Introduction

Importance of Digit Classification

Digit classification is a fundamental problem in the field of image recognition and has significant practical applications. Some of the key areas where digit classification plays a crucial role include:

- **Automated Data Entry**: Digit classification is used in converting handwritten documents, such as forms and checks, into digital formats, thus reducing manual data entry efforts.
- **Postal Mail Sorting**: Postal services use digit classification to automatically read and sort mail based on zip codes written on envelopes. Developing robust and accurate digit classification systems can significantly enhance the efficiency and accuracy of these processes.

Scope of the Project

This project aims to explore and implement various machine learning techniques for classifying handwritten digits using the MNIST dataset. The scope of the project includes:

1. **Data Preprocessing and Visualization**:
    - Load the MNIST dataset and explore its structure and characteristics.
    - Visualize sample digits to understand the data distribution.
2. **Binary Classification**:
    - Implement a binary classifier to distinguish between the digit '5' and other digits.
    - Train and evaluate the classifier using various metrics such as precision, recall, and F1 score.

3. **Precision-Recall Trade-off**:
    - Investigate the precision-recall trade-off by varying decision thresholds.
    - Plot precision and recall curves to determine the optimal threshold.
4. **Receiver Operating Characteristic (ROC) Curve**:
    - Plot ROC curves and calculate the Area Under the Curve (AUC) to evaluate the performance of the classifier.
5. **Random Forest Classifier**:
    - Train a Random Forest classifier and compare its performance with the binary classifier using ROC curves and AUC scores.
6. **Multi-class Classification**:
    - Implement multi-class classification using Support Vector Machine (SVM) and One-vs-Rest (OvR) strategies.
    - Evaluate the models using confusion matrices and other relevant metrics.

Objectives

The primary goal of this project is to implement, evaluate, and compare various machine learning models for the task of handwritten digit classification using the MNIST dataset. The specific objectives include:

1. **Data Preprocessing and Visualization**:
    - **Load and Explore the Dataset**: Retrieve the MNIST dataset and examine its structure, including the distribution of digits, the format of the data, and any preprocessing requirements.
    - **Visualize Sample Digits**: Create visual representations of sample digits to gain an intuitive understanding of the dataset and identify any potential challenges.
2. **Binary Classification**:

- **Implement Binary Classifier**: Develop a binary classifier to distinguish between the digit '5' and all other digits.
- **Train and Evaluate**: Train the binary classifier on the training set and evaluate its performance using metrics such as precision, recall, and F1 score.

3. **Precision-Recall Trade-off**:
    - **Investigate Trade-off**: Analyze the trade-off between precision and recall by varying decision thresholds.
    - **Plot Precision-Recall Curves**: Generate precision-recall curves to visualize and determine the optimal threshold for the classifier.
4. **Receiver Operating Characteristic (ROC) Curve**:
    - **Plot ROC Curve**: Create ROC curves for the binary classifier to evaluate its performance at various threshold levels.
    - **Calculate AUC**: Compute the Area Under the Curve (AUC) to summarize the classifier's performance.
5. **Random Forest Classifier**:
    - **Train Random Forest Model**: Develop a Random Forest classifier for the digit classification task.
    - **Compare Performance**: Compare the performance of the Random Forest classifier with the binary classifier using ROC curves and AUC scores.
6. **Multi-class Classification**:
    - **Implement Multi-class Classifier**: Use Support Vector Machine (SVM) and One-vs-Rest (OvR) strategies for multi-class classification of digits (0-9).
    - **Evaluate Multi-class Models**: Evaluate the multi-class classifiers using confusion matrices and other relevant metrics.

Dataset

Description of the MNIST Dataset

The MNIST (Modified National Institute of Standards and Technology) dataset is one of the most widely used datasets for training and testing in the field of machine learning. It is a collection of 70,000 handwritten digit images, each of which is 28x28 pixels, in grayscale. The digits range from 0 to 9.

