Pattern Recognition and Machine Learning

PA4-Report

Soumen Kumar
Roll No-B22ES006

Google Colab Notebook Link

Access my notebook: PA4-B22ES006.ipynb.

Question-1: Linear Discriminant Analysis

1 Introduction

This part of the report presents the detailed analysis and computations performed in the Question 1 of PRML Lab Assignment-4. For the first question, the Python code calculates various terms related to Linear Discriminant Analysis (LDA) using given input data.

2 Computed Terms

2.1 Difference of Class-Wise Means

The difference of class-wise means for binary class classification is computed as:

$$m_1 - m_2 = \frac{1}{N_1}\sum_{i=1}^{N_1} x_i^{(1)} - \frac{1}{N_2}\sum_{i=1}^{N_2} x_i^{(2)}$$

where:

- N_1 is the number of samples in class 1.
- N_2 is the number of samples in class 2.
- $x_i^{(1)}$ represents the ith sample from class 1.
- $x_i^{(2)}$ represents the ith sample from class 2.

2.2 Total Within-class Scatter Matrix S_W

The total within-class scatter matrix for binary class classification is computed as:

$$S_W = \sum_{i=1}^{N_1}(x_i^{(1)} - m_1)(x_i^{(1)} - m_1)^T + \sum_{i=1}^{N_2}(x_i^{(2)} - m_2)(x_i^{(2)} - m_2)^T$$

where:

- m_1 is the mean vector of class 1.
- m_2 is the mean vector of class 2.
- $x_i^{(1)}$ represents the ith sample from class 1.
- $x_i^{(2)}$ represents the ith sample from class 2.

2.3 Between-class Scatter Matrix S_B

The between-class scatter matrix for binary class classification is computed as:

$$S_B = \sum_{i=1}^{N} (\mathbf{m}_i - \mathbf{m})(\mathbf{m}_i - \mathbf{m})^T$$

Where:

- N is the number of classes (in this case, 2).
- $\mathbf{m}_i$ is the mean vector of class i.
- $\mathbf{m}$ is the overall mean vector.

2.4 EigenVectors of $S_W^{-1} S_B$

The eigenvectors of the matrix $S_W^{-1} S_B$ corresponding to the highest eigenvalue for binary class classification are computed using the same method as before. These eigenvectors represent the directions in which the data varies the most between classes.

The eigenvectors of a matrix are computed as follows:

$$\text{EigenVectors} = \text{eigenvectors of } (S_W^{-1} S_B)$$

2.5 Projection According to LDA

For binary class classification, the projection of a given 2-D point according to LDA is computed as:

$$\text{Projection} = w^T x$$

where:

- w is the LDA projection vector.
- x is the input 2-D point.

3 Task 2: Projection Vector(LDA)

Linear Discriminant Analysis (LDA) is a dimensionality reduction technique commonly used for feature projection and classification tasks. Linear Discriminant Analysis (LDA) is a dimensionality reduction technique used in machine learning to find the linear combination of features that best separates two or more classes of objects or events. The goal of LDA is to project the features in the dataset onto a lower-dimensional space with good class separability to avoid overfitting and reduce computational costs. In this report, we visualize the LDA projection vector on a plot to understand its discriminant direction for binary classification.

3.1 Data Description

The dataset used in this analysis consists of 2-dimensional samples with binary class labels. It contains a total of [2000] samples, each described by [2] features.

3.2 Steps of LDA

The Linear Discriminant Analysis (LDA) algorithm involves the following steps:

1. **Compute the mean vectors for each class**: For each class in the dataset, calculate the mean vector by taking the average of the feature values for all samples belonging to that class. This results in a mean vector for each class, representing the central tendency of the feature values within that class.

2. : The within-class scatter matrix measures the spread of the data within each class. It is calculated by summing the scatter matrices of individual classes. The scatter matrix for each class is computed by taking the outer product of the deviation of each sample from the class mean.

3. **Compute the between-class scatter matrix (S_B): The between-class scatter matrix quantifies the spread between the class means**. It is computed by taking the outer product of the deviation of each class mean from the overall mean. This matrix captures the separation between different classes.

