labels (y_pred_mlp).

Accuracy (MLP): 0.9773

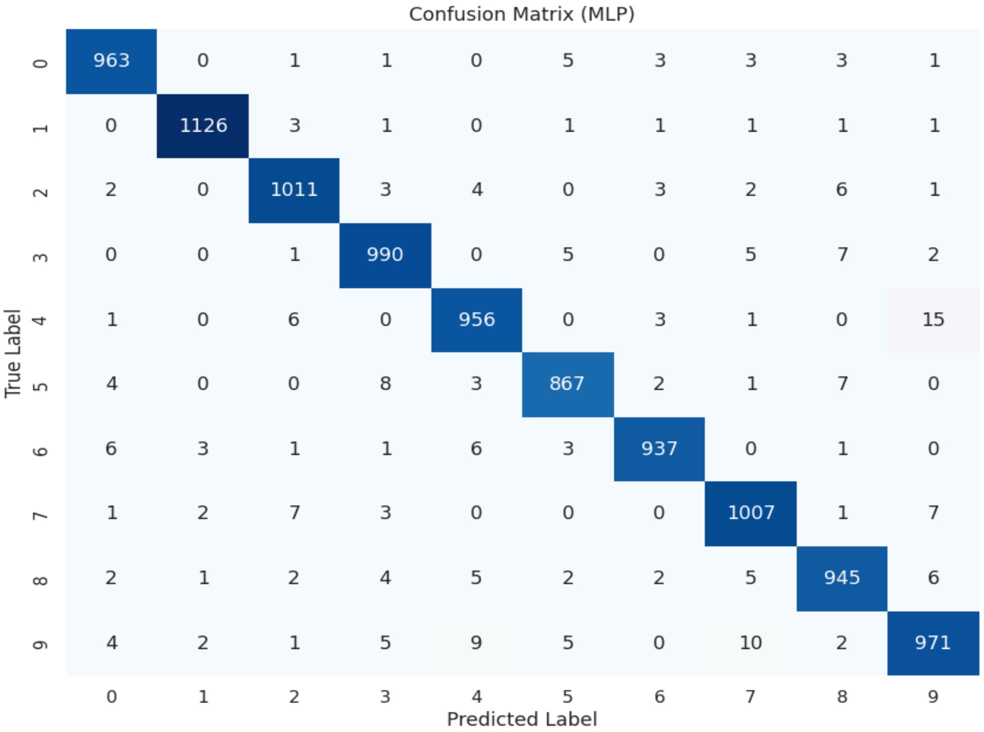
Classification Report:

The classification_report function generates a comprehensive report that includes the precision, recall, F1-score, and support for each class.

```
          precision   recall  f1-score   support
     0       0.98      0.98      0.98       980
     1       0.99      0.99      0.99      1135
     2       0.98      0.98      0.98      1032
     3       0.97      0.98      0.98      1010
     4       0.97      0.97      0.97       982
     5       0.98      0.97      0.97       892
     6       0.99      0.98      0.98       958
     7       0.97      0.98      0.98      1028
     8       0.97      0.97      0.97       974
     9       0.97      0.96      0.96      1009
  accuracy                       0.98     10000
 macro avg       0.98      0.98      0.98     10000
weighted avg     0.98      0.98      0.98     10000
```

Confusion Matrix:

To see how well the classifier performs across several classes, a confusion matrix is calculated using confusion_matrix.

