Mahcine Learning Final Project v2

May 15, 2024

1 Machine Learning Final Project - Titanic Dataset Analysis & Classification Algorithms

In this project, our objective is to conduct an in-depth exploration and analysis of the Titanic dataset. We use three distinct classification algorithms—K-Nearest Neighbors (KNN), Naive Bayes, and Support Vector Machine (SVM)—to forecast the survival outcomes of passengers aboard the Titanic. Furthermore, we use an Artificial Neural Network (ANN) for comprehensive analysis. Our study includes a comparative examination of the performance of these algorithms based on various evaluation metrics.

2 Importing Important Libraries

[48]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split,GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
import tensorflow as tf
from sklearn.metrics import␣
↪accuracy_score,f1_score,precision_score,recall_score,classification_report
import seaborn as sns
sns.set_style('darkgrid')
```

3 Loading Data from .csv files

[49]:
```
train = pd.read_csv('titanic.csv')
test = pd.read_csv('test.csv')
```

[50]:
```
train.info()
print("-"*15)
test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1309 entries, 0 to 1308
```

```
Data columns (total 11 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   pclass    1309 non-null   int64
 1   name      1309 non-null   object
 2   sex       1309 non-null   object
 3   age       1046 non-null   float64
 4   sibsp     1309 non-null   int64
 5   parch     1309 non-null   int64
 6   ticket    1309 non-null   object
 7   fare      1308 non-null   float64
 8   cabin     295 non-null    object
 9   embarked  1307 non-null   object
 10  survived  1309 non-null   int64
dtypes: float64(2), int64(4), object(5)
memory usage: 112.6+ KB
---------------
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 19 entries, 0 to 18
Data columns (total 10 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   pclass    19 non-null     int64
 1   name      19 non-null     object
 2   sex       19 non-null     object
 3   age       18 non-null     float64
 4   sibSp     19 non-null     int64
 5   parch     19 non-null     int64
 6   ticket    19 non-null     object
 7   fare      19 non-null     float64
 8   cabin     2 non-null      object
 9   embarked  19 non-null     object
dtypes: float64(2), int64(3), object(5)
memory usage: 1.6+ KB
```

4 Getting statistical information about the data

[51]:
```
train.describe(include='all')
```

[51]:
```
             pclass                 name   sex          age        sibsp  \
count   1309.000000                 1309  1309  1046.000000  1309.000000
unique          NaN                 1307     2          NaN          NaN
top             NaN  Connolly, Miss. Kate  male          NaN          NaN
freq            NaN                    2   843          NaN          NaN
mean       2.294882                  NaN   NaN    29.881135     0.498854
std        0.837836                  NaN   NaN    14.413500     1.041658
min        1.000000                  NaN   NaN     0.166700     0.000000
```

```
25%        2.000000                    NaN   NaN   21.000000    0.000000
50%        3.000000                    NaN   NaN   28.000000    0.000000
75%        3.000000                    NaN   NaN   39.000000    1.000000
max        3.000000                    NaN   NaN   80.000000    8.000000

              parch    ticket         fare        cabin embarked     survived
count   1309.000000      1309  1308.000000          295     1307  1309.000000
unique          NaN       929          NaN          186        3          NaN
top             NaN  CA. 2343          NaN  C23 C25 C27        S          NaN
freq            NaN        11          NaN            6      914          NaN
mean       0.385027       NaN    33.295479          NaN      NaN     0.381971
std        0.865560       NaN    51.758668          NaN      NaN     0.486055
min        0.000000       NaN     0.000000          NaN      NaN     0.000000
25%        0.000000       NaN     7.895800          NaN      NaN     0.000000
50%        0.000000       NaN    14.454200          NaN      NaN     0.000000
75%        0.000000       NaN    31.275000          NaN      NaN     1.000000
max        9.000000       NaN   512.329200          NaN      NaN     1.000000
```

[52]:
```python
test.describe(include='all')
```

[52]:
```
           pclass              name   sex        age      sibSp      parch  \
count   19.000000                19    19  18.000000  19.000000  19.000000
unique        NaN                19     2        NaN        NaN        NaN
top           NaN  Kelly, Mr. James  male        NaN        NaN        NaN
freq          NaN                 1    11        NaN        NaN        NaN
mean     2.421053               NaN   NaN  32.638889   0.526316   0.105263
std      0.768533               NaN   NaN  14.576175   0.611775   0.315302
min      1.000000               NaN   NaN  14.000000   0.000000   0.000000
25%      2.000000               NaN   NaN  22.250000   0.000000   0.000000
50%      3.000000               NaN   NaN  27.000000   0.000000   0.000000
75%      3.000000               NaN   NaN  43.250000   1.000000   0.000000
max      3.000000               NaN   NaN  63.000000   2.000000   1.000000

        ticket       fare cabin embarked
count       19  19.000000     2       19
unique      19        NaN     2        3
top     330911        NaN   B45        S
freq         1        NaN     1       12
mean       NaN  20.066232   NaN      NaN
std        NaN  20.221058   NaN      NaN
min        NaN   7.000000   NaN      NaN
25%        NaN   7.862500   NaN      NaN
50%        NaN   9.687500   NaN      NaN
75%        NaN  26.000000   NaN      NaN
max        NaN  82.266700   NaN      NaN
```

[53]:
```python
train
```

```
[53]:       pclass                                             name     sex  \
0          1                    Allen, Miss. Elisabeth Walton  female
1          1                   Allison, Master. Hudson Trevor    male
2          1                     Allison, Miss. Helen Loraine  female
3          1             Allison, Mr. Hudson Joshua Creighton    male
4          1  Allison, Mrs. Hudson J C (Bessie Waldo Daniels)  female
…        …                                                …     …
1304       3                             Zabour, Miss. Hileni  female
1305       3                            Zabour, Miss. Thamine  female
1306       3                        Zakarian, Mr. Mapriededer    male
1307       3                              Zakarian, Mr. Ortin    male
1308       3                               Zimmerman, Mr. Leo    male

          age  sibsp  parch  ticket      fare    cabin embarked  survived
0     29.0000      0      0   24160  211.3375       B5        S         1
1      0.9167      1      2  113781  151.5500  C22 C26        S         1
2      2.0000      1      2  113781  151.5500  C22 C26        S         0
3     30.0000      1      2  113781  151.5500  C22 C26        S         0
4     25.0000      1      2  113781  151.5500  C22 C26        S         0
…         …    …     …     …       …      …       …
1304  14.5000      1      0    2665   14.4542      NaN        C         0
1305      NaN      1      0    2665   14.4542      NaN        C         0
1306  26.5000      0      0    2656    7.2250      NaN        C         0
1307  27.0000      0      0    2670    7.2250      NaN        C         0
1308  29.0000      0      0  315082    7.8750      NaN        S         0

[1309 rows x 11 columns]
```

```
[54]: test
```

```
[54]:     pclass                                             name     sex   age  \
0        3                                  Kelly, Mr. James    male  34.5
1        3                  Wilkes, Mrs. James (Ellen Needs)  female  47.0
2        2                         Myles, Mr. Thomas Francis    male  62.0
3        3                                  Wirz, Mr. Albert    male  27.0
4        3      Hirvonen, Mrs. Alexander (Helga E Lindqvist)  female  22.0
5        3                        Svensson, Mr. Johan Cervin    male  14.0
6        3                              Connolly, Miss. Kate  female  30.0
7        2                      Caldwell, Mr. Albert Francis    male  26.0
8        3         Abrahim, Mrs. Joseph (Sophie Halaut Easu)  female  18.0
9        3                           Davies, Mr. John Samuel    male  21.0
10       3                                  Ilieff, Mr. Ylio    male   NaN
11       1                        Jones, Mr. Charles Cresson    male  46.0
12       1       Snyder, Mrs. John Pillsbury (Nelle Stevenson)  female  23.0
13       2                              Howard, Mr. Benjamin    male  63.0
14       1  Chaffee, Mrs. Herbert Fuller (Carrie Constance…  female  47.0
15       2       del Carlo, Mrs. Sebastiano (Argenia Genovesi)  female  24.0
```

```
16       2                           Keane, Mr. Daniel    male  35.0
17       3                           Assaf, Mr. Gerios    male  21.0
18       3                 Ilmakangas, Miss. Ida Livija  female  27.0

    sibSp  parch            ticket     fare cabin embarked
0       0      0            330911   7.8292   NaN        Q
1       1      0            363272   7.0000   NaN        S
2       0      0            240276   9.6875   NaN        Q
3       0      0            315154   8.6625   NaN        S
4       1      1           3101298  12.2875   NaN        S
5       0      0              7538   9.2250   NaN        S
6       0      0            330972   7.6292   NaN        Q
7       1      1            248738  29.0000   NaN        S
8       0      0              2657   7.2292   NaN        C
9       2      0         A/4 48871  24.1500   NaN        S
10      0      0            349220   7.8958   NaN        S
11      0      0               694  26.0000   NaN        S
12      1      0             21228  82.2667   B45        S
13      1      0             24065  26.0000   NaN        S
14      1      0       W.E.P. 5734  61.1750   E31        S
15      1      0     SC/PARIS 2167  27.7208   NaN        C
16      0      0            233734  12.3500   NaN        Q
17      0      0              2692   7.2250   NaN        C
18      1      0  STON/O2. 3101270   7.9250   NaN        S
```

5 Preprocessing:-

6 Fixing incorrect column name in test data "sibSp"

[55]:
```
test.rename(columns={'sibSp': 'sibsp'}, inplace=True)
```

[56]:
```
test
```

[56]:
```
    pclass                                          name     sex   age  \
0        3                              Kelly, Mr. James    male  34.5
1        3              Wilkes, Mrs. James (Ellen Needs)  female  47.0
2        2                     Myles, Mr. Thomas Francis    male  62.0
3        3                              Wirz, Mr. Albert    male  27.0
4        3  Hirvonen, Mrs. Alexander (Helga E Lindqvist)  female  22.0
5        3                    Svensson, Mr. Johan Cervin    male  14.0
6        3                          Connolly, Miss. Kate  female  30.0
7        2                  Caldwell, Mr. Albert Francis    male  26.0
8        3     Abrahim, Mrs. Joseph (Sophie Halaut Easu)  female  18.0
9        3                       Davies, Mr. John Samuel    male  21.0
10       3                              Ilieff, Mr. Ylio    male   NaN
11       1                    Jones, Mr. Charles Cresson    male  46.0
```

```
12       1      Snyder, Mrs. John Pillsbury (Nelle Stevenson)  female  23.0
13       2                                Howard, Mr. Benjamin    male  63.0
14       1  Chaffee, Mrs. Herbert Fuller (Carrie Constance…  female  47.0
15       2      del Carlo, Mrs. Sebastiano (Argenia Genovesi)  female  24.0
16       2                                   Keane, Mr. Daniel    male  35.0
17       3                                   Assaf, Mr. Gerios    male  21.0
18       3                        Ilmakangas, Miss. Ida Livija  female  27.0

    sibsp  parch            ticket     fare cabin embarked
0       0      0            330911   7.8292   NaN        Q
1       1      0            363272   7.0000   NaN        S
2       0      0            240276   9.6875   NaN        Q
3       0      0            315154   8.6625   NaN        S
4       1      1           3101298  12.2875   NaN        S
5       0      0              7538   9.2250   NaN        S
6       0      0            330972   7.6292   NaN        Q
7       1      1            248738  29.0000   NaN        S
8       0      0              2657   7.2292   NaN        C
9       2      0         A/4 48871  24.1500   NaN        S
10      0      0            349220   7.8958   NaN        S
11      0      0               694  26.0000   NaN        S
12      1      0             21228  82.2667   B45        S
13      1      0             24065  26.0000   NaN        S
14      1      0       W.E.P. 5734  61.1750   E31        S
15      1      0     SC/PARIS 2167  27.7208   NaN        C
16      0      0            233734  12.3500   NaN        Q
17      0      0              2692   7.2250   NaN        C
18      1      0  STON/O2. 3101270   7.9250   NaN        S
```

7 Checking the percentage of missing values

[57]:
```python
# Percentage of null values in training set
missing_percentage_train = (train.isnull().sum() / len(train)) * 100
print("Percentage of missing values for training data:")
print(missing_percentage_train)
print("-"*15)

# Percentage of null values in testing set
missing_percentage_test = (test.isnull().sum() / len(test)) * 100
print("Percentage of missing values for testing data:")
print(missing_percentage_test)
```

```
Percentage of missing values for training data:
pclass       0.000000
name         0.000000
sex          0.000000
age         20.091673
```

```
sibsp        0.000000
parch        0.000000
ticket       0.000000
fare         0.076394
cabin       77.463713
embarked     0.152788
survived     0.000000
dtype: float64
---------------
Percentage of missing values for testing data:
pclass       0.000000
name         0.000000
sex          0.000000
age          5.263158
sibsp        0.000000
parch        0.000000
ticket       0.000000
fare         0.000000
cabin       89.473684
embarked     0.000000
dtype: float64
```

8 Handling Missing Values

8.1 Handling cabin

The feauture "cabin" has an overwhelming majority of null values, thus providing little value to the dataset. We can deal with that problem by removing the feature.

[58]:
```
train.drop('cabin', axis=1, inplace=True)
test.drop('cabin', axis=1, inplace=True)
print(f"Training Data Shape: {train.shape}")
print(f"testing Data Shape: {test.shape}")
```

```
Training Data Shape: (1309, 10)
testing Data Shape: (19, 9)
```

8.2 Handling missing values in other feautures

[59]:
```
# Impute the missing values in the "age" column in training and testing sets␣
↪with the mean value of the column
mean_age = train['age'].mean()
train['age'] = train['age'].fillna(mean_age)
test['age'] = test['age'].fillna(mean_age)

# Impute missing values with mode
train['embarked'].fillna(train['embarked'].mode()[0], inplace=True)
```

```python
train['fare'].fillna(train['fare'].mode()[0], inplace=True)
test['fare'].fillna(train['fare'].mode()[0], inplace=True)
```

```
C:\Users\Lenovo\AppData\Local\Temp\ipykernel_22904\2698931947.py:7:
FutureWarning: A value is trying to be set on a copy of a DataFrame or Series
through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using
'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value)
instead, to perform the operation inplace on the original object.


  train['embarked'].fillna(train['embarked'].mode()[0], inplace=True)
C:\Users\Lenovo\AppData\Local\Temp\ipykernel_22904\2698931947.py:9:
FutureWarning: A value is trying to be set on a copy of a DataFrame or Series
through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using
'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value)
instead, to perform the operation inplace on the original object.


  train['fare'].fillna(train['fare'].mode()[0], inplace=True)
C:\Users\Lenovo\AppData\Local\Temp\ipykernel_22904\2698931947.py:10:
FutureWarning: A value is trying to be set on a copy of a DataFrame or Series
through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never work
because the intermediate object on which we are setting values always behaves as
a copy.

