Programming Assignment 4

Lab 7 and Lab 8

Neermita Bhattacharya

B22CS092

CSL2050

B22CS092_all.ipynb

Linear Discriminant Analysis

Introduction:

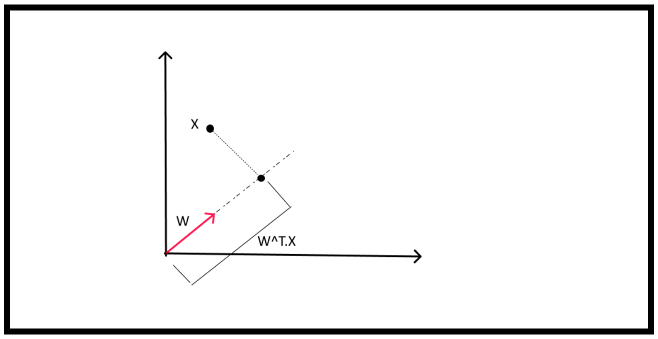
Linear Discriminant Analysis (LDA) is most commonly used as dimensionality reduction technique in the pre-processing step for pattern-classification and machine learning applications.

The goal is to project a dataset onto a lower-dimensional space with good class-separability in order to avoid overfitting. It identifies patterns in features to distinguish between different classes.

LDA helps in finding a straight line or plane that achieves maximum separation between samples of different classes and minimum separation within the class samples itself. By maximizing the separation between classes, it enables accurate classification of new data points.

Linear Discriminant Analysis makes some assumptions about the data:

- It assumes that the data follows a **Normal or Gaussian distribution**, meaning each feature forms a bell-shaped curve when plotted.
- Each of the classes has **identical covariance matrices.**
- w^T here is the eigenvector of $Sw^{-1}Sbw = \lambda w$ that we will compute.



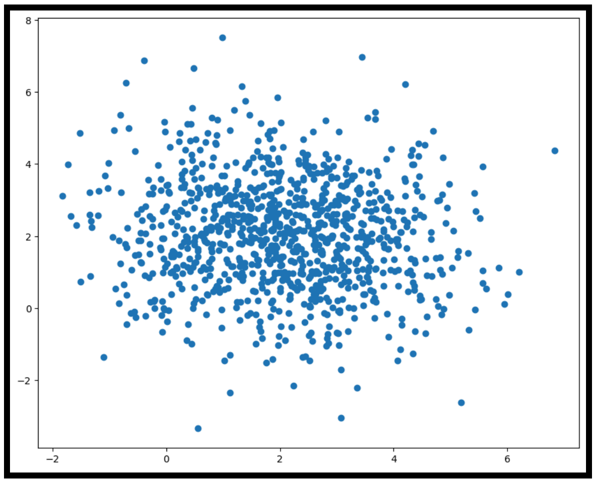
Task-1 (25 pts): Compute the following terms and print them: (For this Task, you are given a sample data.csv and helper code pa 4 problem 1 task 1.py, use it for the template, and write the function definitions there). (i). Difference of class wise means = m1 - m2 (ii). Total Within-class Scatter Matrix SW (iii). Between-class Scatter Matrix SB (iv). The EigenVector of matrix S^2wSb corresponding to the highest EigenValue (v). For any input 2-D point, print its projection according to LDA. Deliverable: (i) myLDA.py that performs all these tasks we will test it on our version of data.csv (ii) For this task there is no requirement of any report.

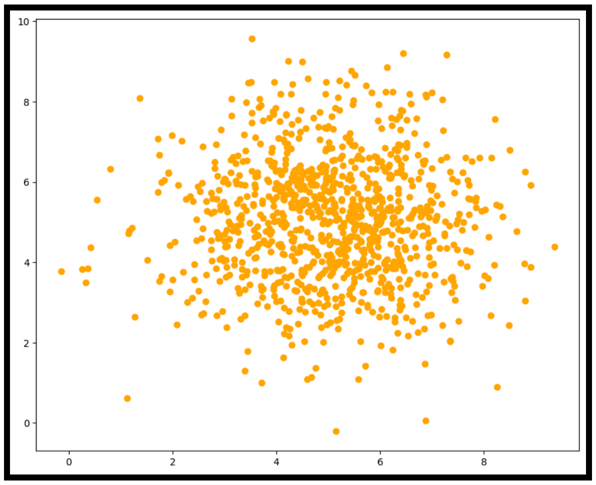
- Using pandas, 'data.csv' is directly read and stored as a dataframe with headings: f1 (feature 1), f2 (feature 2), class (the labels 0/1).
- Note that I did not use np.vstack for creating X; I directly created a dataframe and will be processing the data from there itself.
-

```
           f1        f2  class
0    1.861898  2.722963    0.0
1    2.235896 -2.157721    0.0
2    1.692817  0.995896    0.0
```

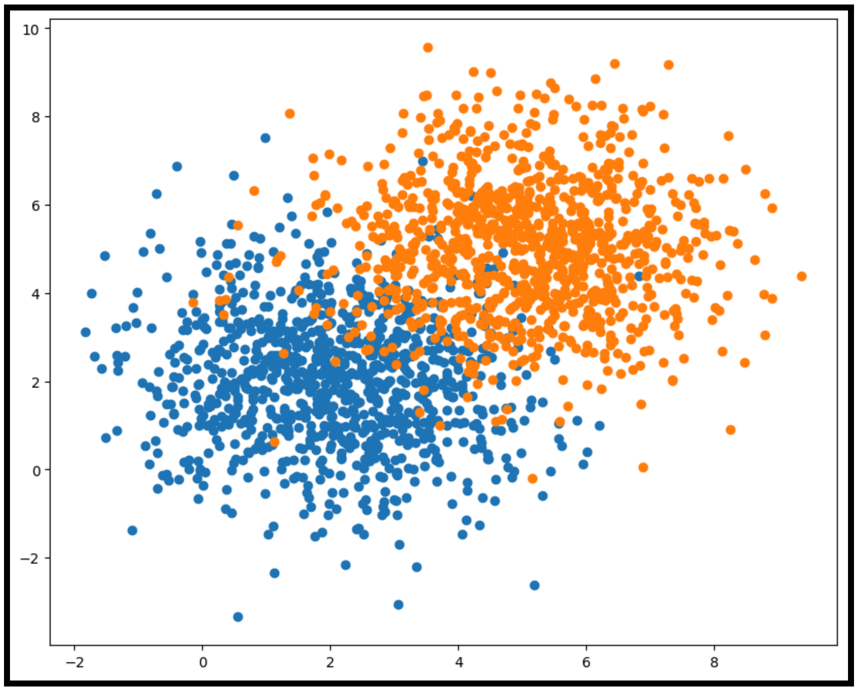
```
3       2.584343  3.722926     0.0
4       1.689087 -0.838214     0.0
...          ...       ...     ...
1995   6.234800  5.117729     1.0
1996   6.803708  3.127105     1.0
1997   4.466789  3.444926     1.0
1998   5.658980  4.043341     1.0
1999   1.980180  7.158152     1.0
          [2000 rows x 3 columns]
```

- Visualization of the dataset:





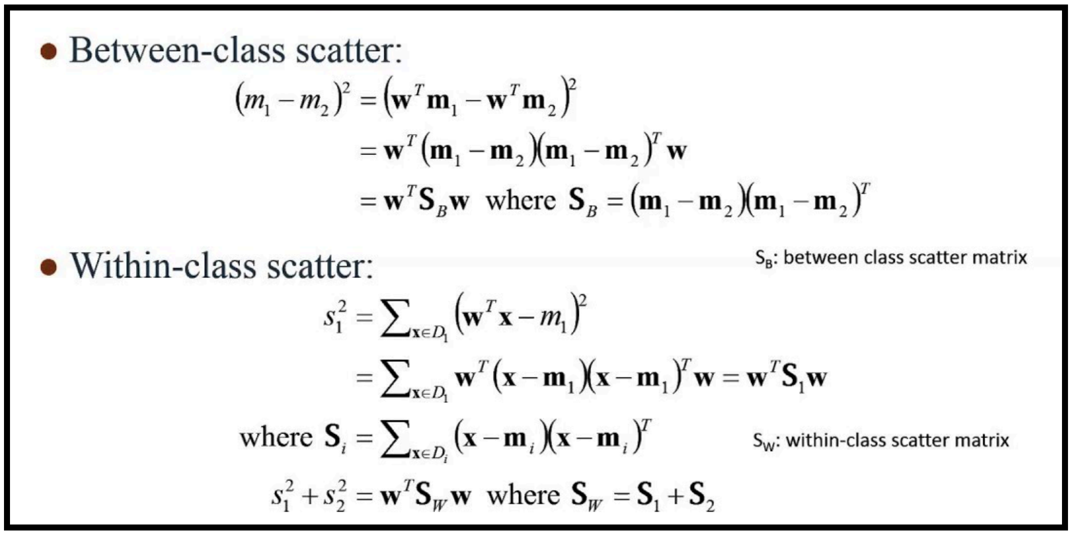
`Class 0.0`　　　　　　`Class 1.0`



- To compute the mean difference vector between two classes, we first calculate the respective means of class=0.0 (m_1) and class=1.0 (m_2).
- Each of these mean vectors will be a 2x1 vector since we have two features.

-
  ```
  Mean of samples with class 0.0:  [2.049236220970497,
  2.0276954531455904]                                   ,
  ```

-
  ```
  Mean of samples with class 1.0:  [5.034786691276668,
  5.051953981970655]                                    ,
  ```

-
  ```
  Difference in class means: [-2.98555047 -3.02425853]
  ```
- Difference is negative since we did m_1(0.0)-m_2(1.0).
- If m_1 was mean of samples in class 1.0 and m_2 mean of samples in class 0.0, then mean difference would be: ` [2.98555047 3.02425853]`
- Next, we define within class scatter matrix and between class scatter matrix.
- In LDA, we aim to minimize the within class scatter (so that it's easy to club classes together to form a group) and maximize the between class scatter (so that different classes can be easily distinguished).
- 

