PATTERN RECOGNITION

FALL 2023

Seraina Kytili 9728
Matthieu Ndumbi Lukuenya 9217

DATA OVERVIEW

- The dataset is loaded from dataset.csv file.
- The samples are 280.
- Each sample has a 2D vector along with a label (1, 2, or 3).
- This dataset will be used in parts A, B and C.

```
from tabulate import tabulate
print(tabulate(df, headers='keys', tablefmt='psql'))
```

```
+-----+-----------+-----------+-----+
|     |         0 |         1 |   2 |
|-----+-----------+-----------+-----|
|   0 |  1.8036   |  4.4229   |   3 |
|   1 |  3.4615   |  4.1436   |   2 |
|   2 |  2.1873   |  3.9964   |   1 |
|   3 |  3.0933   |  2.9056   |   1 |
|   4 |  1.7586   |  2.4109   |   1 |
|   5 |  1.3935   |  3.3955   |   3 |
|   6 | -0.25885  | -0.30159  |   1 |
|   7 |  7.1342   |  4.0605   |   1 |
|   8 |  3.1281   |  3.4291   |   2 |
|   9 |  5.7726   |  0.95443  |   2 |
|  10 |  1.6966   |  2.3042   |   1 |
|  11 |  4.5855   |  6.8982   |   1 |
|  12 |  1.9932   |  3.8719   |   1 |
|  13 |  1.8076   |  1.4564   |   1 |
|  14 |  2.5472   |  5.1554   |   3 |
|  15 |  4.9085   |  3.5222   |   2 |
|  16 |  4.8933   |  3.7063   |   1 |
|  17 |  0.85293  |  0.56615  |   1 |
|  18 |  2.1049   |  2.9689   |   1 |
```

IMPORTS AND PREPROCESSING

- Import the libraries needed.
- Load the dataset.
- Separate features and labels.
- Split train and test datasets.

```
# Preprocessing.
# Insert the dataset into a Dataframe called movies_df
df = pd.read_csv("../../dataset.csv", header=None)

# Separate into X (features) and y (label)
X = df.iloc[:, :-1] # Select all columns except the last one
y = df.iloc[:, -1] # Select the last column

# Split data 50% - 50%
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.5,
                                                    random_state=77)

# Create a dictionary for the mean error value of each classifier
mean_errors = {}
```

A. BAYESIAN CLASSIFIER WITH MAXIMUM LIKELIHOOD ESTIMATION

- The goal is to implement a Bayesian classifier using Maximum Likelihood Estimation and compare different covariance table approaches.
- We'll compare the performance of the classifier when assuming the same covariance matrix for all classes versus different covariance matrices for each class.

A.1 - CLASSIFIER WITH SAME COVARIANCE

- **Classifier Type:** Gaussian Naive Bayes.
- **Assumption:** Same covariance matrix for all classes.
- **Training:** Fit the classifier with the training data.
- **Predictions:** Use the trained classifier to make predictions on the test set.
- **Error Calculation:** Calculate the mean error by comparing the predicted labels with the true labels. Utilize the accuracy score as the error metric.
- **Result Display:** Print and store the mean error for the case of the same covariance matrix.

```
# Part A.1
# Train a Naive Bayes classifier with MLE and assuming the same covariance
# matrix for all classes
classifier = GaussianNB()
classifier.fit(X_train, y_train)

# Make predictions on the test set
ySame_pred = classifier.predict(X_test)

# Calculate error
mean_error_same_covariance  = 1 - accuracy_score(y_test, ySame_pred)
mean_errors["Same Covariance"] = mean_error_same_covariance
print(f"Error of same covariance: {mean_error_same_covariance:.3f}")
```

```
Error of same covariance: 0.150
```

A.2 – CLASSIFIER WITH DIFFERENT COVARIANCES

- **Classifier Type:** Quadratic Discriminant Analysis (QDA)
- **Assumption:** Different covariance matrix for all classes.
- **Training:** Fit the classifier with the training data.
- **Predictions:** Use the trained classifier to make predictions on the test set.
- **Error Calculation:** Calculate the mean error by comparing the predicted labels with the true labels. Utilize the accuracy score as the error metric.
- **Result Display:** Print and store the mean error for the case of the same covariance matrix.

```
# Part A.2
# Initialize a Quadratic Discriminant Analysis (QDA) classifier
qda = QuadraticDiscriminantAnalysis()

# Fit the model to the training data
qda.fit(X_train, y_train)

# Predict the labels for the test set
yDiff_pred = qda.predict(X_test)

# Calculate error
mean_error_diff_covariance  = 1 - accuracy_score(y_test, yDiff_pred)
mean_errors["Different Covariance"] = mean_error_diff_covariance
print(f"Error of different covariance: {mean_error_diff_covariance:.3f}")
```

A.3 - FUNCTION

- **Function Purpose**: Define a function to plot decision regions and mark misclassified points.
- **Input Parameters**:

  X: Feature matrix (2D),

  y: True labels,

  classifier: Trained classifier or distribution models,

  predictions: Predictions made by the classifier,

  title: Title for the plot,

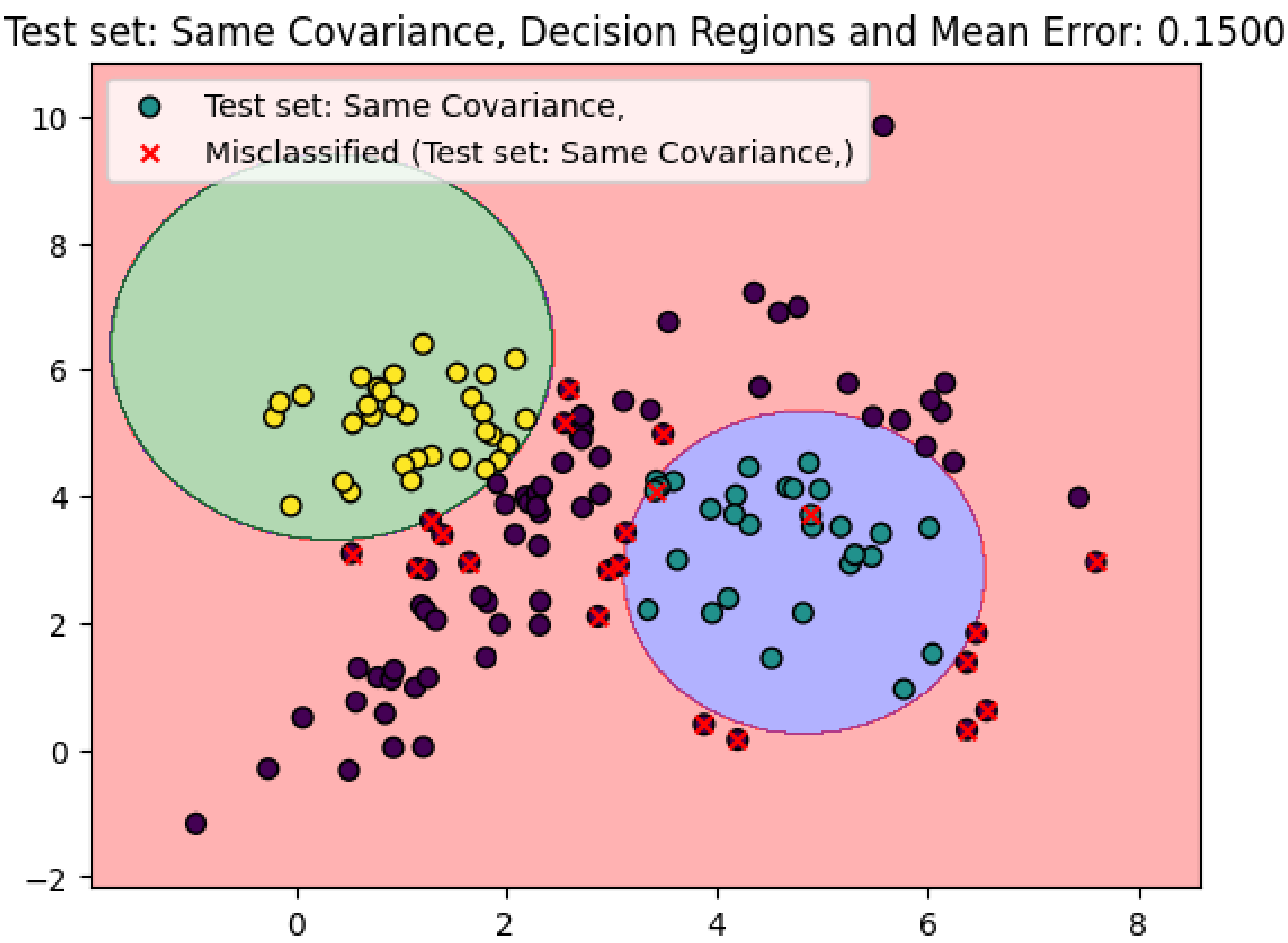
  resolution: Step size in the mesh for decision region plotting.
- **Meshgrid Creation**: Generate a meshgrid based on feature ranges and the specified resolution.
- **Decision Regions Plotting:** Use the classifier to predict classes for meshgrid points.