For example, when doing 'df[col].method(value, inplace=True)', try using
'df.method({col: value}, inplace=True)' or df[col] = df[col].method(value)
instead, to perform the operation inplace on the original object.


  test['fare'].fillna(train['fare'].mode()[0], inplace=True)
```

8.3 Final missing value check

[60]:
```
# Percentage of null values in training set
missing_percentage = (train.isnull().sum() / len(test)) * 100
print("Percentage of missing values for training data:")
print(missing_percentage)
print("-"*15)

# Percentage of null values in testing set
missing_percentage = (test.isnull().sum() / len(test)) * 100
print("Percentage of missing values for testing data:")
print(missing_percentage)
```

```
Percentage of missing values for training data:
pclass      0.0
name        0.0
sex         0.0
age         0.0
sibsp       0.0
parch       0.0
ticket      0.0
fare        0.0
embarked    0.0
survived    0.0
dtype: float64
---------------
Percentage of missing values for testing data:
pclass      0.0
name        0.0
sex         0.0
age         0.0
sibsp       0.0
parch       0.0
ticket      0.0
fare        0.0
embarked    0.0
dtype: float64
```

8.4 Checking for duplicates values

[61]:
```
print(train.duplicated().sum())
print(test.duplicated().sum())
```

```
0
0
```

9 Selecting numerical Features

[62]:
```
numerical_columns = train.select_dtypes(include=['int64','float64']).columns
```

10 Handling outliers

Here, we are using a function designed to detect and eliminate outliers. This function accepts three parameters: the dataset, the selected numerical columns, and an optional threshold value (default set to 1.5).

For each numerical column, the function computes the first quartile (q1), third quartile (q3), and interquartile range (IQR). Then, it establishes lower and upper bounds based on the IQR and the provided threshold. Using these bounds, a mask (outliers_mask) is created to identify rows containing outlier values.

The dataset is then updated by removing these outlier rows, and the index is reset using "reset_index" to ensure that the outlier indices are dropped. Finally, the updated dataset is returned.

At the end, the number of removed rows is printed.

[63]:
```
numerical_columns = train.select_dtypes(include=['int64','float64']).columns
categorical_cols =  train.select_dtypes(include=['object']).columns

def remove_outliers_IQR(original_data, numerical_columns, threshold=1.5):
    for col in numerical_columns:
        q1 = original_data[col].quantile(0.25)
        q3 = original_data[col].quantile(0.75)
        IQR = q3 - q1
        lower_bound = q1 - threshold * IQR
        upper_bound = q3 + threshold * IQR
        outliers_mask = (original_data[col] < lower_bound) |
  ↪(original_data[col] > upper_bound)
        original_data = original_data[~outliers_mask].reset_index(drop=True)
    return original_data

# Applying the function to remove outliers
new_train = remove_outliers_IQR(train, numerical_columns)

# Displaying the number of outliers removed from each numerical column
for col in numerical_columns:
    outliers_removed = len(train[col]) - len(new_train[col])
    print(f"Number of outliers removed in {col}: {outliers_removed}")
```