  - Between-class scatter:

  $$(m_1 - m_2)^2 = \left(\mathbf{w}^T\mathbf{m}_1 - \mathbf{w}^T\mathbf{m}_2\right)^2$$
  $$= \mathbf{w}^T(\mathbf{m}_1 - \mathbf{m}_2)(\mathbf{m}_1 - \mathbf{m}_2)^T\mathbf{w}$$
  $$= \mathbf{w}^T\mathbf{S}_B\mathbf{w} \text{ where } \mathbf{S}_B = (\mathbf{m}_1 - \mathbf{m}_2)(\mathbf{m}_1 - \mathbf{m}_2)^T$$

  - Within-class scatter:

  S_B: between class scatter matrix

  $$s_1^2 = \sum_{\mathbf{x}\in D_1}\left(\mathbf{w}^T\mathbf{x} - m_1\right)^2$$
  $$= \sum_{\mathbf{x}\in D_1}\mathbf{w}^T(\mathbf{x} - \mathbf{m}_1)(\mathbf{x} - \mathbf{m}_1)^T\mathbf{w} = \mathbf{w}^T\mathbf{S}_1\mathbf{w}$$
  $$\text{where } \mathbf{S}_i = \sum_{\mathbf{x}\in D_i}(\mathbf{x} - \mathbf{m}_i)(\mathbf{x} - \mathbf{m}_i)^T$$

  S_W: within-class scatter matrix

  $$s_1^2 + s_2^2 = \mathbf{w}^T\mathbf{S}_W\mathbf{w} \text{ where } \mathbf{S}_W = \mathbf{S}_1 + \mathbf{S}_2$$

  Where $y = w^T$ x
- *m1* and *m2* are the means of the newly projected data points.
- The LDA goal is to maximize J(w)= $(w^Tm1 - w^Tm2)^2/(S1^2 + S2^2)$
- Sw is calculated as:

-
  ```
  [[4421.02288527  -237.41918148]
   [-237.41918148 4556.36685753]]
  ```

- Sb is calculated as:

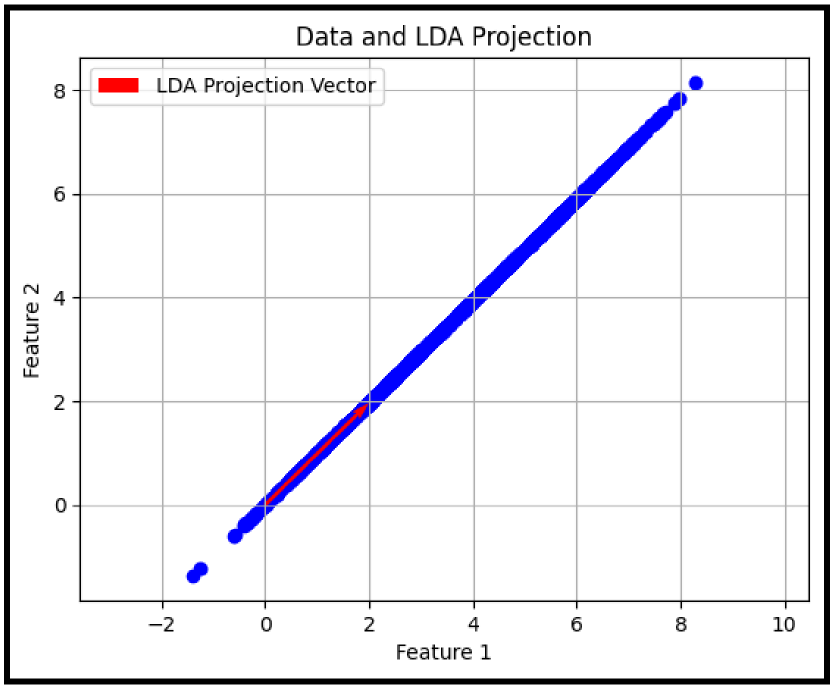
-
  ```
  [[8.91351161  9.02907647]
   [9.02907647 9.14613965]]
  ```

- Using the matrix $Sw^{-1}Sb$:
    - We calculate the eigenvalues.
    - Arrange them in descending order.
    - Acquire the largest eigenvalue.
    - Find its corresponding eigenvector.
    - Finally, project the data onto this vector.
- The Eigenvalues in decreasing order: **0.004248221663209285, 0.0**
- **Eigenvector 1: [[0.71310953] [0.70105263]], Eigenvalue 1: 4.25e-03**
- **Eigenvector 2: [[-0.71164636][0.70253787]], Eigenvalue 2: 0.00e+00**
- GetLDAProjectionVector(X,df) will return the eigenvector with the highest eigenvalue. This value is stored in w.
- w_normalized = w / np.linalg.norm(w)
- Printing both w and w_normalized, we see that they are both:

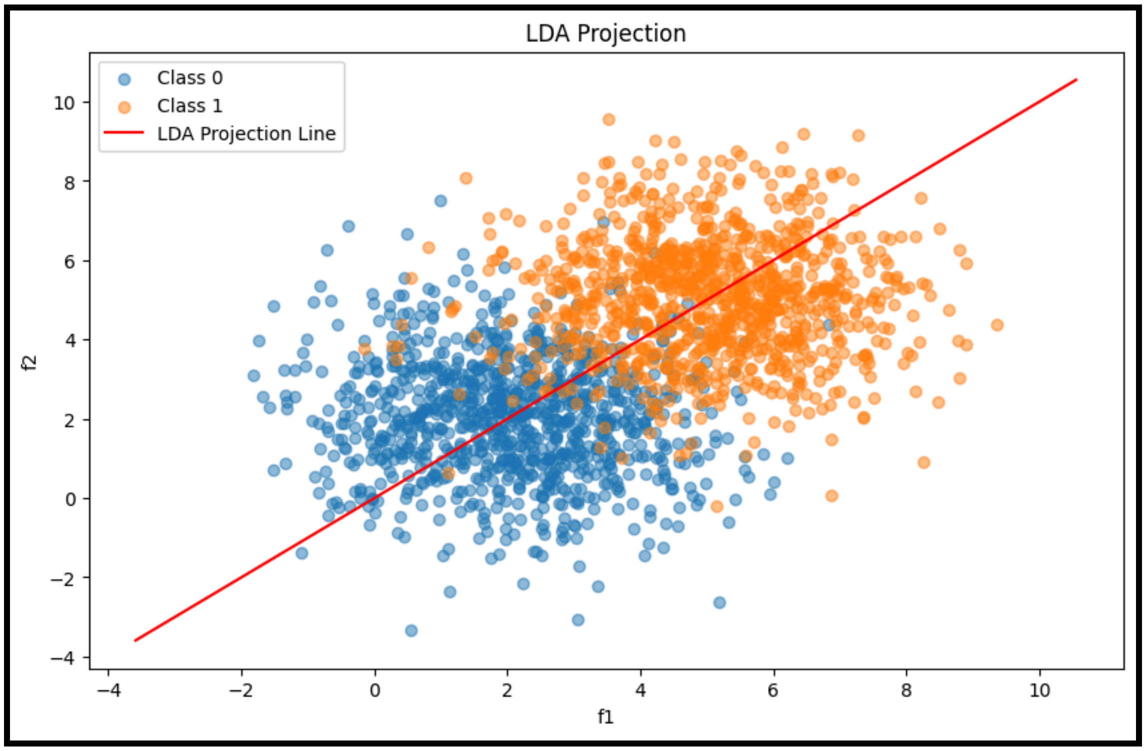
-
  ```
  [0.71310953 0.70105263] [0.71310953 0.70105263]
  ```
- **Hence, the vector returned to us is already normalized.**

Task-2 (5 pts): Show the LDA projection vector on a plot.

