Homework 2, STAT 5241

Zongyi Liu

Mar 1, 2025

1 Questions

1.1 Question 1

To address the challenge of predicting autism in a European population using a model trained on US data, the team must carefully design their machine learning process to account for the significant differences in key features such as ethnicity, demographics, and environmental factors. The first step involves thorough data preprocessing to align and normalize features across the two datasets, ensuring consistency and handling missing values through imputation. Feature engineering should focus on creating universal patterns rather than region-specific ones. Given the distribution shift between the training and testing datasets, the team should employ domain adaptation techniques such as transfer learning, domain-adversarial neural networks (DANN), or covariate shift correction to improve generalization. Robust models like random forests, gradient boosting machines (e.g., XGBoost, LightGBM), or regularized linear models are recommended due to their ability to handle mixed data types and resist overfitting to region-specific patterns. Ensemble methods can further enhance robustness by combining multiple models.

Hyperparameter tuning should be conducted using k-fold cross-validation on the US dataset, with careful attention to ensuring the validation splits are representative. If a small subset of European data is available, it can be used for fine-tuning to improve generalization. Bayesian optimization or grid search can efficiently explore the hyperparameter space, focusing on key parameters such as regularization terms, learning rates, and model complexity. For evaluation, the primary metric should be the F1 score, which balances precision and recall and is particularly suitable for imbalanced datasets. Secondary metrics like AUC-ROC, precision, recall, and confusion matrices provide additional insights into model performance, while calibration metrics such as the Brier score ensure probabilistic predictions are reliable.

During testing, the model should be evaluated on the European dataset without further tuning, and performance metrics should be reported alongside cross-validation results from the US dataset. Error analysis is crucial to identify misclassified cases and understand whether errors stem from feature distribution differences or other factors. Additionally, the team must consider bias and fairness, ensuring the model does not disproportionately misclassify specific subgroups, and prioritize interpretability using methods like SHAP or LIME to align predictions with clinical knowledge. Continuous monitoring of the model's performance post-deployment is also recommended to detect any data drift over time. By integrating these steps, the team can develop a model that generalizes effectively to the European population while maintaining robustness, fairness, and transparency.

1.2 Question 2

1.2.1 Part A

We need to pre-process the data. There are columns with many missing values, which should be considered to drop first. Then we can drop rows with missing values. If we simply drop rows with missing data, or drop the rows with missing data before deleting columns with significant number of missing data, it will distorted the regression results.

The result I got using self-coded K-fold CV from scratch is as below:

```
Best RBF Kernel Parameters: lambda=0.01, gamma=0.1
Best Polynomial Kernel Parameters: lambda=0.01, degree=2
RBF Kernel Test MSE: 0.014702546829905011
Polynomial Kernel Test MSE: 0.009525269150252024
```

1.2.2 Part B

And comparing the results of scratch code and `sklearn` code, the hyperparameters are the same:

```
Custom RBF Kernel Parameters: lambda=0.01, gamma=0.1
Custom Polynomial Kernel Parameters: lambda=0.01, degree=2
scikit-learn RBF Kernel Parameters: {'alpha': 0.01, 'gamma': 0.1}
scikit-learn Polynomial Kernel Parameters: {'alpha': 0.01, 'degree': 2}
```

The tuned parameters are the same for the built-in model of `sklearn` and the cross-validation I coded up by myself. Plugging those hyper-parameters back, we would get the same MSE as we did in Part A.

1.2.3 Part C

I redid the OLS model in Homework 1, and calculated the MSE using code below

```
    residuals = y_new - y_pred

    # Calculate MSE
    mse = np.mean(residuals**2)
```

The result is as below:

```
Mean Squared Error (MSE): 0.0165
```

Here the non-linear model performed better than the linear model. First of all, linear model is a simple way to fit the data, which is highly likely to be influenced by the noise in the dataset, even though it might has lower MSE. It might cause the problem of overfitting. Whereas the CV can split the dataset into several subsets, and evaluate the model on multiple test and training sets, which would make it much generalizable, and overcome the problem of overfitting. And as we mentioned before, linear models just build the model based on data in the dataset, if the training set is not representative, it might not be a good model; but the CV is more generalizable, giving it ability to comprehensively reflect model performance.

Secondly, CV has hyper-parameters tuning, from which the optimal hyper-parameters can be selected, improving model performance.

Another aspect is that OLS lacks a regularization mechanism, making it susceptible to issues like high-dimensional data or multicollinearity. Where as we implemented regularization here, which helps control model complexity, prevent overfitting, and improve generalization. As for the efficiency to use data, the OLS just use one set of data, whereas CV divides the data multiple times, ensuring that all data points are used for both training and validation.

Finally, if we simply compare the MSE, which is the prediction errors given by different models, the MSE of cross-validated models are still smaller than those of the OLS model.

1.3 Question 3

1.3.1 Part A

Here I listed the confusion matrices, test accuracies given by python for 5 methods below. I did not generate the confusion matrix picture due to the restriction of page-count. Full reports, including `recall` rate, `f1-score`, etc, were inserted in the Appendix by me.

Firstly, for Logistic Regression in One-vs-Rest setting, I got the report

```
Test Accuracy: 93.54%
Confusion Matrix:
[[1328   39   52]
 [  54 1195   45]
 [  39   50 1254]]
```

For Multinomial Regression

```
Test Accuracy: 93.12%
Confusion Matrix:
[[1324   43   52]
 [  51 1201   42]
 [  39   52 1252]]
```