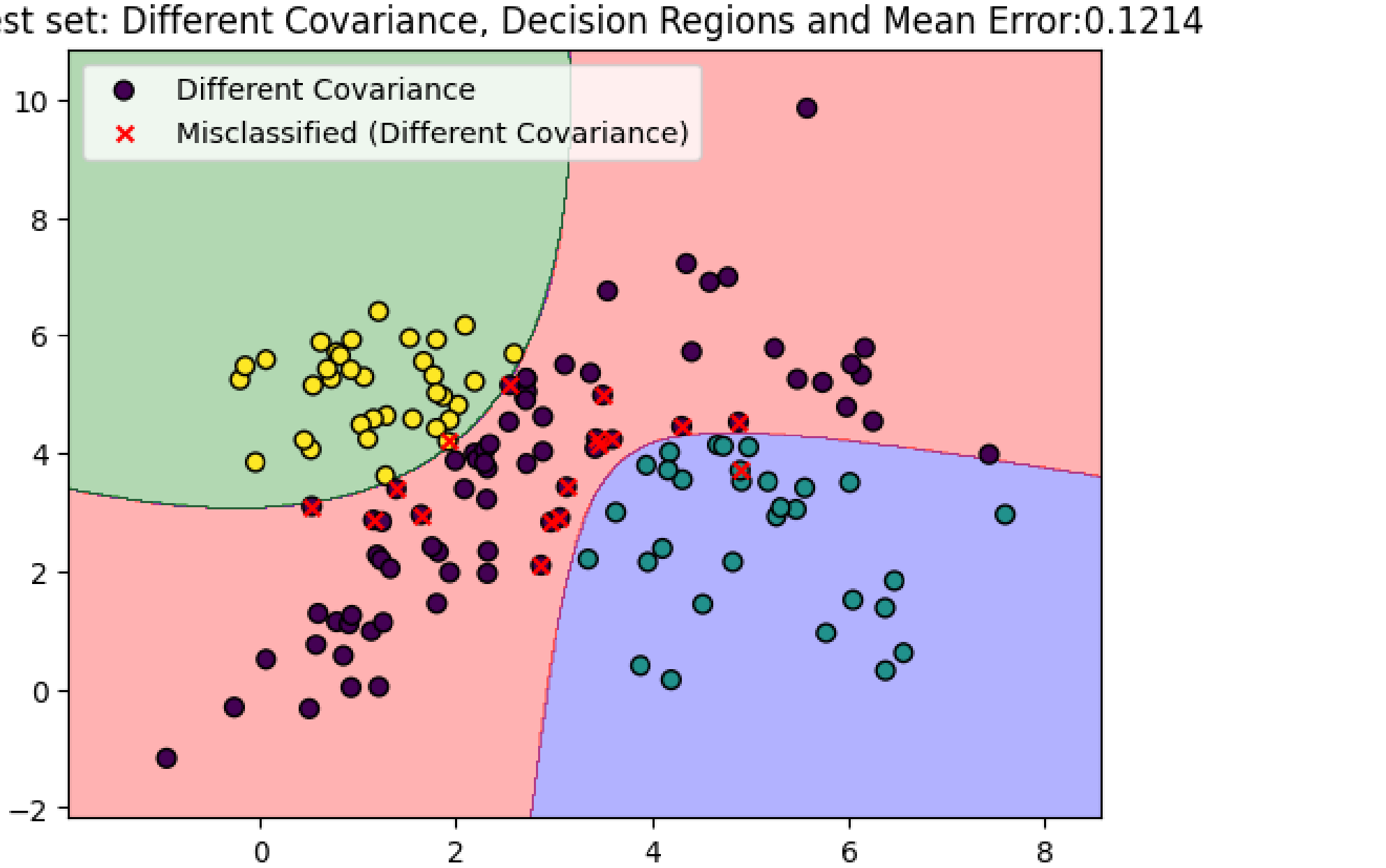
```
# Part A.3
# Define the function for plotting the regions and the misclassified points
def plot_decision_regions_and_misclassified(X, y, classifier, predictions,
                                            title, resolution=0.02):
    h = resolution  # step size in the mesh

    x_min, x_max = X.iloc[:, 0].min() - 1, X.iloc[:, 0].max() + 1
    y_min, y_max = X.iloc[:, 1].min() - 1, X.iloc[:, 1].max() + 1

    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    fig, ax = plt.subplots()

    # Plot decision regions
    if hasattr(classifier, 'predict'):
        Z = classifier.predict(np.c_[xx.ravel(), yy.ravel()])
    else:  # In the case of Naive Bayes where we use logpdf
        Z = np.argmax([d.logpdf(np.c_[xx.ravel(), yy.ravel()]) for d in
                       classifier], axis=0)

    Z = Z.reshape(xx.shape)
```

A.3 - FUNCTION

- **Background Plotting:** Fill decision regions with distinct colors.
- **Scatter Plot:** Display test points with predicted class colors.
- **Misclassified Points:** Identify and mark misclassified points with a red 'x'.
- **Legend and Labels:** Include a legend with the title and mean error information.
- **Mean Error Calculation:** Calculate mean error by comparing true labels with predictions.
- **Plot Title:** Set the plot title to include test set details, decision regions, and mean error.
- **Display Plot:** Show the generated plot.

```
    unique_classes = np.unique(y)
    markers = ['o', '^', 's', 'D', 'v', 'p', '*', 'H', '+', 'x']
    colors = ['red', 'blue', 'green', 'purple', 'orange', 'brown', 'pink',
              'gray', 'cyan', 'yellow']
    cmap_background = ListedColormap(colors[:len(unique_classes)])
    ax.contourf(xx, yy, Z, cmap=cmap_background, alpha=0.3)

    # Plot the test points
    scatter_classifier = ax.scatter(X.iloc[:, 0], X.iloc[:, 1], c=predictions,
                                    cmap='viridis',
                                    edgecolor='k',
                                    s=40,
                                    label=title)

    # Mark misclassified points
    misclassified_indices = y != predictions
    ax.scatter(X.loc[misclassified_indices, 0], X.loc[misclassified_indices, 1],
               marker='x', color='red', s=30, label=f'Misclassified ({title})')

    mean_error = 1 - accuracy_score(y, predictions)
    # Set plot labels and legend
    ax.set_title(f'Test set: {title}, Decision Regions and Mean Error:' +
                 f'{mean_error:.4f}')
    ax.legend()

    # Show the plot
    plt.show()
```


Test set: Same Covariance, Decision Regions and Mean Error: 0.1500
Test set: Same Covariance,
Misclassified (Test set: Same Covariance,)
10
8
6
4
2
0
−2
0
2
4
6
8


Test set: Different Covariance, Decision Regions and Mean Error:0.1214
Different Covariance
Misclassified (Different Covariance)
10
8
6
4
2
0
−2
0
2
4
6
8

A.3 - RESULTS

- The lower mean error in the model with different covariances suggests improved modeling of class-specific variations, indicating that this approach might be more suitable for the specific dataset. However, a comprehensive analysis, considering various metrics and potential trade-offs, will provide a more thorough understanding of model performance.

Same Cov	Different Cov
0.150	0.121

B. K-NN CLASSIFIER

- The goal is to provide insights into the behavior of the k-NN classifier under different parameterizations.
- Emphasize the comparative aspect with the results from the previous part.