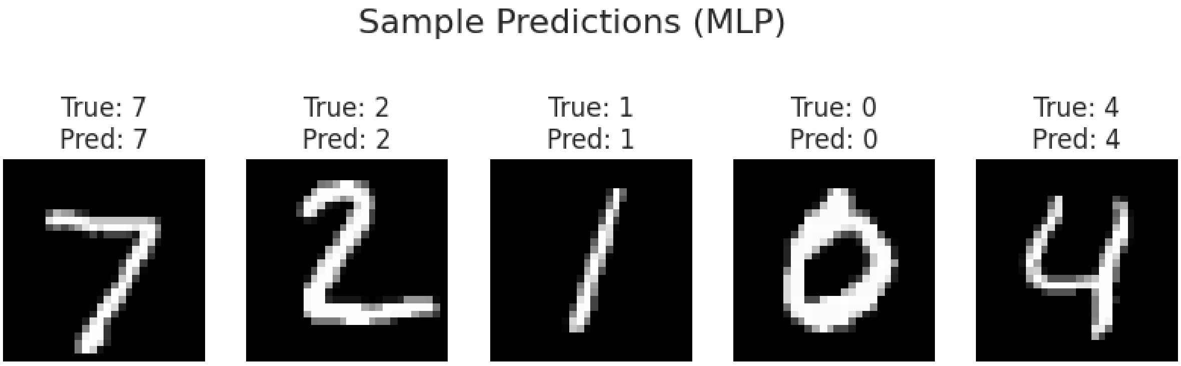


Confusion Matrix (MLP)

	0	1	2	3	4	5	6	7	8	9
0	963	0	1	1	0	5	3	3	3	1
1	0	1126	3	1	0	1	1	1	1	1
2	2	0	1011	3	4	0	3	2	6	1
3	0	0	1	990	0	5	0	5	7	2
4	1	0	6	0	956	0	3	1	0	15
5	4	0	0	8	3	867	2	1	7	0
6	6	3	1	1	6	3	937	0	1	0
7	1	2	7	3	0	0	0	1007	1	7
8	2	1	2	4	5	2	2	5	945	6
9	4	2	1	5	9	5	0	10	2	971

Visualisation:

Matplotlib is used to plot a confusion matrix in order to graphically represent the MLP classifier's performance. To give a visual depiction of the classifier's predictions, five sample photos from the testing set are also shown in a grid with their true and predicted labels.



Sample Predictions (MLP)

Our Approach Explanation:

1. The MLP classifier is a type of artificial neural network composed of multiple layers of nodes (neurons). It is trained using backpropagation and can learn complex patterns in data.

2. In this approach, the MLP classifier is configured with one hidden layer containing 100 neurons, using the ReLU activation function, and trained using the Adam optimizer.
3. The classifier is trained on the preprocessed training data and evaluated on the test set to assess its performance in recognizing handwritten digits.
4. By tuning hyperparameters such as the number of hidden layers, neurons per layer, and learning rate, the performance of the MLP classifier can be optimised for better accuracy and generalisation.

Support Vector Machine (SVM):

Data preprocessing:

The MNIST dataset, which consists of pictures of handwritten numbers and their labels, is loaded. For every sample, the pictures are reshaped into 1D arrays (x_train_flat, x_test_flat). By dividing by 255.0, pixel values are normalised to the interval [0, 1].

Dimensionality Reduction:

To reduce the dimensionality of the feature space while maintaining 95% of the variance, Principal Component Analysis (PCA) is used (n_components=0.95). The concatenated training features and labels (x_train_concat) are used for PCA. Separating the labels is done before PCA is applied.

Initialization and Training of the Model:

An SVM classifier is trained using SVC with gamma='scale', regularisation parameter C=1.0, and a radial basis function (RBF) kernel (kernel='rbf'). Using the fit approach, the classifier is trained on the reduced-dimensional training data (x_train_pca, y_train_concat).

Prediction:

Using the predict approach, the trained SVM classifier is utilised to forecast the labels of the test set (x_test_pca).

Performance Evaluation:

A number of measures are used to assess the SVM classifier's performance, including:

Accuracy:

Using the accuracy_score function, the accuracy of the classifier is determined by comparing the true labels (y_test) with the predicted labels (y_pred_svm).