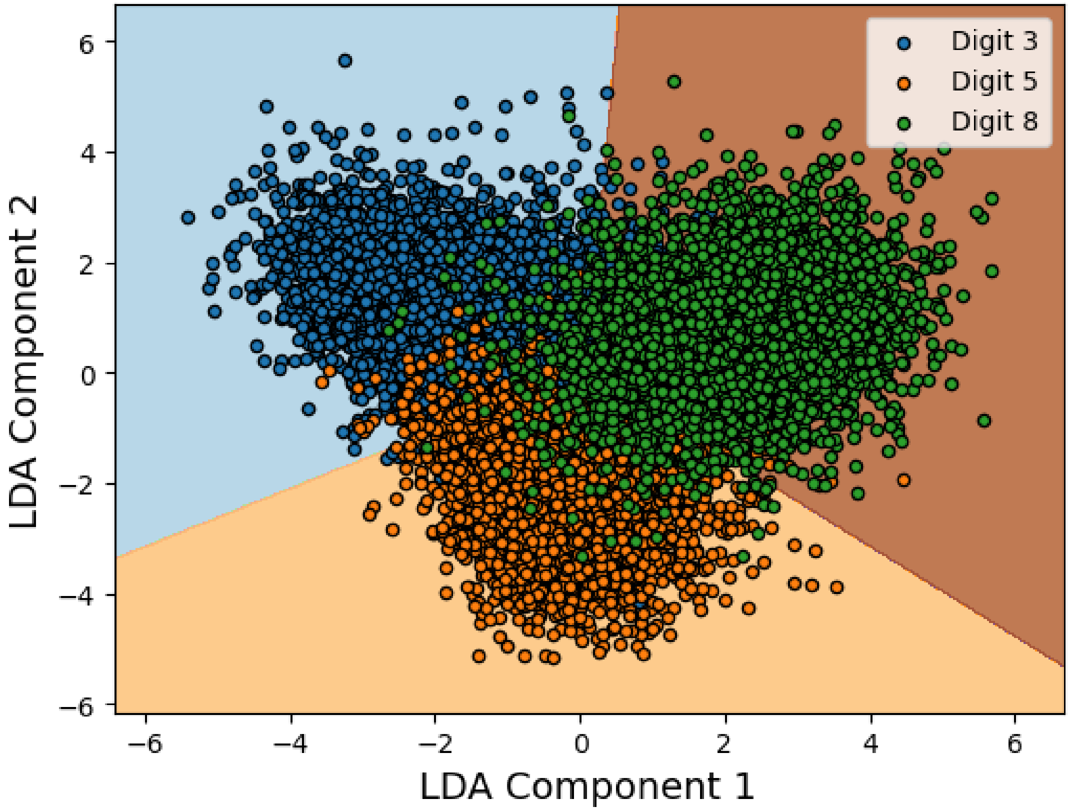
For Naive Bayes

```
Test Accuracy: 50.22%
Confusion Matrix:
[[ 595   26  798]
 [  95  138 1061]
 [  18   21 1304]]
```

For Linear Discriminant Analysis

```
Test Accuracy (LDA): 91.69%
Confusion Matrix (LDA):
[[1285   69   65]
 [  47 1202   45]
 [  34   77 1232]]
```

Figure 1: Decision Boundary after LDA

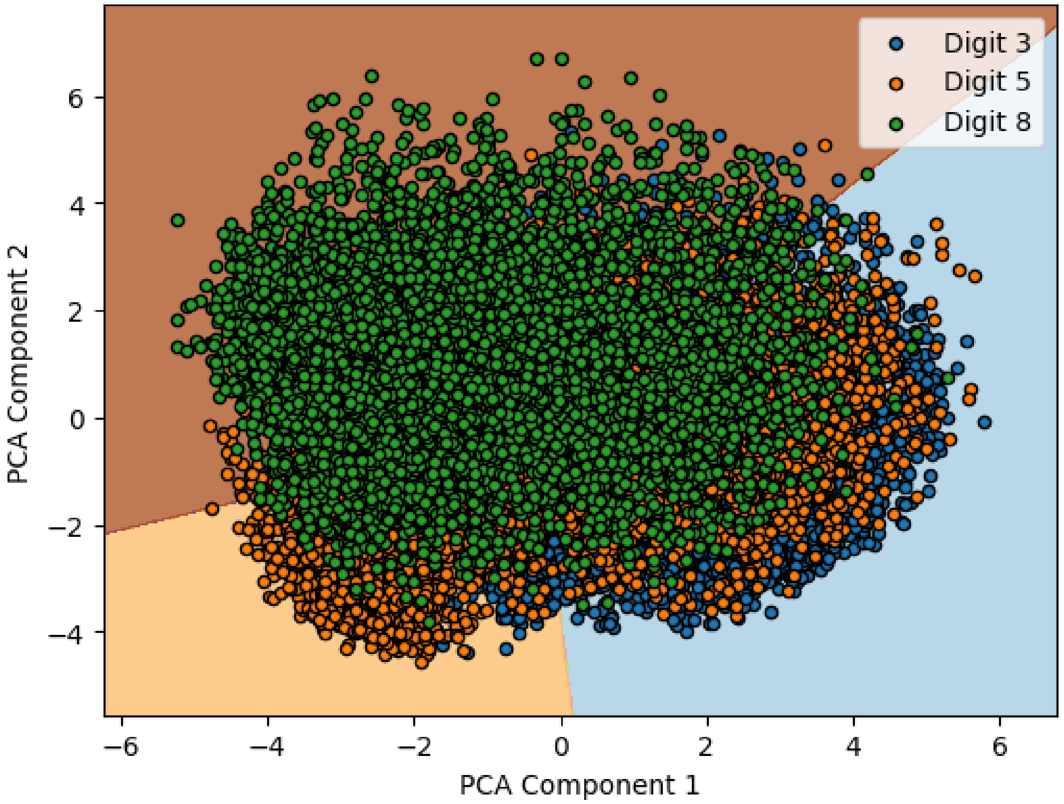


Finally, for Linear Support Vector Machine:

```
Test Accuracy (Linear SVM): 92.73%
Confusion Matrix (Linear SVM):
[[1319   44   56]
 [  57 1193   44]
 [  39   55 1249]]
```

If we want to plot the decision boundary of Linear SVM, we must first do a Principle Component Analysis (PCA).