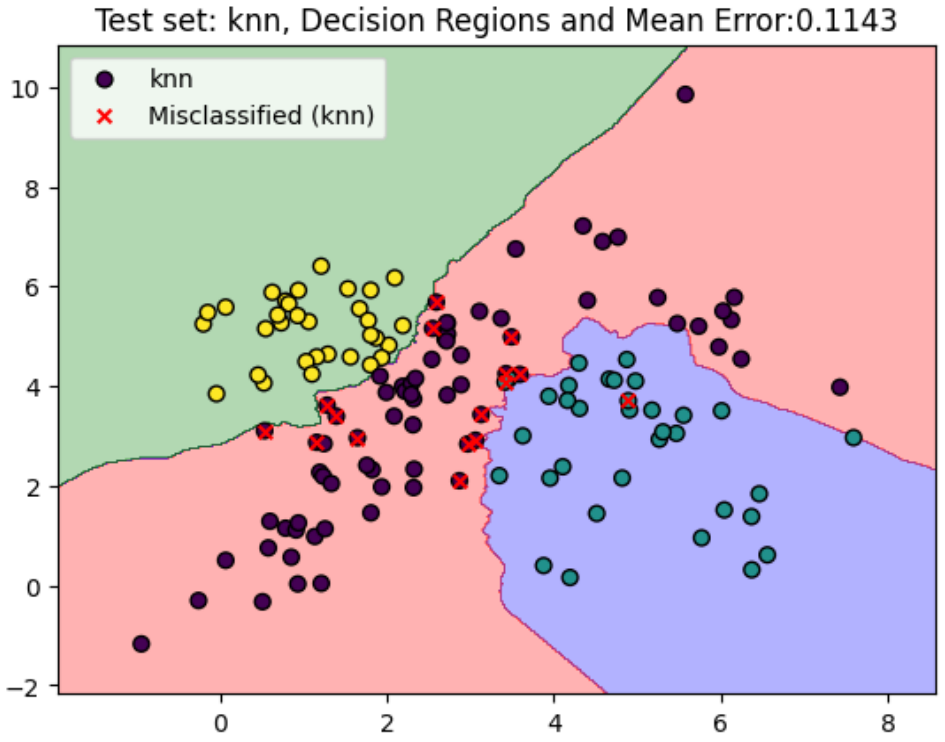
B.1 – CREATE K-NN CLASSIFIER

- **Classifier Type:** k-NN Classifier (Minkowski p=3)
- **Find the Optimal k:**
  - Determine the number of neighbors (k) based on the square root of the length of the sample dataset.
  - If k is divisible by 3, increment it by 1 to ensure diversity.
- **Training:** Fit the classifier with the training data.
- **Predictions:** Use the trained classifier to make predictions on the test set.
- **Error Calculation:** Calculate the mean error by comparing the predicted labels with the true labels. Utilize the accuracy score as the error metric.
- **Result Display:** Print and store the mean error for the case of the same covariance matrix.

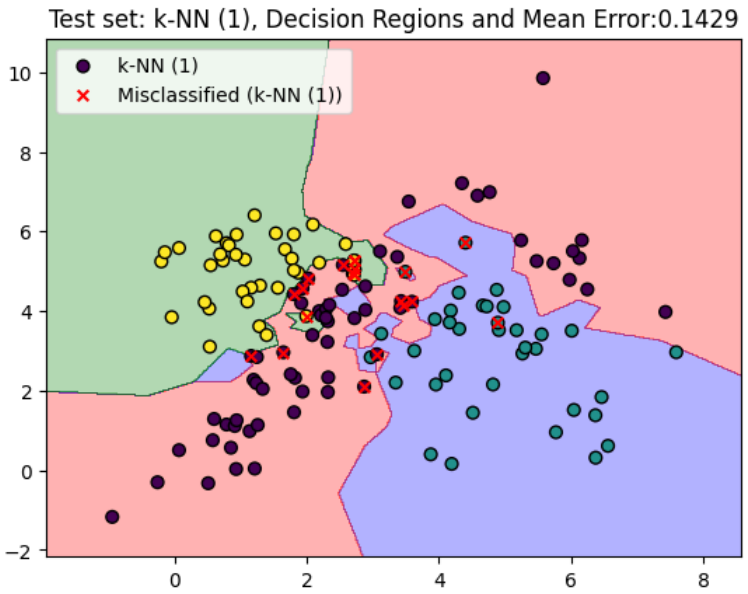
```
# Part B.1
# Find the k (number of neighbors) based on the length of the sample dataset
N = len(df)
k = int(math.sqrt(N))
if k % 3 == 0:
  k += 1

# Train k-NN classifier
knn_classifier = KNeighborsClassifier(n_neighbors=k, p=3)
knn_classifier.fit(X_train, y_train)

# Make predictions on test dataset
yKNN_pred = knn_classifier.predict(X_test)

# Calculate error for k-NN classifier
mean_error_KNN = 1 - accuracy_score(y_test, yKNN_pred)
mean_errors["KNN"] = mean_error_KNN
print(f"Error of KNN: {mean_error_KNN:.3f}")
```

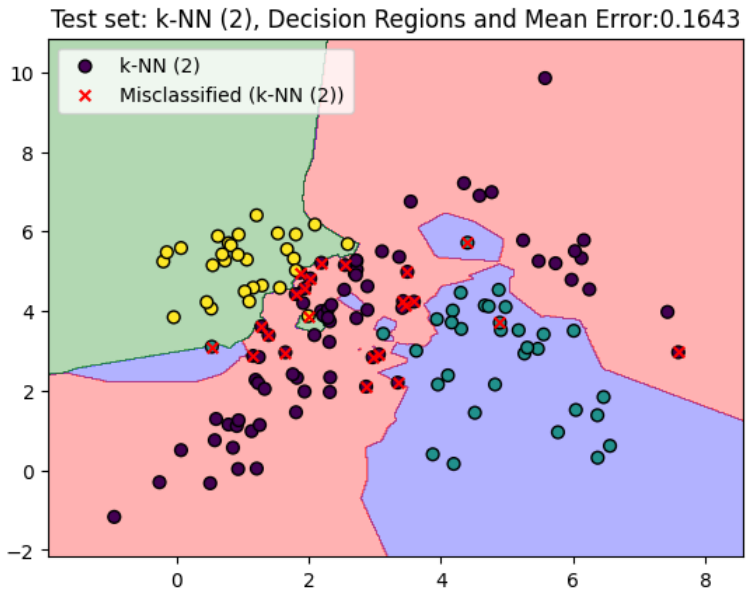
B.2 – PLOT THE TEST DATASET

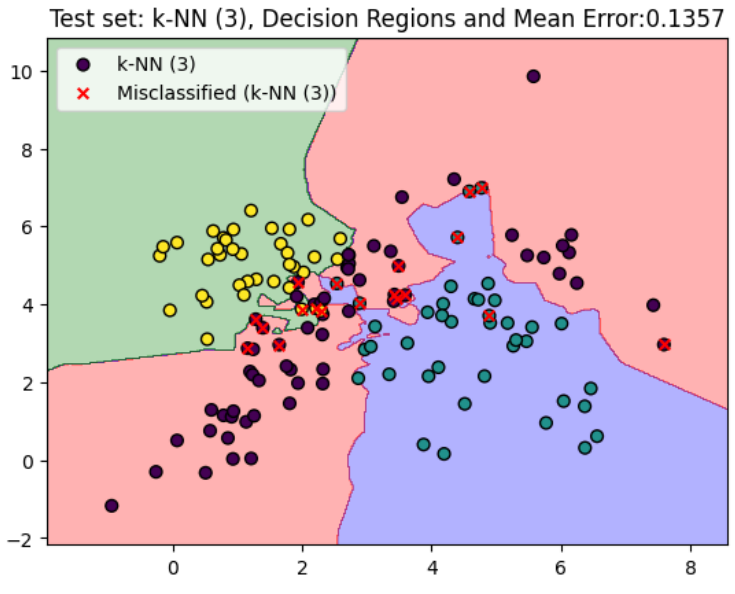
- The function explained in A.3 has been used to display the results.
- It seems that knn is more accurate than all Bayesian Classifiers with Maximum Likelihood Estimation.