4. **Compute the eigenvalues and eigenvectors of the matrix** ($S_W^{-1}S_B$): The LDA projection vector is obtained by solving the generalized eigenvalue problem $(S_W^{-1}S_B)v = \lambda v$, where v represents the eigenvector and λ is the corresponding eigenvalue. These eigenvectors represent the directions in which the data is most discriminative.

5. **Select the eigenvector corresponding to the largest eigenvalue as the projection vector** : After computing the eigenvectors and eigenvalues, select the eigenvector corresponding to the largest eigenvalue. This

eigenvector represents the direction of maximum separation between the classes and is chosen as the LDA projection vector.

Following these steps, LDA identifies the optimal linear combination of features that maximizes the separation between different classes in the dataset, facilitating effective dimensionality reduction and classification.

3.3 LDA Projection Vector Visualization

To visualize the LDA projection vector:

- We compute the LDA projection vector using the `GetLDAProjectionVector` function.
- Original data points are plotted on a scatter plot, with each class represented by a different color.
- The LDA projection vector is scaled and plotted as an arrow on the same plot.

```
# Optimized graph
X_float = X.astype(float)
class_0_samples = X_float[X_float[:, -1] == 0][:, :-1]
class_1_samples = X_float[X_float[:, -1] == 1][:, :-1]

# Computing the mean of each class
mean_0 = np.mean(class_0_samples, axis=0)
mean_1 = np.mean(class_1_samples, axis=0)

# Plotting
plt.figure(figsize=(8, 6))
plt.scatter(class_0_samples[:, 0], class_0_samples[:, 1], color='red', label='Class 0')
plt.scatter(class_1_samples[:, 0], class_1_samples[:, 1], color='blue', label='Class 1')
plt.scatter(*mean_0, color='black', marker='x', s=100, label='Mean Class 0')
plt.scatter(*mean_1, color='black', marker='o', s=100, label='Mean Class 1')

# Scaling the LDA projection vector for visualization
w=GetLDAProjectionVector(X)
proj_line = w * 10

# LDA direction
plt.quiver(0, 0, proj_line[0], proj_line[1], color='green', scale=1, scale_units='xy', angles='xy', width=0.005, label='LDA Projection Vector')

plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('LDA Projection Vector Visualization')
plt.legend()
plt.grid(True)
plt.axis('equal')
plt.show()
```

This code segment generates a plot where the data points of each class are coloured differently, and the LDA projection vector is shown as a green arrow. This visualization aids in understanding how the LDA projection vector separates the data points into different classes.

The scatter plot shows the data points of different classes. Each class is represented by a different color: red for Class 0 and blue for Class 1. Additionally, the plot includes the LDA projection vector, shown as a green arrow. This visualization helps understand how the LDA projection vector separates the data points into different classes.