Accuracy (SVM): 0.9832

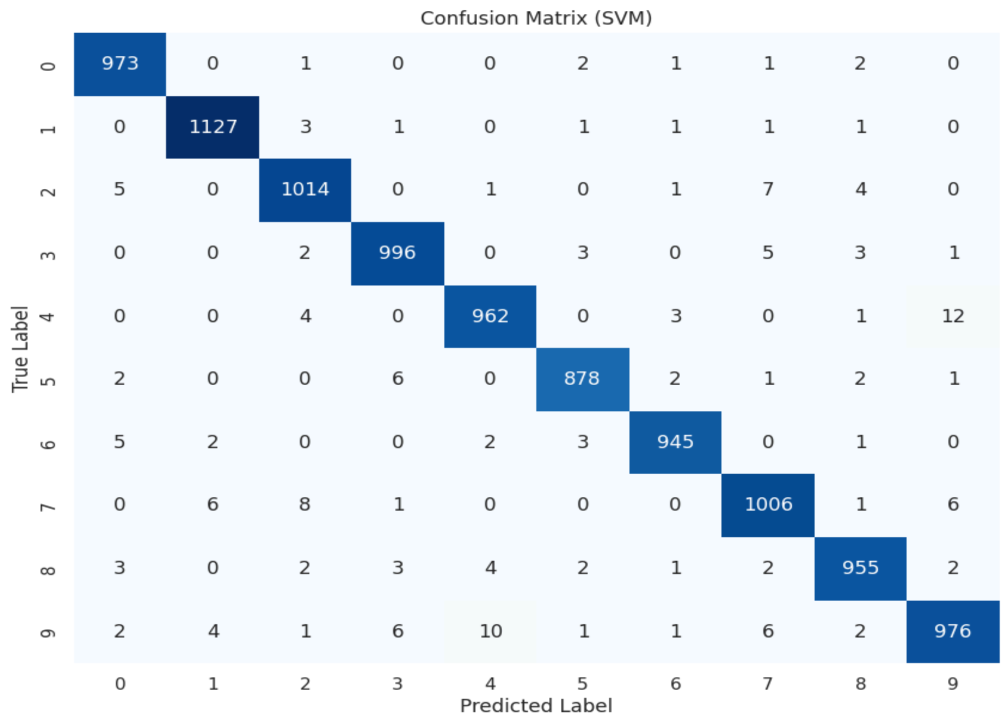
Classification Report:

The classification_report function generates a comprehensive report that includes the precision, recall, F1-score, and support for each class.

```
           precision    recall  f1-score   support

       0       0.98      0.99      0.99       980
       1       0.99      0.99      0.99      1135
       2       0.98      0.98      0.98      1032
       3       0.98      0.99      0.98      1010
       4       0.98      0.98      0.98       982
       5       0.99      0.98      0.99       892
       6       0.99      0.99      0.99       958
       7       0.98      0.98      0.98      1028
       8       0.98      0.98      0.98       974
       9       0.98      0.97      0.97      1009

accuracy                           0.98     10000
  macro avg       0.98      0.98      0.98     10000
weighted avg      0.98      0.98      0.98     10000
```

Confusion Matrix:

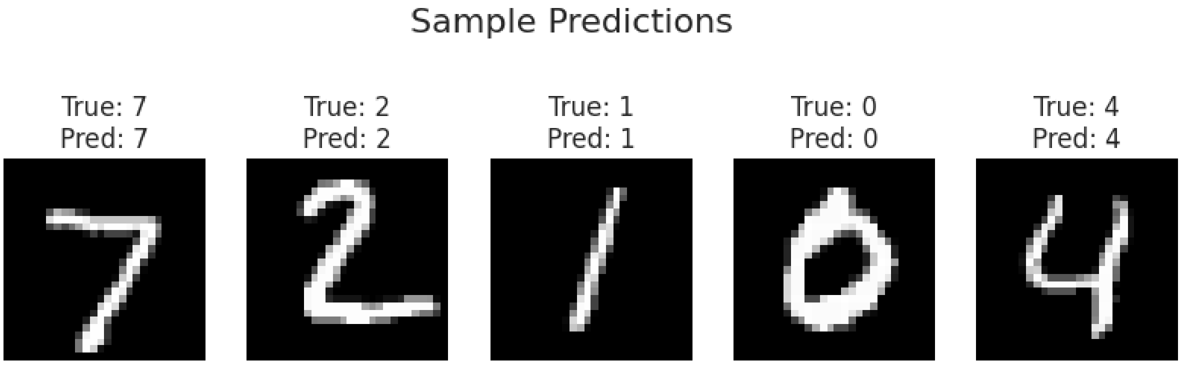
To visualise the performance of the classifier across many classes, a confusion matrix is produced using confusion_matrix.