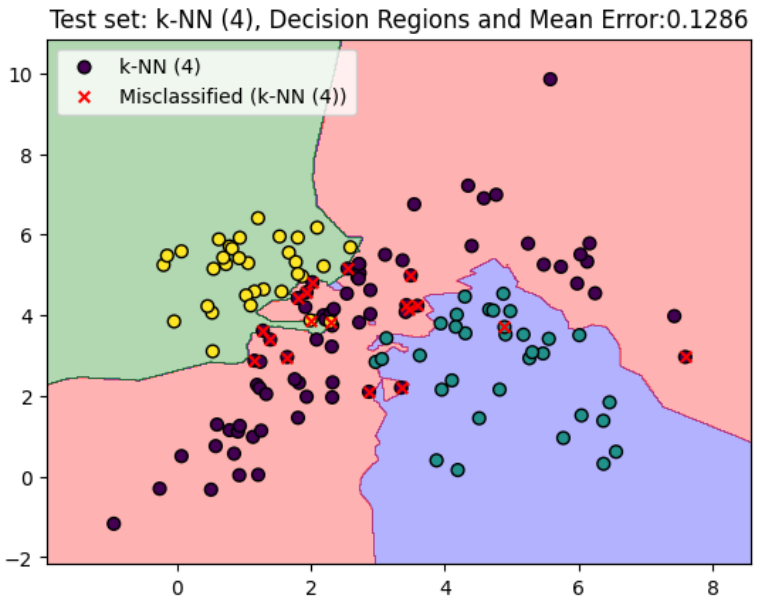


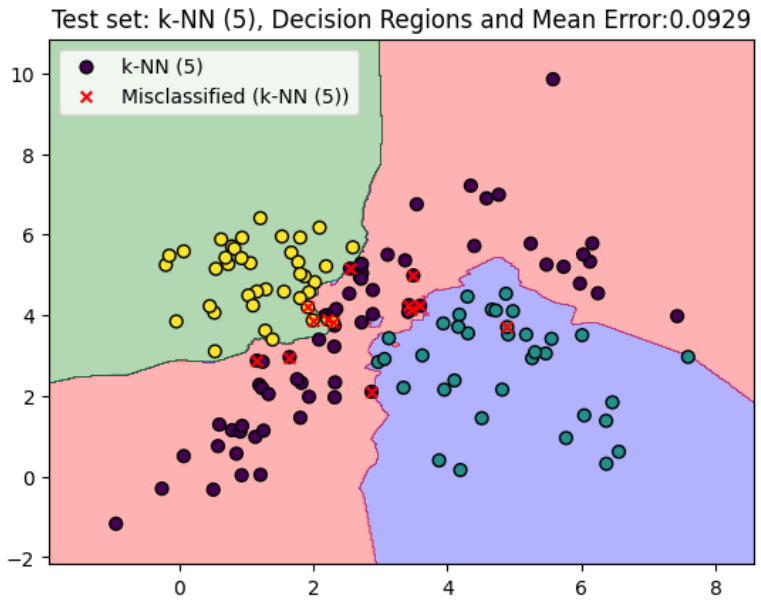
B.3 – PLOT CLASSIFIERS WITH DIFFERENT # OF NEIGHBORS

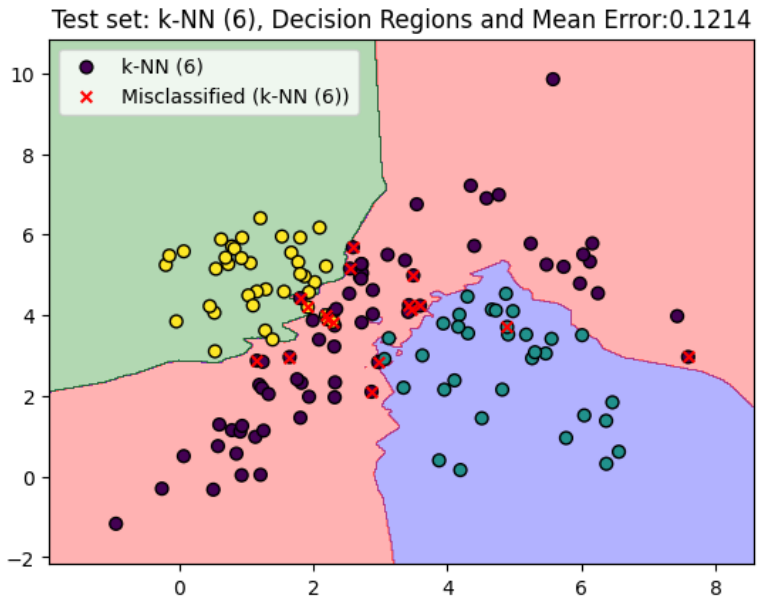


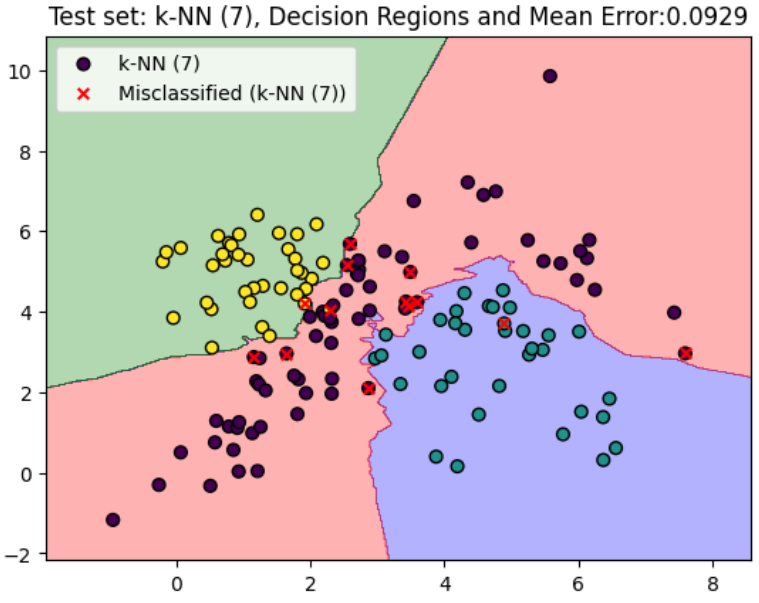


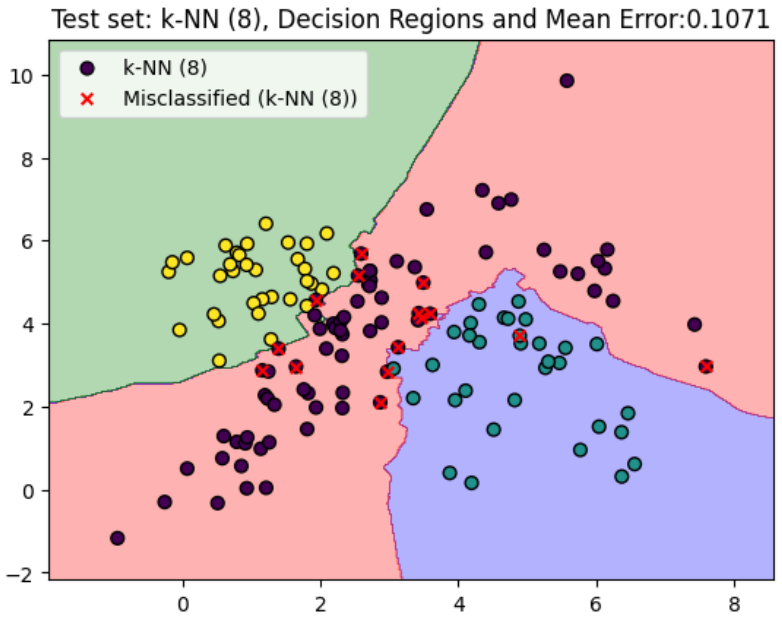


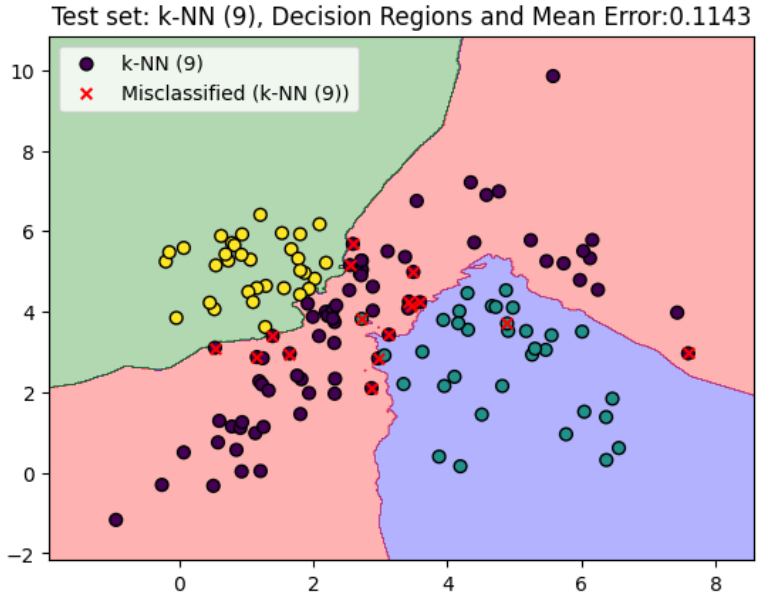

Test set: k-NN (4), Decision Regions and Mean Error:0.1286
k-NN (4)
Misclassified (k-NN (4))