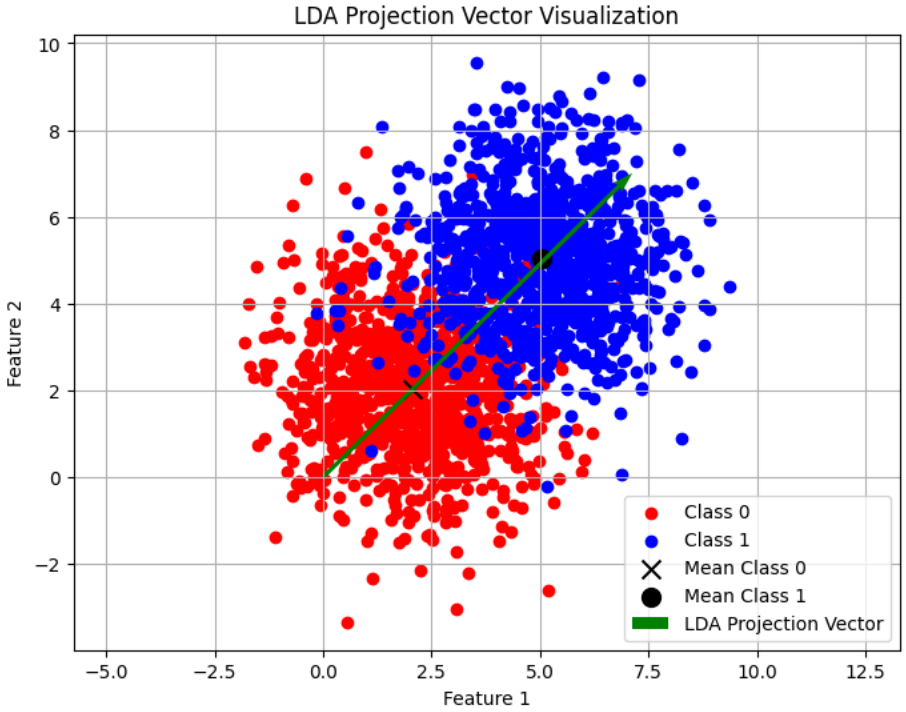

Figure 1: LDA Projection Vector Visualization

3.4 Results and Observations

From the visualization:

- We observe that the LDA projection vector points in a direction that maximizes the separation between the two classes.
- The data points belonging to different classes are well-separated along the direction of the projection vector.

3.5 Conclusion

The LDA projection vector provides a clear visual representation of the discriminant direction for binary classification tasks. It helps understand how the features contribute to class separation and aids in building more effective classification models.

4 Task3: Comparing Performance of 1-NN Classifier

In this part, we analyze the performance of a 1-NN classifier on both the original dataset and the dataset after applying Linear Discriminant Analysis (LDA) for feature projection.

4.1 Data Description

The dataset used in this analysis is also the same one(consisting of 2000 samples of 2D points and their corresponding binary labels) used in the previous task.

4.2 1-NN Classifier

The 1-NN classifier is implemented as follows:

```
class OneNNClassifier:
    def __init__(self):
        pass

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train

    def predict(self, X_test):
        y_pred = []
        for x_test in X_test:
            nearest_neighbor_index = self.
    find_nearest_neighbor_index(x_test)
            nearest_neighbor_label = self.y_train[
    nearest_neighbor_index]
            y_pred.append(nearest_neighbor_label)
        return np.array(y_pred)

    def find_nearest_neighbor_index(self, x_test):
        min_distance = float('inf')
        nearest_neighbor_index = None
        for i, x_train in enumerate(self.X_train):
            distance = np.linalg.norm(x_test - x_train)
            if distance < min_distance:
                min_distance = distance
                nearest_neighbor_index = i
        return nearest_neighbor_index
```

The `OneNNClassifier` class represents a 1-Nearest Neighbor (1-NN) Classifier.

- **Methods:**
  - `fit(X_train, y_train)`: Fit the classifier to the training data.
  - `predict(X_test)`: Predict the labels for the test data.
  - `find_nearest_neighbor_index(x_test)`: Find the index of the nearest neighbor for a given test instance.

4.3 Data Loading and Preprocessing(Normalization)

The dataset is loaded and preprocessed as follows:

```
# Loading the dataset
data_url = 'https://raw.githubusercontent.com/anandmishra22/PRML-Spring-2023/main/programmingAssignment/PA-4/data.csv'
df = pd.read_csv(data_url,header=None)

# Extracting features and labels
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values

# Splitting the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=19)

# Standardizing the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

4.4 Training and Testing

The 1-NN classifier is trained and tested on both the original and LDA-projected datasets:

```
# Train and test 1-NN classifier on the original data
knn= OneNNClassifier()
knn.fit(X_train, y_train)
predictions_original = knn.predict(X_test)
accuracy_original = accuracy_score(y_test, predictions_original)

# Performing (LDA) for feature projection
lda = LinearDiscriminantAnalysis(n_components=1)
X_train_lda = lda.fit_transform(X_train, y_train)
X_test_lda = lda.transform(X_test)