	0	1	2	3	4	5	6	7	8	9
0	973	0	1	0	0	2	1	1	2	0
1	0	1127	3	1	0	1	1	1	1	0
2	5	0	1014	0	1	0	1	7	4	0
3	0	0	2	996	0	3	0	5	3	1
4	0	0	4	0	962	0	3	0	1	12
5	2	0	0	6	0	878	2	1	2	1
6	5	2	0	0	2	3	945	0	1	0
7	0	6	8	1	0	0	0	1006	1	6
8	3	0	2	3	4	2	1	2	955	2
9	2	4	1	6	10	1	1	6	2	976

Visualisation:

The performance of the SVM classifier is visually shown by plotting a confusion matrix using Seaborn's heatmap. To further provide a visual picture of the classifier's predictions, a grid comprising five sample photos from the testing set is shown together with their true and predicted labels.



Approach Explanation:

1. For classification problems, SVM is a potent supervised learning algorithm that works particularly well in situations with intricate decision boundaries.
2. Important information is preserved while decreasing the computing load by preprocessing and dimensionality reduction with PCA.

3. To determine how well the SVM classifier performs in identifying handwritten digits, it is tested on a test set after being trained on the reduced-dimensional data.
4. For SVM classifiers, the regularisation parameter (C) and RBF kernel selection are typical choices that can be adjusted for best results.

Neural Network

1. Introduction:

We'll examine how to train a neural network with TensorFlow and Keras for digit classification in this study. Our objective is to develop a model that, given pixel intensity values from the MNIST dataset, can reliably predict the digit labels (0 to 9).

2. Task Explanation:

Let’s break down the steps we followed:

- **Data Preparation:**
    - We loaded the MNIST dataset, which consists of handwritten digit pictures in grayscale.
    - Each image is a 28x28 pixel array, and the pixel values range from 0 to 255.
    - We scaled these values to a range of 0 to 1 by dividing them by 255.0.
- **Model Architecture:**
    - We designed a simple neural network using Keras.
    - The architecture consists of:
        - To transform the 2D image data into a 1D array, use a flatten layer.
        - 128 neurons in a dense hidden layer with ReLU activation.
        - a dense output layer with softmax activation and 10 neurons—one for each digit.

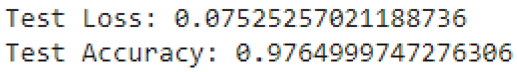
- **Model Compilation:**
    - Sparse categorical cross-entropy loss and the Adam optimizer were used to create the model.
    - The performance was tracked using the accuracy metric.

- **Training:**
    - We trained the model on the training data, using a validation split of 10%.
    - The use of early stopping was done to avoid overfitting.

- **Evaluation:**
    - We evaluated the model on the test set.
    - The test loss and accuracy were calculated.