Test set: k-NN (5), Decision Regions and Mean Error:0.0929
k-NN (5)
Misclassified (k-NN (5))

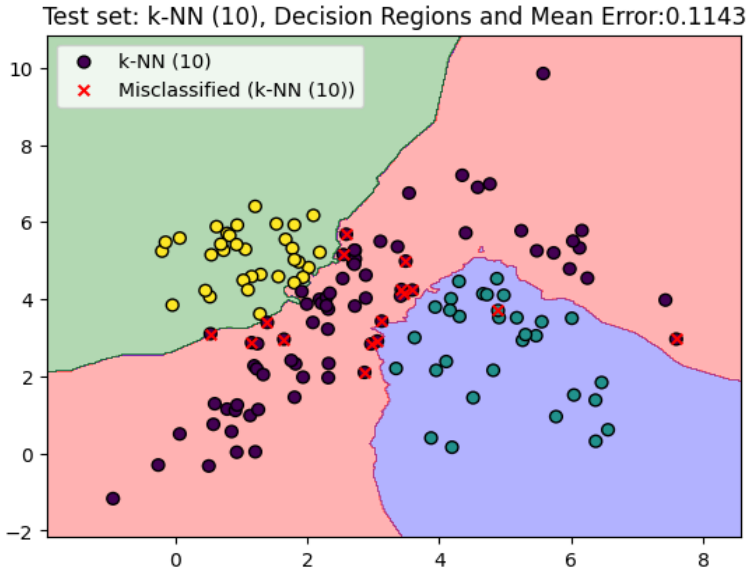

Test set: k-NN (6), Decision Regions and Mean Error:0.1214
k-NN (6)
Misclassified (k-NN (6))


Test set: k-NN (7), Decision Regions and Mean Error:0.0929
k-NN (7)
Misclassified (k-NN (7))


Test set: k-NN (8), Decision Regions and Mean Error:0.1071
k-NN (8)
Misclassified (k-NN (8))


Test set: k-NN (9), Decision Regions and Mean Error:0.1143
k-NN (9)
Misclassified (k-NN (9))

B.3 – PLOT CLASSIFIERS WITH DIFFERENT # OF NEIGHBORS



- The increasing values of k in KNN demonstrate the classic bias-variance trade-off. Smaller values of k may lead to overfitting, while larger values may lead to oversmoothing.
- The observed decrease in mean error suggests that a moderate value of k, such as 5, is providing a good balance between bias and variance in this particular case.
- The KNN classifier with k=5 appears to outperform both Maximum Likelihood Bayes models in terms of mean error.

C. SVM CLASSIFIERS

- The goal is to gain a nuanced understanding of SVM classifiers, their performance with different kernels, and provide a visual representation of the dataset to aid interpretation.
- Emphasize the comparative aspect with the results from the previous part.

C.1 – LINEAR SVM

- **Classifier Type:** Support Vector Machine (SVM) with Linear Kernel.
- **Kernel Choice:** Linear kernel is employed for the SVM classifier.
- **Training:** Train the SVM classifier using the training data (X_train, y_train).
- **Predictions:** Utilize the trained SVM classifier to make predictions on the test set (X_test).
- **Error Calculation:** Calculate the mean error by comparing the predicted labels (ySVMLinear_pred) with the true labels (y_test). The accuracy score is employed as the error metric.
- **Result Display:** Print and store the mean error for the SVM classifier with a linear kernel. The mean error is displayed for analysis and comparison with other classifiers.

```
# Part C.1
# Create an SVM classifier with linear kernel
SVM_classifier = SVC(kernel='linear')
SVM_classifier.fit(X_train, y_train)

# Make predictions on the test set
ySVMLinear_pred = SVM_classifier.predict(X_test)

# Calculate error
mean_error_SVM_linear  = 1 - accuracy_score(y_test, ySVMLinear_pred)
mean_errors["SVM linear"] = mean_error_SVM_linear
print(f"Error of SVM linear: {mean_error_SVM_linear:.3f}")
```

C.2 – RBF SVM

- **Classifier Type:** Support Vector Machine (SVM) with RBF Kernel.
- **Grid Search for Hyperparameter Tuning:** Define a parameter grid (param_grid_rbf) specifying different values for 'C' (regularization parameter) and 'gamma' (kernel coefficient).
- **Model Initialization:** Create an SVM classifier with an RBF kernel (svm_classifier_rbf).
- **Grid Search Execution:**
  - Perform a grid search with cross-validation (GridSearchCV) on the RBF kernel SVM using the defined parameter grid.
  - Evaluate performance using 5-fold cross-validation and the accuracy scoring metric.

```
# Part C.2
# Define the parameter grid for RBF kernel
param_grid_rbf = {
    'C': [0.1, 1, 10],
    'gamma': [0.1, 1, 'scale', 'auto']
}

# Create an SVM classifier with RBF kernel
svm_classifier_rbf = SVC(kernel='rbf')

# Perform grid search for RBF kernel with cross-validation
grid_search_rbf = GridSearchCV(svm_classifier_rbf, param_grid_rbf, cv=5,
                               scoring='accuracy')
grid_search_rbf.fit(X_train, y_train)
```

C.2 – RBF SVM

- **Results Compilation:** Retrieve the results of the grid search and organize them into a DataFrame (results_df). Display the grid search results, including hyperparameters and corresponding mean test scores.
- **Training with Best Parameters:** Train a new SVM classifier with the RBF kernel using the best hyperparameters (best_svm_classifier_rbf).
- **Predictions:** Utilize the trained SVM classifier to make predictions on the test set (X_test).
- **Error Calculation:** Calculate the mean error by comparing the predicted labels (y_pred_rbf) with the true labels (y_test). The accuracy score is employed as the error metric.
- **Result Display:** Display the best hyperparameters for the RBF kernel. Print the error on the test dataset, providing a quantitative measure of the model's predictive accuracy.