Figure 2: Decision Boundary after PCA

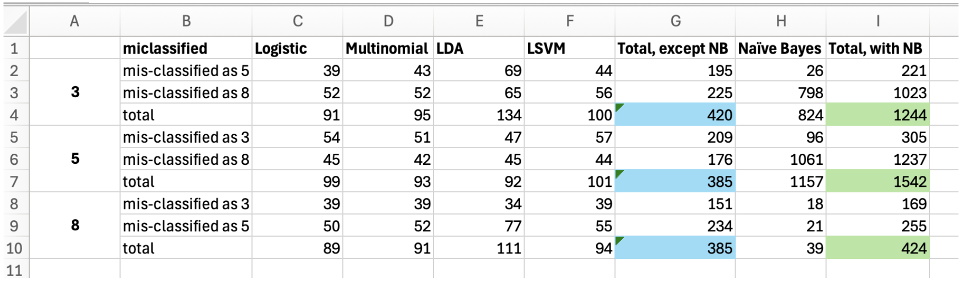


Overall, if we directly compare the test accuracy evaluated by python, the logistic regression has the highest accuracy rate, whereas the Naive Bayes has the least accuracy.

In the python built-in function `confusion matrix()`, the vertical axis represents the true values, whereas the horizontal axis represents the predictive value. Thus we can added the misclassified terms accordingly. However, I noticed that there was a large proportion of data misclassified in Naive Bayes case, so I splitted them into two parts, firstly I calculated the total misinterpreted values for 4 methods other than Naive Bayes; here the most often misclassified digit is 3, and has 420 counts. And for 5 and 8, the numbers are equal, which is 385.

If we add up the NB method, overall, digit 5 is more likely to be misclassified, and this trend towards error reaches to peak when we use Naive Bayes method. The total number of misclassified 5 is 1542, and 3 is 1244, and 8 is 424.

Figure 3: Count of Misclassified Digits



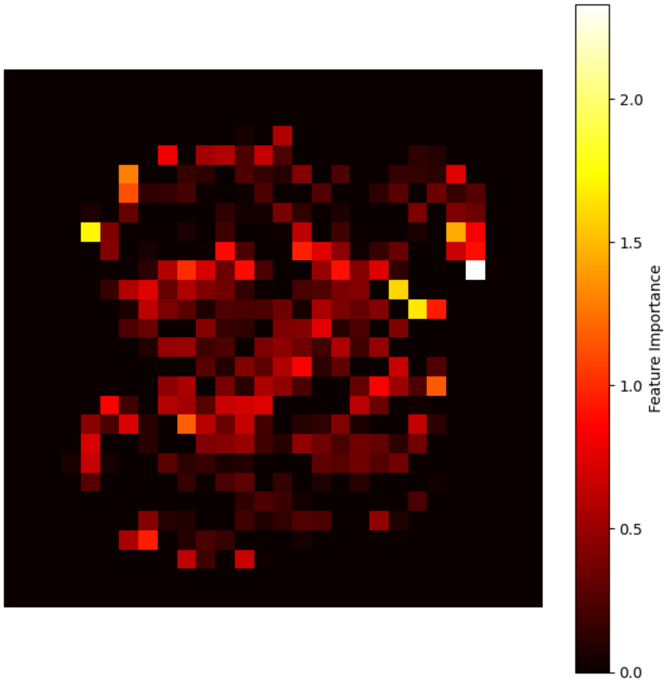
	A	B	C	D	E	F	G	H	I
1		**miclassified**	**Logistic**	**Multinomial**	**LDA**	**LSVM**	**Total, except NB**	**Naïve Bayes**	**Total, with NB**
2	**3**	mis-classified as 5	39	43	69	44	195	26	221
3		mis-classified as 8	52	52	65	56	225	798	1023
4		total	91	95	134	100	420	824	1244
5	**5**	mis-classified as 3	54	51	47	57	209	96	305
6		mis-classified as 8	45	42	45	44	176	1061	1237
7		total	99	93	92	101	385	1157	1542
8	**8**	mis-classified as 3	39	39	34	39	151	18	169
9		mis-classified as 5	50	52	77	55	234	21	255
10		total	89	91	111	94	385	39	424
11									

1.3.2 Part B

The group-lasso regularized multinomial logistic regression will help us to select features that separate the three digits, and we can end up getting a heatmap. In the heatmap, bright regions represent pixels that are highly important for distinguishing between the digits 3, 5, and 8. The model relies heavily on these pixels for making predictions.

And dark regions represent pixels that contribute little to the classification task. The model considers these pixels unimportant.

Figure 4: Headmap showing Features that Separate the Three Digits


0.0
0.5
1.0
1.5
2.0
Feature Importance

2 Appendix

Supplementary Github Repo

2.1 Question 2

2.1.1 Setup

```python
# Same as Homework 1

from ucimlrepo import fetch_ucirepo


# fetch dataset
communities_and_crime = fetch_ucirepo(id=183)

# data (as pandas dataframes)
X = communities_and_crime.data.features
y = communities_and_crime.data.targets

# metadata
print(communities_and_crime.metadata)

# variable information
print(communities_and_crime.variables)

# Inspect the shape of X and y
print(X.shape)  # Should be (1994, 127)
print(y.shape)  # Should be (1994, 1)

# Check for missing values
print(X.isnull().sum())  # Count of missing values per feature

# Inspect the first few rows of X and y
print(X.head())
print(y.head())
X = X.iloc[:, 5:]
print(X.dtypes) # There are object columns within the data. The object data type is the default type for columns containing text (strings) in a pandas DataFrame.
```

Drop columns and rows with missing values:

```python
# Convert all values to numeric (force non-numeric to NaN)
X = X.applymap(pd.to_numeric, errors=’coerce’)

# Replace "?" with NaN
X.replace("?", np.nan, inplace=True)

# Check the number of missing values in each column
missing_counts = X.isnull().sum()

# Determine threshold for column removal (e.g., remove if >50% missing)
threshold = 0.5 * len(X)  # Adjust this threshold as needed
cols_to_drop = missing_counts[missing_counts > threshold].index

# Drop those columns
X_cleaned = X.drop(columns=cols_to_drop)

X_with_y = pd.concat([X_cleaned, y], axis=1)

# Remove rows containing missing values in the combined DataFrame
X_with_y_cleaned = X_with_y.dropna()

```

```
X_with_y_cleaned

# Separate X and y after cleaning
X_final = X_with_y_cleaned.drop(columns=['ViolentCrimesPerPop'])
y_final = X_with_y_cleaned['ViolentCrimesPerPop']
```