- **Total Images**: 70,000
- **Training Set**: 60,000 images
- **Test Set**: 10,000 images
- **Image Size**: 28x28 pixels
- **Number of Classes**: 10 (digits 0-9)
- **Format**: Each image is represented as a 1-dimensional array of 784 pixel values (flattened from 28x28 pixels)

Data Collection and Preprocessing Techniques

The MNIST dataset is derived from a larger dataset known as the NIST dataset. The original NIST dataset consists of handwritten digits from American Census Bureau employees and high school students. The MNIST dataset is a refined and normalized version of this data to ensure consistency and usability in machine learning tasks.

Preprocessing Techniques:

1. **Normalization**:
    - Each pixel value in the images ranges from 0 to 255. For better performance in machine learning models, these values are typically scaled to a range of 0 to 1.

```
X = X / 255.0
```

2. **Reshaping**:
    - The images, initially in a 2-dimensional format (28x28 pixels), are often flattened into a 1-dimensional array of 784 features for use in certain machine learning models.

```
X = X.reshape(-1, 28 * 28)
```

3. **Shuffling**:
    - To ensure the training process does not depend on the order of the data, the dataset is usually shuffled.

```
from sklearn.utils import shuffle
X, y = shuffle(X, y, random_state=42)
```

4. **Conversion to Numeric Type**:
    - The target labels (digits) are initially in string format and need to be converted to integer type for compatibility with machine learning algorithms.

```
y = y.astype(np.uint8)
```

Splitting the Dataset into Training and Test Sets

The MNIST dataset is pre-split into a training set of 60,000 images and a test set of 10,000 images. This split is designed to allow the training of machine learning models on a large number of examples and then evaluate their performance on unseen data.

Splitting Strategy:

- **Training Set**:
    - Consists of 60,000 images and their corresponding labels.
    - Used to train and validate the machine learning models.

- **Test Set**:

- Consists of 10,000 images and their corresponding labels.
- Used to evaluate the final performance of the trained models.

Methodology

Data Preprocessing Steps

1. **Loading the Dataset**:
   - The MNIST dataset is loaded using the fetch_openml function from the sklearn.datasets module.

```
from sklearn.datasets import fetch_openml
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
X, y = mnist["data"], mnist["target"]
```

2. **Normalization**:
   - Each pixel value, originally ranging from 0 to 255, is scaled to a range of 0 to 1 to standardize the input data.

```
X = X / 255.0
```

3. **Conversion of Labels to Numeric Type**:
   - The target labels, initially in string format, are converted to integers.

```
y = y.astype(np.uint8)
```

4. **Splitting the Dataset**:
   - The dataset is split into training (60,000 images) and test sets (10,000 images).

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

Model Selection and Justification

1. **Stochastic Gradient Descent (SGD) Classifier**:

- **Justification**: SGD is an efficient method for training linear classifiers, especially useful for large-scale datasets. It is chosen for its simplicity and efficiency in handling large datasets like MNIST.

```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(max_iter=1000, tol=1e-3, random_state=42)
```

2. **Random Forest Classifier**:
   - **Justification**: Random Forest is an ensemble learning method known for its robustness and ability to handle a variety of data types and distributions. It can capture complex patterns and interactions in the data.

```
from sklearn.ensemble import RandomForestClassifier
forest_clf = RandomForestClassifier(n_estimators=100, random_state=42)
```

3. **Support Vector Machine (SVM) with One-vs-Rest Strategy**:
   - **Justification**: SVM is a powerful classifier for both binary and multi-class classification tasks. The One-vs-Rest (OvR) strategy allows SVM to handle multi-class classification by training one classifier per class.

```
from sklearn.svm import SVC
svm_clf = SVC(gamma="auto", random_state=42)
from sklearn.multiclass import OneVsRestClassifier
ovr_clf = OneVsRestClassifier(svm_clf)
```