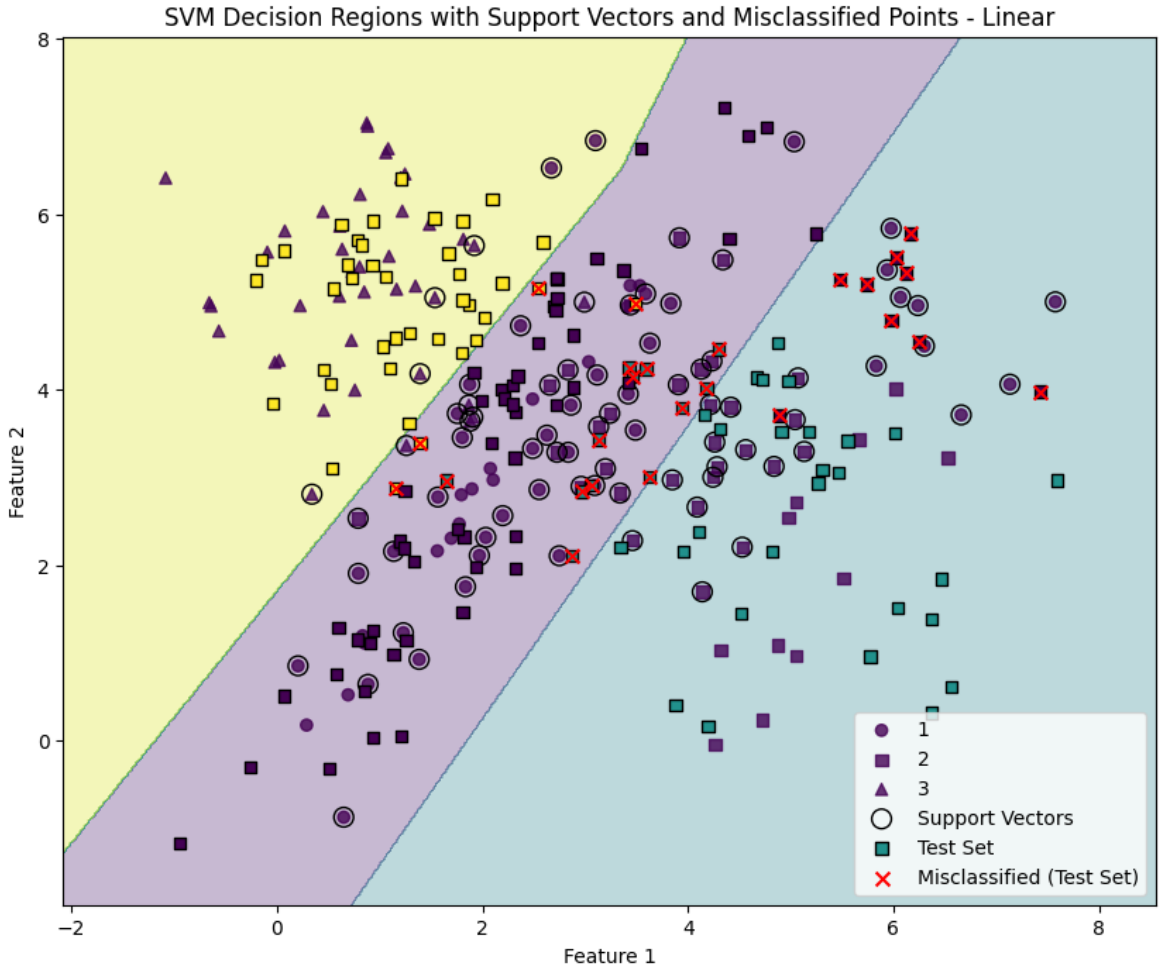
```
# Get the results as a DataFrame
results_df = pd.DataFrame(grid_search_rbf.cv_results_)

# Display the results, including hyperparameters and accuracy
print("Grid Search Results:")
print(results_df[['params', 'mean_test_score']])

# Get the best parameters for RBF kernel
best_params_rbf = grid_search_rbf.best_params_

# Train the model with the best parameters for RBF kernel
best_svm_classifier_rbf = SVC(kernel='rbf', **best_params_rbf)
best_svm_classifier_rbf.fit(X_train, y_train)

# Make predictions on the test data
y_pred_rbf = best_svm_classifier_rbf.predict(X_test)

# Calculate accuracy on the test data
accuracy_test = 1 - accuracy_score(y_test, y_pred_rbf)

print(f"\nBest hyperparameters for RBF kernel: {best_params_rbf}")
print(f"Error on the test dataset: {accuracy_test:.3f}")
```

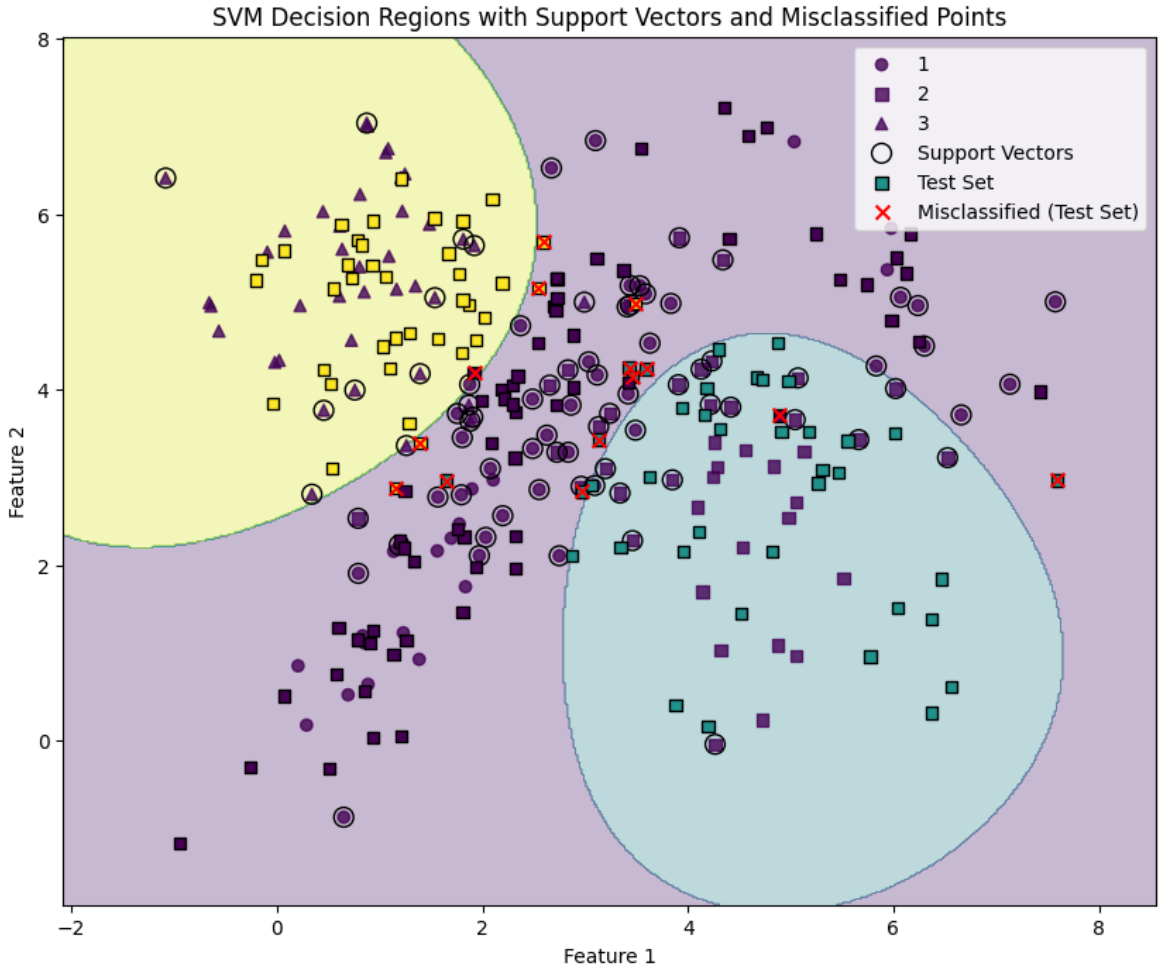
C.2 – RBF SVM RESULTS

```
Error of SVM linear: 0.179
Grid Search Results:
                          params  mean_test_score
0       {'C': 0.1, 'gamma': 0.1}         0.750000
1         {'C': 0.1, 'gamma': 1}         0.628571
2   {'C': 0.1, 'gamma': 'scale'}         0.800000
3    {'C': 0.1, 'gamma': 'auto'}         0.757143
4         {'C': 1, 'gamma': 0.1}         0.864286
5           {'C': 1, 'gamma': 1}         0.814286
6     {'C': 1, 'gamma': 'scale'}         0.835714
7      {'C': 1, 'gamma': 'auto'}         0.835714
8        {'C': 10, 'gamma': 0.1}         0.821429
9          {'C': 10, 'gamma': 1}         0.807143
10   {'C': 10, 'gamma': 'scale'}         0.821429
11    {'C': 10, 'gamma': 'auto'}         0.828571

Best hyperparameters for RBF kernel: {'C': 1, 'gamma': 0.1}
Error on the test dataset: 0.100
```

- As 'C' decreases:
  - **Grid Search Scores:** The mean test scores generally decrease, indicating that lower values of 'C' result in less regularization and potentially overfitting.
  - **Best Hyperparameter Selection:** The grid search selects {'C': 1} as the best-performing 'C' value, striking a balance between regularization and capturing complex patterns.
- As 'gamma' decreases:
  - **Grid Search Scores:** The mean test scores generally decrease, suggesting that lower values of 'gamma' result in a smoother decision boundary.
  - **Best Hyperparameter Selection:** The grid search selects {'gamma': 0.1} as the best-performing 'gamma' value, indicating a preference for a smoother decision boundary.