Standardize it:

```
Y = y_final.to_numpy()

# Standardize X (centered and scaled)
scaler = StandardScaler(with_mean=True, with_std=True)
X_standardized = scaler.fit_transform(X_final)
```

2.1.2 Split into Different Sets

```
# Set a random seed for reproducibility
rng = default_rng(1)

# Define the proportions for the split
train_prop = 0.6
validation_prop = 0.2
test_prop = 0.2

# Calculate the number of observations for each split
total_samples = X_with_y_cleaned.shape[0]
train_size = int(train_prop * total_samples)
validation_size = int(validation_prop * total_samples)
test_size = total_samples - train_size - validation_size

# Create a random permutation of row indices
indices = rng.choice(np.arange(total_samples), size=(total_samples), replace=False)

# Split the dataset into train, validation, and test sets
y_train = Y[indices[:train_size]]
y_val = Y[indices[(train_size + 1):(train_size + validation_size)]]
y_test = Y[indices[(train_size + validation_size + 1):]]

X_train = X_standardized[indices[:train_size]]
X_val = X_standardized[indices[(train_size + 1):(train_size + validation_size)]]
X_test = X_standardized[indices[(train_size + validation_size + 1):]]
```

2.1.3 Using K-fold CV from scratch to tune Kernel Ridge Regularization

```
import numpy as np
from sklearn.metrics import mean_squared_error

# Define the RBF kernel function
def rbf_kernel(X1, X2, gamma):
pairwise_dists = np.sum(X1**2, axis=1).reshape(-1, 1) + np.sum(X2**2, axis=1) - 2 *
    np.dot(X1, X2.T)
return np.exp(-gamma * pairwise_dists)

# Define the Polynomial kernel function
def polynomial_kernel(X1, X2, degree, c=1):
return (np.dot(X1, X2.T) + c) ** degree

# Kernel Ridge Regression
def kernel_ridge_regression(X_train, y_train, X_test, kernel, lambda_, **
    kernel_params):
"""
```

```
Perform Kernel Ridge Regression.

Parameters:
X_train: Training data (n_samples, n_features).
y_train: Target values (n_samples,).
X_test: Test data (n_samples_test, n_features).
kernel: Kernel function (rbf_kernel or polynomial_kernel).
lambda_: Regularization parameter.
**kernel_params: Kernel-specific parameters (e.g., gamma for RBF, degree for Polynomial).

Returns:
y_pred: Predicted values for X_test.
"""
# Compute the kernel matrix
K_train = kernel(X_train, X_train, **kernel_params)
K_test = kernel(X_test, X_train, **kernel_params)

# Solve for the dual coefficients alpha
n_samples = X_train.shape[0]
alpha = np.linalg.inv(K_train + lambda_ * np.eye(n_samples)) @ y_train

# Predict on the test set
y_pred = K_test @ alpha
return y_pred

# K-Fold Cross-Validation
def k_fold_cross_validation(X_standardized, Y, k, kernel, lambda_range, gamma_range=None, degree_range=None):
"""
Perform K-fold cross-validation to select the best hyperparameters.

Parameters:
X: Input data (n_samples, n_features).
y: Target values (n_samples,).
k: Number of folds.
kernel: Kernel function (rbf_kernel or polynomial_kernel).
lambda_range: List of regularization parameters to try.
gamma_range: List of gamma values for RBF kernel (optional).
degree_range: List of degree values for Polynomial kernel (optional).

Returns:
best_lambda: Best regularization parameter.
best_gamma: Best gamma value (for RBF kernel).
best_degree: Best degree value (for Polynomial kernel).
"""
fold_size = len(X_standardized) // k
best_lambda = None
best_gamma = None
best_degree = None
best_score = float(’inf’)

# Grid search over hyperparameters
for lambda_ in lambda_range:
if kernel == rbf_kernel:
# RBF kernel: only gamma is needed
for gamma in (gamma_range if gamma_range is not None else [1.0]):
scores = []
for i in range(k):
# Split into training and validation sets
val_indices = range(i * fold_size, (i + 1) * fold_size)
train_indices = np.setdiff1d(range(len(X)), val_indices)