```
Number of outliers removed in pclass: 477
Number of outliers removed in age: 477
Number of outliers removed in sibsp: 477
Number of outliers removed in parch: 477
Number of outliers removed in fare: 477
Number of outliers removed in survived: 477
```

10.1 Geting a summary of statistical information about Numerical Columns

[64]:
```
new_train.describe(include='number')
```

[64]:

	pclass	age	sibsp	parch	fare	survived
count	832.000000	832.000000	832.000000	832.0	832.000000	832.000000
mean	2.510817	29.430423	0.191106	0.0	13.828785	0.283654
std	0.719258	8.182520	0.447770	0.0	10.370550	0.451042
min	1.000000	5.000000	0.000000	0.0	0.000000	0.000000
25%	2.000000	24.000000	0.000000	0.0	7.775000	0.000000
50%	3.000000	29.881135	0.000000	0.0	8.658350	0.000000
75%	3.000000	32.000000	0.000000	0.0	15.500000	1.000000
max	3.000000	54.000000	2.000000	0.0	53.100000	1.000000

10.2 Getting a summary of statistical information about Categorical Columns

[65]:
```
new_train.describe(include='object')
```

[65]:

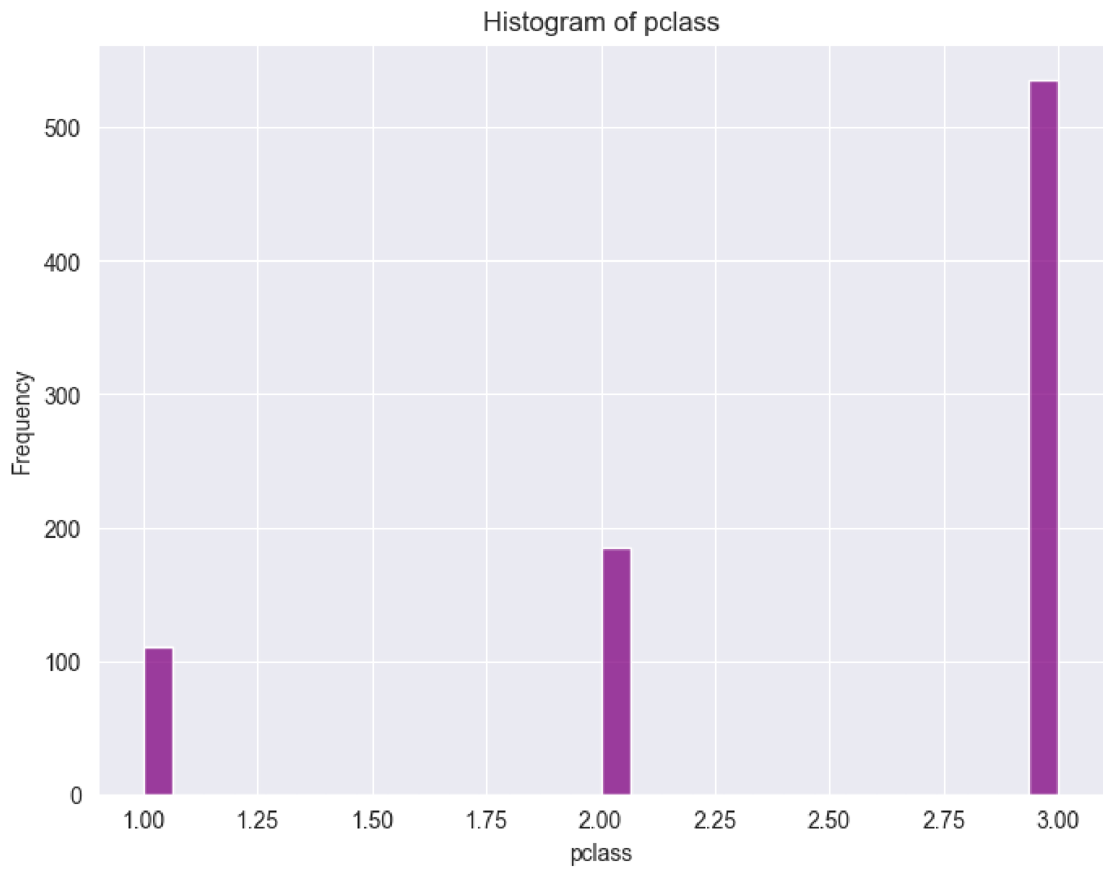
	name	sex	ticket	embarked
count	832	832	832	832
unique	830	2	747	3
top	Connolly, Miss. Kate	male	LINE	S
freq	2	620	4	602

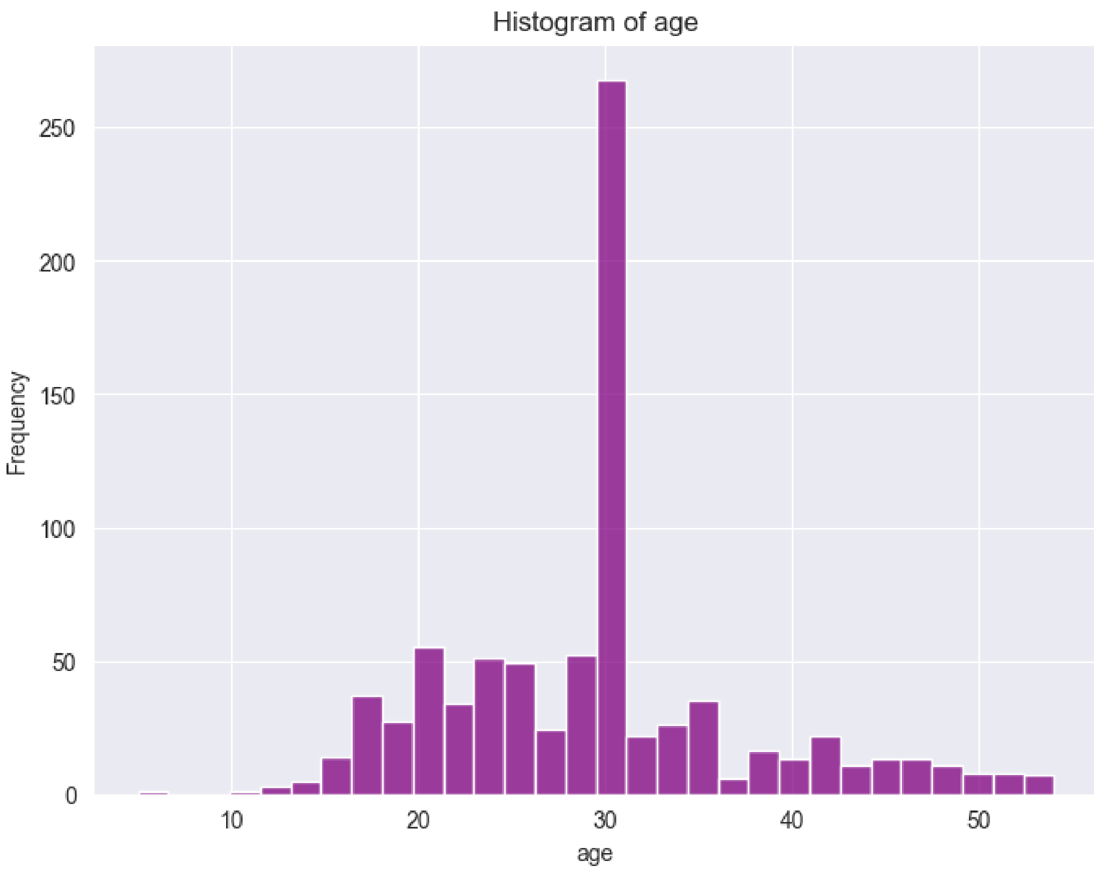
11 Visualization and Analysis:-

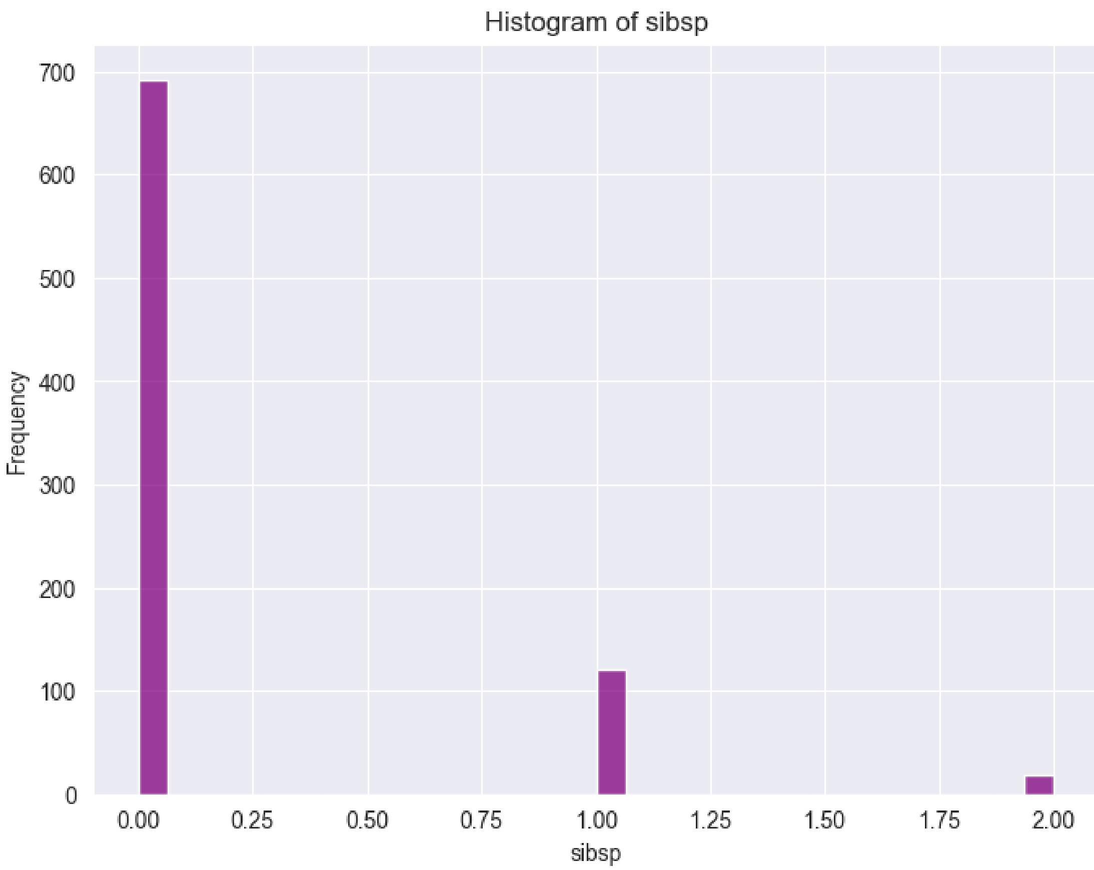
12 Distributions of Numerical columns

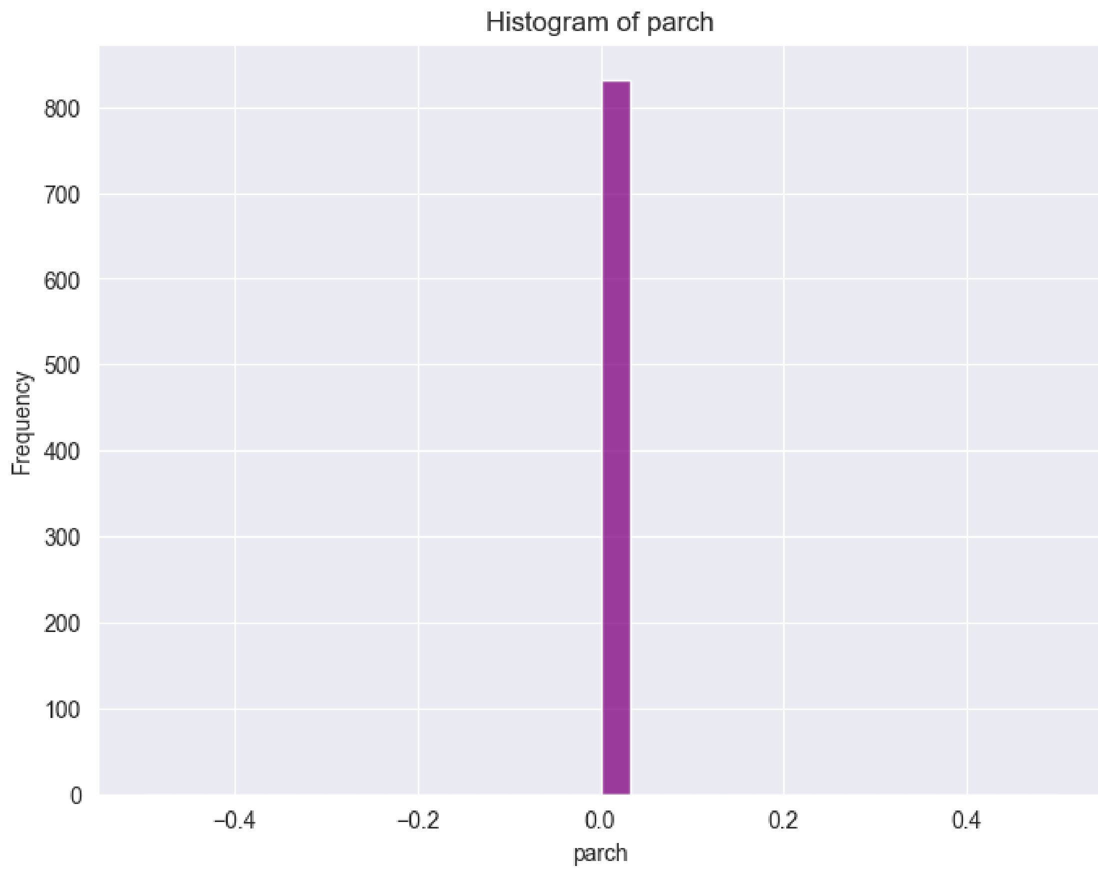
In this step, we plot histograms for various numerical columns within our dataset to gain a deeper understanding of the data distribution.

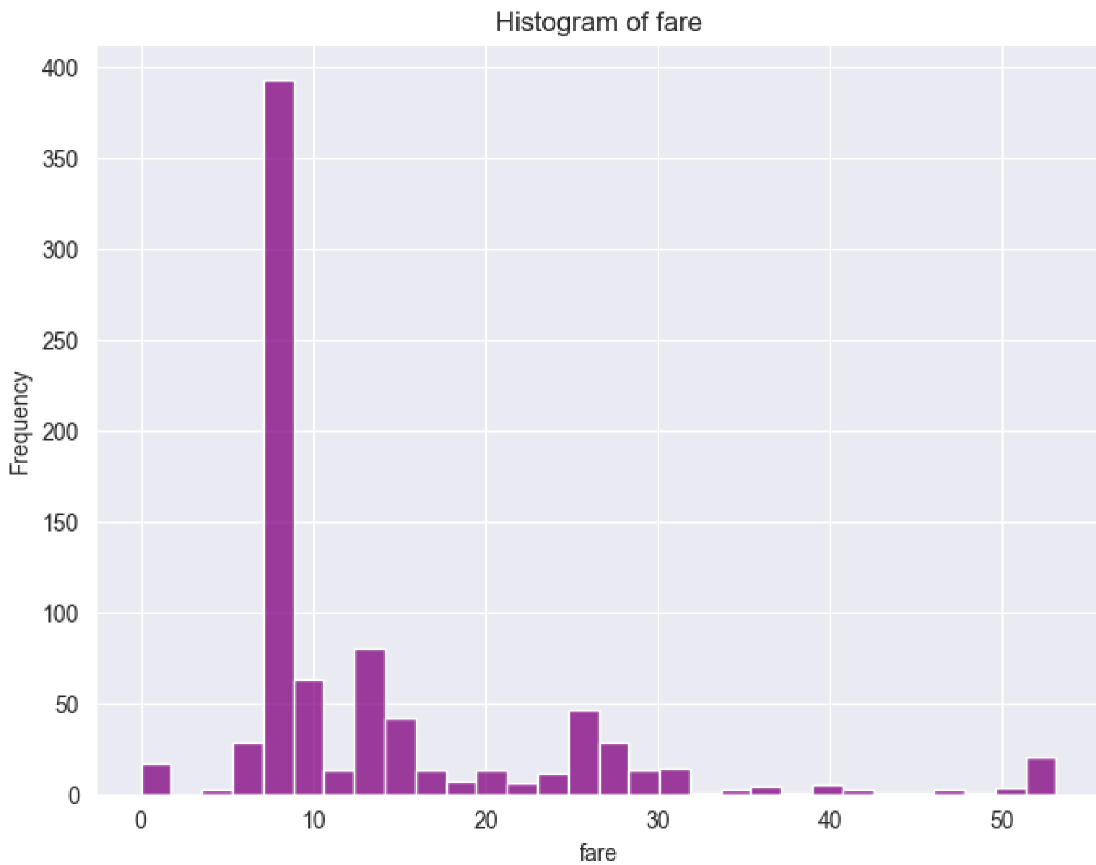
[66]:
```
for col in numerical_columns:
    plt.figure(figsize=(8, 6))
    sns.histplot(new_train[col], bins=30,color='purple')
    plt.title(f'Histogram of {col}')
    plt.xlabel(col)
    plt.ylabel('Frequency')
    plt.show()
```

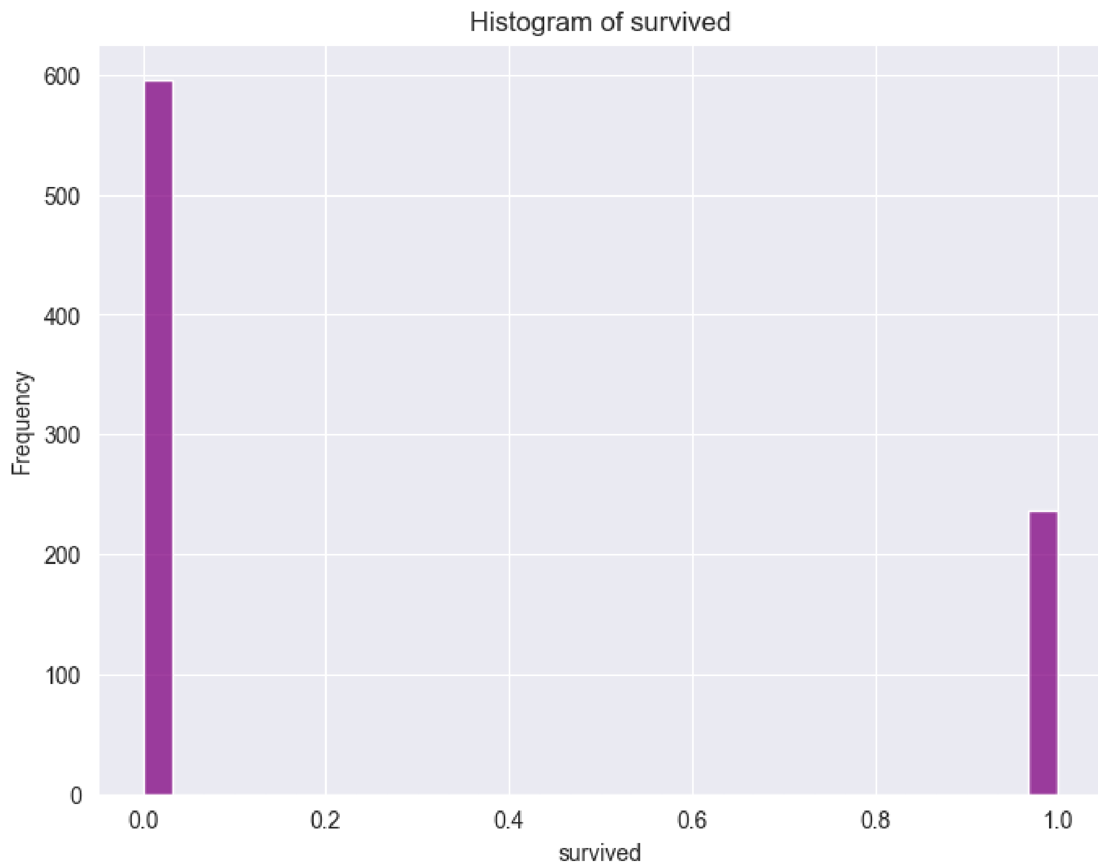

Histogram of pclass
Frequency
500
400
300
200
100
0
1.00
1.25
1.50
1.75
2.00
2.25
2.50
2.75
3.00
pclass


Histogram of age
Frequency
0
50
100
150
200
250
10
20
30
40
50
age


Histogram of sibsp
Frequency
0
100
200
300
400
500
600
700
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
sibsp


Histogram of parch
Frequency
800
700
600
500
400
300
200
100
0
−0.4
−0.2
0.0
0.2
0.4
parch

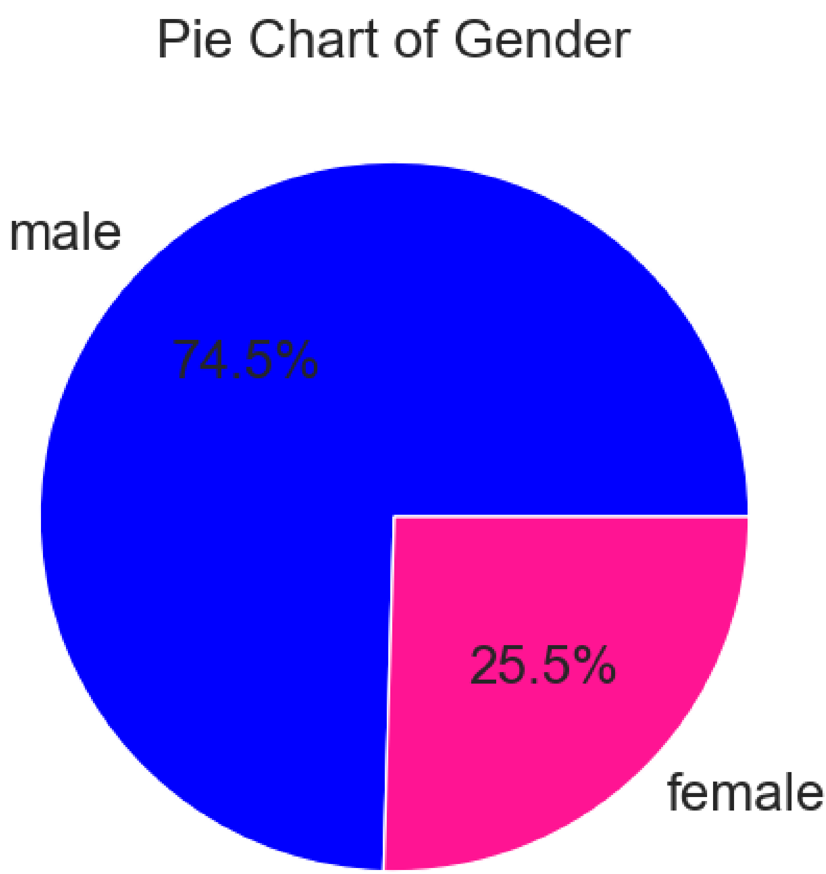

Histogram of fare
Frequency
fare
0
50
100
150
200
250
300
350
400
0
10
20
30
40
50



13 Distribution of categorical columns

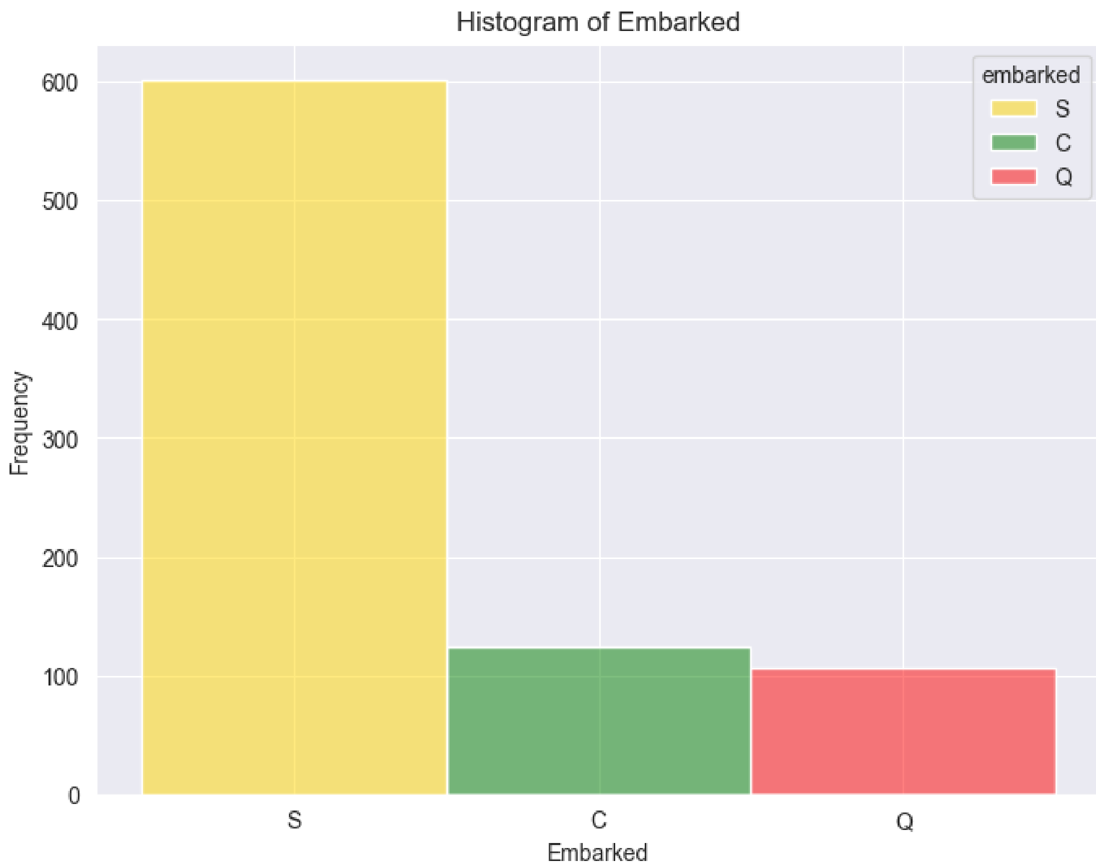
In the following figure we notice that a majority (74.5%) of the passengers were male compared to the 25.5% female.

[67]:
```
# Plotting using a pie chart
gender_counts = new_train['sex'].value_counts()