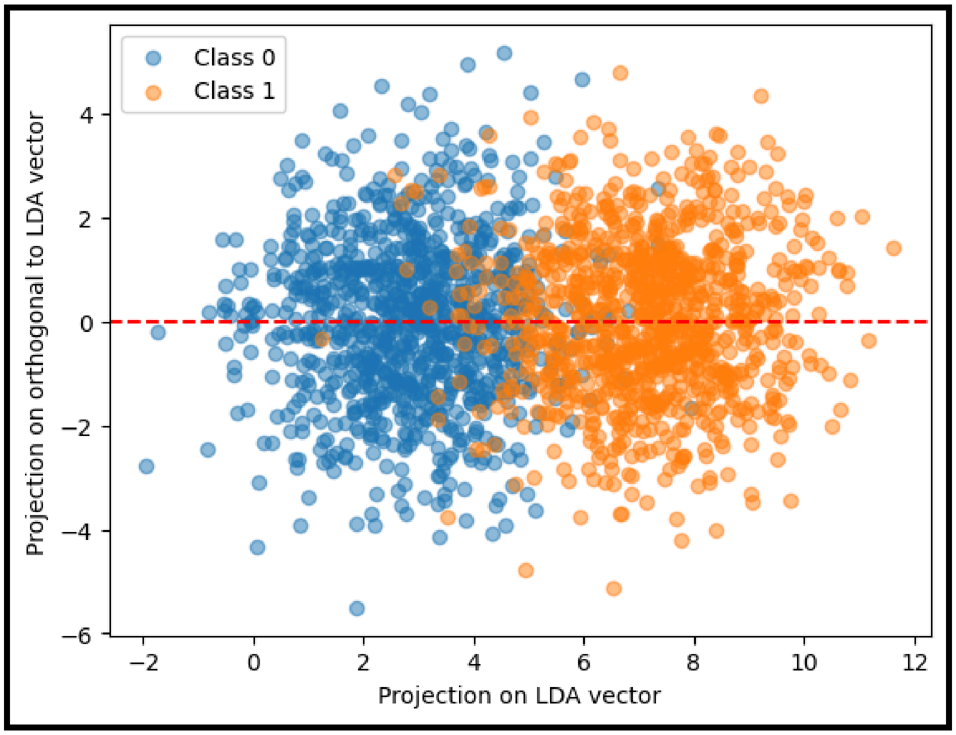
- Visualization of the eigenvector using plt.quiver() function.
- First, all the projections of the sample points are plotted along the eigenvector. Then the LDA projection vector is plotted.



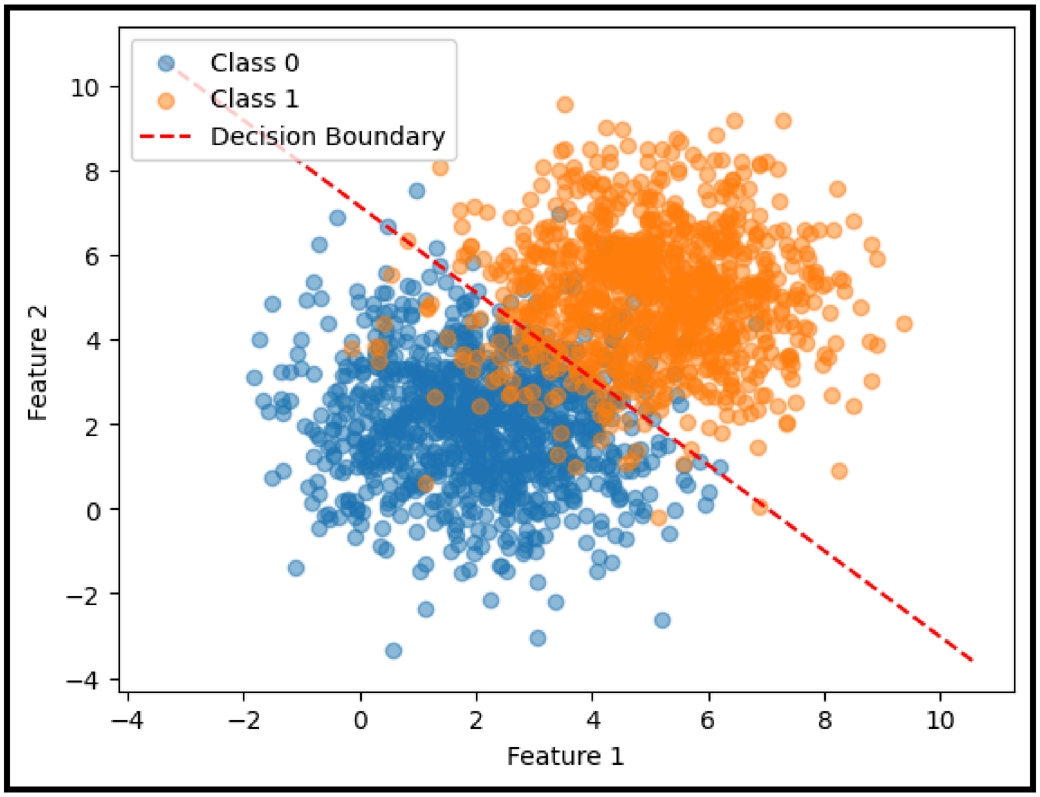
- Plotting the LDA line along with the sample points:



- Computing and plotting the rotated data points too, when the LDA line is the x-axis and its orthogonal counterpart is the y-axis.



- After projecting all the data points onto the LDA line, we would get a point on the line where maximum separation of classes could be obtained. Extending this point orthogonally to the LDA line, we essentially get the decision boundary.



- This is plotted above along with all the sample points.
- We see that the decision boundary manages to separate the classes to a good degree (accuracy discussed below).

Task-3 (10 pts): Compare the performance of 1-NN neighbor classifier on original data vs projected data. Write down your observations.

Table to compare accuracies

Seed value	Test size	Accuracy	
42		Original Data	Projected Data
	0.2	88.75%	88%
	0.3	89%	88.5%
	0.5	90.3%	89.3%
	0.7	89.57142857142857%	89.5%
	0.9	87.88888888888889%	89.16666666666667%
1000	0.2	89.75%	88.75%
	0.3	90%	88.66666666666667%
	0.5	89.9%	88.9%
	0.7	89.85714285714286%	89.14285714285715%
	0.9	88.61111111111111%	89.88888888888888%

Please find the code for this task on Google Colab too.

FINAL OBSERVATIONS:

- Accuracy of original data is generally larger than accuracy of projected data.
- LDA aims to do two things at once: Maximize scatter between classes and minimize scatter within classes. Hence, some data may not be labeled correctly while trying to achieve both.
- Higher accuracy reported if train and test split equally. (0.5 or 0.3 test size)
- During dimensionality reduction, some information from the original features may be lost. If this lost information is critical for classification, it can lead to a reduction in accuracy on the projected data.
- The LDA projection axis might not align perfectly with the directions that maximize between class separability in the original feature space. As a result, some information might be lost by the LDA projection.
- As testing size increases, the original data accuracy decreases and projected data accuracy increases (as expected since samples are increasing in the testing environment).
- The difference in accuracy between the original and projected data is relatively small (0.75% for test size=0.2 and rs=42). This difference could be within the variability of the dataset or due to random fluctuations.
- Accuracies reported are pretty high considering we used a 1-NN classifier.
- This is due to the fact that LDA aims to find a projection that maximizes the distance between the means of different classes while minimizing scatter within each class.
- This results in a reduced-dimensional space where classes are well-separated, making it easier for 1NN to classify new instances based on their nearest neighbors

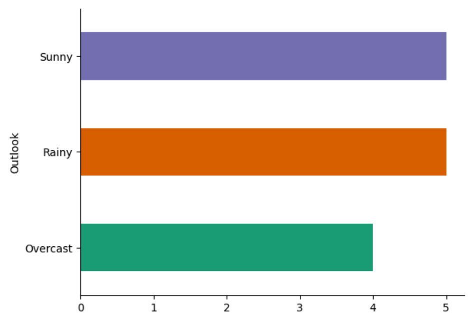
Naive Bayes

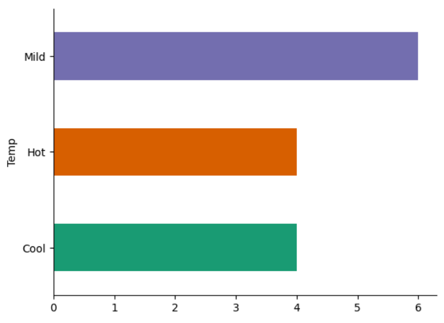
Introduction:

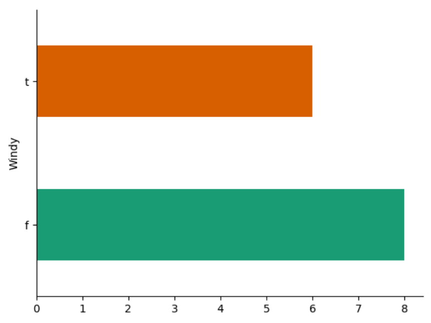
Naive Bayes is a supervised learning algorithm to predict posterior probabilities. Naive Bayes is called **naive** because it assumes that each input variable is independent. This is a strong assumption and unrealistic for real data; however, very useful for other applications.

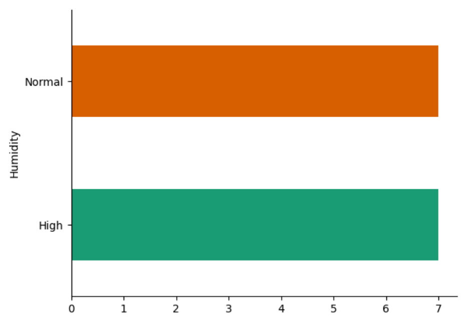
Data pre-processing:

- The dataset is read directly from the csv file in the github page.
- 4 features are present: Outlook, Temp, Humidity and Windy.
- The final labels to assess are whether a person played or not (Play= yes or no) given a set of features.