```

```
X_train, y_train = X[train_indices], y[train_indices]
X_val, y_val = X[val_indices], y[val_indices]

# Train and predict
y_pred = kernel_ridge_regression(X_train, y_train, X_val, kernel, lambda_, gamma=
    gamma)

# Compute the validation score (e.g., mean squared error)
score = mean_squared_error(y_val, y_pred)
scores.append(score)

# Average score across folds
avg_score = np.mean(scores)
if avg_score < best_score:
best_score = avg_score
best_lambda = lambda_
best_gamma = gamma

elif kernel == polynomial_kernel:
# Polynomial kernel: degree and optionally c are needed
for degree in (degree_range if degree_range is not None else [2]):
scores = []
for i in range(k):
# Split into training and validation sets
val_indices = range(i * fold_size, (i + 1) * fold_size)
train_indices = np.setdiff1d(range(len(X)), val_indices)

X_train, y_train = X[train_indices], y[train_indices]
X_val, y_val = X[val_indices], y[val_indices]

# Train and predict
y_pred = kernel_ridge_regression(X_train, y_train, X_val, kernel, lambda_, degree=
    degree)

# Compute the validation score (e.g., mean squared error)
score = mean_squared_error(y_val, y_pred)
scores.append(score)

# Average score across folds
avg_score = np.mean(scores)
if avg_score < best_score:
best_score = avg_score
best_lambda = lambda_
best_degree = degree

return best_lambda, best_gamma, best_degree

# Example usage
if __name__ == "__main__":
# Generate synthetic data
np.random.seed(42)
X = np.random.rand(100, 2)  # 100 samples, 2 features
y = 3 * X[:, 0] + 5 * X[:, 1] + np.random.randn(100) * 0.1  # Linear relationship
    with noise

# Split data into train, validation, and test sets
train_size = int(0.6 * len(X))
val_size = int(0.2 * len(X))
test_size = len(X) - train_size - val_size

indices = np.random.permutation(len(X))
X_train, y_train = X[indices[:train_size]], y[indices[:train_size]]
```

```
    X_val, y_val = X[indices[train_size:train_size + val_size]], y[indices[train_size:train_size + val_size]]
    X_test, y_test = X[indices[train_size + val_size:]], y[indices[train_size + val_size:]]

    # Define hyperparameter ranges
    lambda_range = [0.01, 0.1, 1, 10]
    gamma_range = [0.01, 0.1, 1]  # For RBF kernel
    degree_range = [2, 3, 4, 5, 6, 7]      # For Polynomial kernel

    # Perform K-fold cross-validation for RBF kernel
    best_lambda_rbf, best_gamma_rbf, _ = k_fold_cross_validation(
    X_train, y_train, k=5, kernel=rbf_kernel, lambda_range=lambda_range, gamma_range=gamma_range
    )
    print(f"Best␣RBF␣Kernel␣Parameters:␣lambda={best_lambda_rbf},␣gamma={best_gamma_rbf}")

    # Perform K-fold cross-validation for Polynomial kernel
    best_lambda_poly, _, best_degree_poly = k_fold_cross_validation(
    X_train, y_train, k=5, kernel=polynomial_kernel, lambda_range=lambda_range, degree_range=degree_range
    )
    print(f"Best␣Polynomial␣Kernel␣Parameters:␣lambda={best_lambda_poly},␣degree={best_degree_poly}")

    # Evaluate on the test set with the best parameters
    y_pred_rbf = kernel_ridge_regression(X_train, y_train, X_test, rbf_kernel, best_lambda_rbf, gamma=best_gamma_rbf)
    y_pred_poly = kernel_ridge_regression(X_train, y_train, X_test, polynomial_kernel, best_lambda_poly, degree=best_degree_poly)

    print(f"RBF␣Kernel␣Test␣MSE:␣{mean_squared_error(y_test,␣y_pred_rbf)}")
    print(f"Polynomial␣Kernel␣Test␣MSE:␣{mean_squared_error(y_test,␣y_pred_poly)}")
```

2.1.4 Redo the OLS as in Homework 1

```
    # Step 1: Separate y and X
    X_new = X_final
    y_new = y_final

    # Step 2: Add a constant to X (for the intercept term)
    X_new = sm.add_constant(X_new)

    # Step 3: Fit the OLS model
    model = sm.OLS(y_new, X_new)

    results = model.fit()

    # Step 4: View the results
    print(results.summary())
```

```
                             OLS Regression Results
==============================================================================
Dep. Variable:     ViolentCrimesPerPop   R-squared:                       0.696
Model:                             OLS   Adj. R-squared:                  0.680
Method:                  Least Squares   F-statistic:                     43.26
Date:                 Fri, 07 Mar 2025   Prob (F-statistic):               0.00
Time:                         13:15:31   Log-Likelihood:                 1261.1
No. Observations:                 1993   AIC:                            -2320.
Df Residuals:                     1892   BIC:                            -1755.
Df Model:                          100
Covariance Type:             nonrobust
=========================================================================================
coef    std err          t      P>|t|      [0.025      0.975]
-----------------------------------------------------------------------------------------
const                     0.5504      0.203      2.712      0.007       0.152       0.948
population                0.1840      0.397      0.463      0.643      -0.595       0.963
householdsize            -0.0223      0.086     -0.259      0.796      -0.191       0.147
racepctblack              0.2049      0.051      4.008      0.000       0.105       0.305
racePctWhite             -0.0492      0.059     -0.837      0.403      -0.164       0.066
racePctAsian             -0.0144      0.034     -0.420      0.674      -0.082       0.053
racePctHisp               0.0609      0.053      1.139      0.255      -0.044       0.166
agePct12t21               0.1104      0.106      1.043      0.297      -0.097       0.318
agePct12t29              -0.2292      0.156     -1.467      0.143      -0.536       0.077
agePct16t24              -0.1302      0.164     -0.793      0.428      -0.452       0.192
agePct65up                0.0497      0.103      0.481      0.630      -0.153       0.253
numbUrban                -0.2964      0.387     -0.766      0.444      -1.055       0.462
pctUrban                  0.0467      0.016      2.989      0.003       0.016       0.077
medIncome                -0.1998      0.173     -1.158      0.247      -0.538       0.139
pctWWage                 -0.2016      0.089     -2.259      0.024      -0.377      -0.027
pctWFarmSelf              0.0488      0.020      2.422      0.016       0.009       0.088
pctWInvInc               -0.1731      0.068     -2.563      0.010      -0.306      -0.041
pctWSocSec                0.0762      0.107      0.712      0.477      -0.134       0.286
pctWPubAsst               0.0050      0.046      0.108      0.914      -0.085       0.095
pctWRetire               -0.0900      0.037     -2.445      0.015      -0.162      -0.018
medFamInc                 0.2880      0.160      1.797      0.073      -0.026       0.602
perCapInc                 0.0955      0.189      0.506      0.613      -0.274       0.465
whitePerCap              -0.3510      0.152     -2.303      0.021      -0.650      -0.052
blackPerCap              -0.0288      0.025     -1.131      0.258      -0.079       0.021
indianPerCap             -0.0357      0.019     -1.841      0.066      -0.074       0.002
AsianPerCap               0.0216      0.019      1.145      0.252      -0.015       0.059
OtherPerCap               0.0438      0.019      2.341      0.019       0.007       0.081
HispPerCap                0.0357      0.025      1.435      0.151      -0.013       0.085
NumUnderPov               0.1112      0.138      0.805      0.421      -0.160       0.382
PctPopUnderPov           -0.1721      0.063     -2.745      0.006      -0.295      -0.049
PctLess9thGrade          -0.0999      0.068     -1.474      0.141      -0.233       0.033
PctNotHSGrad              0.0525      0.096      0.548      0.584      -0.136       0.241
PctBSorMore               0.0504      0.077      0.651      0.515      -0.101       0.202
PctUnemployed             0.0045      0.041      0.111      0.911      -0.075       0.084
PctEmploy                 0.2485      0.079      3.151      0.002       0.094       0.403
PctEmplManu              -0.0658      0.032     -2.054      0.040      -0.129      -0.003
PctEmplProfServ          -0.0267      0.041     -0.654      0.513      -0.107       0.053
PctOccupManu              0.0723      0.055      1.318      0.188      -0.035       0.180
PctOccupMgmtProf          0.1226      0.086      1.419      0.156      -0.047       0.292
MalePctDivorce            0.4585      0.248      1.851      0.064      -0.027       0.944
MalePctNevMarr            0.2267      0.068      3.339      0.001       0.094       0.360
FemalePctDiv              0.1627      0.309      0.526      0.599      -0.444       0.770
TotalPctDiv              -0.5619      0.519     -1.084      0.279      -1.579       0.455
PersPerFam               -0.1405      0.168     -0.834      0.404      -0.471       0.190
PctFam2Par                0.0186      0.160      0.117      0.907      -0.294       0.331
PctKids2Par              -0.3227      0.155     -2.080      0.038      -0.627      -0.018
PctYoungKids2Par         -0.0323      0.048     -0.670      0.503      -0.127       0.062
PctTeen2Par              -0.0029      0.043     -0.069      0.945      -0.087       0.081
```

