


B22EE036_PRML_PA4_REPORT

Author: K. K. N. SHYAM SATHVIK

Roll No: B22EE036

Link to the Colab File:

Google Colaboratory

 https://colab.research.google.com/drive/1dJpabL_LCr5Zy_1DS_YBVstkM145sx7u?usp=sharing



▼ Linear Discriminant Analysis (LDA)

Linear Discriminant Analysis (LDA) is a statistical method used for dimensionality reduction while preserving as much of the class discriminatory information as possible. Unlike PCA, which does not take class labels into account and focuses on capturing the directions of maximum variance, LDA aims to model the difference between classes. It is particularly useful in the preprocessing steps for pattern classification and machine learning applications.

Task-1: LDA Implementation and Analysis

Implementation of Linear Discriminant Analysis:

▼ Step 1:

- Consider a dataset of m samples, each belonging to one of L classes. The feature space of the dataset is n -dimensional, represented as:

$$X = \{x_1, x_2, x_3, x_4, \dots, x_n\}$$

with each x_i being an n -dimensional feature vector. The dataset can be represented as:

$$X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix}$$

▼ Step 2 : Compute the Mean Vectors

- Compute the mean vectors for each class, which involves calculating the average of each feature for the samples within each class.

- The mean vector of class i is denoted by μ_i and is calculated as:

$$\mu_i = \frac{1}{n_i} \sum_{x \in D_i} x$$

where D_i represents the set of all samples belonging to class i and n_i is the number of samples in class i .

```
def ComputeMeanDiff(X):
    # Separate the samples by class
    class_zero = X[X[:, 2] == 0][:, :2]
    class_one = X[X[:, 2] == 1][:, :2]
    # means of each class
    mean_zero = np.mean(class_zero, axis=0)
    mean_one = np.mean(class_one, axis=0)
    # difference of class wise means
    mean_diff = mean_zero - mean_one
    return mean_diff
```

▼ Step 3 : Compute the Between-Class Scatter Matrix (S_B) and Within-Class Scatter Matrix (S_W)

- The within-class scatter matrix S_W quantifies the spread of the class samples around their respective mean vectors, whereas the between-class scatter matrix S_B measures the separation between the different class means.
- S_W and S_B are given by:

$$S_W = \sum_{i=1}^L \sum_{x \in D_i} (x - \mu_i)(x - \mu_i)^T$$

$$S_B = \sum_{i=1}^L n_i (\mu_i - \mu)(\mu_i - \mu)^T$$

where μ is the overall mean of the dataset, and μ_i is the mean vector of class i .

```
def ComputeSW(X):
    # Separate the samples by class
    class_zero = X[X[:, 2] == 0][:, :2]
    class_one = X[X[:, 2] == 1][:, :2]
    # scatter matrices for each class
    S_zero = np.dot((class_zero - np.mean(class_zero, axis=0)).T,
                    (class_zero - np.mean(class_zero, axis=0)))
    S_one = np.dot((class_one - np.mean(class_one, axis=0)).T,
                   (class_one - np.mean(class_one, axis=0)))
```

```

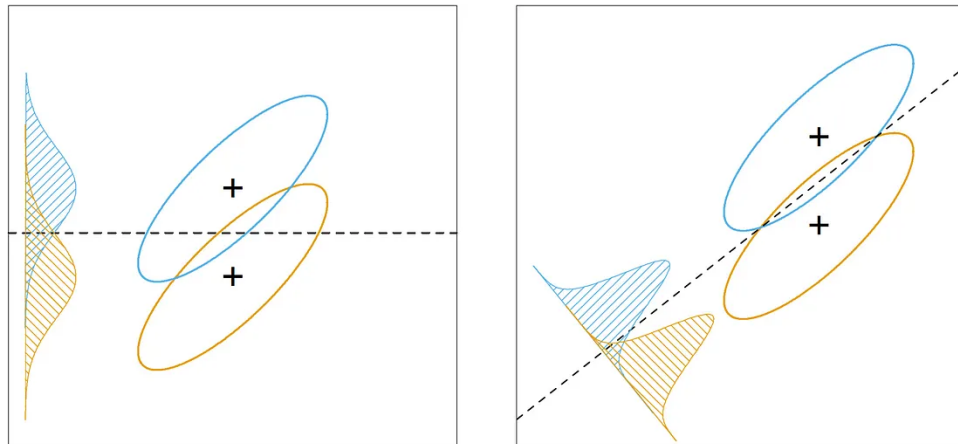
# total within-class scatter matrix
S_w = S_zero + S_one
return S_w

def ComputeSB(X):
    # overall mean
    overall_mean = np.mean(X[:, :2], axis=0)
    # means of each class
    mean_diff = ComputeMeanDiff(X)
    mean_zero = overall_mean - mean_diff / 2
    mean_one = overall_mean + mean_diff / 2
    # between-class scatter matrix
    S_b = np.outer(mean_diff, mean_diff)
    return S_b