# Train and test 1-NN classifier on the LDA projected data
knn_lda = OneNNClassifier()
knn_lda.fit(X_train_lda, y_train)
predictions_lda = knn_lda.predict(X_test_lda)
accuracy_lda = accuracy_score(y_test, predictions_lda)
```

4.5 Results and Observations

The accuracy of the 1-NN classifier on the original and LDA-projected datasets is summarized in Table 1.

Table 1: Accuracy of 1-NN Classifier

Dataset	**Accuracy**
Original Data	0.8925
LDA Projected Data	0.9100

Based on the experiment, the 1-NN classifier demonstrated a slightly higher accuracy when applied to the LDA-projected data than the original dataset. Specifically, the accuracy achieved with the LDA-projected data was 0.91, whereas it was 0.8925 with the original data.

This suggests that the LDA projection helped improve the discriminative power of the features, leading to better classification performance. However, the difference in accuracy between the original and projected datasets was relatively small, indicating that the original features already contained valuable information for classification.

4.6 Conclusion

So In this part, we evaluated the performance of a 1-NN classifier on a binary classification task using both the original dataset and the dataset after applying LDA for feature projection. In conclusion, while LDA projection can enhance the separability of classes and improve classification accuracy, the effectiveness of the improvement may depend on the specific characteristics of the dataset and the chosen classification algorithm.

Question-2: Naive Bayes Analysis

1 Introduction

In this part of report,we analyse the Naive Bayes algorithm applied to a given dataset. We perform various tasks, including data loading, exploration, probability calculations, and predictions. Laplace smoothing is also applied to handle zero probabilities effectively.

2 Task 0: Data Loading and Exploration

The dataset is loaded from a given URL using pandas. Categorical distributions are visualized using Seaborn's countplot.

```
1 url = "https://raw.githubusercontent.com/anandmishra22/PRML-Spring
      -2023/main/programmingAssignment/PA-4/naive_bayes.csv"
```

```
data = pd.read_csv(url)
# Displaying the dataset
display(data)
# Plotting categorical distributions
categorical_features = ['Outlook', 'Temp', 'Humidity', 'Windy', '
    Play']
for feature in categorical_features:
    plt.figure(figsize=(8, 6))
    sns.countplot(data=data, x=feature)
    plt.title(f"Distribution of {feature}")
    plt.xlabel(feature)
    plt.ylabel("Count")
    plt.show()

#Task_0
# Splitting dataset into train and test sets
train_set, test_set = train_test_split(data, test_size=2,
    random_state=29)
print("Train set:")
display(train_set)
print("\nTest set:")
display(test_set)
```

3 Task 1: Calculating Prior Probabilities

In Task 1, we are calculating the prior probabilities for the classes in the dataset. Prior probabilities represent the likelihood of each class occurring in the dataset before considering any specific features.

```
#Task1
# Calculating prior probabilities
prior_counts = train_set['Play'].value_counts()
total_samples = prior_counts.sum()

p_play_yes = prior_counts.get('yes', 0) / total_samples
p_play_no = prior_counts.get('no', 0) / total_samples

print("Prior Probabilities:")
print("P(Play=yes):", p_play_yes)
print("P(Play=no):", p_play_no)
```

Table 2: Prior Probabilities

Class	**Probability**
Play=yes	0.6667
Play=no	0.3333

The table above shows the calculated prior probabilities for the classes "yes" and "no". The probability of playing being "yes" is approximately 0.667, while the probability of playing being "no" is approximately 0.333. These prior probabilities provide a baseline understanding of the distribution of classes in the dataset before any features are considered.

4 Task 2: Calculating Likelihood Probabilities

In Task 2, we calculate the likelihood probabilities for each feature value given a specific class. Likelihood probabilities represent the probability of observing each feature value given the class label.

```
#Task2
# Calculating likelihood probabilities
likelihood_probabilities = {}

for feature in train_set.columns[:-1]:
    for category in train_set['Play'].unique():
        subset = train_set[train_set['Play'] == category]
        counts = subset.groupby(feature).size()
        total_category = subset.shape[0]

        for value, count in counts.items():
            likelihood_probabilities[(feature, value, category)] =
    count / total_category