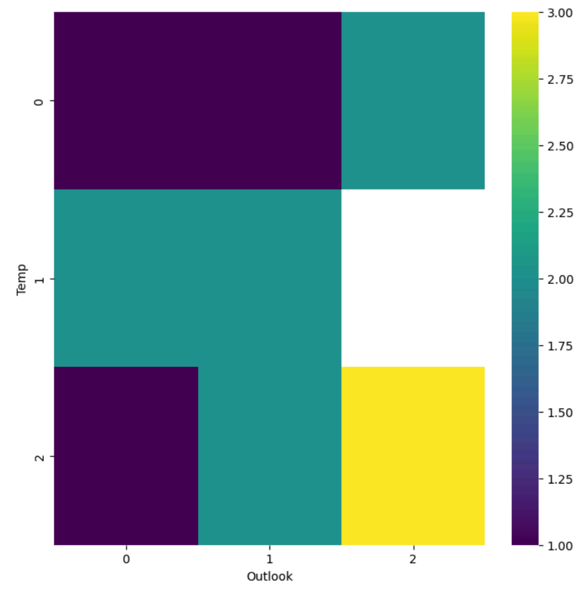


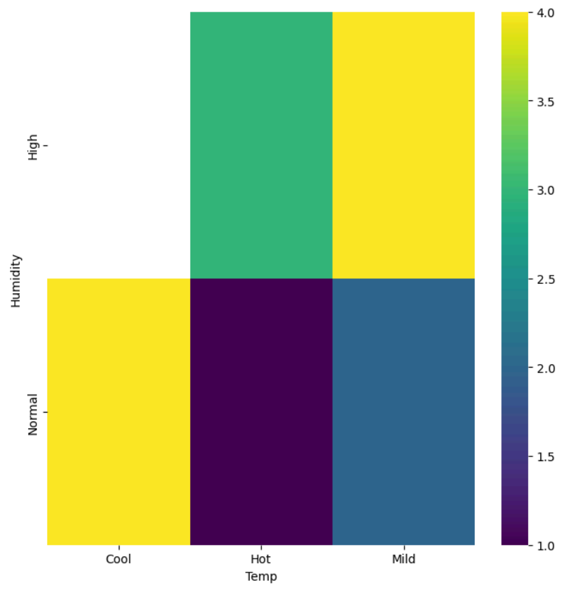


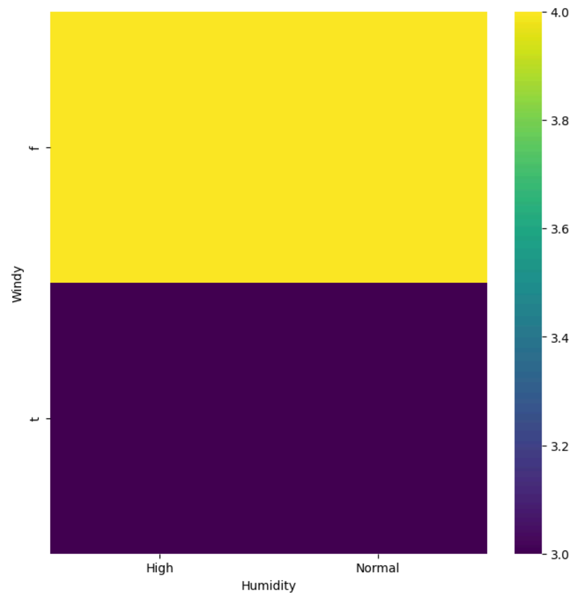


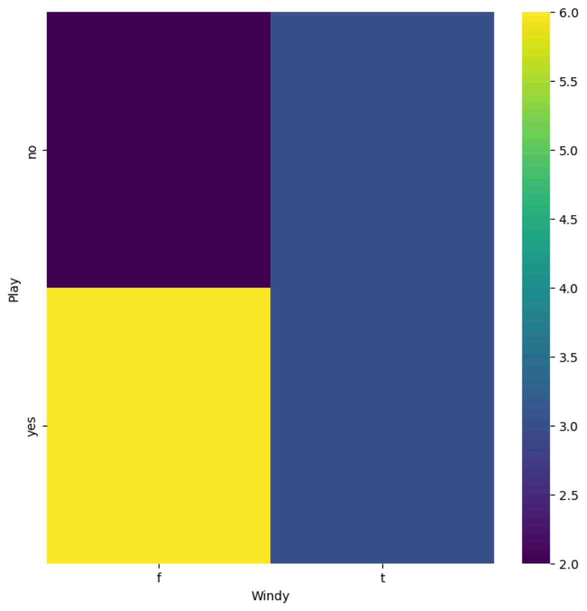


Variation plots among different features:

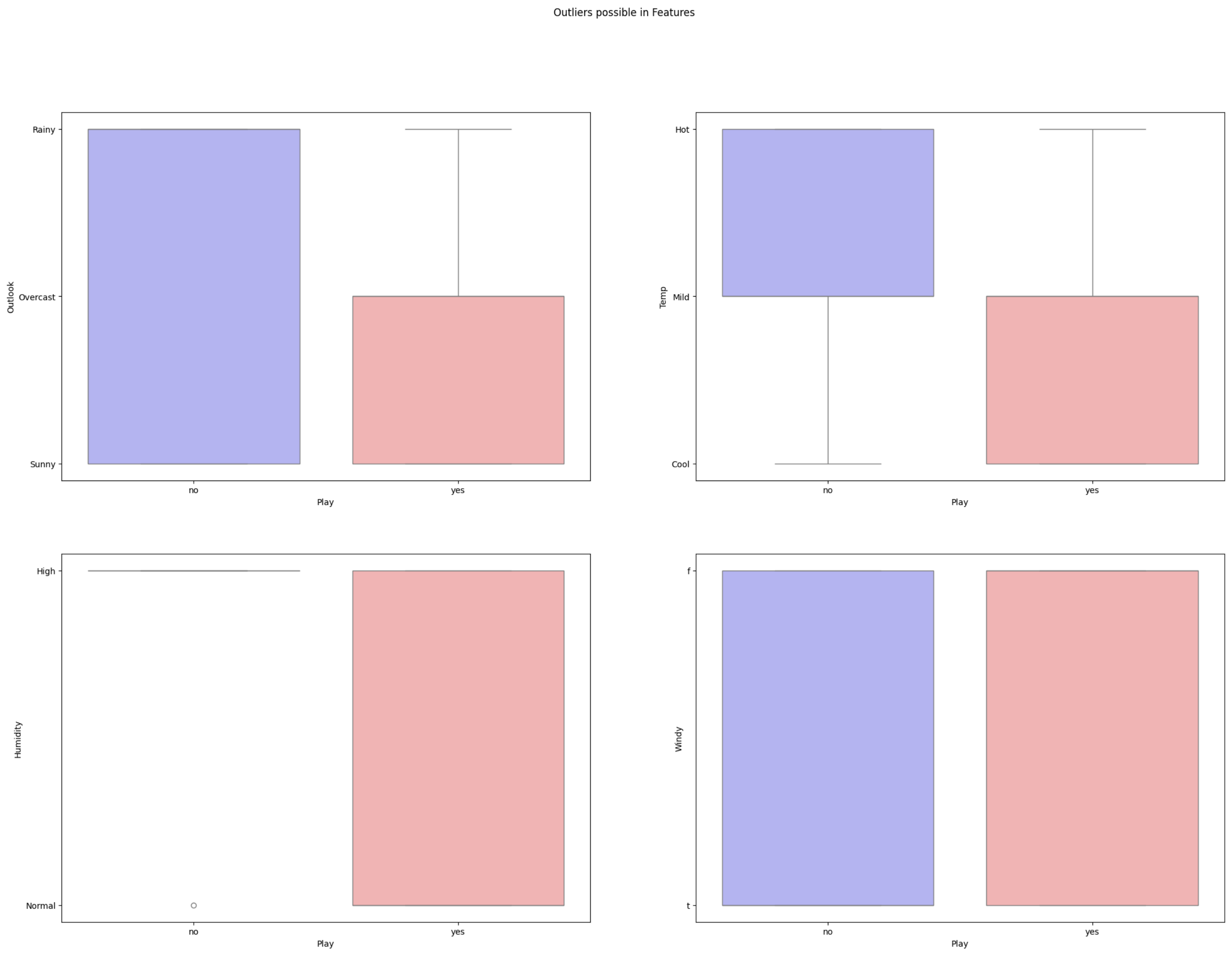








- The dataset is checked for missing values.
- There are no missing values. Hence, we proceed further.
- Outliers existing in each feature for a particular class must be investigated.