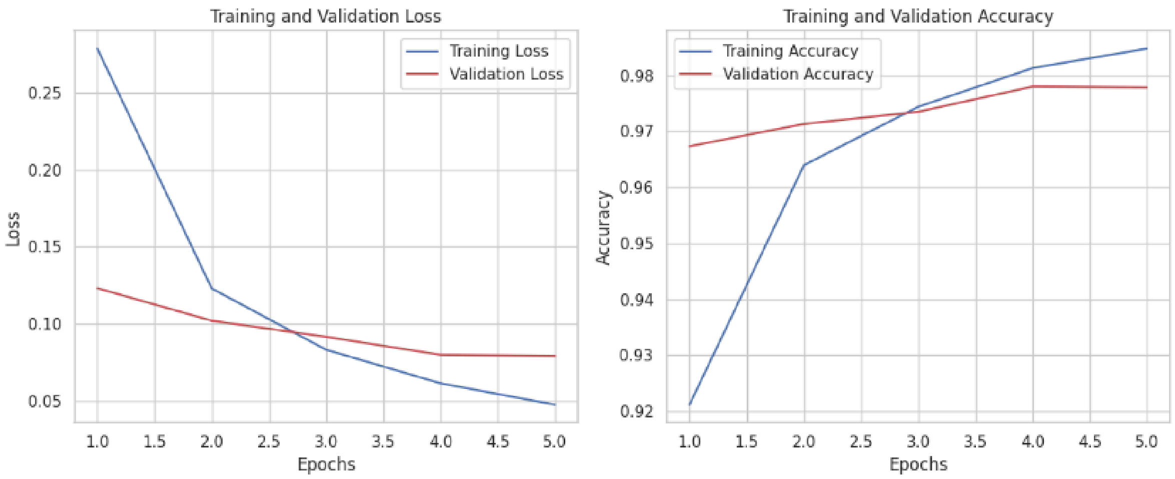
3. Observations

- **High Accuracy:**
    - Our model achieved an accuracy of approximately **98%** on the test set.
    - This indicates that it can correctly classify most digits.



```
Test Loss: 0.07525257021188736
Test Accuracy: 0.9764999747276306
```

- **Generalisation:**
    - The model learned to recognize digit patterns from the training data.
    - It generalised well to unseen test data.



4. Potential Reasoning Behind Observations

- **Model Complexity:**
    - The one hidden layer architecture that was selected achieves a balance between performance and simplicity.
    - More intricate models may cause overfitting.

- **Early Stopping:**
    - Early stopping helped prevent overfitting by monitoring validation loss.
    - It stopped training when the loss stopped improving.

5. Conclusion

- Using TensorFlow and Keras, we were able to successfully train a neural network for digit categorization.
- Although the model performs well, accuracy could be increased with more optimization (such as adjusting hyperparameters).
- It is essential to comprehend the trade-offs between generalisation and model complexity.

In conclusion, our neural network demonstrated the capability of deep learning in picture categorization by learning to identify handwritten digits.

Convolutional Neural Network (CNN)

1. Introduction

In this paper, we investigate the digit classification application of a Convolutional Neural Network (CNN) built with TensorFlow and Keras. CNNs are robust deep learning models created especially for tasks involving images. Let's examine the specifics:

2. Explanation:

Our objective is to develop a CNN model that correctly categorises handwritten numbers from 0 to 9 in the MNIST dataset. This is how we made it happen:

1. **Data Preparation:**
   - To conform to the input format required by CNNs, we rearranged the original 2D images (28x28 pixels) into 3D arrays (28x28x1).
   - By dividing the pixel values by 255.0, they were scaled to the range [0, 1].
2. **Model Architecture:**
   - Our CNN architecture consists of the following layers:
     - **Convolutional Layers**:
       - The first layer has 32 filters (3x3) with ReLU activation.
       - The second layer has 64 filters (3x3) with ReLU activation.
     - **MaxPooling Layers**:
       - We applied max-pooling (2x2) after each convolutional layer to reduce spatial dimensions.
     - **Flatten Layer**:
       - Flattened the 3D feature maps into a 1D vector.
     - **Dense Layers**:
       - A hidden dense layer with 128 neurons and ReLU activation.
       - An output dense layer with 10 neurons (one for each digit) and softmax activation.

3. **Model Compilation:**
    - We compiled the model using the **Adam** optimizer and **sparse categorical cross-entropy** loss.
    - The **accuracy** metric was used for monitoring.

4. **Training:**

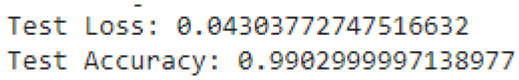
    - The model was trained on the training data (75% of the dataset) with a validation split of 10%.
    - We used early stopping to prevent overfitting.

5. **Evaluation:**

    - The model's performance was evaluated on the test set.