2.2 Question 3

2.2.1 Read the Data

```
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Load MNIST dataset and filter digits 3, 5, and 8
from sklearn.datasets import fetch_openml

# Fetch MNIST from openml
mnist = fetch_openml(’mnist_784’, version=1, as_frame=False)
X, y = mnist["data"], mnist["target"]

mnist_df = pd.DataFrame(np.concatenate((mnist[’target’].reshape(-1, 1), mnist[’data’
    ]), axis=1),
columns= [’target’] +  mnist[’feature_names’])

mnist_df
```

Then we keep only 3, 5, and 8.

```
import numpy as np
import pandas as pd
from sklearn.datasets import fetch_openml

# Fetch MNIST from OpenML
mnist = fetch_openml(’mnist_784’, version=1, as_frame=False)
X, y = mnist["data"], mnist["target"]

# Convert labels to integers
y = y.astype(int)

# Correct filtering: Keep only digits 3, 5, and 8
selected_digits = {3, 5, 8}
filter_mask = np.isin(y, list(selected_digits))

X_filtered = X[filter_mask]  # Keep only selected digits
y_filtered = y[filter_mask]  # Keep corresponding labels

# Print dataset size
print(f"Original␣dataset␣size:␣{X.shape[0]}")
print(f"Filtered␣dataset␣size␣(only␣3,␣5,␣8):␣{X_filtered.shape[0]}")

# Convert to DataFrame (Optional)
mnist_df = pd.DataFrame(np.column_stack((y_filtered, X_filtered)),
columns=[’target’] + mnist.feature_names)

# Display the dataframe
mnist_df
```

2.2.2 Logistic Regression with OvR

```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import numpy as np
import matplotlib.pyplot as plt

# 1. Load MNIST
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
X, y = mnist.data, mnist.target
y = y.astype(int)

# filter 3, 5, 8
selected_digits = [3, 5, 8]
mask = np.isin(y, selected_digits)
X_filtered, y_filtered = X[mask], y[mask]

X_filtered = X_filtered / 255.0

# Divide the sets
X_train, X_test, y_train, y_test = train_test_split(X_filtered, y_filtered, test_size=0.2, random_state=42)

# 2. Train the Logistic Regression (One-vs-Rest)
log_reg_ovr = LogisticRegression(multi_class='ovr', solver='liblinear', max_iter=100, random_state=42)
log_reg_ovr.fit(X_train, y_train)

# 3. Evaluate
y_pred = log_reg_ovr.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Test␣Accuracy:␣{accuracy␣*␣100:.2f}%")

# Confusion Matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion␣Matrix:")
print(conf_matrix)

# Class Report
class_report = classification_report(y_test, y_pred)
print("Classification␣Report:")
print(class_report)

# 4. Visualize
num_images = 5
indices = np.random.choice(len(X_test), num_images, replace=False)

plt.figure(figsize=(10, 5))
for i, index in enumerate(indices):
plt.subplot(1, num_images, i + 1)
plt.imshow(X_test[index].reshape(28, 28), cmap='gray')
plt.title(f"Pred:␣{y_pred[index]}\nTrue:␣{y_test[index]}")
plt.axis('off')
plt.show()
```

```
Test Accuracy: 93.12%
Confusion Matrix:
[[1328   39   52]
[  54 1195   45]
[  39   50 1254]]
Classification Report:
precision    recall  f1-score   support

3       0.93      0.94      0.94      1419
5       0.93      0.92      0.93      1294
8       0.93      0.93      0.93      1343

accuracy                         0.93      4056
macro avg       0.93      0.93      0.93      4056
weighted avg       0.93      0.93      0.93      4056
```

2.2.3 Multinomial Regression

```
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import numpy as np
import matplotlib.pyplot as plt

# Similar steps

mnist = fetch_openml(’mnist_784’, version=1, as_frame=False)
X, y = mnist.data, mnist.target
y = y.astype(int)

selected_digits = [3, 5, 8]
mask = np.isin(y, selected_digits)
X_filtered, y_filtered = X[mask], y[mask]

X_filtered = X_filtered / 255.0

X_train, X_test, y_train, y_test = train_test_split(X_filtered, y_filtered, test_size=0.2, random_state=42)

log_reg_multinomial = LogisticRegression(multi_class=’multinomial’, solver=’lbfgs’, max_iter=100, random_state=42)
log_reg_multinomial.fit(X_train, y_train)

y_pred = log_reg_multinomial.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Test␣Accuracy:␣{accuracy␣*␣100:.2f}%")

conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion␣Matrix:")
print(conf_matrix)

class_report = classification_report(y_test, y_pred)
print("Classification␣Report:")
print(class_report)

num_images = 5
indices = np.random.choice(len(X_test), num_images, replace=False)

plt.figure(figsize=(10, 5))
for i, index in enumerate(indices):
plt.subplot(1, num_images, i + 1)
plt.imshow(X_test[index].reshape(28, 28), cmap='gray')
plt.title(f"Pred:␣{y_pred[index]}\nTrue:␣{y_test[index]}")
plt.axis('off')
plt.show()
```

```
Test Accuracy: 93.12%
Confusion Matrix:
[[1324   43   52]
 [  51 1201   42]
 [  39   52 1252]]
Classification Report:
precision    recall  f1-score   support

3       0.94      0.93      0.93      1419
5       0.93      0.93      0.93      1294
8       0.93      0.93      0.93      1343

accuracy                           0.93      4056
macro avg       0.93      0.93      0.93      4056
weighted avg       0.93      0.93      0.93      4056
```