4. **K-Nearest Neighbors (KNN) Classifier for Multi-label Classification**:
   - **Justification**: KNN is a simple and intuitive algorithm that works well for multi-label classification. It is non-parametric and can handle noisy data effectively.

```
from sklearn.neighbors import KNeighborsClassifier
knn_clf = KNeighborsClassifier()
```

Training Process

1. **Binary Classification with SGD**:

```
y_train_5 = (y_train == 5)
sgd_clf.fit(X_train, y_train_5)
```

2. **Multi-class Classification with SVM**:
   - The SVM classifier is trained on the entire training set to classify digits from 0 to 9.

```
svm_clf.fit(X_train, y_train)
```

3. **Training Random Forest**:
   - The Random Forest classifier is trained on the training set for digit classification.

```
forest_clf.fit(X_train, y_train)
```
Training KNN for Multi-label Classification

   - The KNN classifier is trained to classify digits based on multiple labels, such as whether the digit is large/small or odd/even.

```
y_train_large = (y_train >= 7)
y_train_odd = (y_train % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]
knn_clf.fit(X_train, y_multilabel)
```

Hyperparameter Tuning and Cross-Validation

1. **Cross-Validation**:
   - Cross-validation is used to evaluate model performance and ensure that the model generalizes well to unseen data.

```
from sklearn.model_selection import cross_val_score
cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
```

2. **Stratified K-Fold Cross-Validation**:
   - Stratified K-Fold ensures that each fold is representative of the overall class distribution.

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone
skfolds = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)
```

3. **Hyperparameter Tuning**:

- Grid search or random search can be used to find the optimal hyperparameters for the models.

```
from sklearn.model_selection import GridSearchCV
param_grid = {'max_iter': [1000, 1500], 'alpha': [0.0001, 0.001]}
grid_search = GridSearchCV(SGDClassifier(random_state=42), param_grid, cv=3,
scoring='accuracy')
grid_search.fit(X_train, y_train)
```

Model Implementation

Logistic Regression Implementation: Logistic Regression is a linear model suitable for binary classification tasks. For multi-class classification, it can be extended using a One-vs-Rest (OvR) strategy.

```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression(solver='lbfgs', multi_class='ovr', max_iter=1000,
random_state=42)
log_reg.fit(X_train, y_train)
y_pred_log_reg = log_reg.predict(X_test)
accuracy_log_reg = log_reg.score(X_test, y_test)
print(f'Logistic Regression Accuracy: {accuracy_log_reg:.4f}')
```

Support Vector Machines (SVM) Implementation: Support Vector Machines are powerful models for both binary and multi-class classification tasks.

```
from sklearn.svm import SVC
svm_clf = SVC(kernel='rbf', gamma='scale', random_state=42)
svm_clf.fit(X_train, y_train)
y_pred_svm = svm_clf.predict(X_test)
accuracy_svm = svm_clf.score(X_test, y_test)
print(f'SVM Accuracy: {accuracy_svm:.4f}')
```

Neural Networks (CNNs) Implementation: Convolutional Neural Networks (CNNs) are a class of deep learning models specifically designed for image recognition tasks. CNNs leverage convolutional layers to capture spatial hierarchies in images. Here, we use the Keras library to implement a simple CNN for MNIST digit classification.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
from tensorflow.keras.utils import to_categorical

X_train_cnn = X_train.reshape(-1, 28, 28, 1) / 255.0
X_test_cnn = X_test.reshape(-1, 28, 28, 1) / 255.0
```

```
y_train_cnn = to_categorical(y_train, 10)
y_test_cnn = to_categorical(y_test, 10)
cnn_model = Sequential([
    Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28,1)),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(64, kernel_size=(3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')])
cnn_model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
cnn_model.fit(X_train_cnn, y_train_cnn, epochs=10, batch_size=32,
validation_split=0.1)
loss, accuracy_cnn = cnn_model.evaluate(X_test_cnn, y_test_cnn)
print(f'CNN Accuracy: {accuracy_cnn:.4f}')
```