print("\nLikelihood Probabilities:")
for key, value in likelihood_probabilities.items():
    print(f"P({key[0]}={key[1]}|Play={key[2]}): {value}")
```

Table 3: Likelihood Probabilities

Feature	Value	Probability
Outlook	Overcast, Play=yes	0.5
	Rainy, Play=yes	0.25
	Sunny, Play=yes	0.25
	Rainy, Play=no	0.5
	Sunny, Play=no	0.5
Temp	Cool, Play=yes	0.25
	Hot, Play=yes	0.25
	Mild, Play=yes	0.5
	Cool, Play=no	0.25
	Hot, Play=no	0.5
	Mild, Play=no	0.25
Humidity	High, Play=yes	0.375
	Normal, Play=yes	0.625
	High, Play=no	0.75
	Normal, Play=no	0.25
Windy	f, Play=yes	0.625
	t, Play=yes	0.375
	f, Play=no	0.25
	t, Play=no	0.75

The table above shows the calculated likelihood probabilities for each feature-

value pair conditioned on the classes "yes" and "no". These probabilities represent the likelihood of observing a specific feature value given a particular class. For example, the probability of observing "Overcast" weather conditions given that the class is "yes" (indicating playing) is 0.5. Similarly, the probability of observing "High" humidity conditions given that the class is "no" (indicating not playing) is 0.75.

- The likelihood of playing when the outlook is Overcast is 0.5, indicating a moderate preference for playing during Overcast conditions.
- For Rainy and Sunny outlooks, the likelihood of playing is lower, with probabilities of 0.25 for both. This suggests that playing is less favored during Rainy and Sunny weather conditions.
- Mild temperatures have the highest likelihood of playing with a probability of 0.5, indicating a strong preference for playing during Mild weather.
- Cooler and Hotter temperatures have lower likelihoods of playing, with probabilities of 0.25 each. This suggests that extreme temperature conditions are less conducive to playing.
- Normal humidity conditions have a higher likelihood of playing (0.625) compared to High humidity conditions (0.375). This indicates that normal humidity is more favorable for outdoor activities.
- When it's not windy (f), there's a higher likelihood of playing (0.625) compared to when it's windy (t), where the likelihood is lower (0.375). This suggests that windy conditions are less preferred for outdoor activities.

5 Task 3: Calculating Posterior Probabilities for Testing Split

In Task 3, we calculate the posterior probabilities for each sample in the testing set. Posterior probabilities represent the probability of each class given the observed data.

```
#Task3
# Calculating posterior probabilities for testing split
posterior_probabilities = []

for index, row in test_set.iterrows():
    posterior_yes = p_play_yes
    posterior_no = p_play_no
    for feature in test_set.columns[:-1]:
        feature_value = row[feature]

        # Handling missing likelihood probabilities
        likelihood_yes = likelihood_probabilities.get((feature,
    feature_value, 'yes'), 1e-6)
        likelihood_no = likelihood_probabilities.get((feature,
    feature_value, 'no'), 1e-6)
```

```
14
15          posterior_yes *= likelihood_yes
16          posterior_no *= likelihood_no
17
18      # Normalizing the posterior probabilities
19      sum_posterior = posterior_yes + posterior_no
20      posterior_yes /= sum_posterior
21      posterior_no /= sum_posterior
22
23      posterior_probabilities.append(('yes', posterior_yes, 'no',
        posterior_no))
24
25  print("\nPosterior Probabilities for Testing Split:")
26  for i, result in enumerate(posterior_probabilities):
27      print(f"Test Sample {i+1}:")
28      print("P(Play=yes|data):", result[1])
29      print("P(Play=no|data):", result[3])
```

- We iterate over each sample in the testing set using `test_set.iterrows()`.
- For each sample, we initialize the posterior probabilities for both classes ('yes' and 'no') to their prior probabilities (`p_play_yes` and `p_play_no`).
- We then iterate over each feature of the sample using `test_set.columns[:-1]`.
- For each feature, we obtain the corresponding feature value for the current sample.
- We handle missing likelihood probabilities using Laplace smoothing by setting a small value (`1e-6`) if the likelihood probability is not found in the `likelihood_probabilities` dictionary.
- We update the posterior probabilities by multiplying with the likelihood probabilities for each feature value.
- After processing all features, we normalize the posterior probabilities by dividing each by the sum of both ('yes' and 'no') posterior probabilities.
- Finally, we store the posterior probabilities for the current sample in the `posterior_probabilities` list.
- The posterior probabilities for each sample are printed, indicating the probability of each class given the observed data.

Table 4: Posterior Probabilities for Testing Split

Test Sample	**P(Play=yes—data)**	**P(Play=no—data)**
1	0.8621	0.1379
2	0.7143	0.2857

Based on the posterior probabilities for the testing split:

- For Test Sample 1, the probability of playing given the observed data is 0.8621, while the probability of not playing is 0.1379. This suggests a high likelihood of playing based on the observed weather conditions.
- For Test Sample 2, the probability of playing given the observed data is 0.7143, while the probability of not playing is 0.2857. This indicates a moderate likelihood of playing based on the observed weather conditions.

Based on the observed weather conditions for each test sample, these posterior probabilities provide valuable insights into the likelihood of playing or not playing.

6 Task 4: Making Predictions

Predictions are made by comparing the posterior probabilities of each class. The prediction is made by choosing the class (i.e., Play = yes or Play = no) with the the higher posterior probability for each test instance.