Do the Learning Curve

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split, learning_curve
from sklearn.linear_model import LogisticRegression

# 1. Load and filter the MNIST dataset
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
X, y = mnist.data, mnist.target
y = y.astype(int)

# Filter out samples with labels 3, 5, 8
selected_digits = [3, 5, 8]
mask = np.isin(y, selected_digits)
X_filtered, y_filtered = X[mask], y[mask]

# Scale pixel values from [0, 255] to [0, 1]
X_filtered = X_filtered / 255.0

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_filtered, y_filtered, test_size=0.2, random_state=42)

# 2. Define the Logistic Regression model
log_reg = LogisticRegression(solver='lbfgs', max_iter=100, random_state=42)

# 3. Compute the learning curve
train_sizes, train_scores, val_scores = learning_curve(
log_reg, X_train, y_train, cv=5, scoring='accuracy', train_sizes=np.linspace(0.1, 1.0, 10)
)

# Calculate mean and standard deviation of training and validation scores
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
val_scores_mean = np.mean(val_scores, axis=1)
val_scores_std = np.std(val_scores, axis=1)

# 4. Plot the learning curve
plt.figure(figsize=(10, 6))
```

```
    plt.plot(train_sizes, train_scores_mean, label='Training␣Accuracy', color='blue', marker='o')
    plt.fill_between(train_sizes, train_scores_mean - train_scores_std, train_scores_mean + train_scores_std, alpha=0.15, color='blue')
    plt.plot(train_sizes, val_scores_mean, label='Validation␣Accuracy', color='green', marker='o')
    plt.fill_between(train_sizes, val_scores_mean - val_scores_std, val_scores_mean + val_scores_std, alpha=0.15, color='green')

    # plt.title('Learning Curve for Logistic Regression', fontsize=16)
    plt.xlabel('Training␣Set␣Size', fontsize=14)
    plt.ylabel('Accuracy', fontsize=14)
    plt.legend(loc='best')
    plt.grid(True)
    plt.show()
```

Do the Validation Curve

```
    from sklearn.model_selection import validation_curve

    # Define the range of hyperparameter C (inverse of regularization strength)
    param_range = np.logspace(-4, 4, 10)

    # Compute validation curve
    train_scores, val_scores = validation_curve(
    log_reg, X_train, y_train, param_name='C', param_range=param_range, cv=5, scoring='accuracy'
    )

    # Calculate mean and standard deviation of training and validation scores
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    val_scores_mean = np.mean(val_scores, axis=1)
    val_scores_std = np.std(val_scores, axis=1)

    # Plot the validation curve
    plt.figure(figsize=(10, 6))
    plt.semilogx(param_range, train_scores_mean, label='Training␣Accuracy', color='blue', marker='o')
    plt.fill_between(param_range, train_scores_mean - train_scores_std, train_scores_mean + train_scores_std, alpha=0.15, color='blue')
    plt.semilogx(param_range, val_scores_mean, label='Validation␣Accuracy', color='green', marker='o')
    plt.fill_between(param_range, val_scores_mean - val_scores_std, val_scores_mean + val_scores_std, alpha=0.15, color='green')

    # plt.title('Validation Curve for Logistic Regression', fontsize=16)
    plt.xlabel('Regularization␣Strength␣(C)', fontsize=14)
    plt.ylabel('Accuracy', fontsize=14)
    plt.legend(loc='best')
    plt.grid(True)
    plt.show()
```

2.2.4 Naive Bayes

```
    from sklearn.datasets import fetch_openml
    from sklearn.model_selection import train_test_split
    from sklearn.naive_bayes import GaussianNB
    from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
    import numpy as np
    import matplotlib.pyplot as plt

    # 1. Load and filter the MNIST dataset
    mnist = fetch_openml('mnist_784', version=1, as_frame=False)
    X, y = mnist.data, mnist.target
    y = y.astype(int)

    # Filter out samples with labels 3, 5, 8
    selected_digits = [3, 5, 8]
    mask = np.isin(y, selected_digits)
    X_filtered, y_filtered = X[mask], y[mask]

    # Scale pixel values from [0, 255] to [0, 1]
    X_filtered = X_filtered / 255.0

    # Split the dataset into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X_filtered, y_filtered, test_size=0.2, random_state=42)

    # 2. Train a Gaussian Naive Bayes model
    naive_bayes = GaussianNB()
    naive_bayes.fit(X_train, y_train)

    # 3. Evaluate the model
    y_pred = naive_bayes.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print(f"Test Accuracy: {accuracy * 100:.2f}%")

    # Confusion matrix
    conf_matrix = confusion_matrix(y_test, y_pred)
    print("Confusion Matrix:")
    print(conf_matrix)

    # Classification report
    class_report = classification_report(y_test, y_pred)
    print("Classification Report:")
    print(class_report)

    # 4. Visualize the results
    num_images = 5
    indices = np.random.choice(len(X_test), num_images, replace=False)

    plt.figure(figsize=(10, 5))
    for i, index in enumerate(indices):
    plt.subplot(1, num_images, i + 1)
    plt.imshow(X_test[index].reshape(28, 28), cmap='gray')
    plt.title(f"Pred: {y_pred[index]}\nTrue: {y_test[index]}")
    plt.axis('off')
    plt.show()
```

```
Test Accuracy: 50.22%
Confusion Matrix:
[[ 595   26  798]
 [  95  138 1061]
 [  18   21 1304]]
Classification Report:
precision    recall  f1-score   support