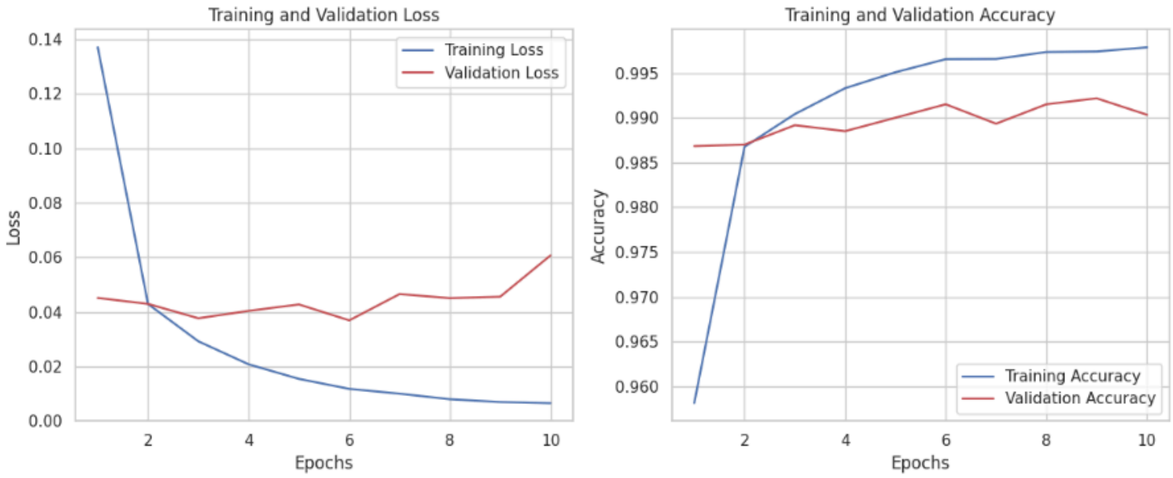
3. Observations

- **High Accuracy:**
    - Our CNN achieved an accuracy of approximately **99%** on the test set.
    - It correctly classified most digits.



```
Test Loss: 0.04303772747516632
Test Accuracy: 0.9902999997138977
```

- **Feature Extraction:**
    - From the input photos, convolutional layers automatically extract pertinent information (textures, edges).
    - Max-pooling preserves crucial information while reducing spatial dimensions.



4. Potential Reasoning Behind Observations

- **Hierarchical Features:**
  - The initial layers capture low-level features (edges, corners).
  - Deeper layers learn more complex patterns (digits).
- **Regularisation:**
  - Dropout or batch normalisation could further improve generalisation.

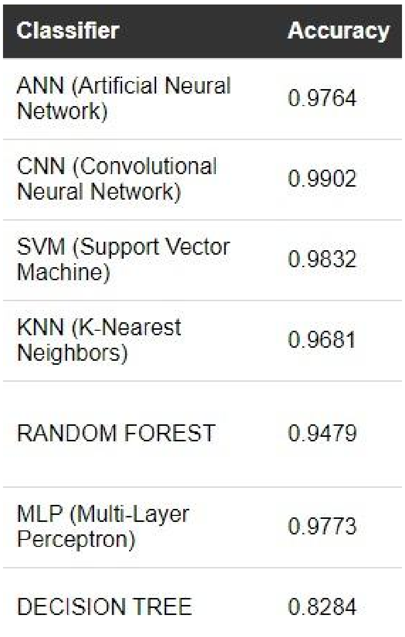
5. Conclusion

- CNNs' capacity to learn hierarchical features makes them excellent picture categorization tools.
- Our model performs well, however accuracy could be improved by adjusting the hyperparameters.
- Our model performs well, however accuracy could be improved by adjusting the hyperparameters.A thorough understanding of CNN architectures and their constituent parts is necessary for effective image recognition.