```

▼ Step 4 : Compute Eigenvalues and Eigenvectors

- Solve the eigenvalue problem for the matrix $S_W^{-1}S_B$ to find the directions that best separate the classes. The eigenvectors correspond to the directions for maximum separation, and the eigenvalues represent the effectiveness of these directions in separating the classes.
- The goal is to select the top k eigenvectors based on their corresponding eigenvalues, which gives the axes that maximize class separability.



```

def GetLDAProjectionVector(X):
    # Get SW and SB
    S_w = ComputeSW(X)
    S_b = ComputeSB(X)
    # inverse of Sw
    S_w_inv = np.linalg.inv(S_w)
    # Sw^-1 dot Sb
    S_w_inv_Sb = np.dot(S_w_inv, S_b)
    # Compute the eigenvectors and eigenvalues

```

```
eigenvalues, eigenvectors = np.linalg.eig(S_w_inv_Sb)
# Get the eigenvector corresponding to the highest eigenvalue
max_index = np.argmax(eigenvalues)
w = eigenvectors[:, max_index]
return w
```

▼ Step 5 : Project the Data onto the New Feature Space

- Once the eigenvectors (W) are determined, project the original dataset onto the new subspace using the transformation:

$$X_{\text{reduced}} = XW$$

where X_{reduced} is the matrix representing the data in the reduced-dimensional space.

- This projection maximizes the class separability and hence is optimal for classification tasks.
- Once the eigenvectors (W) are determined, project the original dataset onto the new subspace using the transformation:

$$X_{\text{reduced}} = XW$$

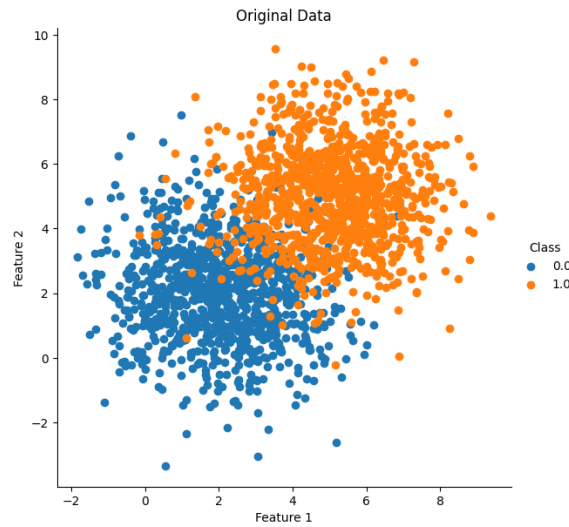
where X_{reduced} is the matrix representing the data in the reduced-dimensional space.

```
def project(x, y, w):
    # Project the point (x, y) using the LDA projection vector w
    point = np.array([x, y])
    projection = np.dot(point, w)
    return projection
```

Task-2:

LDA Projection Vector and Transformation Results

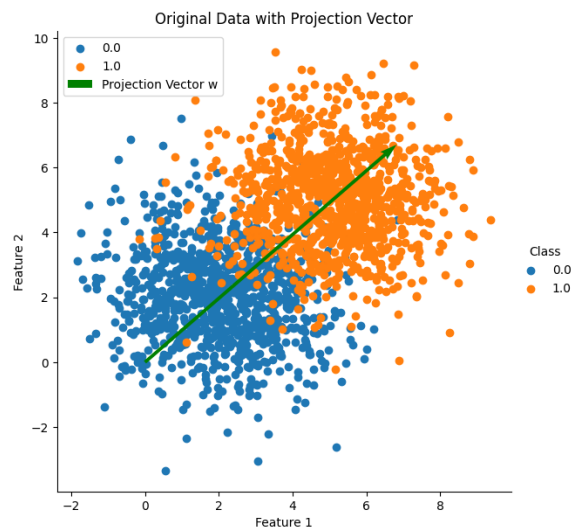
Original Dataset:



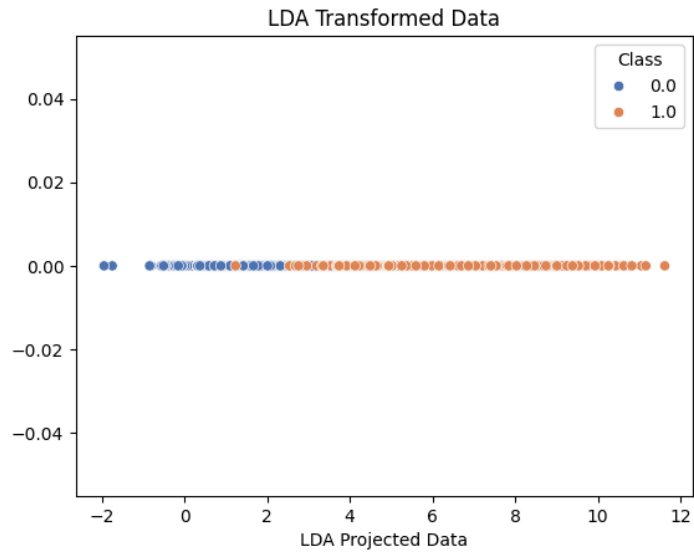
Projection Vector:



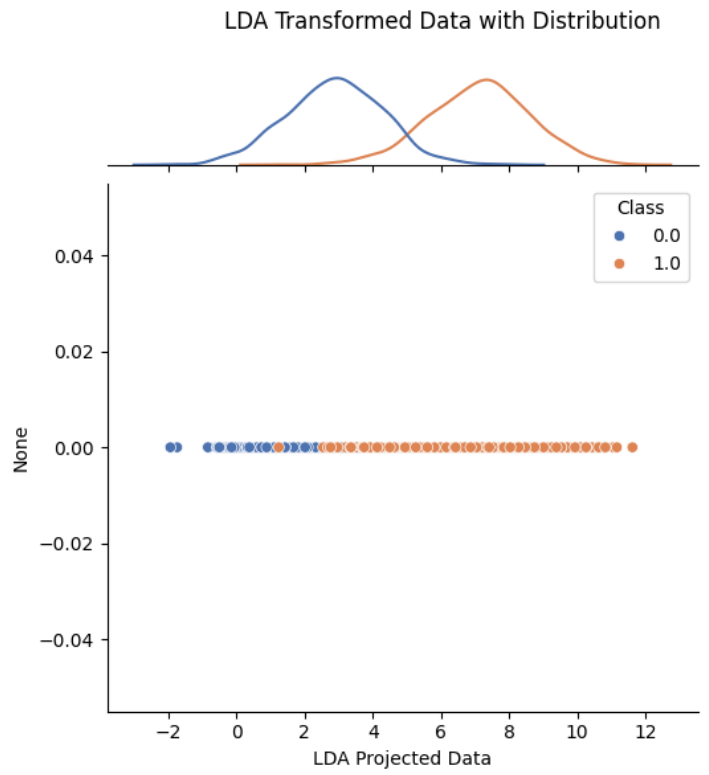
The green line represents the projection vector.