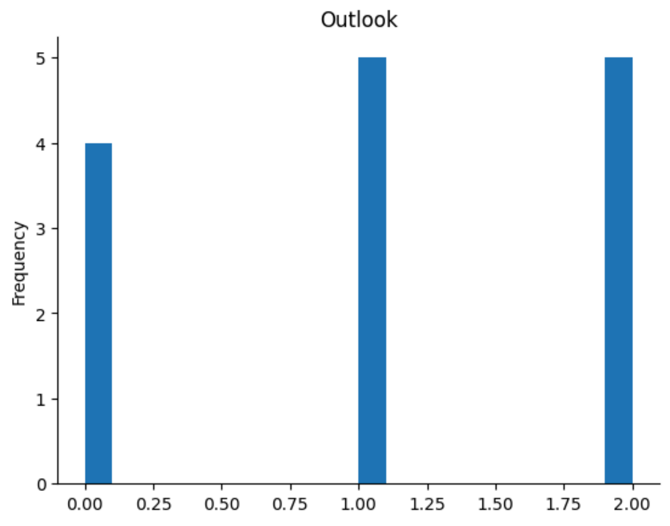
- The box-plots show us that there aren't any outliers present (according to the 1.5IQR rule).
- Since the complete dataset is categorical, I used **LabelEncoder** to encode them into numerical values.

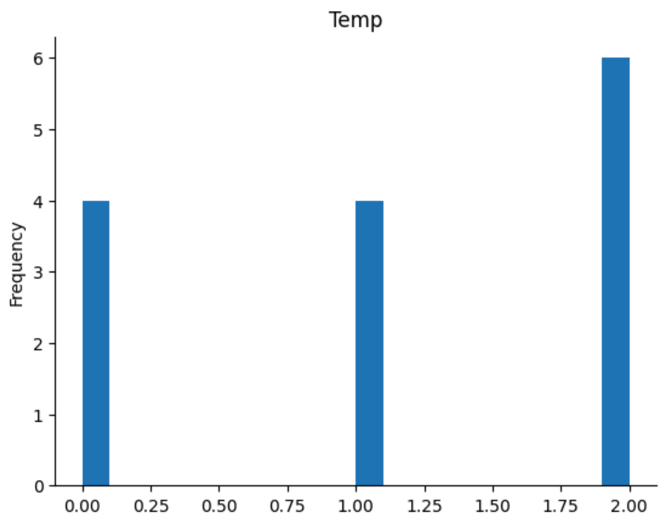
	Outlook	Temp	Humidity	Windy	Play
0	1	1	0	0	0
1	1	1	0	1	0
2	0	1	0	0	1
3	2	2	0	0	1
4	2	0	1	0	1
5	2	0	1	1	0
6	0	0	1	1	1
7	1	2	0	0	0
8	1	0	1	0	1
9	2	2	1	0	1
10	1	2	1	1	1
11	0	2	0	1	1
12	0	1	1	0	1
13	2	2	0	1	0

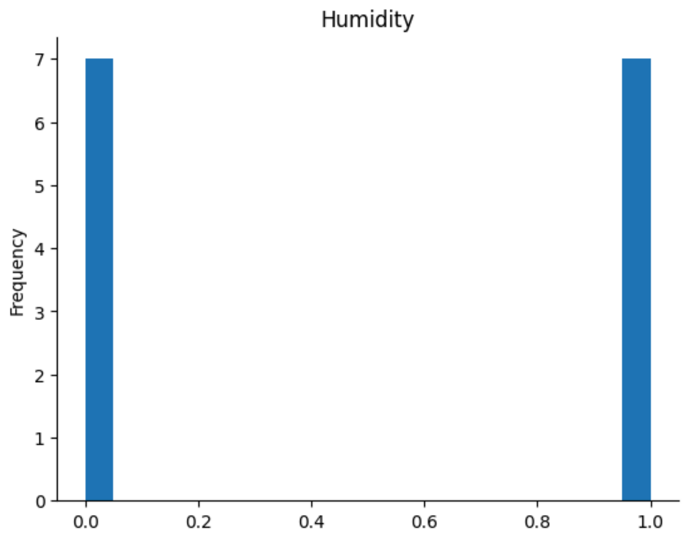
- LabelEncoder uses alphabetical ordering from observation.
- Feature **Outlook** is divided into 3 categories: **Overcast, Rainy and Sunny.**
- Feature **Temp** is divided into 3 categories: **Cool, Hot, Mild.**
- Feature **Humidity** is divided into 2 categories: **High, Normal.**
- Feature **Windy** is divided into two categories: **f, t.**
- Target **Play** is divided into 2 categories: **no, yes.**
- **Using LabelEncoder :-**
  Outlook: 0=Overcast, 1= Rainy, 2= Sunny.
  For Temp: 0=Cool, 1=Hot, 2=Mild.
  For Humidity: 0=High, 1=Normal.

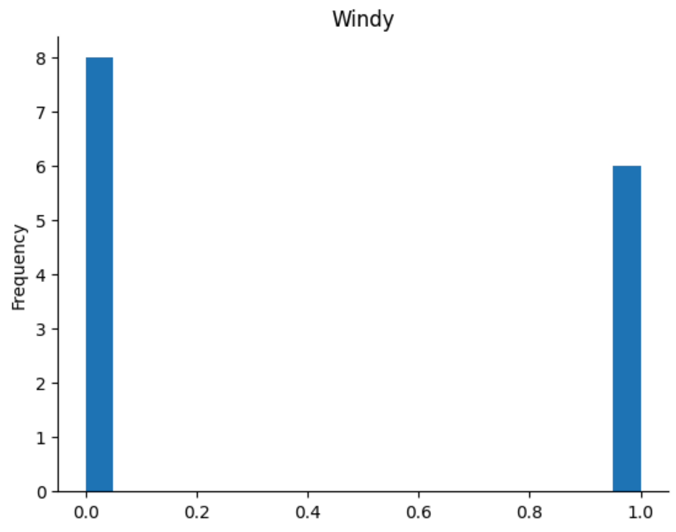
For Windy: 0=f, 1=t.
For Play: 0=no, 1=yes

- **Plots after encoding into numerical values:**









Task-0 (0 pt): Split the dataset into train-test so that randomly chosen 12 out of 14 samples go to train split and the remaining two samples go to test split.

- Splitting the dataset into X features and y labels.

```
X = nb1.drop('Play', axis=1)
y = nb1['Play']
```

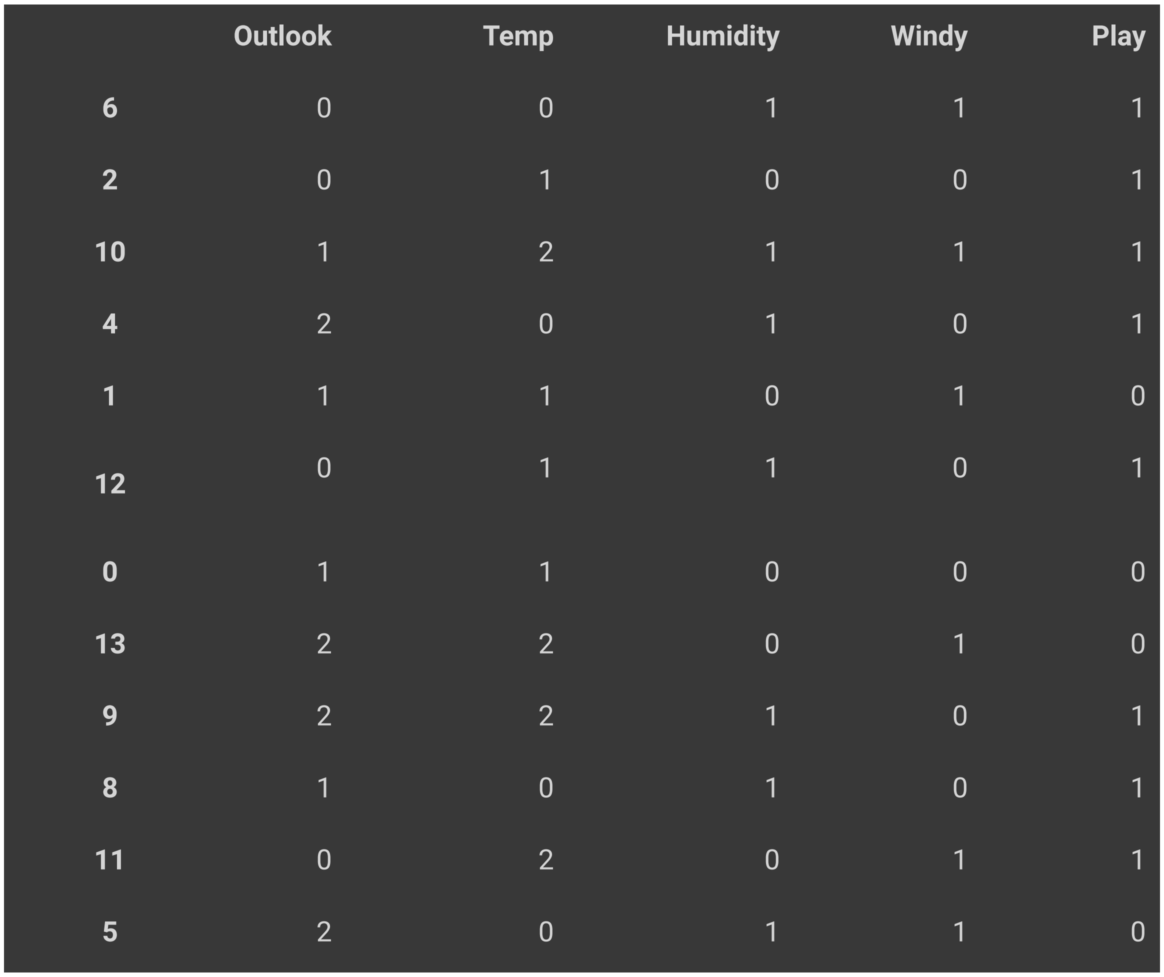
- There are 14 samples in this data set. To keep 2 as test and 12 as train, the test size should be approximately 0.13.
- The entire problem statement is repeated for two random states: **1 and 42.**

RANDOM STATE = 1

Task-1 (5 pts): Calculate Prior Probabilities, i.e. the probability of playing (P(Play=yes)) and not playing (P(Play=no)).