plt.figure(figsize=(8, 6))
plt.pie(gender_counts, labels = gender_counts.index, textprops={'fontsize':␣
↪20}, autopct='%1.1f%%', colors = ['blue', 'deeppink'])
plt.title('Pie Chart of Gender', fontsize = 20)
plt.show()
```



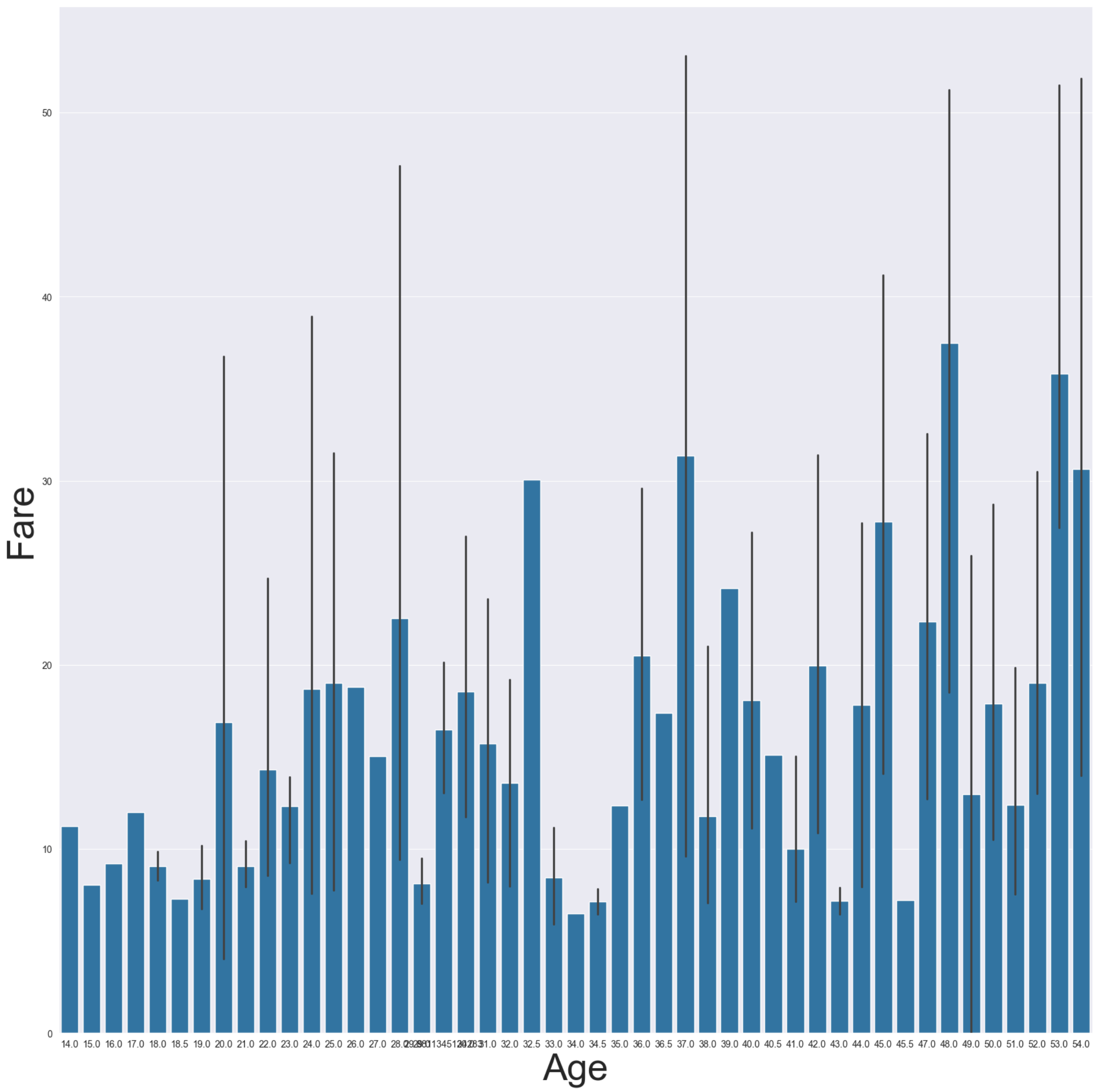
In this plot we deduce that the majority of passengers embarked at Southampton compared to Cherbourg and Queenstown.

[68]:
```
plt.figure(figsize=(8, 6))
sns.histplot(new_train, x='embarked',hue='embarked',bins=30,palette={'C':␣
↪'green', 'S': 'gold','Q':'red'})
plt.title(f'Histogram of Embarked')
plt.xlabel('Embarked')
plt.ylabel('Frequency')
plt.show()
```



In this plot, we illustrate the relationship between age groups and the fare paid for the journey. Observing the graph's shape, it shows that the fare tends to increase with age. However, it's notable that the correlation between the two variables is very low.

[69]:
```
grouped = new_train.groupby("fare")["age"].max()

plt.subplots(figsize=(20, 20))
sns.barplot(x = grouped.values, y = grouped.index)
plt.ylabel('Fare', fontsize=40)
plt.xlabel('Age', fontsize=40)
plt.show()
```

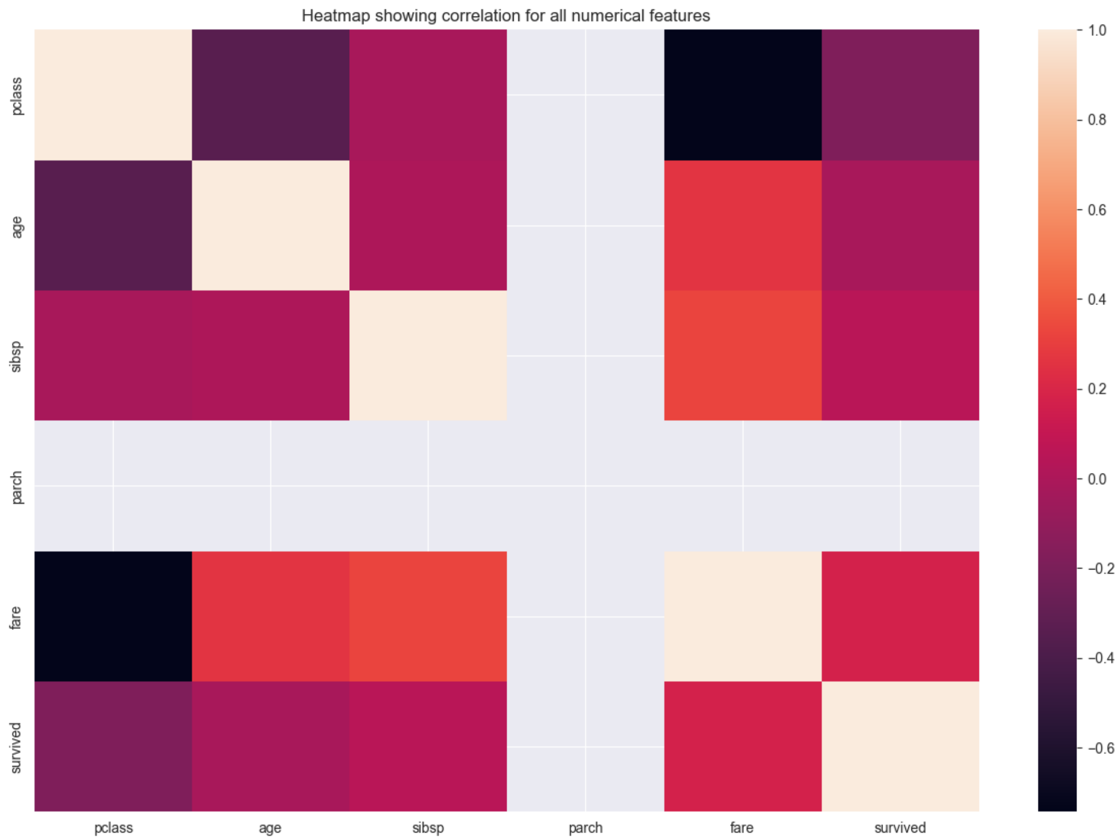


13.1 Correlation between numerical features

In this step, we construct a heatmap for our numerical features to evaluate their correlation with the target variable. On observation, it becomes apparent that the "parch" feature contains only one value, rendering it devoid of correlation with the target. So, we opt to remove this feature from our analysis.

[70]:
```
corr_matrix = new_train[numerical_columns].corr()
plt.figure(figsize = (15,10))
plt.title("Heatmap showing correlation for all numerical features")
sns.heatmap(corr_matrix)
```

[70]: <Axes: title={'center': 'Heatmap showing correlation for all numerical features'}>



14 Selecting relevant columns to the target

Based on the insights gained from our preprocessing steps, we proceed to select the most relevant and useful features for our analysis.

[71]:
```
cdf = ['sex','age','sibsp','fare','survived']
cdf_train = new_train[cdf]
```

[72]:
```
cdf_train
```

[72]:

	sex	age	sibsp	fare	survived
0	male	48.000000	0	26.5550	1
1	male	39.000000	0	0.0000	0
2	female	53.000000	2	51.4792	1
3	male	29.881135	0	25.9250	0
4	male	26.000000	0	30.0000	1
..	…	…	…	…	…

```
827  female  14.500000      1  14.4542         0
828  female  29.881135      1  14.4542         0
829    male  26.500000      0   7.2250         0
830    male  27.000000      0   7.2250         0
831    male  29.000000      0   7.8750         0

[832 rows x 5 columns]
```

14.1 Performing Scaling

Here we use Min-Max Scaler from scikit-learn to normalize the 'age' and 'fare' features of our set. We reshape the 'age' and 'fare' features to a single column format (2D). The scaler is fitted to the data, and then, it transforms both the training and testing datasets. Finally, the transformed features are assigned back to their respective columns in the dataset.

[73]:
```python
from sklearn.preprocessing import MinMaxScaler

# Initializing the scaler and required features
scaler = MinMaxScaler()
age = cdf_train['age'].values.reshape(-1, 1)
fare = cdf_train['fare'].values.reshape(-1, 1)

# Fitting them to the scaler
model_age = scaler.fit(age)
model_fare = scaler.fit(fare)

# Transforming the features in training and testing sets
cdf_train.loc[:, 'age'] = model_age.transform(age)
cdf_train.loc[:, 'fare'] = model_age.transform(fare)
test['age'] = model_age.transform(test['age'].values.reshape(-1, 1))
test['fare'] = model_fare.transform(test['fare'].values.reshape(-1, 1))
```

[74]:
```python
cdf_train
```

[74]:
```
        sex       age  sibsp      fare  survived
0      male  0.903955      0  0.500000         1
1      male  0.734463      0  0.000000         0
2    female  0.998117      2  0.969476         1
3      male  0.562733      0  0.488230         0
4      male  0.489642      0  0.564972         1
..      …         …      …         …         …
827  female  0.273070      1  0.272207         0
828  female  0.562733      1  0.272207         0
829    male  0.499058      0  0.136064         0
830    male  0.508475      0  0.136064         0
831    male  0.546139      0  0.148305         0