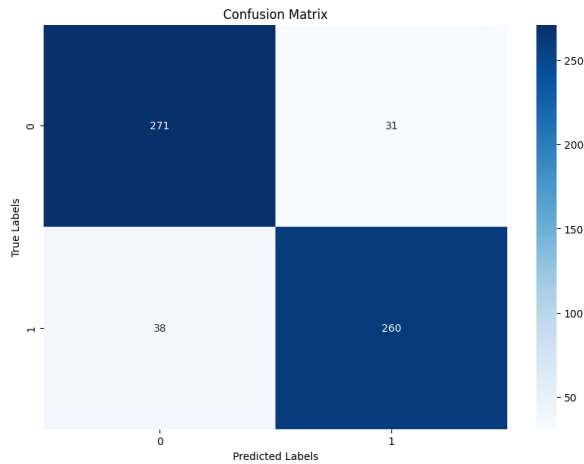
LDA Transformed Dataset:



Distribution of Transformed Data Set:



Projected Data 1-NN Classifier Performance:

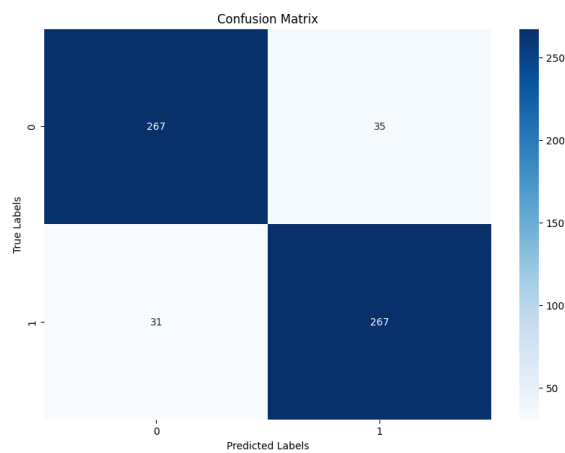


Class	Precision	Recall	F1-Score
0.0	0.88	0.90	0.89
1.0	0.89	0.87	0.88
Accuracy			0.89
Macro Avg	0.89	0.88	0.88
Weighted Avg	0.89	0.89	0.88

💡 Accuracy: 0.885

- Worse recall.
- Better Precision.
- Around the same accuracy.

Original Data 1-NN Classifier Performance:



Class	Precision	Recall	F1-Score
0.0	0.90	0.88	0.89
1.0	0.88	0.90	0.89
Accuracy			0.89
Macro Avg	0.89	0.89	0.89
Weighted Avg	0.89	0.89	0.89

- Better Recall.
- Worse Precision.
- Same accuracy.

▼ Conclusion from the above trained Model:

- The trained 1-NN model seems to be as performant as the model trained on the original data, with slightly increased precision and slightly worse recall. This makes sense since the original data is transformed and outliers/noisy data has been removed, making the model more robust to unseen samples. We can conclude that LDA

▼ Naive Bayes

The Naive Bayes classifier is a simple, yet effective algorithm based on Bayes' theorem with the assumption of independence among predictors. Despite its simplicity, it has shown remarkable performance in various applications, especially in text classification and spam filtering. The classifier calculates the probability of each class given a set of features, and assigns the class with the highest probability to the input data. Its efficiency and speed come from its ability to handle an immense amount of data with ease. The "naive" aspect refers to the assumption that all **features contribute independently** to the probability of each class, simplifying computation but sometimes oversimplifying real-world interactions.

GAUSSIAN NAIVE BAYES CLASSIFIER

"Gaussian" because this is a normal distribution

This is our prior belief

$$P(\text{class} | \text{data}) = \frac{P(\text{data} | \text{class}) \times P(\text{class})}{P(\text{data})}$$

We don't calculate this in naive bayes classifiers

ChrisAlbon



Naive Bayes assumes independence among the features.