```
#Task4
# Making predictions
predictions = ['yes' if result[1] > result[3] else 'no' for result
    in posterior_probabilities]
print("\nPredictions:")
for i, prediction in enumerate(predictions):
    print(f"Test Sample {i+1}: Predicted Play = {prediction}")
```

Table 5: Predictions for Test Samples

Test Sample	Predicted Play
1	yes
2	yes

Based on the predictions for the test samples:

- For Test Sample 1, the model predicts playing ('yes') based on the observed data.
- For Test Sample 2, the model also predicts playing ('yes') based on the observed data.

These predictions indicate that the model predicts playing for both test samples based on the observed weather conditions.

7 Task 5: Laplace Smoothing

Laplace smoothing, also known as add-one smoothing or add-k smoothing, is a technique used to address the issue of zero probabilities in probability estimation, particularly in the context of Naive Bayes classifiers. When calculating probabilities from data, it's common to encounter situations where certain

events have never been observed in the training data, resulting in zero probabilities. This can lead to overly confident predictions and poor generalization performance.

Laplace smoothing addresses this issue by adding a small value (often denoted as α) to the counts of each possible outcome for every feature. This ensures that no probability estimate becomes zero, and it also helps to distribute the probability mass more evenly among the different outcomes.

The formula for calculating the smoothed probability of an event x given a class y is as follows:

$$P(x|y) = \frac{\text{count}(x|y) + \alpha}{\text{count}(y) + \alpha \cdot \text{num\_possible\_outcomes}}$$

Where:

- $\text{count}(x|y)$ is the count of event x given class y in the training data.
- $\text{count}(y)$ is the total count of class y in the training data.
- num_possible_outcomes is the total number of possible outcomes for the given feature.
- α is the Laplace smoothing parameter.

In the context of Laplace smoothing, the parameter α represents the amount of pseudo-counts added to each observed count during probability estimation. Laplace smoothing is also known as add-one smoothing, where α is typically set to 1, hence the name.

When $\alpha = 1$, Laplace smoothing adds a single pseudo-count to each observed count. This ensures that even if a particular event has not been observed in the training data (i.e., it has a count of 0), it still has a non-zero probability estimate after smoothing.

The choice of α affects the degree of smoothing applied to the probability estimates. Smaller values of α result in less smoothing, while larger values lead to more aggressive smoothing. A common practice is to set α to a small positive value (such as 1) to provide a minimal level of smoothing without significantly altering the underlying distribution of the data.

In summary, setting α to 1 in Laplace smoothing ensures that all events have non-zero probability estimates, thereby mitigating the issue of zero probabilities and improving the robustness of the model's predictions.

```
#Task5
# Laplace smoothing parameter
alpha = 1

# Calculateing likelihood probabilities with Laplace smoothing
likelihood_probabilities_smoothed = {}

for feature in train_set.columns[:-1]:
    for category in train_set['Play'].unique():
        subset = train_set[train_set['Play'] == category]
```

```
        counts = subset.groupby(feature).size()
        total_category = subset.shape[0]
        total_unique_values = len(train_set[feature].unique())

        for value in train_set[feature].unique():
            count = counts.get(value, 0)
            likelihood_probabilities_smoothed[(feature, value, category)] = (count + alpha) / (total_category + alpha * total_unique_values)


# Calculating posterior probabilities for testing split with Laplace smoothing
posterior_probabilities_smoothed = []

# Calculating prior counts for Laplace smoothing
play_yes_count = train_set['Play'].value_counts().get('yes', 0)
play_no_count = train_set['Play'].value_counts().get('no', 0)

for index, row in test_set.iterrows():
    posterior_yes = p_play_yes
    posterior_no = p_play_no
    for feature in test_set.columns[:-1]:
        feature_value = row[feature]