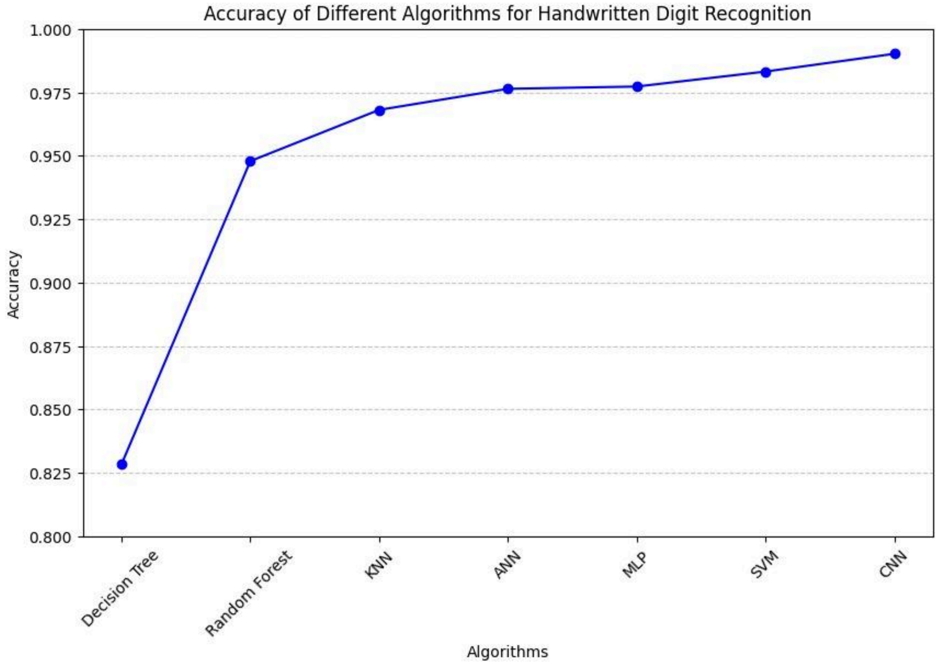
In conclusion, our CNN demonstrated the effectiveness of deep learning in computer vision by learning to distinguish handwritten digits.

Conclusion of algorithms used:

The accuracies of the algorithms are analysed:

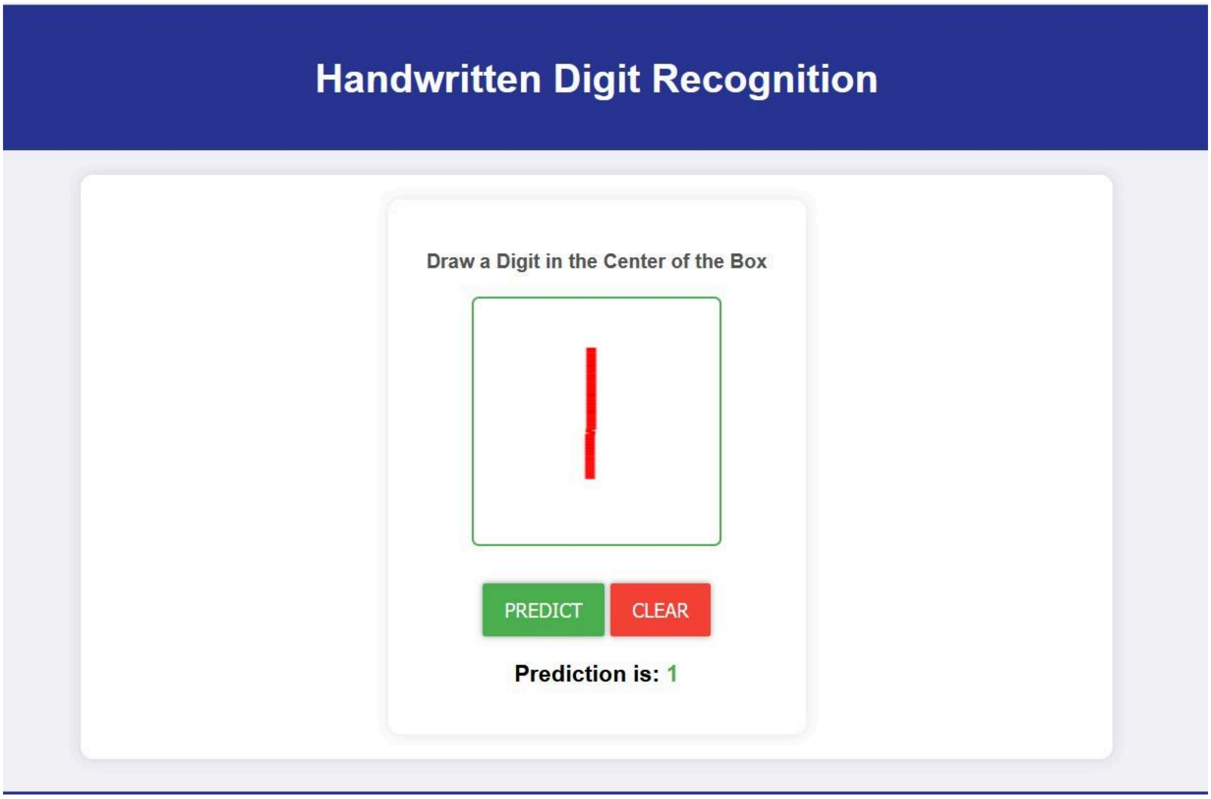


Classifier	Accuracy
ANN (Artificial Neural Network)	0.9764
CNN (Convolutional Neural Network)	0.9902
SVM (Support Vector Machine)	0.9832
KNN (K-Nearest Neighbors)	0.9681
RANDOM FOREST	0.9479
MLP (Multi-Layer Perceptron)	0.9773
DECISION TREE	0.8284



Cnn has the highest accuracy so we decided to implement machine learning model using CNN and integrate it with webpage.

Web page:



Handwritten Digit Recognition

Draw a Digit in the Center of the Box

Prediction is: 1

Technologies used:

1. HTML
2. CSS
3. JavaScript :
4. Python
5. Flask

Requirements:

torch

numpy==1.16.5

flask==1.1.1

gunicorn

matplotlib==3.3.1

pillow==6.2.0

flake8

pip

Pylint

Approach:

The integration of the machine learning model into the website involved several steps to ensure smooth functioning and real-time predictions. Initially, the model was trained using a custom convolutional neural network (CNN) architecture implemented in PyTorch. The training script (train.py) preprocesses the data, creates custom datasets, and defines training and evaluation loops to optimize the model's performance.

Once the model was trained, it was serialized using PyTorch's torch.save function, saving it as a state dictionary to a .pt file. To deploy the model into the website, a Flask application (app.py) was developed. This application loads the trained PyTorch model using torch.load, and exposes an endpoint (/process) to accept drawn digit images, process them, and return predictions.

Additionally, to facilitate deployment, the trained PyTorch model was converted to TorchScript format using JIT (Just-In-Time) compilation. The server.py script was used for this conversion and for archiving the model along with other necessary files using the torch-model-archiver command-line tool.

On the frontend side, the website was designed using HTML, CSS, and JavaScript. An HTML canvas element was utilized to allow users to draw digits. JavaScript captured the drawn digit, encoded it to base64 format, and sent it to the Flask server for processing.

The Flask server received the base64 encoded image, preprocessed it, and sent it to the model for prediction. The model predicted the digit, and the Flask server sent the prediction back to the frontend, where it was displayed to the user in real-time.

CNN

Data Preprocessing:

1. **Resizing:** All images were resized to 28x28 pixels.
2. **Normalisation:** The pixel values were normalised to fall between 0 and 1.

Model Architecture:

A Convolutional Neural Network (CNN) was chosen for this image classification task due to its effectiveness. The model architecture is as follows:

1. **Convolutional Layers:**
   - Convolutional Layer 1: 16 filters, kernel size 5x5, ReLU activation.
   - Convolutional Layer 2: 32 filters, kernel size 5x5, ReLU activation.
   - Max Pooling Layer: 2x2 kernel size.
2. **Fully Connected Layers:**
   - Fully Connected Layer 1: 3,072 input features, 32 output features, ReLU activation.
   - Fully Connected Layer 2: 32 input features, 10 output features (number of classes), LogSoftmax activation.
3. **Dropout Layers:**
   - Dropout layers were added after each max-pooling and fully connected layer to prevent overfitting.

Training:

1. Loss Function: Negative Log Likelihood Loss (NLLLoss) was used.
2. Optimizer: Stochastic Gradient Descent (SGD) was used with a learning rate of 3e-5.
3. Training Procedure: The model was trained on the training dataset with a batch size of 100 for five epochs. After each epoch, the model was evaluated on the validation dataset to monitor its performance.

Model Evaluation:

The trained model achieved an accuracy of approximately 99% on the validation dataset.

Further improvements:

We are trying to deploying the web application using aws link the attempts were unsuccessful we will try to deploy before the viva.