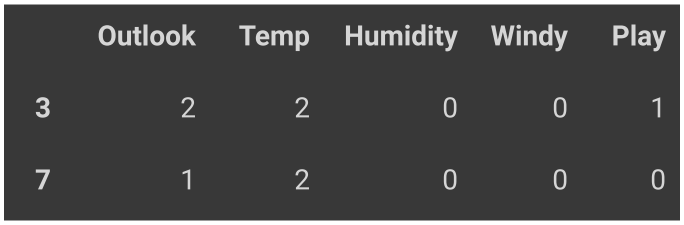
New dataframes of only training samples and test samples are created respectively.

traindf:



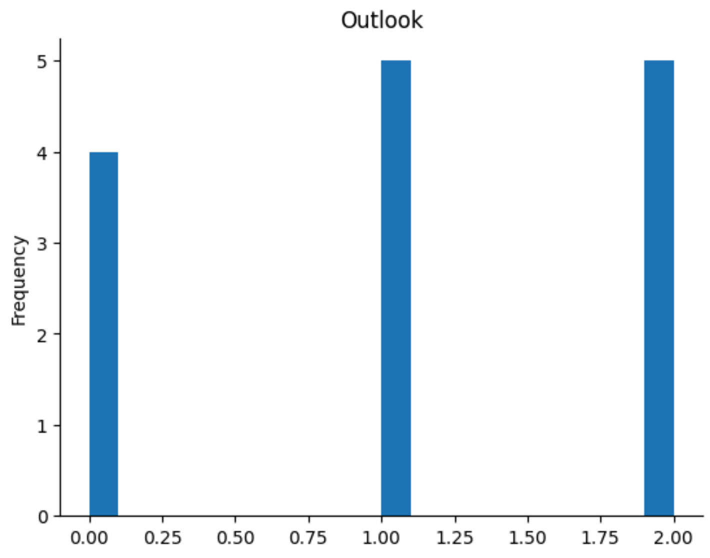
	Outlook	Temp	Humidity	Windy	Play
6	0	0	1	1	1
2	0	1	0	0	1
10	1	2	1	1	1
4	2	0	1	0	1
1	1	1	0	1	0
12	0	1	1	0	1
0	1	1	0	0	0
13	2	2	0	1	0
9	2	2	1	0	1
8	1	0	1	0	1
11	0	2	0	1	1
5	2	0	1	1	0

testdf:



	Outlook	Temp	Humidity	Windy	Play
3	2	2	0	0	1
7	1	2	0	0	0

- Prior probabilities are calculated directly from the classes/labels given in the training dataset.
- `1    8    //1 is simply Play=Yes`
- `0    4    //0 is simply Play=No`
- Hence the probabilities respectively are :-
- `P(Play=Yes):  0.6666666666666666 //Pyes`
- `P(Play=No):   0.3333333333333333 //Pno`



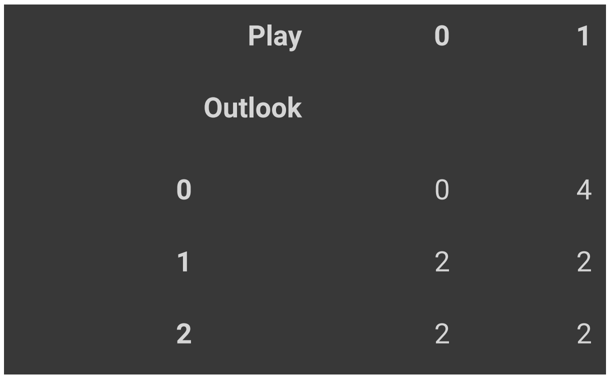
Frequency of classes in Outlook in the training data.

Task-2 (10 pts): Calculate Likelihood Probabilities: i.e. the likelihood probabilities for each feature given the class (Play = yes or Play = no). For example, calculate P(Outlook = Sunny|Play = yes), P(Temperature = Mild|Play = yes), and so on.

- For each given class, the likelihood probability is found by calculating the probability of each feature given that class occurs.
- $$\underbrace{P(c \mid x)}_{\text{Posterior Probability}} = \frac{\overbrace{P(x \mid c)}^{\text{Likelihood}}\,\overbrace{P(c)}^{\text{Class Prior Probability}}}{\underbrace{P(x)}_{\text{Predictor Prior Probability}}}$$

$$P(c \mid X) = P(x_1 \mid c) \times P(x_2 \mid c) \times \cdots \times P(x_n \mid c) \times P(c)$$

- We found Pyes and Pno as the class prior probabilities. For likelihood probabilities, we use the **crosstab()** function to easily group data by a specific column and display the corresponding target values.
- Crosstab for Outlook

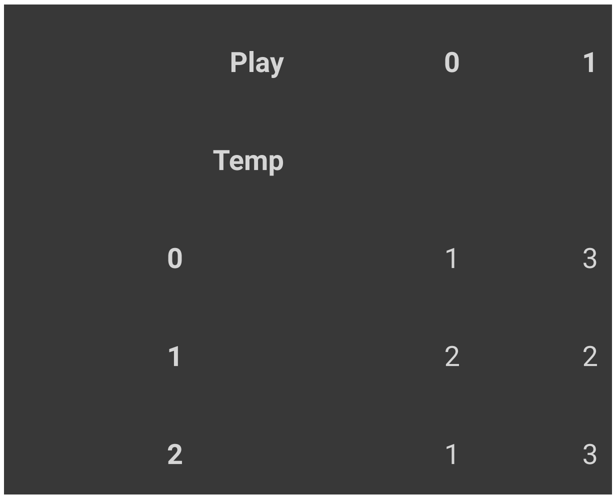


Play	0	1
Outlook		
0	0	4
1	2	2
2	2	2

- From this table, the conditional probabilities are manually assigned according to whichever class/label they belong to.

```
● P(Outlook=Overcast | Play=yes) :  0.5
● P(Outlook=Rainy | Play=yes) :    0.25
● P(Outlook=Sunny | Play=yes) :    0.25
● P(Outlook=Overcast | Play=no):      0
● P(Outlook=Rainy | Play=no) :      0.5
● P(Outlook=Sunny | Play=no) :      0.5
```

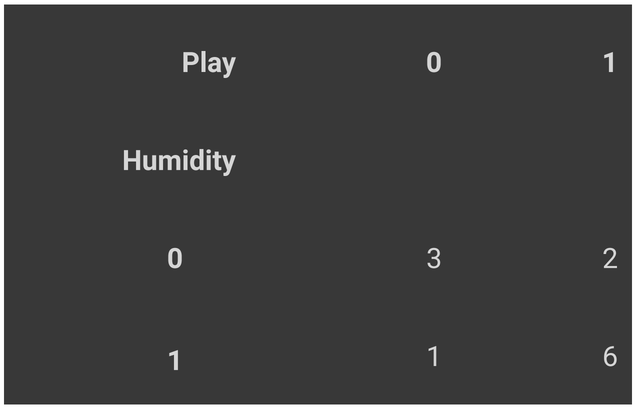
- Crosstab for Temp



Play	0	1
Temp		
0	1	3
1	2	2
2	1	3

```
● P(Temp= Cool | Play=yes) :   0.375
● P(Temp= Hot | Play=yes) :     0.25
● P(Temp= Mild | Play=yes) :   0.375
● P(Temp= Cool | Play=no):      0.25
● P(Temp= Hot | Play=no):        0.5
● P(Temp= Mild | Play=no):      0.25
```

- Crosstab for Humidity



Play	0	1
Humidity		
0	3	2
1	1	6

```
● P(Humidity= High | Play=yes) :     0.25
● P(Humidity= Normal | Play=yes) :   0.75
● P(Humidity= High | Play=no) :      0.75
● P(Humidity= Normal | Play=no) :    0.25
```

- Crosstab for Windy

Play	0	1
Windy		
0	1	5
1	3	3

```
• P(Windy= f | Play=yes) :  0.625
• P(Windy= t | Play=yes) :  0.375
• P(Windy= f | Play=no) :    0.25
• P(Windy= t| Play=no) :     0.75
```

Task-3 (10 pts): Calculate Posterior Probabilities: Using the Naive Bayes formula, calculate the posterior probabilities for both classes (Play = yes and Play = no) for the testing split.