        # Using Laplace smoothed likelihood probabilities
        likelihood_yes = likelihood_probabilities_smoothed.get((feature, feature_value, 'yes'), alpha / (play_yes_count + alpha * len(train_set[feature].unique())))
        likelihood_no = likelihood_probabilities_smoothed.get((feature, feature_value, 'no'), alpha / (play_no_count + alpha * len(train_set[feature].unique())))

        posterior_yes *= likelihood_yes
        posterior_no *= likelihood_no

    # Normalizing the posterior probabilities
    sum_posterior = posterior_yes + posterior_no
    posterior_yes /= sum_posterior
    posterior_no /= sum_posterior

    posterior_probabilities_smoothed.append(('yes', posterior_yes, 'no', posterior_no))


# Making predictions with Laplace smoothed probabilities
predictions_smoothed = []

for result in posterior_probabilities_smoothed:
    # Check which class has higher posterior probability
    if result[1] > result[3]:
        predictions_smoothed.append('yes')
    else:
        predictions_smoothed.append('no')
```

Table 6: Likelihood Probabilities with Laplace Smoothing

Feature	**Value**	**Probability**
Outlook	Overcast, Play=yes	0.4545
	Rainy, Play=yes	0.2727
	Sunny, Play=yes	0.2727
	Overcast, Play=no	0.1429
	Rainy, Play=no	0.4286
	Sunny, Play=no	0.4286
Temp	Cool, Play=yes	0.2727
	Mild, Play=yes	0.4545
	Hot, Play=yes	0.2727
	Cool, Play=no	0.2857
	Mild, Play=no	0.2857
	Hot, Play=no	0.4286
Humidity	Normal, Play=yes	0.6
	High, Play=yes	0.4
	Normal, Play=no	0.3333
	High, Play=no	0.6667
Windy	t, Play=yes	0.4
	f, Play=yes	0.6
	t, Play=no	0.6667
	f, Play=no	0.3333

Table 7: Posterior Probabilities for Testing Split with Laplace Smoothing

Test Sample	**P(Play=yes—data)**	**P(Play=no—data)**
1	0.7974	0.2026
2	0.6862	0.3138

Table 8: Predictions for Test Samples with Laplace Smoothing

Test Sample	**Predicted Play**
1	yes
2	yes

8 Observations

8.1 Comparison

This table compares the likelihood probabilities with and without Laplace smoothing.

Table 9: Comparison of Likelihood Probabilities

Feature	**Without Laplace Smoothing**	**With Laplace Smoothing**
Outlook Overcast, Play=yes	0.5	0.4545
Outlook Rainy, Play=yes	0.25	0.2727
Outlook Sunny, Play=yes	0.25	0.2727
Outlook Overcast, Play=no	0.5	0.1429
Outlook Rainy, Play=no	0.5	0.4286
Outlook Sunny, Play=no	0.5	0.4286
Temp Cool, Play=yes	0.25	0.2727
Temp Mild, Play=yes	0.5	0.4545
Temp Hot, Play=yes	0.25	0.2727
Temp Cool, Play=no	0.25	0.2857
Temp Mild, Play=no	0.25	0.2857
Temp Hot, Play=no	0.5	0.4286
Humidity Normal, Play=yes	0.625	0.6
Humidity High, Play=yes	0.375	0.4
Humidity Normal, Play=no	0.25	0.3333
Humidity High, Play=no	0.75	0.6667
Windy t, Play=yes	0.375	0.4
Windy f, Play=yes	0.625	0.6
Windy t, Play=no	0.25	0.6667
Windy f, Play=no	0.75	0.3333

- **Outlook:**
  - For the 'Overcast' condition, the likelihood probability decreases slightly with Laplace smoothing from 0.5 to 0.4545 for 'Play=yes'.
  - For the 'Rainy' and 'Sunny' conditions, the likelihood probabilities increase slightly with Laplace smoothing, indicating a more balanced distribution across classes.
- **Temperature:**
  - For 'Cool' and 'Hot' temperatures, the likelihood probabilities remain relatively unchanged with Laplace smoothing.
  - For 'Mild' temperature, the likelihood probability increases slightly with Laplace smoothing from 0.5 to 0.4545 for 'Play=yes'.

- **Humidity:**
  - For 'Normal' humidity, the likelihood probability decreases slightly with Laplace smoothing from 0.625 to 0.6 for 'Play=yes'.
  - For 'High' humidity, the likelihood probability decreases slightly with Laplace smoothing from 0.75 to 0.6667 for 'Play=no'.
- **Windy:**
  - For 'f' (false) value of 'Windy', the likelihood probability increases slightly with Laplace smoothing from 0.625 to 0.6 for 'Play=yes'.
  - For 't' (true) value of 'Windy', the likelihood probability decreases slightly with Laplace smoothing from 0.75 to 0.6667 for 'Play=no'.