C.3 – RBF SVM VISUALIZING RESULTS



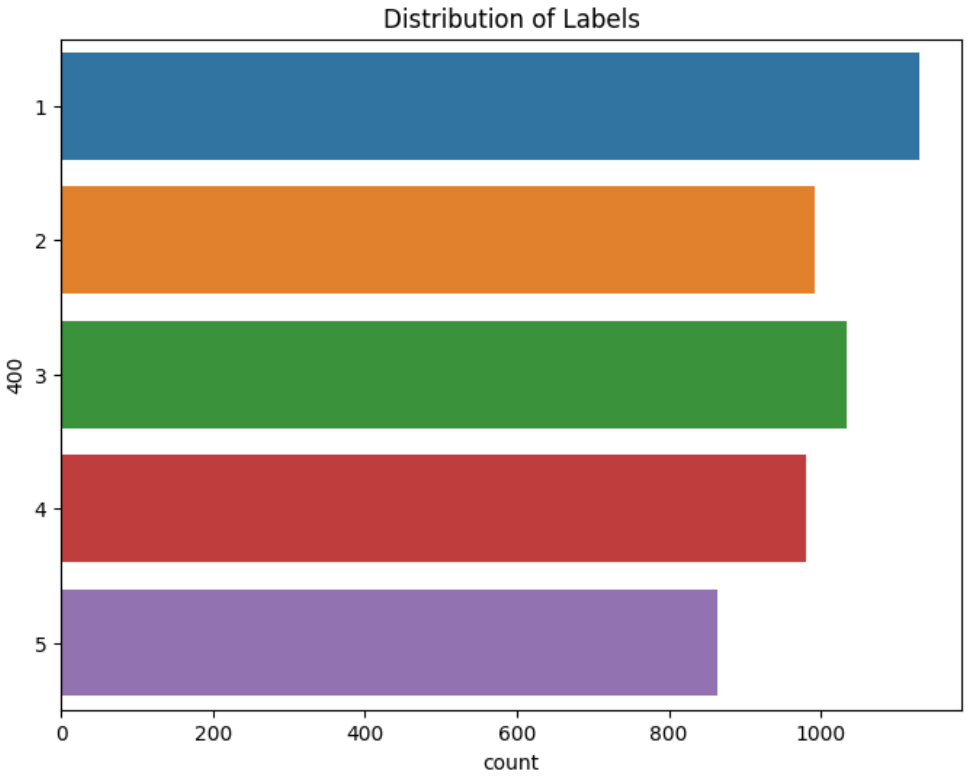


C. CONCLUSION

- The model achieving the lowest error, with a value of 0.093, corresponds to K-Nearest Neighbors (KNN) with k=5, whereas the second lowest error is recorded by the SVM with an RBF kernel, yielding a value of 0.100.
- The SVM with RBF kernel's ability to model complex, non-linear relationships, optimal hyperparameter tuning, robustness to noise, and consideration of global decision boundaries contribute to its superior performance compared to KNN with k=5 in this specific scenario.

D. DATA OVERVIEW

- The dataset is loaded from datasetC.csv file.
- There are 5000 samples.
- Each sample has a 400D vector (features) along with a label (1, 2, 3, 4 or 5).
- The dataset loaded from datasetTestC.csv file will be used as input for the final predictions.
- The datasetC is splitted into train, validation and test datasets
- The datasetC is balanced and there are no missing values.
- Given that the classes are balanced, metrics like accuracy, precision, recall, and F1 score should be informative. The accuracy metric will be used.



D.1 NAIVE BAYES CLASSIFIER

- The Naive Bayes classifier is a probabilistic machine learning model based on Bayes' theorem.
- It assumes that the features are conditionally independent given the class label, which is a simplifying but powerful assumption.
- Naive Bayes is effective in many real-world applications, particularly in text classification and spam filtering.
- It's computationally efficient, easy to implement, and requires relatively small amounts of training data.
- While the "naive" assumption might not hold in all cases, Naive Bayes can still perform surprisingly well, especially when dealing with high-dimensional data or situations where the assumption of independence is reasonable.

Mean_test_score
0.794

D.2 LOGISTIC REGRESSION

- Logistic Regression is a statistical method used for binary and multiclass classification problems.
- Despite its name, it is a classification algorithm rather than a regression algorithm.
- Logistic Regression models the probability that a given instance belongs to a particular class.
- It employs the logistic function (sigmoid function) to transform a linear combination of features into values between 0 and 1, representing probabilities.

D.2 LOGISTIC REGRESSION - HYPERPARAMETER TUNING

1. **C:** Inverse of regularization strength; smaller values specify stronger regularization, helping prevent overfitting. Increase 'C' for less regularization (allowing the model to fit the training data more closely), decrease for stronger regularization (to generalize better to unseen data).
2. **solver :** Optimization algorithm to use in the optimization problem; 'liblinear' is suitable for small datasets or when dealing with binary classification, while 'lbfgs' is efficient for large datasets and multiclass problems.
3. **max_iter:** Maximum number of iterations for the solver to converge; increase if convergence is not reached within the default number of iterations.

```
param_grid_lr = {
    'C': [0.001, 0.01, 0.1, 1, 10],
    'solver': ['liblinear', 'lbfgs'],
    'max_iter': [100, 200, 300]
}
```

c	solver	Max_iter	Mean_test_score
0.01	lbfgs	100	0.818286
0.01	lbfgs	200	0.818286
…	…	…	…
10	lbfgs	100	0.722571

D.3 K-NEAREST NEIGHBORS (KNN) CLASSIFIER

- K-Nearest Neighbors (KNN) is a simple and intuitive classification algorithm that operates based on the idea of proximity, assigning labels to data points by considering the majority class among their k nearest neighbors.
- KNN relies on distance metrics, commonly using Euclidean distance, to measure the similarity between data points. The choice of distance metric is crucial and depends on the nature of the data.
- Number of Neighbors (k): The hyperparameter 'k' determines the number of neighbors considered during classification. A smaller 'k' may lead to more flexible decision boundaries, but it can also make the model sensitive to noise.

D.3 KNN - HYPERPARAMETER TUNING

- **n_neighbors:** Increasing: Smoother decision boundary, less sensitivity to local noise. Decreasing: More sensitivity to local variations, potential overfitting.
- **Weights**:
  - 'uniform': Treats all neighbors equally, uniform voting.
  - 'distance': Gives more weight to closer neighbors, emphasizing local patterns.
- **P**:
  - 'p=1' (Manhattan): Angular decision boundary, considers absolute differences.
  - 'p=2' (Euclidean): Smoother boundary, straight-line distance.

```
param_grid_knn = {
    'n_neighbors': [3, 5, 7, 9],
    'weights': ['uniform', 'distance'],
    'p': [1, 2]
}
```

n_neighbors	weights	p	mean_test_score
9	Distance	2	0.741714
7	Distance	2	0.734571
…	…	…	…
3	Uniform	1	0.643143

Best score in extended search

```
Best Parameters:  {'n_neighbors': 23, 'p': 2, 'weights': 'distance'}
Best Accuracy on Training Set:  0.7545714285714287
```

D.4 SUPPORT VECTOR MACHINE

- Support Vector Machines (SVM) are powerful supervised learning algorithms designed for classification and regression tasks.
- SVMs excel in high-dimensional datasets and are particularly effective when dealing with complex decision boundaries.
- As an ensemble method, SVMs make predictions by finding the optimal hyperplane that maximizes the margin between different classes.
- SVMs can handle both linear and non-linear relationships through the use of various kernel functions.
- One key strength of SVMs lies in their ability to generalize well to new, unseen data, making them suitable for diverse applications in machine learning.

D.4 SVM - HYPERPARAMETER TUNING

- **C**: the penalty parameter, which represents misclassification or error term. The misclassification or error term tells the SVM optimization how much error is bearable. This is how you can control the trade-off between decision boundary and misclassification term. when C is high it will classify all the data points correctly, also there is a chance to overfit.
- **kernel**: The main function of the kernel is to take low dimensional input space and transform it into a higher-dimensional space. It is mostly useful in non-linear separation problem.
- **gamma**: It defines how far influences the calculation of plausible line of separation. when gamma is higher, nearby points will have high influence; low gamma means far away points also be considered to get the decision boundary.

```
param_grid_svm = {
    'C': [0.1, 1, 10],
    'kernel': ['linear', 'rbf', 'poly'],
    'gamma': ['scale', 'auto']
}
```

C	Kernel	Gamma	mean_test_score
1	scale	rbf	0.841143
10	scale	rbf	0.840286
10	auto	rbf	0.838857
1	auto	rbf	0.8388571
…	…	…	…
0.1	scale	poly	0.229429
0.1	auto	poly	0.227429

D.5 MULTIPLAYER PERCEPTRON (MLP) CLASSIFIER

- Multilayer Perceptron (MLP) Classifier is a versatile neural network algorithm used for both classification and regression tasks.
- It is a type of feedforward artificial neural network with an input layer, one or more hidden layers, and an output layer.
- MLPs can learn complex relationships in data, making them suitable for tasks with intricate patterns.
- They use backpropagation for training, adjusting weights to minimize the difference between predicted and actual values.
- While MLPs can model non-linear relationships effectively, they may require careful tuning of hyperparameters to prevent overfitting and achieve optimal performance.