- **testdf**

	Outlook	Temp	Humidity	Windy	Play
3	2	2	0	0	1
7	1	2	0	0	0

- These test cases do not have much variation and are hence, bad. The probability of Outlook being 2 or 1 is the same and all the other features remain the same. Hence we see at the end, that the final posterior probabilities are the same in both test cases.
- Posterior probability is given by P(class | set of features). We look at the specific set of features for the test set and compute P(yes|features) and P(no|features). Whichever one is greater, we assign that label/class to it.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

  - In its simplest form, this is what posterior probability is. P(B)= P(features) remains the same for both posterior probabilities. Hence, we can ignore it during comparison.

- **Test Case 1:**

  We see that Outlook=Sunny, Temp= Mild, Humidity=High, Windy=false

  - **P(yes|sunny, mild, high, false)= P(sunny|yes)P(mild|yes)P(high|yes)P(false|yes)P(yes)/P(features)**
  - **P(no/sunny, mild, high, false) = P(sunny/no)P(mild/no)P(high/no)P(false/no)P(no)/P(features)**
  - **We have all these values except P(features) which is not really needed.**
  - **Yet, it could be easily computed as :- P(Outlook=Sunny)*P(Temp=Mild)*P(Humidity=High)*P(Windy=False) = (4/12)*(4/12)*(5/12)*(6/12)= 0.023 = PX.**

```
Pyes1= (PSunny_yes*PMild_yes*PHigh_yes*Pf_yes*Pyes)/PX
Pno1= (PSunny_no*PMild_no*PHigh_no*Pf_no*Pno)/PX
```

  - **The results are:**

```
P(yes/features for testcase 1):  0.421875
P(no/features for testcase 1):  0.3375
```

  - **P(yes/features)>P(no/features) hence we predict sample 3(test case 1) as Play=yes**

    - Playtestcase 1= 1
    - This is the correct label as seen from the testdf table.
- **Test Case 2:**

  We see that Outlook=Rainy, Temp=Mild, Humidity=High, Windy=false.

    - **P(yes/rainy, mild, high, false)=P(rainy/yes)P(mild/yes)P(high/yes)P(false/yes)P(yes)/PY.**
    - **P(no/rainy, mild, high, false) = P(rainy/no)P(mild/no)P(high/no)P(false/no)P(no)/PY.**
    - **PY= P(features).**
    - **PY=(4/12)*(4/12)*(5/12)*(6/12).**
    - `P(yes/features for testcase 2):  0.421875`
    - `P(no/features for testcase 2):   0.3375`
    - **P(yes/features)>P(no/features) hence we predict sample 7(test case 2) as Play=yes**
    - **Playtestcase 2 = 1**
    - **This is the wrong label as seen from the testdf table.**

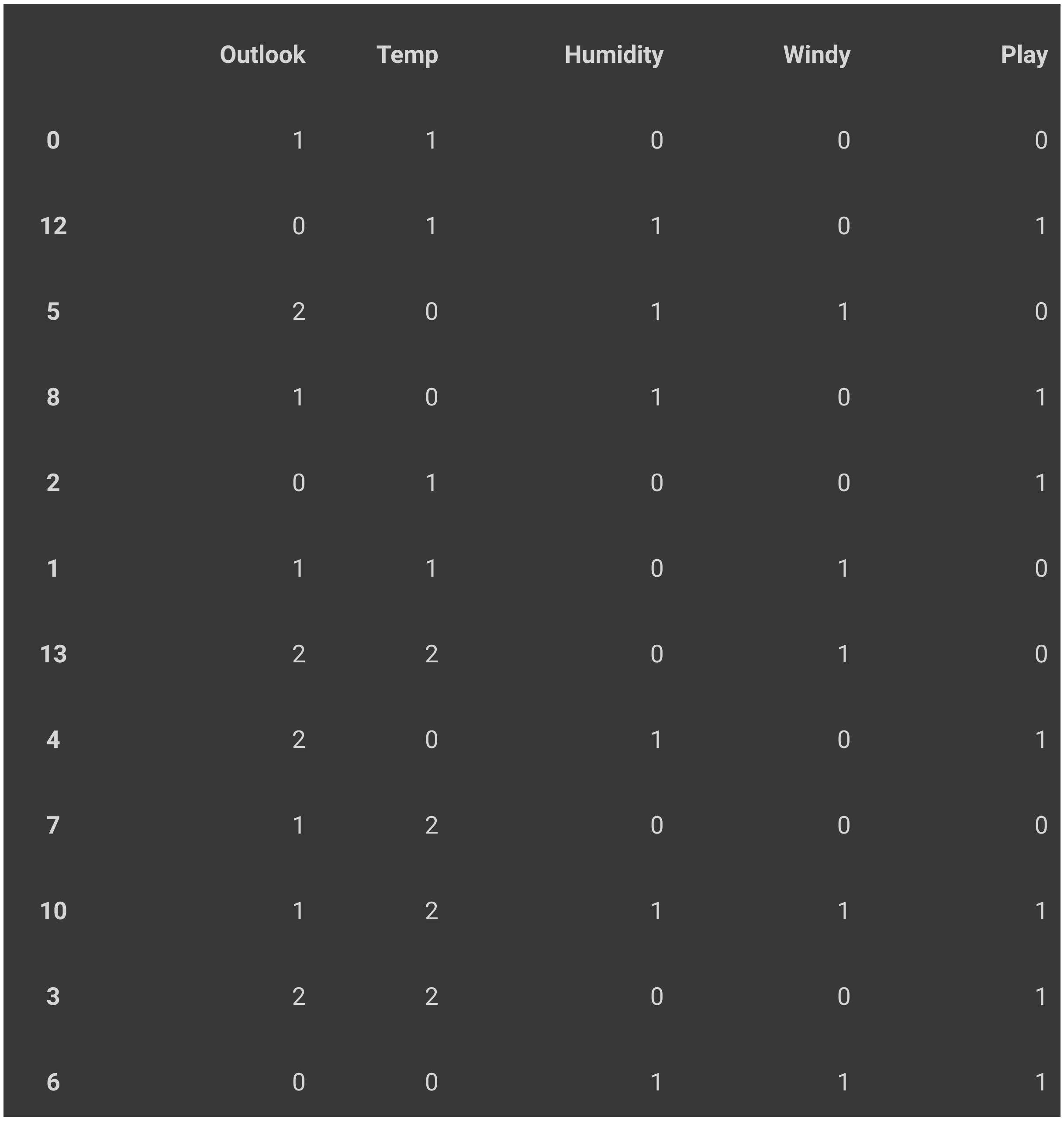
Task-4 (5 pts): Make Predictions: Based on the posterior probabilities, predict whether the given test split examples will result in playing the sport or not.

- TESTCASE 1:
    - `P(yes/features for testcase 1):  0.421875`
    - `P(no/features for testcase 1):  0.3375`
    - P(yes)>P(no). Hence Play=yes.
    - Correct prediction.
- TESTCASE 2:
    - `P(yes/features for testcase 2):  0.421875`
    - `P(no/features for testcase 2):   0.3375`
    - P(yes)> P(no). Hence Play=yes.
    - Wrong prediction.

RANDOM STATE = 42

Task-1 (5 pts): Calculate Prior Probabilities, i.e. the probability of playing (P(Play=yes)) and not playing (P(Play=no)).

traindf2:



	Outlook	Temp	Humidity	Windy	Play
0	1	1	0	0	0
12	0	1	1	0	1
5	2	0	1	1	0
8	1	0	1	0	1
2	0	1	0	0	1
1	1	1	0	1	0
13	2	2	0	1	0
4	2	0	1	0	1
7	1	2	0	0	0
10	1	2	1	1	1
3	2	2	0	0	1
6	0	0	1	1	1

testdf2:

	Outlook	Temp	Humidity	Windy	Play
9	2	2	1	0	1
11	0	2	0	1	1

- Prior probabilities of Play=yes and Play=no

```
Pyesf=7/12
Pnof=5/12
```

Task-2, Task-3, Task-4 are all repeated in the same way.