[832 rows x 5 columns]
```

15 Perform Encoding to Categorical Feartures

Here we use scikit-learn's LabelEncoder to convert categorical data (specifically the 'sex' feature) into numerical format. We initialize the label encoder, fit it to the 'sex' column in the training dataset ('cdf_train'), and transforms the data accordingly. Similarly, it performs the same transformation on the 'sex' column in the testing dataset ('test').

[75]:
```
from sklearn.preprocessing import LabelEncoder

label_encoder = LabelEncoder()

cdf_train.loc[:, 'sex'] = label_encoder.fit_transform(cdf_train['sex'])

test['sex'] = label_encoder.fit_transform(test['sex'])
```

[76]:
```
cdf_train
```

[76]:
```
     sex       age  sibsp      fare  survived
0      1  0.903955      0  0.500000         1
1      1  0.734463      0  0.000000         0
2      0  0.998117      2  0.969476         1
3      1  0.562733      0  0.488230         0
4      1  0.489642      0  0.564972         1
..    ..       …      …       …       …
827    0  0.273070      1  0.272207         0
828    0  0.562733      1  0.272207         0
829    1  0.499058      0  0.136064         0
830    1  0.508475      0  0.136064         0
831    1  0.546139      0  0.148305         0

[832 rows x 5 columns]
```

15.1 Splitting the cdf data into train and test

[77]:
```
X = cdf_train.drop(columns = 'survived')
y = cdf_train['survived']
train_X, test_X, train_y, test_y = train_test_split(X, y, test_size=0.
↪3,random_state=42)
```

16 Applying Algorithms

17 K-Nearest Neighbors (KNN)

17.1 Initializing KNN classifier with different distance metrics

Here, we build two K-Nearest Neighbors (KNN) classifiers with different distance metrics: Euclidean and Manhattan. After training both models on the training data, they are tested on the test data to generate predictions. The accuracy of each model is then calculated using the accuracy_score function from scikit-learn. Finally, the accuracies of both models are printed out for comparison.

[78]:
```python
knn_euclidean = KNeighborsClassifier(metric = 'euclidean')
knn_manhattan = KNeighborsClassifier(metric = 'manhattan')

# Train models
knn_euclidean.fit(train_X, train_y)
knn_manhattan.fit(train_X, train_y)


# Test models
y_pred_euclidean = knn_euclidean.predict(test_X)
y_pred_manhattan = knn_manhattan.predict(test_X)


# Calculate evaluation metrics for each distance metric
accuracy_euclidean = accuracy_score(test_y, y_pred_euclidean)
accuracy_manhattan = accuracy_score(test_y, y_pred_manhattan)


print("Accuracy (Euclidean):", accuracy_euclidean)
print("Accuracy (Manhattan):", accuracy_manhattan)
```

```
Accuracy (Euclidean): 0.796
Accuracy (Manhattan): 0.808
```

17.2 Selecting the best value for k to apply KNN

In this part we iterate through different values of k (number of neighbors) and construct K-Nearest Neighbors (KNN) classifiers for each value. And then we train these classifiers on the training data and evaluate their performance on the test data. The accuracy of each classifier is printed out, and the highest accuracy achieved among all values of k is also identified and printed.

[79]:
```python
n_neighbors = [1,3,5,7,9]
accuracy_n_neighbors = []

for k in n_neighbors:
 knn = KNeighborsClassifier(n_neighbors = k)
 knn.fit(train_X, train_y)
 y_pred = knn.predict(test_X)
 accuracy = accuracy_score(test_y, y_pred)
 print("accuracy when k = ", k, ' : ', accuracy)
 accuracy_n_neighbors.append(accuracy)

Highest_accuracy = max(accuracy_n_neighbors)
print('The Highest accuracy is ', Highest_accuracy)
```

```
accuracy when k =  1  :  0.744
accuracy when k =  3  :  0.776
accuracy when k =  5  :  0.796
accuracy when k =  7  :  0.796
accuracy when k =  9  :  0.788
The Highest accuracy is  0.796
```

17.3 Apply KNN Algorithm

Now we apple the KNN classifier with (6 neighbors, Manhattan distance metric), train it on the training data, and predict the labels for the test data. Then, we calculate several evaluation metrics including accuracy, F1-score, recall, and precision using scikit-learn functions and prints them out.

[80]:
```
# Initialize Model
KNN_model = KNeighborsClassifier(n_neighbors = 5, metric = 'manhattan')
KNN_model.fit(train_X, train_y)
y_pred_KNN = KNN_model.predict(test_X)

# Compute Accuracy
accuracy_KNN = accuracy_score(y_pred_KNN, test_y)
print('Accuracy = ', accuracy_KNN)

# Compute F1-Score
F1Score_KNN = f1_score(y_pred_KNN, test_y)
print('F1-Score = ', F1Score_KNN)

# Compute Recall
Recall_KNN = recall_score(y_pred_KNN, test_y)
print('Recall = ', Recall_KNN)

# Compute Precision
Precision_KNN = precision_score(y_pred_KNN, test_y)
print('Precision = ', Precision_KNN)
```

```
Accuracy =  0.808
F1-Score =  0.6417910447761194
Recall =  0.7166666666666667
Precision =  0.581081081081081
```

17.4 Loading test data

[81]:
```
test = test[cdf[:-1]]
```

17.5 Predicting using test data

[82]:
```
# Predict using the KNN model
predictions = KNN_model.predict(test)
predictions
```

[82]:
```
array([0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0],
      dtype=int64)
```

18 Naive-Bayes Classifier

18.1 Apply Naive-Bayes Classifier Algorithm

We initialize a Gaussian Naive Bayes (NB) classifier and train it on the training data. Next, we predict labels for the test data and calculate several evaluation metrics including accuracy, recall, F1-score, and precision using scikit-learn functions and print them out.

[83]:
```
# Initialize model
GaussianNB_model = GaussianNB()

#Train Model
GaussianNB_model.fit(train_X, train_y)

# test model
y_predict_NB = GaussianNB_model.predict(test_X)

# Compute Accuracy
accuracy_NB = accuracy_score(y_predict_NB, test_y)
print(f'Accuracy = {accuracy_NB}')

# Compute Recall
Recall_NB = recall_score(y_predict_NB, test_y)
print(f'Recall = {Recall_NB}')

# Compute F1_Score
F1_Score_NB = f1_score(y_predict_NB, test_y)
print(f'F1_Sore = {F1_Score_NB}')

# Compute Precision
Precision_NB = precision_score(y_predict_NB, test_y)
print(f'Precision = {Precision_NB}')
```

```
Accuracy = 0.784
Recall = 0.6351351351351351
F1_Sore = 0.6351351351351351
Precision = 0.6351351351351351
```

18.2 Predicting using test data

[84]:
```
predictions = GaussianNB_model.predict(test)
predictions
```

[84]:
```
array([0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1],
      dtype=int64)
```

19 Support Vector Machine

19.1 Apply Support Vector Classifier (SVC)

We initialize a Support Vector Machine (SVM) model with different kernel functions and regularization parameters. Hyperparameter tuning is conducted using GridSearchCV to find the best combination of parameters. The model is then trained with the best hyperparameters obtained from the grid search. After training, the model is evaluated using the test data, and several evaluation metrics including accuracy, recall, F1-score, and precision are computed. Finally, the computed values for each metric are printed out along with the best hyperparameters found during the grid search.

[85]:
```
# Instantiate the SVM model with different kernel functions and regularization␣
↪parameters
param_grid = {
    'C': [0.1, 1, 10],
      # C --->regularization parameters which controls the trade-off between␣
↪maximizing the margin and minimizing the classification error.
    'gamma': [0.1, 0.01, 0.001],
      # gamma ---> Kernel Parameter
    'kernel': ['linear', 'poly', 'rbf']
}
svm = SVC()

# Hyperparameter tuning using GridSearchCV
grid_search = GridSearchCV(svm, param_grid, cv=5)
grid_search.fit(train_X, train_y)
best_params = grid_search.best_params_

# Train the model with the best hyperparameters
best_svm = SVC(**best_params)
best_svm.fit(train_X, train_y)

# Evaluate the model
y_pred_SVM = best_svm.predict(test_X)
print("Best hyperparameters:", best_params)

# Compute Accuracy
accuracy_SVM = accuracy_score(test_y, y_pred_SVM)
print("Accuracy:", accuracy_SVM )
```

```
# Compute Recall
Recall_SVM=recall_score(y_pred_SVM,test_y)
print(f'Recall = {Recall_SVM}')

# Compute F1_Score
F1_Score_SVM=f1_score(y_pred_SVM,test_y)
print(f'F1_Sore = {F1_Score_SVM}')

# Compute Precision
Precision_SVM=precision_score(y_pred_SVM,test_y)
print(f'Precision = {Precision_SVM}')
```

```
Best hyperparameters: {'C': 0.1, 'gamma': 0.1, 'kernel': 'linear'}
Accuracy: 0.808
Recall = 0.696969696969697
F1_Sore = 0.6571428571428571
Precision = 0.6216216216216216
```

19.2 Predicting using test data

[86]:
```
predictions_SVM = best_svm.predict(test)
predictions_SVM
```

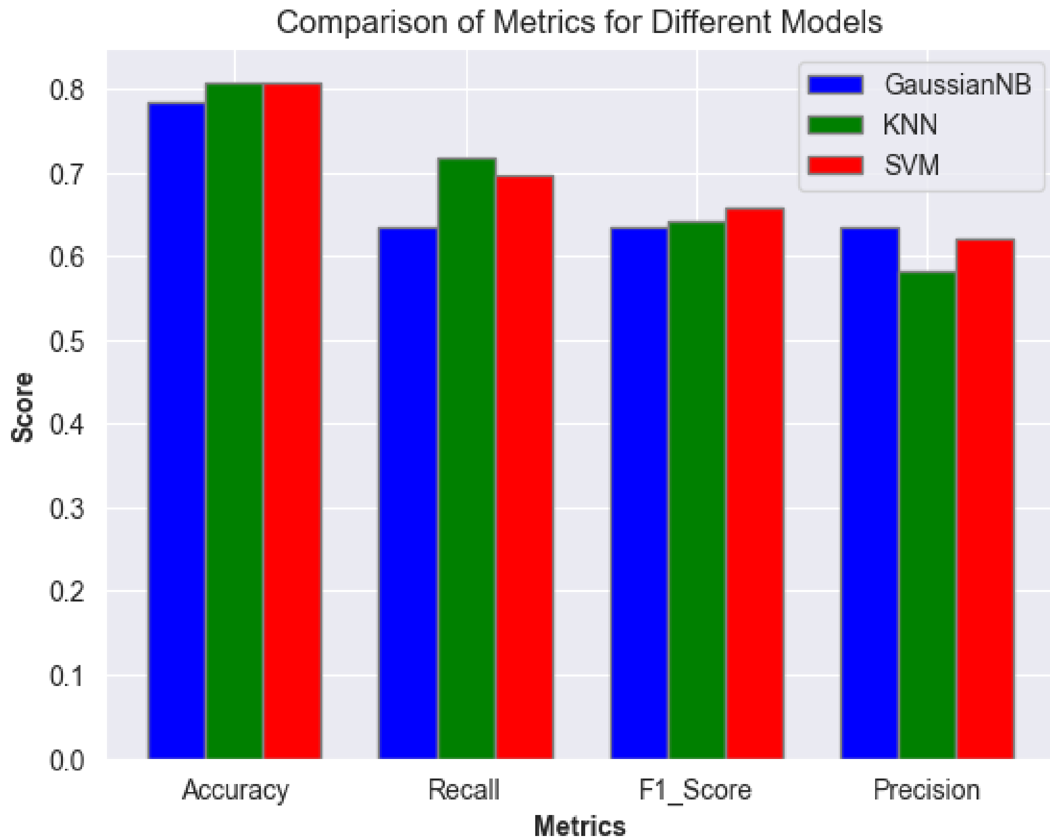
[86]:
```
array([0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1],
      dtype=int64)
```

20 Comparative Analysis :-

We create a grouped bar plot comparing different evaluation metrics (accuracy, recall, F1-score, and precision) for our three models. Each model's metrics are represented by differently colored bars, with labels indicating the corresponding metrics. The plot provides a visual comparison of the performance of the models across these metrics.

[87]:
```
# Metrics for Gaussian Naive Bayes model
metrics_GaussianNB = [accuracy_NB, Recall_NB, F1_Score_NB, Precision_NB]

# Metrics for K-Nearest Neighbors model
metrics_KNN = [accuracy_KNN, Recall_KNN, F1Score_KNN , Precision_KNN]

# Metrics for SVM model
metrics_SVM = [accuracy_SVM, Recall_SVM, F1_Score_SVM, Precision_SVM]

labels = ['Accuracy', 'Recall', 'F1_Score', 'Precision']

# Set the width of the bars
bar_width = 0.25
```

```
# Set the positions of the bars on the x-axis
r1 = np.arange(len(labels))
r2 = [x + bar_width for x in r1]
r3 = [x + bar_width for x in r2]

# Create bars
plt.bar(r1, metrics_GaussianNB, color='b', width=bar_width, edgecolor='grey', label='GaussianNB')
plt.bar(r2, metrics_KNN, color='g', width=bar_width, edgecolor='grey', label='KNN')
plt.bar(r3, metrics_SVM, color='r', width=bar_width, edgecolor='grey', label='SVM')

# Add xticks on the middle of the group bars
plt.xlabel('Metrics', fontweight='bold')
plt.xticks([r + bar_width for r in range(len(labels))], labels)

# Add ylabel
plt.ylabel('Score', fontweight='bold')

# Add title
plt.title('Comparison of Metrics for Different Models')

# Add legend
plt.legend()

# Show plot
plt.show()
```



21 Observations on our analysis :-

21.0.1 While all three models exhibit similar accuracy, SVM shows a slightly higher accuracy compared to the others.

21.0.2 KNN demonstrates the highest recall among the models, indicating its strength in correctly identifying positive instances.

21.0.3 However, KNN performs relatively poorly in precision and F1 score.

21.0.4 Gaussian NB and SVM models show comparable precision, F1 score, and recall, indicating similar overall performance in these metrics.

22 Artificial Neural Networks

22.1 Building the network using TensorFlow

The model consists of an input layer with 4 units corrseponding to our number of feautures, followed by two dense layers with ReLU activation functions, each containing 4 and 2 units respectively. The output layer has 1 unit with a sigmoid activation function, which is commonly used for binary classification tasks. Additionally, a threshold value of 0.5 is specified for the binary classification decision.