D.5 MLP - HYPERPARAMETER TUNING

- **hidden_layer_sizes:** Configures the neural network architecture by specifying the size of hidden layers, influencing the model's capacity to learn intricate patterns.
- **activation:** Determines the activation function for hidden layers, introducing non-linearities crucial for capturing complex relationships in the data.
- **solver:** Chooses the optimization algorithm (e.g., 'adam', 'sgd') for weight updates during training, impacting the efficiency and effectiveness of the learning process.
- **alpha:** Controls overfitting through L2 regularization, penalizing large weights and ensuring a balance between fitting the data and preventing excessive complexity.

```
param_grid_mlp = {
    'hidden_layer_sizes': [(100,), (50, 50), (50, 30, 20)],
    'activation': ['relu', 'tanh'],
    'solver': ['adam', 'sgd'],
    'alpha': [0.0001, 0.001, 0.01],
}
```

hidden_layer_sizes	activation	solver	alpha	mean_test_score
(100,)	relu	adam	0.01	0.8120
(100,)	relu	adam	0.0001	0.8102
…	…	…	…	…
(50, 30, 20)	tanh	sgd	0.001	0.7683

D.6 RANDOM FOREST CLASSIFIER

- Random forests are for supervised machine learning, where there is a labeled target variable.
- Random forests can be used for solving regression (numeric target variable) and classification (categorical target variable) problems.
- Random forests are an ensemble method, meaning they combine predictions from other models.
- Each of the smaller models in the random forest ensemble is a decision tree.

D.6 RANDOM FOREST - HYPERPARAMETER TUNING

- **n_estimators**: the number of decision trees in the forest. Increasing this hyperparameter generally improves the performance of the model but also increases the computational cost of training and predicting.
- **max_depth**: the maximum depth of each decision tree in the forest. Setting a higher value for max_depth can lead to overfitting while setting it too low can lead to underfitting.
- **min_samples_split**: determines the minimum number of decision tree observations in any given node in order to split.
- **min_samples_leaf:** specifies the minimum amount of samples that a node must hold after getting split. It also helps to reduce overfitting when we have ample amount of parameters.

```
param_grid_rf = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
}
```

Max_depth	Min_samples_leaf	Min_samples_Split	N_estimators	Mean_test_score
20	1	10	200	0.732000
20	4	10	200	0.730000
…	…	…	…	…
20	1	2	50	0.624571

Best score in extended search

```
Best Parameters:  {'max_depth': 40, 'min_samples_leaf': 4, 'min_samples_split': 10, 'n_estimators': 650}
Best Accuracy on Training Set:  0.7485714285714286
```

D.7 XGBOOST

- XGBoost is a supervised machine learning algorithm suitable for both regression and classification tasks.
- As an ensemble method, it combines predictions from multiple decision trees to create a robust and accurate model. Each individual model within the XGBoost ensemble is a decision tree.
- The algorithm employs gradient boosting to build trees sequentially, correcting errors from previous iterations.
- Notable features of XGBoost include regularization for controlling model complexity and effective handling of missing values during training.
- Unlike some other classifiers, XGBoost assumes that the class labels should be integers starting from 0.

D.7 XGBOOST - HYPERPARAMETER TUNING

- **n_estimators:** It determines the total number of trees in the ensemble.
- **learning_rate:** It scales the contribution of each tree. A lower learning rate requires more boosting rounds but can lead to a more robust model.
- **max_depth:** It controls the maximum depth of each tree in the ensemble. Deeper trees can capture more complex patterns but may lead to overfitting.
- **subsample:** It determines the fraction of training data to be used in each boosting round. A value less than 1.0 introduces randomness and helps prevent overfitting.
- **colsample_bytree:** It controls the fraction of features (columns) to be used in each boosting round. It introduces randomness to prevent overfitting and improve generalization.

```
param_grid_xgb = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 4, 5],
    'subsample': [0.8, 0.9, 1.0],
    'colsample_bytree': [0.8, 0.9, 1.0]
}
```

n_estimators	learning_rate	max_depth	subsample	colsample_bytree	mean_test_score
200	0.2	3	0.8	0.9	0.79028
200	0.2	3	0.8	0.8	0.78828
200	0.2	3	0.8	1.0	0.78400
…	…	…	…	…	…

D.8 VOTING CLASSIFIER

- Ensemble methods, such as the Voting Classifier, combine multiple individual models to improve overall predictive performance.
- The Voting Classifier aggregates the predictions of diverse base models, such as Decision Trees, Support Vector Machines, or Logistic Regression, and produces a final prediction based on a majority vote or weighted combination.
- This approach enhances robustness, generalization, and mitigates the risk of overfitting, making it particularly effective in scenarios where individual models may have limitations.
- Ensemble methods are widely employed in machine learning for achieving higher accuracy and stability across a variety of applications.
- In ensemble models, it's beneficial to include diverse base models that have different strengths and weaknesses.

D.8 VOTING CLASSIFIER

- Soft Voting: The final prediction is based on the average probability across all base models. Soft voting is often preferred when base models can provide probability estimates, as it takes into account the confidence of each model's predictions.
- Hard voting: The final prediction is based on the majority vote of the base models.

```
Best Parameters:  {'rf__n_estimators': 100, 'svm__C': 10}
Best Accuracy on Training Set:  0.8428571428571144
Accuracy on Validation Set with the best model:  0.84
Accuracy on Test Set with the best model:  0.844
```

```
Best Parameters:  {'rf__n_estimators': 200, 'svm__C': 10}
Best Accuracy on Training Set:  0.74
Accuracy on Validation Set with the best model:  0.737333
Accuracy on Test Set with the best model:  0.744
```

```python
# Define the base models
model_svm = SVC(probability=True)
model_rf = RandomForestClassifier(random_state=42)

# Define the ensemble model
model_vc = VotingClassifier(estimators=[('svm', model_svm), ('rf', model_rf)],
                            voting='soft')
```

D.9 STACKING

- Stacking is an ensemble learning technique that involves combining the predictions of multiple base models to create a more powerful and accurate meta-model.
- This approach leverages the diverse strengths of individual models, mitigating their weaknesses and enhancing overall predictive performance.
- Stacking is particularly effective when different models excel in capturing distinct patterns or nuances within the data, providing a comprehensive and robust solution to complex prediction tasks.
- The process typically consists of two stages: in the first stage, various base models are trained independently on the training data; in the second stage, a meta-model is trained to make predictions based on the outputs of the base models.

D.10 BAGGED DECISION TREES

- Bagged Decision Trees, an ensemble technique, involve training multiple instances of a base Decision Tree model on different bootstrap samples of the training data.
- By aggregating the predictions of individual trees through voting, Bagged Decision Trees enhance model robustness and generalization, reducing the risk of overfitting.
- BDT, or Bootstrap Aggregating of Decision Trees, is primarily employed to reduce overfitting and enhance the robustness of decision tree models.
- The main parameter to consider is the number of base decision trees (n_estimators), which controls the diversity and strength of the ensemble.
- This technique is particularly beneficial when dealing with high-variance models, and its effectiveness becomes pronounced in scenarios where individual decision trees are prone to overfitting.

Classifiers					Mean_error
Voting (Soft):	SVC	RandomForestClassifier	-	-	0.848
Voting (Soft):	SVC	-	MLPClassifier	-	0.833
Voting (Soft):	SVC	RandomForestClassifier	MLPClassifier	-	0.825
Voting (Hard):	SVC	RandomForestClassifier	MLPClassifier	XGBClassifier	0.824
Voting (Soft):	SVC	RandomForestClassifier	MLPClassifier	XGBClassifier	0.848
Voting (Soft):	SVC	RandomForestClassifier	-	XGBClassifier	0.853
Voting (Soft):	SVC	-	-	XGBClassifier	0.848
Stacking	OneVsRest	RandomForestClassifier	MLPClassifier	Logistic Regression (Final)	0.849
Bagged Decision Trees	DecisionTreeClassifier				0.688
Stacking	SVC	RandomForestClassifier	KNN	RandomForestClassifier (Final)	0.858
AdaBoostClassifier	DecisionTreeClassifier				0.356

THANKS FOR YOUR TIME!!