3       0.84      0.42      0.56      1419
5       0.75      0.11      0.19      1294
8       0.41      0.97      0.58      1343

accuracy                           0.50      4056
macro avg       0.67      0.50      0.44      4056
weighted avg       0.67      0.50      0.45      4056
```

2.2.5 Linear Discriminant Analysis

```python
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import numpy as np
import matplotlib.pyplot as plt

# 1. Load and filter the MNIST dataset
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
X, y = mnist.data, mnist.target
y = y.astype(int)

# Filter out samples with labels 3, 5, 8
selected_digits = [3, 5, 8]
mask = np.isin(y, selected_digits)
X_filtered, y_filtered = X[mask], y[mask]

# Scale pixel values from [0, 255] to [0, 1]
X_filtered = X_filtered / 255.0

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_filtered, y_filtered, test_size=0.2, random_state=42)

# 2. Train a Linear Discriminant Analysis (LDA) model
lda = LinearDiscriminantAnalysis()
lda.fit(X_train, y_train)

# 3. Evaluate the model
y_pred = lda.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy (LDA): {accuracy * 100:.2f}%")

# Confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix (LDA):")
print(conf_matrix)

# Classification report
class_report = classification_report(y_test, y_pred)
print("Classification Report (LDA):")
print(class_report)

# 4. Visualize the results
```

```python
num_images = 5
indices = np.random.choice(len(X_test), num_images, replace=False)

plt.figure(figsize=(10, 5))
for i, index in enumerate(indices):
plt.subplot(1, num_images, i + 1)
plt.imshow(X_test[index].reshape(28, 28), cmap='gray')
plt.title(f"Pred:␣{y_pred[index]}\nTrue:␣{y_test[index]}")
plt.axis('off')
plt.show()
```

```
Test Accuracy (LDA): 91.69%
Confusion Matrix (LDA):
[[1285   69   65]
 [  47 1202   45]
 [  34   77 1232]]
Classification Report (LDA):
precision    recall  f1-score   support

3       0.94      0.91      0.92      1419
5       0.89      0.93      0.91      1294
8       0.92      0.92      0.92      1343

accuracy                           0.92      4056
macro avg       0.92      0.92      0.92      4056
weighted avg       0.92      0.92      0.92      4056
```

2.2.6 Linear SVM

```python
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
import numpy as np
import matplotlib.pyplot as plt

# 1. Load and filter the MNIST dataset
mnist = fetch_openml('mnist_784', version=1, as_frame=False)
X, y = mnist.data, mnist.target
y = y.astype(int)

# Filter out samples with labels 3, 5, 8
selected_digits = [3, 5, 8]
mask = np.isin(y, selected_digits)
X_filtered, y_filtered = X[mask], y[mask]

# Scale pixel values from [0, 255] to [0, 1]
X_filtered = X_filtered / 255.0

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_filtered, y_filtered, test_size=0.2, random_state=42)

# 2. Train a Linear SVM model with One-vs-Rest strategy
linear_svm = LinearSVC(multi_class='ovr', max_iter=10000, random_state=42)  # Use One-vs-Rest
linear_svm.fit(X_train, y_train)

# 3. Evaluate the model
y_pred = linear_svm.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
```

```
print(f"Test␣Accuracy␣(Linear␣SVM):␣{accuracy␣*␣100:.2f}%")

# Confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion␣Matrix␣(Linear␣SVM):")
print(conf_matrix)

# Classification report
class_report = classification_report(y_test, y_pred)
print("Classification␣Report␣(Linear␣SVM):")
print(class_report)

# 4. Visualize the results
num_images = 5
indices = np.random.choice(len(X_test), num_images, replace=False)

plt.figure(figsize=(10, 5))
for i, index in enumerate(indices):
plt.subplot(1, num_images, i + 1)
plt.imshow(X_test[index].reshape(28, 28), cmap='gray')
# plt.title(f"Pred: {y_pred[index]}\nTrue: {y_test[index]}")
plt.axis('off')
plt.show()
```

```
Test Accuracy (Linear SVM): 92.73%
Confusion Matrix (Linear SVM):
[[1319   44   56]
[  57 1193   44]
[  39   55 1249]]
Classification Report (Linear SVM):
precision    recall  f1-score   support

3       0.93      0.93      0.93      1419
5       0.92      0.92      0.92      1294
8       0.93      0.93      0.93      1343

accuracy                            0.93      4056
macro avg       0.93      0.93      0.93      4056
weighted avg       0.93      0.93      0.93      4056
```

2.2.7 Plot the Confusion Matrix

```
from sklearn.metrics import confusion_matrix
import seaborn as sns

# Logistic Regression Confusion Matrix
log_reg_pred = log_reg.predict(X_test)
log_reg_cm = confusion_matrix(y_test, log_reg_pred)

# Multinomial Regression Confusion Matrix
multinomial_pred = multinomial_reg.predict(X_test)
multinomial_cm = confusion_matrix(y_test, multinomial_pred)

# Naive Bayes Confusion Matrix
naive_bayes_pred = naive_bayes.predict(X_test)
naive_bayes_cm = confusion_matrix(y_test, naive_bayes_pred)

# LDA Confusion Matrix
lda_pred = lda.predict(X_test)
lda_cm = confusion_matrix(y_test, lda_pred)

```

```
    # SVM Confusion Matrix
    svm_pred = svm.predict(X_test)
    svm_cm = confusion_matrix(y_test, svm_pred)

    def plot_confusion_matrix(cm, class_names):
    plt.figure(figsize=(6, 5))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names,
        yticklabels=class_names)
    plt.xlabel('Predicted')
    plt.ylabel('True')
    plt.title('Confusion Matrix')
    plt.show()

    class_names = ['3', '5', '8']

    # Plot confusion matrices
    plot_confusion_matrix(log_reg_cm, class_names)
    plot_confusion_matrix(multinomial_cm, class_names)
    plot_confusion_matrix(naive_bayes_cm, class_names)
    plot_confusion_matrix(lda_cm, class_names)
    plot_confusion_matrix(svm_cm, class_names)
```