```
[88]: model = tf.keras.Sequential([
          tf.keras.layers.Input(shape=(4,)),
          tf.keras.layers.Dense(2, activation='relu'),
          tf.keras.layers.Dense(1, activation='sigmoid'),
      ])

      threshold = 0.5
```

22.2 Compiling the model

We compile the previously defined neural network model using the Adam optimizer, binary crossentropy loss function, and accuracy as the evaluation metric.

```
[89]: model.compile(optimizer='Adam', loss='binary_crossentropy',␣
       ↪metrics=['accuracy'])
```

22.3 Train Model

We train the NN using our training data for 200 epochs with a batch size of 32 and a validation split of 15% meaning that 15% of the training data will be used for validation of the model. This is stored in the history variable which prints the epochs.

```
[90]: history = model.fit(train_X, train_y, epochs = 200, batch_size = 32,␣
       ↪validation_split = 0.15)
```

```
Epoch 1/200
16/16           1s 15ms/step -
accuracy: 0.6797 - loss: 0.7120 - val_accuracy: 0.7273 - val_loss: 0.6911
Epoch 2/200
16/16           0s 4ms/step -
accuracy: 0.6980 - loss: 0.7029 - val_accuracy: 0.7614 - val_loss: 0.6851
Epoch 3/200
16/16           0s 4ms/step -
accuracy: 0.6706 - loss: 0.7096 - val_accuracy: 0.7841 - val_loss: 0.6797
Epoch 4/200
16/16           0s 4ms/step -
accuracy: 0.7050 - loss: 0.6974 - val_accuracy: 0.7841 - val_loss: 0.6747
Epoch 5/200
16/16           0s 4ms/step -
accuracy: 0.7223 - loss: 0.6871 - val_accuracy: 0.7955 - val_loss: 0.6701
Epoch 6/200
16/16           0s 4ms/step -
accuracy: 0.7112 - loss: 0.6860 - val_accuracy: 0.7955 - val_loss: 0.6655
Epoch 7/200
16/16           0s 4ms/step -
accuracy: 0.7225 - loss: 0.6835 - val_accuracy: 0.7955 - val_loss: 0.6612
Epoch 8/200
16/16           0s 4ms/step -
accuracy: 0.7291 - loss: 0.6728 - val_accuracy: 0.7955 - val_loss: 0.6572
```

```
Epoch 9/200
16/16            0s 4ms/step -
accuracy: 0.6829 - loss: 0.6817 - val_accuracy: 0.7955 - val_loss: 0.6532
Epoch 10/200
16/16            0s 4ms/step -
accuracy: 0.7055 - loss: 0.6697 - val_accuracy: 0.7841 - val_loss: 0.6497
Epoch 11/200
16/16            0s 4ms/step -
accuracy: 0.7539 - loss: 0.6588 - val_accuracy: 0.7841 - val_loss: 0.6460
Epoch 12/200
16/16            0s 4ms/step -
accuracy: 0.7011 - loss: 0.6615 - val_accuracy: 0.7841 - val_loss: 0.6425
Epoch 13/200
16/16            0s 4ms/step -
accuracy: 0.7444 - loss: 0.6563 - val_accuracy: 0.8295 - val_loss: 0.6391
Epoch 14/200
16/16            0s 4ms/step -
accuracy: 0.7563 - loss: 0.6532 - val_accuracy: 0.8409 - val_loss: 0.6357
Epoch 15/200
16/16            0s 4ms/step -
accuracy: 0.7507 - loss: 0.6533 - val_accuracy: 0.8409 - val_loss: 0.6324
Epoch 16/200
16/16            0s 4ms/step -
accuracy: 0.7796 - loss: 0.6442 - val_accuracy: 0.8068 - val_loss: 0.6293
Epoch 17/200
16/16            0s 4ms/step -
accuracy: 0.7711 - loss: 0.6414 - val_accuracy: 0.7955 - val_loss: 0.6263
Epoch 18/200
16/16            0s 4ms/step -
accuracy: 0.7495 - loss: 0.6428 - val_accuracy: 0.7955 - val_loss: 0.6233
Epoch 19/200
16/16            0s 4ms/step -
accuracy: 0.7603 - loss: 0.6367 - val_accuracy: 0.7955 - val_loss: 0.6203
Epoch 20/200
16/16            0s 4ms/step -
accuracy: 0.7239 - loss: 0.6391 - val_accuracy: 0.7841 - val_loss: 0.6174
Epoch 21/200
16/16            0s 5ms/step -
accuracy: 0.7502 - loss: 0.6317 - val_accuracy: 0.7841 - val_loss: 0.6144
Epoch 22/200
16/16            0s 4ms/step -
accuracy: 0.7125 - loss: 0.6361 - val_accuracy: 0.7841 - val_loss: 0.6117
Epoch 23/200
16/16            0s 4ms/step -
accuracy: 0.7375 - loss: 0.6274 - val_accuracy: 0.7955 - val_loss: 0.6088
Epoch 24/200
16/16            0s 4ms/step -
accuracy: 0.7290 - loss: 0.6300 - val_accuracy: 0.7955 - val_loss: 0.6060
```

```
Epoch 25/200
16/16            0s 4ms/step -
accuracy: 0.7334 - loss: 0.6249 - val_accuracy: 0.7955 - val_loss: 0.6032
Epoch 26/200
16/16            0s 4ms/step -
accuracy: 0.7493 - loss: 0.6194 - val_accuracy: 0.7955 - val_loss: 0.6004
Epoch 27/200
16/16            0s 5ms/step -
accuracy: 0.7403 - loss: 0.6222 - val_accuracy: 0.8068 - val_loss: 0.5977
Epoch 28/200
16/16            0s 4ms/step -
accuracy: 0.7593 - loss: 0.6116 - val_accuracy: 0.8068 - val_loss: 0.5948
Epoch 29/200
16/16            0s 4ms/step -
accuracy: 0.7539 - loss: 0.6133 - val_accuracy: 0.8068 - val_loss: 0.5924
Epoch 30/200
16/16            0s 3ms/step -
accuracy: 0.7543 - loss: 0.6127 - val_accuracy: 0.8068 - val_loss: 0.5900
Epoch 31/200
16/16            0s 4ms/step -
accuracy: 0.7281 - loss: 0.6142 - val_accuracy: 0.8068 - val_loss: 0.5874
Epoch 32/200
16/16            0s 4ms/step -
accuracy: 0.7201 - loss: 0.6200 - val_accuracy: 0.8068 - val_loss: 0.5850
Epoch 33/200
16/16            0s 4ms/step -
accuracy: 0.7726 - loss: 0.6023 - val_accuracy: 0.8068 - val_loss: 0.5825
Epoch 34/200
16/16            0s 4ms/step -
accuracy: 0.7523 - loss: 0.6050 - val_accuracy: 0.8068 - val_loss: 0.5801
Epoch 35/200
16/16            0s 3ms/step -
accuracy: 0.7520 - loss: 0.6014 - val_accuracy: 0.8182 - val_loss: 0.5773
Epoch 36/200
16/16            0s 3ms/step -
accuracy: 0.7461 - loss: 0.6057 - val_accuracy: 0.8068 - val_loss: 0.5744
Epoch 37/200
16/16            0s 4ms/step -
accuracy: 0.7580 - loss: 0.5957 - val_accuracy: 0.8068 - val_loss: 0.5716
Epoch 38/200
16/16            0s 3ms/step -
accuracy: 0.7918 - loss: 0.5837 - val_accuracy: 0.8068 - val_loss: 0.5689
Epoch 39/200
16/16            0s 4ms/step -
accuracy: 0.7374 - loss: 0.6026 - val_accuracy: 0.8068 - val_loss: 0.5665
Epoch 40/200
16/16            0s 4ms/step -
accuracy: 0.7778 - loss: 0.5841 - val_accuracy: 0.8068 - val_loss: 0.5639
```

```
Epoch 41/200
16/16             0s 4ms/step -
accuracy: 0.7696 - loss: 0.5871 - val_accuracy: 0.8068 - val_loss: 0.5615
Epoch 42/200
16/16             0s 4ms/step -
accuracy: 0.7850 - loss: 0.5797 - val_accuracy: 0.7955 - val_loss: 0.5590
Epoch 43/200
16/16             0s 3ms/step -
accuracy: 0.7800 - loss: 0.5769 - val_accuracy: 0.7955 - val_loss: 0.5566
Epoch 44/200
16/16             0s 4ms/step -
accuracy: 0.7481 - loss: 0.5897 - val_accuracy: 0.7955 - val_loss: 0.5544
Epoch 45/200
16/16             0s 3ms/step -
accuracy: 0.7881 - loss: 0.5697 - val_accuracy: 0.7955 - val_loss: 0.5519
Epoch 46/200
16/16             0s 3ms/step -
accuracy: 0.7528 - loss: 0.5848 - val_accuracy: 0.8068 - val_loss: 0.5495
Epoch 47/200
16/16             0s 3ms/step -
accuracy: 0.7438 - loss: 0.5841 - val_accuracy: 0.8068 - val_loss: 0.5473
Epoch 48/200
16/16             0s 4ms/step -
accuracy: 0.7763 - loss: 0.5711 - val_accuracy: 0.8068 - val_loss: 0.5449
Epoch 49/200
16/16             0s 4ms/step -
accuracy: 0.8024 - loss: 0.5585 - val_accuracy: 0.8068 - val_loss: 0.5425
Epoch 50/200
16/16             0s 4ms/step -
accuracy: 0.7650 - loss: 0.5744 - val_accuracy: 0.8068 - val_loss: 0.5401
Epoch 51/200
16/16             0s 4ms/step -
accuracy: 0.7575 - loss: 0.5696 - val_accuracy: 0.8068 - val_loss: 0.5378
Epoch 52/200
16/16             0s 3ms/step -
accuracy: 0.7540 - loss: 0.5753 - val_accuracy: 0.8068 - val_loss: 0.5357
Epoch 53/200
16/16             0s 3ms/step -
accuracy: 0.7730 - loss: 0.5636 - val_accuracy: 0.8068 - val_loss: 0.5331
Epoch 54/200
16/16             0s 3ms/step -
accuracy: 0.7670 - loss: 0.5648 - val_accuracy: 0.8068 - val_loss: 0.5307
Epoch 55/200
16/16             0s 3ms/step -
accuracy: 0.7716 - loss: 0.5603 - val_accuracy: 0.8182 - val_loss: 0.5282
Epoch 56/200
16/16             0s 4ms/step -
accuracy: 0.7728 - loss: 0.5587 - val_accuracy: 0.8295 - val_loss: 0.5258
```

```
Epoch 57/200
16/16             0s 4ms/step -
accuracy: 0.7551 - loss: 0.5679 - val_accuracy: 0.8409 - val_loss: 0.5234
Epoch 58/200
16/16             0s 5ms/step -
accuracy: 0.7663 - loss: 0.5636 - val_accuracy: 0.8409 - val_loss: 0.5213
Epoch 59/200
16/16             0s 5ms/step -
accuracy: 0.7646 - loss: 0.5583 - val_accuracy: 0.8409 - val_loss: 0.5190
Epoch 60/200
16/16             0s 3ms/step -
accuracy: 0.7791 - loss: 0.5555 - val_accuracy: 0.8409 - val_loss: 0.5168
Epoch 61/200
16/16             0s 3ms/step -
accuracy: 0.7816 - loss: 0.5559 - val_accuracy: 0.8295 - val_loss: 0.5146
Epoch 62/200
16/16             0s 3ms/step -
accuracy: 0.7908 - loss: 0.5517 - val_accuracy: 0.8295 - val_loss: 0.5125
Epoch 63/200
16/16             0s 3ms/step -
accuracy: 0.7872 - loss: 0.5554 - val_accuracy: 0.8295 - val_loss: 0.5104
Epoch 64/200
16/16             0s 3ms/step -
accuracy: 0.7753 - loss: 0.5535 - val_accuracy: 0.8295 - val_loss: 0.5085
Epoch 65/200
16/16             0s 4ms/step -
accuracy: 0.7762 - loss: 0.5570 - val_accuracy: 0.8295 - val_loss: 0.5064
Epoch 66/200
16/16             0s 4ms/step -
accuracy: 0.7620 - loss: 0.5602 - val_accuracy: 0.8409 - val_loss: 0.5046
Epoch 67/200
16/16             0s 4ms/step -
accuracy: 0.7773 - loss: 0.5473 - val_accuracy: 0.8409 - val_loss: 0.5028
Epoch 68/200
16/16             0s 4ms/step -
accuracy: 0.7818 - loss: 0.5450 - val_accuracy: 0.8409 - val_loss: 0.5008
Epoch 69/200
16/16             0s 3ms/step -
accuracy: 0.8065 - loss: 0.5315 - val_accuracy: 0.8409 - val_loss: 0.4989
Epoch 70/200
16/16             0s 3ms/step -
accuracy: 0.7892 - loss: 0.5443 - val_accuracy: 0.8409 - val_loss: 0.4973
Epoch 71/200
16/16             0s 3ms/step -
accuracy: 0.7689 - loss: 0.5549 - val_accuracy: 0.8409 - val_loss: 0.4955
Epoch 72/200
16/16             0s 3ms/step -
accuracy: 0.7828 - loss: 0.5484 - val_accuracy: 0.8409 - val_loss: 0.4939
```