▼ Task-0: Data Preparation

Data Exploration:



Output of `data.describe()`

	Outlook	Temp	Humidity	Windy	Play
count	14	14	14	14	14
unique	3	3	2	2	2
top	Rainy	Mild	High	f	yes
freq	5	6	7	8	9

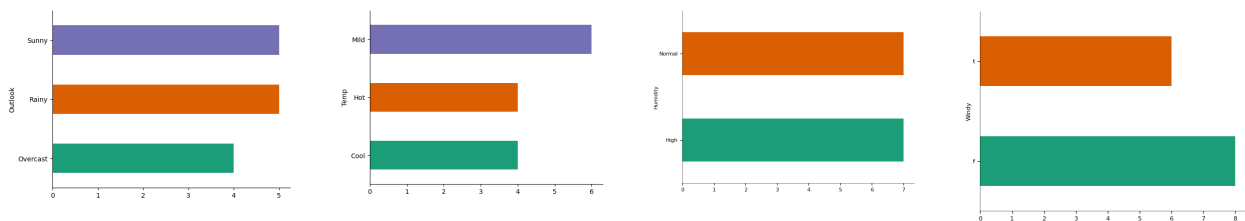


Output of `data.info()` :

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14 entries, 0 to 13
Data columns (total 5 columns):
 #   Column      Non-Null Count  Dtype
---  -
 0   Outlook    14 non-null    object
 1   Temp       14 non-null    object
 2   Humidity   14 non-null    object
 3   Windy      14 non-null    object
 4   Play       14 non-null    object
dtypes: object(5)
memory usage: 688.0+ bytes
```

- The data doesn't have any null values.

Data Visualization



• Task-1: Prior Probabilities Calculation

```
def calculate_prior_probabilities(train):
    """
```

```

Calculates the prior probabilities of the target variable.
"""
prior_play_yes = len(train[train['Play'] == 'yes']) / len(train['Play'])
prior_play_no = 1 - prior_play_yes
return prior_play_yes, prior_play_no

```

- Prior probabilities were calculated by dividing the number of instances of each class (Play=yes, Play=no) by the total number of instances.
- **Prior Probability (P(C))**: The initial probability of an event (class) occurring before any evidence is taken into account.
- For example, :

$$P(\text{Play} = \text{yes}) = \frac{\text{Count of instances where Play} = \text{yes}}{\text{Total number of instances}}$$

$$P(\text{Play} = \text{no}) = \frac{\text{Count of instances where Play} = \text{no}}{\text{Total number of instances}}$$

• Task-2: Likelihood Probabilities Calculation

```

def calculate_likelihood_probabilities(df):
    """
    Calculates the likelihood probabilities for each feature given the class.

    Returns:
        likelihood_probabilities (dict): A dictionary containing likelihood probabilities.
    """
    likelihood_probabilities = {}
    features = ['Outlook', 'Temp', 'Humidity', 'Windy']
    for feature in features:
        likelihood_probabilities[feature] = {}
        u = df[feature].unique()
        # print(u)
        for value in df[feature].unique():
            for play in ['yes', 'no']:
                count = len(df[(df[feature] == value) & (df['Play'] == play)])
                total = len(df[df['Play'] == play])
                likelihood_probabilities[feature][(value, play)] = count / total
    return likelihood_probabilities

```

- Likelihood probabilities for each feature given the class were calculated by analyzing the frequency of each feature value within each class.
- **Likelihood Probability (P(E|C))**: The probability of observing the evidence (feature value) given that the class (C) is true.
- For, example:

$$P(\text{Feature} = \text{Value} | \text{Play} = \text{Class}) = \frac{\text{Count of instances where Feature} = \text{Value and Play} = \text{Class}}{\text{Total number of instances where Play} = \text{Class}}$$

• Task-3: Posterior Probabilities Calculation