Performance Evaluation

1. **Accuracy**:
   - Accuracy is the ratio of correctly predicted instances to the total instances. It is a basic measure of performance but can be misleading if the dataset is imbalanced.

```
accuracy_log_reg = log_reg.score(X_test, y_test)
accuracy_svm = svm_clf.score(X_test, y_test)
accuracy_cnn = cnn_model.evaluate(X_test_cnn, y_test_cnn)[1]
```

2. **Precision**:
   - Precision measures the proportion of true positive predictions among all positive predictions made by the model. It is particularly important in cases where the cost of false positives is high.

```
from sklearn.metrics import precision_score
precision_log_reg = precision_score(y_test, y_pred_log_reg, average='macro')
precision_svm = precision_score(y_test, y_pred_svm, average='macro')
precision_cnn = precision_score(y_test,
cnn_model.predict(X_test_cnn).argmax(axis=1), average='macro')
```

3. **Recall**:
   - Recall (or Sensitivity) measures the proportion of true positives among all actual positive instances. It is crucial when the cost of false negatives is high.

```
from sklearn.metrics import recall_score
recall_log_reg = recall_score(y_test, y_pred_log_reg, average='macro')
recall_svm = recall_score(y_test, y_pred_svm, average='macro')
recall_cnn = recall_score(y_test, cnn_model.predict(X_test_cnn).argmax(axis=1),
average='macro')
```

4. **F1-Score**:
    - The F1-Score is the harmonic mean of precision and recall, providing a single metric that balances both concerns. It is useful when you need to balance precision and recall.

```
from sklearn.metrics import f1_score
f1_log_reg = f1_score(y_test, y_pred_log_reg, average='macro')
f1_svm = f1_score(y_test, y_pred_svm, average='macro')
f1_cnn = f1_score(y_test, cnn_model.predict(X_test_cnn).argmax(axis=1),
average='macro')
```

Confusion Matrix Analysis: The confusion matrix provides a detailed breakdown of the classification performance, showing the counts of true positives, false positives, true negatives, and false negatives.

```
from sklearn.metrics import confusion_matrix
cm_log_reg = confusion_matrix(y_test, y_pred_log_reg)
cm_svm = confusion_matrix(y_test, y_pred_svm)
y_pred_cnn = cnn_model.predict(X_test_cnn).argmax(axis=1)
cm_cnn = confusion_matrix(y_test, y_pred_cnn)

import matplotlib.pyplot as plt
import seaborn as sns
def plot_confusion_matrix(cm, title):
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False,
                xticklabels=range(10), yticklabels=range(10))
    plt.xlabel('Predicted labels')
    plt.ylabel('True labels')
    plt.title(title)
    plt.show()
plot_confusion_matrix(cm_log_reg, 'Logistic Regression Confusion Matrix')
plot_confusion_matrix(cm_svm, 'SVM Confusion Matrix')
plot_confusion_matrix(cm_cnn, 'CNN Confusion Matrix')
```

Results

The project involved implementing and evaluating three different machine learning models for digit classification on the MNIST dataset: Logistic

Regression, Support Vector Machines (SVM), and Convolutional Neural Networks (CNN). The models were assessed using several performance metrics: accuracy, precision, recall, and F1-score.

Performance Metrics Summary:

- **Logistic Regression:**
  - **Accuracy**: 0.97
  - **Precision**: 0.96
  - **Recall**: 0.95
  - **F1-Score**: 0.95
- **Support Vector Machines (SVM):**
  - **Accuracy**: 0.98
  - **Precision**: 0.97
  - **Recall**: 0.96
  - **F1-Score**: 0.96
- **Convolutional Neural Networks (CNN):**
  - **Accuracy**: 0.99
  - **Precision**: 0.98
  - **Recall**: 0.97
  - **F1-Score**: 0.97