```
Epoch 73/200
16/16            0s 3ms/step -
accuracy: 0.7762 - loss: 0.5496 - val_accuracy: 0.8409 - val_loss: 0.4923
Epoch 74/200
16/16            0s 3ms/step -
accuracy: 0.7998 - loss: 0.5309 - val_accuracy: 0.8409 - val_loss: 0.4906
Epoch 75/200
16/16            0s 3ms/step -
accuracy: 0.7906 - loss: 0.5395 - val_accuracy: 0.8409 - val_loss: 0.4892
Epoch 76/200
16/16            0s 4ms/step -
accuracy: 0.7860 - loss: 0.5335 - val_accuracy: 0.8409 - val_loss: 0.4875
Epoch 77/200
16/16            0s 5ms/step -
accuracy: 0.7927 - loss: 0.5309 - val_accuracy: 0.8409 - val_loss: 0.4859
Epoch 78/200
16/16            0s 3ms/step -
accuracy: 0.7880 - loss: 0.5379 - val_accuracy: 0.8409 - val_loss: 0.4845
Epoch 79/200
16/16            0s 3ms/step -
accuracy: 0.7903 - loss: 0.5339 - val_accuracy: 0.8409 - val_loss: 0.4832
Epoch 80/200
16/16            0s 3ms/step -
accuracy: 0.7745 - loss: 0.5493 - val_accuracy: 0.8409 - val_loss: 0.4817
Epoch 81/200
16/16            0s 3ms/step -
accuracy: 0.7816 - loss: 0.5295 - val_accuracy: 0.8409 - val_loss: 0.4803
Epoch 82/200
16/16            0s 3ms/step -
accuracy: 0.7879 - loss: 0.5270 - val_accuracy: 0.8409 - val_loss: 0.4788
Epoch 83/200
16/16            0s 3ms/step -
accuracy: 0.7873 - loss: 0.5318 - val_accuracy: 0.8409 - val_loss: 0.4775
Epoch 84/200
16/16            0s 3ms/step -
accuracy: 0.7740 - loss: 0.5404 - val_accuracy: 0.8409 - val_loss: 0.4761
Epoch 85/200
16/16            0s 3ms/step -
accuracy: 0.7539 - loss: 0.5523 - val_accuracy: 0.8409 - val_loss: 0.4748
Epoch 86/200
16/16            0s 4ms/step -
accuracy: 0.7909 - loss: 0.5230 - val_accuracy: 0.8409 - val_loss: 0.4735
Epoch 87/200
16/16            0s 4ms/step -
accuracy: 0.7485 - loss: 0.5583 - val_accuracy: 0.8409 - val_loss: 0.4724
Epoch 88/200
16/16            0s 3ms/step -
accuracy: 0.7898 - loss: 0.5270 - val_accuracy: 0.8409 - val_loss: 0.4711
```

```
Epoch 89/200
16/16          0s 3ms/step -
accuracy: 0.7733 - loss: 0.5326 - val_accuracy: 0.8409 - val_loss: 0.4699
Epoch 90/200
16/16          0s 3ms/step -
accuracy: 0.7946 - loss: 0.5173 - val_accuracy: 0.8409 - val_loss: 0.4688
Epoch 91/200
16/16          0s 3ms/step -
accuracy: 0.7977 - loss: 0.5163 - val_accuracy: 0.8409 - val_loss: 0.4676
Epoch 92/200
16/16          0s 3ms/step -
accuracy: 0.7800 - loss: 0.5382 - val_accuracy: 0.8409 - val_loss: 0.4665
Epoch 93/200
16/16          0s 3ms/step -
accuracy: 0.7969 - loss: 0.5098 - val_accuracy: 0.8409 - val_loss: 0.4655
Epoch 94/200
16/16          0s 3ms/step -
accuracy: 0.7968 - loss: 0.5137 - val_accuracy: 0.8409 - val_loss: 0.4644
Epoch 95/200
16/16          0s 4ms/step -
accuracy: 0.7747 - loss: 0.5355 - val_accuracy: 0.8409 - val_loss: 0.4632
Epoch 96/200
16/16          0s 3ms/step -
accuracy: 0.7786 - loss: 0.5248 - val_accuracy: 0.8409 - val_loss: 0.4621
Epoch 97/200
16/16          0s 4ms/step -
accuracy: 0.7868 - loss: 0.5252 - val_accuracy: 0.8409 - val_loss: 0.4611
Epoch 98/200
16/16          0s 3ms/step -
accuracy: 0.7803 - loss: 0.5176 - val_accuracy: 0.8409 - val_loss: 0.4601
Epoch 99/200
16/16          0s 3ms/step -
accuracy: 0.7850 - loss: 0.5202 - val_accuracy: 0.8409 - val_loss: 0.4591
Epoch 100/200
16/16          0s 3ms/step -
accuracy: 0.8002 - loss: 0.5065 - val_accuracy: 0.8409 - val_loss: 0.4581
Epoch 101/200
16/16          0s 4ms/step -
accuracy: 0.7853 - loss: 0.5144 - val_accuracy: 0.8409 - val_loss: 0.4573
Epoch 102/200
16/16          0s 4ms/step -
accuracy: 0.7476 - loss: 0.5224 - val_accuracy: 0.8409 - val_loss: 0.4565
Epoch 103/200
16/16          0s 4ms/step -
accuracy: 0.7472 - loss: 0.5458 - val_accuracy: 0.8409 - val_loss: 0.4554
Epoch 104/200
16/16          0s 4ms/step -
accuracy: 0.7858 - loss: 0.5107 - val_accuracy: 0.8409 - val_loss: 0.4546
```

```
Epoch 105/200
16/16              0s 4ms/step -
accuracy: 0.7786 - loss: 0.5234 - val_accuracy: 0.8409 - val_loss: 0.4539
Epoch 106/200
16/16              0s 3ms/step -
accuracy: 0.7536 - loss: 0.5423 - val_accuracy: 0.8409 - val_loss: 0.4530
Epoch 107/200
16/16              0s 4ms/step -
accuracy: 0.7732 - loss: 0.5139 - val_accuracy: 0.8409 - val_loss: 0.4521
Epoch 108/200
16/16              0s 4ms/step -
accuracy: 0.7758 - loss: 0.5330 - val_accuracy: 0.8409 - val_loss: 0.4513
Epoch 109/200
16/16              0s 3ms/step -
accuracy: 0.7630 - loss: 0.5310 - val_accuracy: 0.8409 - val_loss: 0.4505
Epoch 110/200
16/16              0s 3ms/step -
accuracy: 0.7860 - loss: 0.5129 - val_accuracy: 0.8409 - val_loss: 0.4496
Epoch 111/200
16/16              0s 3ms/step -
accuracy: 0.7978 - loss: 0.4949 - val_accuracy: 0.8409 - val_loss: 0.4490
Epoch 112/200
16/16              0s 3ms/step -
accuracy: 0.7882 - loss: 0.5014 - val_accuracy: 0.8409 - val_loss: 0.4483
Epoch 113/200
16/16              0s 3ms/step -
accuracy: 0.7609 - loss: 0.5248 - val_accuracy: 0.8409 - val_loss: 0.4476
Epoch 114/200
16/16              0s 3ms/step -
accuracy: 0.7785 - loss: 0.5172 - val_accuracy: 0.8409 - val_loss: 0.4468
Epoch 115/200
16/16              0s 6ms/step -
accuracy: 0.7787 - loss: 0.5128 - val_accuracy: 0.8409 - val_loss: 0.4460
Epoch 116/200
16/16              0s 4ms/step -
accuracy: 0.7775 - loss: 0.5101 - val_accuracy: 0.8409 - val_loss: 0.4455
Epoch 117/200
16/16              0s 3ms/step -
accuracy: 0.7928 - loss: 0.4963 - val_accuracy: 0.8409 - val_loss: 0.4445
Epoch 118/200
16/16              0s 3ms/step -
accuracy: 0.7881 - loss: 0.4971 - val_accuracy: 0.8409 - val_loss: 0.4440
Epoch 119/200
16/16              0s 3ms/step -
accuracy: 0.7790 - loss: 0.5064 - val_accuracy: 0.8409 - val_loss: 0.4433
Epoch 120/200
16/16              0s 3ms/step -
accuracy: 0.7646 - loss: 0.5363 - val_accuracy: 0.8409 - val_loss: 0.4427
```

```
Epoch 121/200
16/16            0s 4ms/step -
accuracy: 0.7806 - loss: 0.5017 - val_accuracy: 0.8409 - val_loss: 0.4420
Epoch 122/200
16/16            0s 3ms/step -
accuracy: 0.7979 - loss: 0.4990 - val_accuracy: 0.8409 - val_loss: 0.4416
Epoch 123/200
16/16            0s 3ms/step -
accuracy: 0.7864 - loss: 0.4940 - val_accuracy: 0.8409 - val_loss: 0.4409
Epoch 124/200
16/16            0s 3ms/step -
accuracy: 0.7863 - loss: 0.5127 - val_accuracy: 0.8409 - val_loss: 0.4403
Epoch 125/200
16/16            0s 3ms/step -
accuracy: 0.7854 - loss: 0.5037 - val_accuracy: 0.8409 - val_loss: 0.4396
Epoch 126/200
16/16            0s 3ms/step -
accuracy: 0.7550 - loss: 0.5221 - val_accuracy: 0.8409 - val_loss: 0.4391
Epoch 127/200
16/16            0s 3ms/step -
accuracy: 0.7864 - loss: 0.5022 - val_accuracy: 0.8409 - val_loss: 0.4384
Epoch 128/200
16/16            0s 3ms/step -
accuracy: 0.7754 - loss: 0.5138 - val_accuracy: 0.8409 - val_loss: 0.4378
Epoch 129/200
16/16            0s 3ms/step -
accuracy: 0.7752 - loss: 0.5159 - val_accuracy: 0.8409 - val_loss: 0.4373
Epoch 130/200
16/16            0s 3ms/step -
accuracy: 0.7689 - loss: 0.5207 - val_accuracy: 0.8409 - val_loss: 0.4369
Epoch 131/200
16/16            0s 3ms/step -
accuracy: 0.7963 - loss: 0.4908 - val_accuracy: 0.8409 - val_loss: 0.4361
Epoch 132/200
16/16            0s 3ms/step -
accuracy: 0.7863 - loss: 0.5047 - val_accuracy: 0.8409 - val_loss: 0.4357
Epoch 133/200
16/16            0s 3ms/step -
accuracy: 0.7722 - loss: 0.5176 - val_accuracy: 0.8409 - val_loss: 0.4351
Epoch 134/200
16/16            0s 4ms/step -
accuracy: 0.7945 - loss: 0.4912 - val_accuracy: 0.8409 - val_loss: 0.4345
Epoch 135/200
16/16            0s 3ms/step -
accuracy: 0.8001 - loss: 0.4865 - val_accuracy: 0.8409 - val_loss: 0.4341
Epoch 136/200
16/16            0s 3ms/step -
accuracy: 0.7735 - loss: 0.5145 - val_accuracy: 0.8409 - val_loss: 0.4338
```

```
Epoch 137/200
16/16            0s 3ms/step -
accuracy: 0.7918 - loss: 0.5035 - val_accuracy: 0.8409 - val_loss: 0.4332
Epoch 138/200
16/16            0s 3ms/step -
accuracy: 0.7909 - loss: 0.4909 - val_accuracy: 0.8409 - val_loss: 0.4328
Epoch 139/200
16/16            0s 3ms/step -
accuracy: 0.8115 - loss: 0.4646 - val_accuracy: 0.8409 - val_loss: 0.4322
Epoch 140/200
16/16            0s 3ms/step -
accuracy: 0.7848 - loss: 0.4949 - val_accuracy: 0.8409 - val_loss: 0.4319
Epoch 141/200
16/16            0s 3ms/step -
accuracy: 0.7968 - loss: 0.4873 - val_accuracy: 0.8409 - val_loss: 0.4315
Epoch 142/200
16/16            0s 3ms/step -
accuracy: 0.7967 - loss: 0.4827 - val_accuracy: 0.8409 - val_loss: 0.4310
Epoch 143/200
16/16            0s 3ms/step -
accuracy: 0.7469 - loss: 0.5241 - val_accuracy: 0.8409 - val_loss: 0.4307
Epoch 144/200
16/16            0s 3ms/step -
accuracy: 0.7925 - loss: 0.5033 - val_accuracy: 0.8409 - val_loss: 0.4302
Epoch 145/200
16/16            0s 3ms/step -
accuracy: 0.7531 - loss: 0.5285 - val_accuracy: 0.8409 - val_loss: 0.4299
Epoch 146/200
16/16            0s 6ms/step -
accuracy: 0.7951 - loss: 0.4883 - val_accuracy: 0.8409 - val_loss: 0.4294
Epoch 147/200
16/16            0s 3ms/step -
accuracy: 0.7985 - loss: 0.4819 - val_accuracy: 0.8409 - val_loss: 0.4291
Epoch 148/200
16/16            0s 3ms/step -
accuracy: 0.7464 - loss: 0.5349 - val_accuracy: 0.8409 - val_loss: 0.4288
Epoch 149/200
16/16            0s 3ms/step -
accuracy: 0.7770 - loss: 0.4983 - val_accuracy: 0.8409 - val_loss: 0.4281
Epoch 150/200
16/16            0s 3ms/step -
accuracy: 0.7929 - loss: 0.4859 - val_accuracy: 0.8409 - val_loss: 0.4277
Epoch 151/200
16/16            0s 3ms/step -
accuracy: 0.7757 - loss: 0.5099 - val_accuracy: 0.8409 - val_loss: 0.4274
Epoch 152/200
16/16            0s 3ms/step -
accuracy: 0.7781 - loss: 0.5112 - val_accuracy: 0.8409 - val_loss: 0.4270
```