This table compares the posterior probabilities for testing split with and without Laplace smoothing.

Table 10: Comparison of Posterior Probabilities

Test Sample	**Without Laplace Smoothing**	**With Laplace Smoothing**
1	(0.8621, 0.1379)	(0.7974, 0.2026)
2	(0.7143, 0.2857)	(0.6862, 0.3138)

Here are the conclusions drawn from the comparison of posterior probabilities:

- **Test Sample 1:**
  - Without Laplace smoothing: The posterior probability of 'Play=yes' is 0.8621, indicating a high probability of playing, while the posterior probability of 'Play=no' is 0.1379.
  - With Laplace smoothing: The posterior probability of 'Play=yes' decreases slightly to 0.7974, while the posterior probability of 'Play=no' increases slightly to 0.2026.

  Explanation: Laplace smoothing adjusts the probabilities to account for unseen feature-value combinations, resulting in a slight decrease in the probability of 'Play=yes' and a slight increase in the probability of 'Play=no'. However, the overall classification remains the same.
- **Test Sample 2:**
  - Without Laplace smoothing: The posterior probability of 'Play=yes' is 0.7143, while the posterior probability of 'Play=no' is 0.2857.
  - With Laplace smoothing: The posterior probability of 'Play=yes' decreases slightly to 0.6862, while the posterior probability of 'Play=no' increases slightly to 0.3138.

Explanation: Similar to Test Sample 1, Laplace smoothing adjusts the probabilities, leading to minor changes in the posterior probabilities. In this case, the probability of 'Play=yes' decreases slightly, while the probability of 'Play=no' increases slightly.

Overall, Laplace smoothing helps to regularize the posterior probabilities, making them more robust and stable, especially in cases with limited data or unseen feature-value combinations. However, the effect of Laplace smoothing on the classification decisions is minimal, with only slight adjustments to the probabilities.

8.2 Accuracy

- **Accuracy for Normal Data:** 0.5
  The accuracy for the normal data is 0.5, indicating that the model's predictions match the ground truth labels for 50% of the test samples.

- **Accuracy for Smoothed Data:** 0.5
  The accuracy for the smoothed data is also 0.5, which is the same as the accuracy for the normal data. This suggests that Laplace smoothing did not significantly improve the model's performance in this particular case.

8.3 Confusion Matrices

- **Confusion Matrix for Normal Data:**

$$\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$$

  In the confusion matrix for the normal data, there is one true positive (TP) prediction and one true negative (TN) prediction. However, there is also one false positive (FP) prediction and zero false negative (FN) predictions.

- **Confusion Matrix for Smoothed Data:**

$$\begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$$

  The confusion matrix for the smoothed data is identical to the confusion matrix for the normal data. This suggests that Laplace smoothing did not change the distribution of predictions, and the number of correct and incorrect predictions remained the same.

General Observations

- The Outlook of Overcast significantly increases the probability of playing, indicating a strong preference for this weather condition.

- Cooler temperatures are preferred for playing, as seen from the higher probability under the Cool category.
- High humidity is a strong deterrent to playing, whereas normal humidity conditions are favorable.
- Windy conditions are less preferred for playing, as indicated by the lower probability when it is true (t).

9 Conclusion

In this problem, we do a step-by-step analysis using Naive Bayes for the classification of the given dataset. Additionally, Laplace smoothing is applied to handle zero probabilities effectively.