```
def calculate_posterior_probabilities(row, priors, likelihoods):  
    """  
    Calculates the posterior probabilities of the target variable.  
    """  
    prior_play_yes = priors[0]  
    prior_play_no = priors[1]  
    for feature in ['Outlook', 'Temp', 'Humidity', 'Windy']:  
        prior_play_yes *= likelihoods[feature][(row[feature], 'yes')]  
        prior_play_no *= likelihoods[feature][(row[feature], 'no')]  
    return prior_play_yes, prior_play_no
```

- Posterior probabilities for both classes were calculated using the Naive Bayes formula, incorporating prior and likelihood probabilities.
- **Posterior Probability ($P(C|E)$):** The probability of the class after observing the evidence. It's the updated belief about the class given the evidence.
- For example:

$$P(\text{Class}|\text{Features}) = \frac{P(\text{Class}) \times \prod_{i=1}^n P(\text{Feature}_i|\text{Class})}{P(\text{Features})}$$

• Task-4: Making Predictions

```
def make_predictions(test_df, priors, likelihoods):  
    """  
    Returns predictions given a row.  
    """  
    predictions = []  
    for _, row in test_df.iterrows():  
        probs_yes, probs_no = calculate_posterior_probabilities(row, priors, likelihoods)  
        prediction = 'yes' if probs_yes > probs_no else 'no'  
        predictions.append(prediction)  
    return predictions
```

- Predictions were made by choosing the class with the highest posterior probability for the given test split examples.
- Final predictions are made based on which class (Play=yes or Play=no) has the higher posterior probability for a given set of features in the test data.
- For example:

$$P(\text{Class}|\text{Features}) = P(\text{Feature}_1|\text{Class}) \times P(\text{Feature}_2|\text{Class}) \times \dots \times P(\text{Feature}_n|\text{Class}) \times P(\text{Class})$$

• Task-5: Application of Laplace Smoothing

```
def calculate_likelihood_probabilities_with_laplace(df):
    """
    Calculates the likelihood probabilities for each feature given the class.

    Returns:
        likelihood_probabilities (dict): A dictionary containing likelihood probabilities.
    """
    likelihood_probabilities = {}
    features = ['Outlook', 'Temp', 'Humidity', 'Windy']
    laplace_k = 1
    for feature in features:
        likelihood_probabilities[feature] = {}
        values = df[feature].unique()
        for value in values:
            for play in ['yes', 'no']:
                count = len(df[(df[feature] == value) & (df['Play'] == play)]) + laplace_k
                total = len(df[df['Play'] == play]) + laplace_k * len(values)
                likelihood_probabilities[feature][(value, play)] = count / total
    return likelihood_probabilities
```

- Laplace smoothing was applied to recalculate likelihood and posterior probabilities to mitigate the issue of zero probabilities for unseen feature-class combinations. Predictions on the test split were then made using these adjusted probabilities.
- **Laplace Smoothing:** A technique used to address the problem of zero probability in Naive Bayes classifiers by adding a pseudo-count of 1 to each class-feature combination.
- For example:

$$P(\text{Feature} = \text{Value} | \text{Class}) = \frac{\text{Count of instances where Feature} = \text{Value and Play} = \text{Class} + 1}{\text{Total number of instances where Play} = \text{Class} + \text{Number of unique values of the feat}}$$

4.2 Results and Observations



Here, + implies correct prediction and - denotes wrong prediction.

Test Sample	Prediction without Smoothing	Prediction with Smoothing
Sample 1	+	+
Sample 2	+	-

- When the dataset is small, Naive Bayes with Laplace smoothing can perform worse than Naive Bayes without it because the addition of pseudo-counts might have amplified the noise in the data, leading to over-smoothing and inaccurate probabilities. This can hinder the Naive Bayes model's ability to capture underlying patterns effectively.

▼ References

- **LDA**
 - https://youtu.be/azXCzI57Yfc?si=9AX_7KVWouWvwOb1
 - **Naive Bayes**
 - <https://www.kaggle.com/code/dhanishahahaha/naive-bayes-working-without-sklearn-libraries>
-