- All the results are tabulated here :-

```
P(Outlook=Overcast | Play=yes) :  0.42857142857142855
P(Outlook=Rainy | Play=yes) :       0.2857142857142857
P(Outlook=Sunny | Play=yes) :       0.2857142857142857
P(Outlook=Overcast | Play=no):                     0.0
P(Outlook=Rainy | Play=no) :                       0.6
P(Outlook=Sunny | Play=no) :                       0.4
```

```
P(Temp= Cool | Play=yes) :  0.42857142857142855
P(Temp= Hot | Play=yes) :     0.2857142857142857
P(Temp= Mild | Play=yes) :    0.2857142857142857
P(Temp= Cool | Play=no) :                    0.2
P(Temp= Hot | Play=no) :                     0.4
P(Temp= Mild | Play=no) :                    0.4
```

```
P(Humidity= High | Play=yes) :     0.2857142857142857
P(Humidity= Normal | Play=yes) :  0.7142857142857143
P(Humidity= High | Play=no) :                     0.8
P(Humidity= Normal | Play=no) :                   0.2
```

```
P(Windy= f | Play=yes) :  0.7142857142857143
P(Windy= t | Play=yes) :  0.2857142857142857
P(Windy= f | Play=no) :                   0.4
P(Windy= t| Play=no) :                    0.6
```

- Finally, we can compute the posterior probabilities.

```
PX2= (4/12)*(4/12)*(6/12)*(7/12) # probabilities of the given set
#of features.
print("P(features): ",PX2)
Pyesnew1= (PSunny_yes2*PMild_yes2*PNormal_yes2*Pf_yes2*Pyesf)/PX2
Pnonew1= (PSunny_no2*PMild_no2*PNormal_no2*Pf_no2*Pnof)/PX2

print("P(yes/features for testcase 1): ", Pyesnew1)
print("P(no/features for testcase 1): ", Pnonew1)
```

Testcase 1 results:

```
P(features):                  0.032407407407407406
P(yes/features for testcase 1):  0.7496876301541026
P(no/features for testcase 1):  0.16457142857142862
```

- Prediction: Play= Yes.
- Correct prediction.

```
PY2= (3/12)*(4/12)*(6/12)*(5/12)# probabilities of the given set of
#features.
print("P(features): ",PY2)
Pyesnew2= (POverCast_yes2*PMild_yes2*PHigh_yes2*Pt_yes2*Pyesf)/PY2
Pnonew2= (POverCast_no2*PMild_no2*PHigh_no2*Pt_no2*Pnof)/PY2

print("P(yes/features for testcase 1): ", Pyesnew2)
print("P(no/features for testcase 1): ", Pnonew2) #issue here!
```

Testcase 2 results:

```
P(features):                      0.017361111111111112
P(yes/features for testcase 1):   0.3358600583090378
P(no/features for testcase 1):                     0.0
```

- Prediction: Play=Yes.
- Correct prediction.

Table to compare results:

Random state	Test case sample no.	P(yes/features)	P(no/features)	Prediction	Actual label	Accuracy
1	3	0.421875	0.3375	Play=yes	Play=yes	
	7	0.421875	0.3375	Play=yes	Play=no	
						50%
42	9	0.7496876301541026	0.16457142857142862	Play=yes	Play=yes	
	11	0.3358600583090378	0.0	Play=yes	Play=yes	
						100%

Task-5 (10 pts): Use Laplace Smoothing: Laplace smoothing is an essential technique in probabilistic models like Naive Bayes. It mitigates the challenge of zero probabilities for unseen events by introducing a small pseudocount. This adjustment ensures a more reliable and adaptable model, particularly when encountering unobserved combinations of feature values during classification. Incorporating Laplace Smoothing, recalculate the Likelihood and Posterior Probabilities and make predictions on the test split. Report the observed differences in your predictions justify the results in the report.

- What we noticed from the previous example was that P(no/features)=0.
- This was due to P(Outlook=Overcast | Play=no) being 0. The training data did not contain this particular case. If a sample is absent in the training dataset, then we don't have its likelihood.
- To solve this error, we use a technique called Laplace smoothing.
- In statistics, Laplace Smoothing is a technique to smooth categorical data. Laplace Smoothing is introduced to solve the problem of zero probability. By applying this method, prior probability and conditional probability can be written as:

$$\boxed{p_\lambda(C_k) = p_\lambda(Y = C_k) = \frac{\sum_{i=1}^{N} I(y_i = C_k) + \lambda}{N + K\lambda}}$$

$$\boxed{p(x_1 = a_j | y = C_k) = \frac{\sum_{i=1}^{N} I(x_{1i} = a_j, y_i = C_k) + \lambda}{\sum_{i=1}^{N} I(y_i = C_k) + A\lambda}}$$

- **K**: number of different values in Y.
- **A:** denotes the number of different values in aj. Usually the λ in the formula equals to 1 to correct for zero probability.
- Laplace Smoothing will be done on the traindf2 and testdf2 created by using random state = 42.

- **USING λ=1**
    - To calculate Prior Probabilities :-
        - P(Play=yes)= (no. of Play=yes + λ)/(total samples + 2* λ)= (7+1)/(12+2)= 8/14
        - P(Play=no) = 1- P(Play=yes)= 6/14
    - To calculate the likelihoods :-
        - POverCast_yeslp= (3+1)/(7+ 3)
        - PRainy_yeslp= (2+1)/(7 +3)
        - PSunny_yeslp= (2+1)/(7+3)
        - POverCast_nolp= (0+1)/(5+3) #this is usually the place where an error could have occurred. We remove that using laplace smoothing.
        - PRainy_nolp= (3+1)/(5+3)
        - PSunny_nolp= (2+1)/(5+3)
        - `P(Outlook=Overcast | Play=no) after laplace smoothing:  0.125 #not 0 anymore!`
        - Similarly, all other likelihoods are calculated.
    - Making predictions :-
        - Test Case 1: Outlook=Sunny, Temp= Mild, Humidity=Normal, Windy=false.
            - PXe= (4/12)*(4/12)*(6/12)*(7/12).
            - P(features): 0.032407407407407406
            - P(yes/features for testcase 1): **0.7053061224489795**
            - P(no/features for testcase 1): **0.22771761765930862**

        - We notice that the probabilities have gotten closer.
        - Play= yes predicted. Correct prediction.
    - Test Case 2: We see that Outlook=Overcast, Temp= Mild, Humidity=High, Windy=true.
        - PYe=(3/12)*(4/12)*(6/12)*(5/12)
        - P(yes/features for testcase 2): **0.4388571428571427**
        - P(no/features for testcase 2): **0.4723032069970844**
        - **P(no/features) is not 0 anymore.**
        - Play= no predicted. Wrong prediction.
- **USING λ=100**
    - P(yes/features for testcase 1): **0.44234907931576084**
    - P(no/features for testcase 1): **0.41900927608255545**
    - P(yes/features for testcase 2): **0.7868477337212924**
    - P(no/features for testcase 2): **0.7973321141468648**
- **USING λ=1000**
    - P(yes/features for testcase 1): **0.42999476741711434**
    - P(no/features for testcase 1): **0.42757601169858006**
    - P(yes/features for testcase 2): **0.798668349413036**
    - P(no/features for testcase 2): **0.7997333914031256**

FINAL OBSERVATIONS:

For random state 42 train-test split:

- The difference between the likelihood probabilities of features in the same class (Play=yes or Play=no) decreases as you increase λ.
- As λ increases, the likelihood probability moves towards uniform distribution.
- The accuracy dropped after using Laplace smoothing. This is because initially, one probability: P(Play=no|features), was

wrongly assigned to be 0 only because a training sample wasn't present in the data (no sample with [Outlook=Overcast| Play=no] was present). This led to the model assigning the probability 0 to P(no|features) and P(yes|features) seemed to obviously be higher than the former (0.33 compared to 0.0).

- As a result, if any test instance had the label "No" and contained the feature, it would be classified incorrectly since the probability of this combination was zero.
- Although the accuracy was 100% before smoothing, the model predicted it incorrectly simply because it did not have enough training samples.

Smoothing factor λ	**Test Case**	**P(yes|features)**	**P(no|features)**	**Predicted**	**Accuracy**
1	1	0.7053	0.2277	Play=Yes	50%
	2	0.4388	0.4723	Play=No	
100	1	0.4423	0.4190	Play=Yes	50%
	2	0.7868	0.7973	Play=No	
1000	1	0.4299	0.4275	Play=Yes	50%
	2	0.7986	0.7997	Play=No	

- Increasing the training and testing data size would easily solve this issue.
- Laplace smoothing introduces a trade-off between overfitting to the training data (which can lead to 100% accuracy on training data but poor generalization on test data) and smoothing out probabilities to improve generalization.
- In some cases, this trade-off may result in a reduction in accuracy on the training data due to the adjustment of probabilities across all feature-label combinations, but will improve accuracy during testing on a larger scale.
- Naive Bayes assumes all the features are independent of each other. If the model is performing badly, it might be due to some dependencies between the features.

- Correlation Matrix between features of original dataset:

	Outlook	**Temp**	**Humidity**	**Windy**
Outlook	1.000	0.092	8.944272e-02	-7.745967e-02
Temp	0.092036	1.000	-5.144958e-01	2.475369e-02
Humidity	0.089443	-0.514496	1.000	1.602469e-17
Windy	-0.077460	0.024754	1.602469e-17	1.000

- We notice that there does indeed exist dependencies between features.
- Hence, Naive Bayes wouldn't work perfectly on this dataset.

References:

Linear discriminant analysis (LDA) - how to use it as a classifier

Laplace smoothing in Naïve Bayes algorithm | by Vaibhav Jayaswal | Towards Data Science

A Brief Introduction to Linear Discriminant Analysis

Naïve Bayes Classifier with Practical Implementation | by Amir Ali | The Art of Data Scicne | Medium

Naive Bayes working without sklearn libraries :) | Kaggle

How to Identify Box Plot Outliers? Easy Steps

An Introduction to Naïve Bayes Classifier | by Yang | Towards Data Science

THANK YOU