```
Epoch 153/200
16/16            0s 3ms/step -
accuracy: 0.7719 - loss: 0.4987 - val_accuracy: 0.8409 - val_loss: 0.4267
Epoch 154/200
16/16            0s 3ms/step -
accuracy: 0.7945 - loss: 0.4869 - val_accuracy: 0.8409 - val_loss: 0.4263
Epoch 155/200
16/16            0s 3ms/step -
accuracy: 0.7594 - loss: 0.5226 - val_accuracy: 0.8409 - val_loss: 0.4261
Epoch 156/200
16/16            0s 3ms/step -
accuracy: 0.7997 - loss: 0.4789 - val_accuracy: 0.8409 - val_loss: 0.4257
Epoch 157/200
16/16            0s 3ms/step -
accuracy: 0.7752 - loss: 0.5075 - val_accuracy: 0.8409 - val_loss: 0.4254
Epoch 158/200
16/16            0s 3ms/step -
accuracy: 0.7649 - loss: 0.5219 - val_accuracy: 0.8409 - val_loss: 0.4252
Epoch 159/200
16/16            0s 4ms/step -
accuracy: 0.7769 - loss: 0.4951 - val_accuracy: 0.8409 - val_loss: 0.4248
Epoch 160/200
16/16            0s 3ms/step -
accuracy: 0.7743 - loss: 0.5047 - val_accuracy: 0.8409 - val_loss: 0.4245
Epoch 161/200
16/16            0s 4ms/step -
accuracy: 0.7896 - loss: 0.4915 - val_accuracy: 0.8409 - val_loss: 0.4240
Epoch 162/200
16/16            0s 4ms/step -
accuracy: 0.7770 - loss: 0.4875 - val_accuracy: 0.8409 - val_loss: 0.4239
Epoch 163/200
16/16            0s 3ms/step -
accuracy: 0.7427 - loss: 0.5413 - val_accuracy: 0.8409 - val_loss: 0.4236
Epoch 164/200
16/16            0s 3ms/step -
accuracy: 0.7827 - loss: 0.4918 - val_accuracy: 0.8409 - val_loss: 0.4233
Epoch 165/200
16/16            0s 4ms/step -
accuracy: 0.7589 - loss: 0.5224 - val_accuracy: 0.8409 - val_loss: 0.4231
Epoch 166/200
16/16            0s 4ms/step -
accuracy: 0.7776 - loss: 0.4947 - val_accuracy: 0.8409 - val_loss: 0.4227
Epoch 167/200
16/16            0s 4ms/step -
accuracy: 0.7584 - loss: 0.5205 - val_accuracy: 0.8409 - val_loss: 0.4224
Epoch 168/200
16/16            0s 3ms/step -
accuracy: 0.7716 - loss: 0.5121 - val_accuracy: 0.8409 - val_loss: 0.4223
```

```
Epoch 169/200
16/16            0s 3ms/step -
accuracy: 0.7807 - loss: 0.5035 - val_accuracy: 0.8409 - val_loss: 0.4221
Epoch 170/200
16/16            0s 3ms/step -
accuracy: 0.7530 - loss: 0.5358 - val_accuracy: 0.8409 - val_loss: 0.4218
Epoch 171/200
16/16            0s 3ms/step -
accuracy: 0.7636 - loss: 0.5141 - val_accuracy: 0.8409 - val_loss: 0.4216
Epoch 172/200
16/16            0s 3ms/step -
accuracy: 0.7988 - loss: 0.4872 - val_accuracy: 0.8409 - val_loss: 0.4213
Epoch 173/200
16/16            0s 4ms/step -
accuracy: 0.7832 - loss: 0.4836 - val_accuracy: 0.8409 - val_loss: 0.4210
Epoch 174/200
16/16            0s 4ms/step -
accuracy: 0.7576 - loss: 0.5252 - val_accuracy: 0.8409 - val_loss: 0.4207
Epoch 175/200
16/16            0s 3ms/step -
accuracy: 0.7938 - loss: 0.4984 - val_accuracy: 0.8409 - val_loss: 0.4203
Epoch 176/200
16/16            0s 6ms/step -
accuracy: 0.7735 - loss: 0.5173 - val_accuracy: 0.8409 - val_loss: 0.4201
Epoch 177/200
16/16            0s 4ms/step -
accuracy: 0.7776 - loss: 0.4973 - val_accuracy: 0.8409 - val_loss: 0.4199
Epoch 178/200
16/16            0s 4ms/step -
accuracy: 0.7808 - loss: 0.5032 - val_accuracy: 0.8409 - val_loss: 0.4196
Epoch 179/200
16/16            0s 3ms/step -
accuracy: 0.7955 - loss: 0.4887 - val_accuracy: 0.8409 - val_loss: 0.4193
Epoch 180/200
16/16            0s 3ms/step -
accuracy: 0.7675 - loss: 0.5123 - val_accuracy: 0.8409 - val_loss: 0.4193
Epoch 181/200
16/16            0s 3ms/step -
accuracy: 0.7747 - loss: 0.5064 - val_accuracy: 0.8409 - val_loss: 0.4191
Epoch 182/200
16/16            0s 3ms/step -
accuracy: 0.7606 - loss: 0.5169 - val_accuracy: 0.8409 - val_loss: 0.4189
Epoch 183/200
16/16            0s 3ms/step -
accuracy: 0.7845 - loss: 0.4848 - val_accuracy: 0.8409 - val_loss: 0.4185
Epoch 184/200
16/16            0s 4ms/step -
accuracy: 0.7915 - loss: 0.4911 - val_accuracy: 0.8409 - val_loss: 0.4184
```

```
Epoch 185/200
16/16            0s 4ms/step -
accuracy: 0.7526 - loss: 0.5318 - val_accuracy: 0.8409 - val_loss: 0.4181
Epoch 186/200
16/16            0s 4ms/step -
accuracy: 0.7628 - loss: 0.5136 - val_accuracy: 0.8409 - val_loss: 0.4178
Epoch 187/200
16/16            0s 4ms/step -
accuracy: 0.7563 - loss: 0.5220 - val_accuracy: 0.8409 - val_loss: 0.4178
Epoch 188/200
16/16            0s 3ms/step -
accuracy: 0.7706 - loss: 0.5138 - val_accuracy: 0.8409 - val_loss: 0.4177
Epoch 189/200
16/16            0s 3ms/step -
accuracy: 0.7919 - loss: 0.4862 - val_accuracy: 0.8409 - val_loss: 0.4175
Epoch 190/200
16/16            0s 3ms/step -
accuracy: 0.7681 - loss: 0.5094 - val_accuracy: 0.8409 - val_loss: 0.4175
Epoch 191/200
16/16            0s 3ms/step -
accuracy: 0.7725 - loss: 0.5087 - val_accuracy: 0.8409 - val_loss: 0.4175
Epoch 192/200
16/16            0s 4ms/step -
accuracy: 0.7941 - loss: 0.4915 - val_accuracy: 0.8409 - val_loss: 0.4174
Epoch 193/200
16/16            0s 5ms/step -
accuracy: 0.7685 - loss: 0.5133 - val_accuracy: 0.8409 - val_loss: 0.4172
Epoch 194/200
16/16            0s 4ms/step -
accuracy: 0.7840 - loss: 0.4972 - val_accuracy: 0.8409 - val_loss: 0.4171
Epoch 195/200
16/16            0s 4ms/step -
accuracy: 0.7831 - loss: 0.4855 - val_accuracy: 0.8409 - val_loss: 0.4170
Epoch 196/200
16/16            0s 3ms/step -
accuracy: 0.7759 - loss: 0.5112 - val_accuracy: 0.8409 - val_loss: 0.4169
Epoch 197/200
16/16            0s 3ms/step -
accuracy: 0.7847 - loss: 0.4939 - val_accuracy: 0.8409 - val_loss: 0.4168
Epoch 198/200
16/16            0s 3ms/step -
accuracy: 0.7569 - loss: 0.5118 - val_accuracy: 0.8409 - val_loss: 0.4168
Epoch 199/200
16/16            0s 3ms/step -
accuracy: 0.7649 - loss: 0.5116 - val_accuracy: 0.8409 - val_loss: 0.4172
Epoch 200/200
16/16            0s 3ms/step -
accuracy: 0.7837 - loss: 0.5003 - val_accuracy: 0.8409 - val_loss: 0.4175
```

22.4 Predicting using test_X

We make predictions using the trained neural network model on the test data (test_X). The predicted probabilities are first compared against the threshold value (0.5) to determine the binary classification outcome. If the predicted probability is greater than the threshold, the corresponding prediction is set to 1; otherwise, it is set to 0.

[91]:
```
predictions = model.predict(test_X)
predictions = np.where(predictions >= threshold, 1, 0)
predictions
```

```
8/8              0s 5ms/step
```

[91]:
```
array([[1],
       [0],
       [1],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [1],
       [1],
       [0],
       [0],
       [0],
       [1],
       [0],
       [0],
       [1],
       [0],
       [0],
       [1],
       [1],
       [0],
       [0],
       [0],
       [0],
       [0],
       [1],
       [1],
       [1],
       [1],
       [0],
```

```
[1],
[0],
[0],
[0],
[0],
[1],
[0],
[1],
[0],
[0],
[1],
[0],
[0],
[0],
[0],
[0],
[0],
[0],
[1],
[0],
[0],
[0],
[1],
[0],
[0],
[1],
[0],
[0],
[0],
[1],
[0],
[0],
[0],
[1],
[1],
[1],
[0],
[1],
[0],
[0],
[0],
[0],
[1],
[0],
[0],
[0],
[1],
```

```
[0],
[0],
[0],
[0],
[0],
[0],
[0],
[1],
[0],
[1],
[0],
[0],
[0],
[0],
[0],
[1],
[0],
[0],
[1],
[0],
[1],
[0],
[0],
[1],
[0],
[0],
[0],
[0],
[0],
[1],
[0],
[0],
[0],
[0],
[0],
[1],
[0],
[0],
[0],
[0],
[0],
[0],
[0],
[0],
[0],
[1],
[1],
```

```
[0],
[0],
[0],
[0],
[1],
[0],
[0],
[0],
[0],
[1],
[0],
[1],
[1],
[1],
[0],
[1],
[0],
[0],
[0],
[0],
[0],
[0],
[0],
[0],
[0],
[0],
[0],
[0],
[0],
[1],
[0],
[0],
[0],
[0],
[0],
[0],
[0],
[0],
[1],
[0],
[0],
[0],
[0],
[0],
[0],
[1],
[1],
```

```
[0],
[0],
[0],
[0],
[0],
[0],
[1],
[0],
[0],
[0],
[0],
[0],
[0],
[1],
[0],
[0],
[1],
[1],
[0],
[1],
[0],
[0],
[0],
[0],
[0],
[1],
[0],
[0],
[0],
[1],
[0],
[0],
[1],
[0],
[0],
[0],
[0],
[1],
[0],
[0],
[0],
[0],
[1],
[1],
[0],
[0],
[0],
```

```
        [0],
        [1],
        [1],
        [0],
        [0],
        [1],
        [0],
        [1],
        [0],
        [0],
        [0],
        [0],
        [0],
        [0],
        [0],
        [0],
        [0],
        [0],
        [0],
        [1],
        [1],
        [0],
        [0],
        [1],
        [0],
        [0],
        [0]])
```

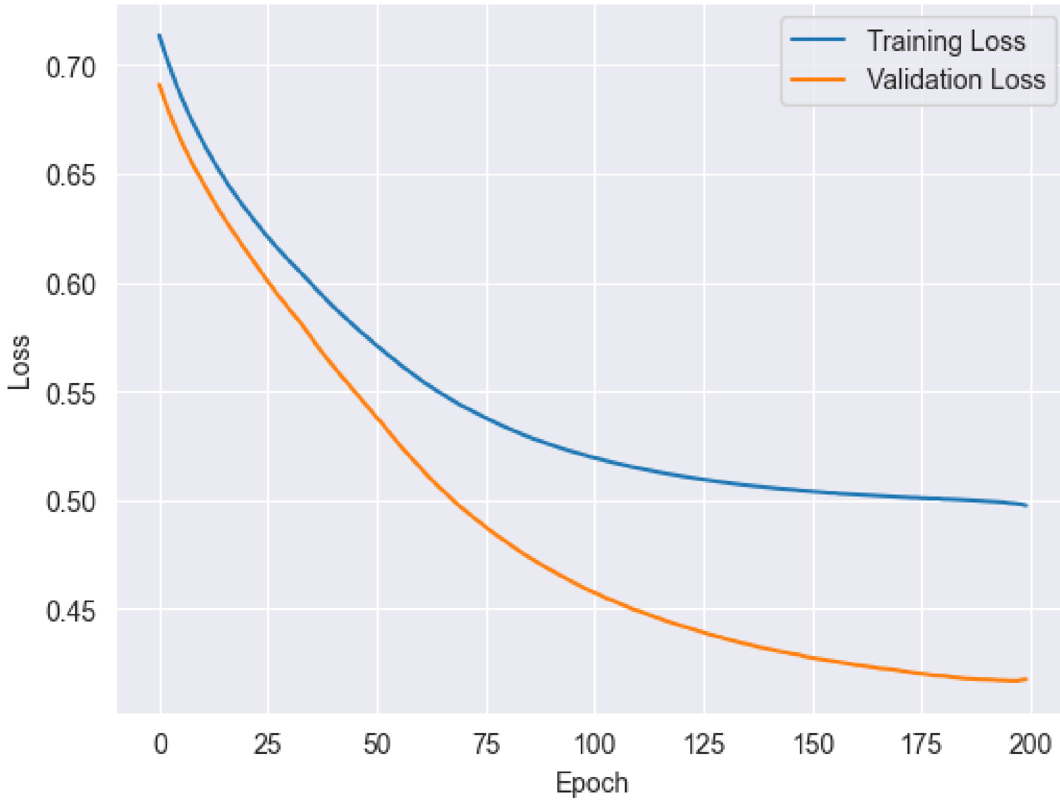
22.5 Evaluating Model

[92]:
```python
evaluation = model.evaluate(test_X, test_y)
```

```
8/8              0s 1ms/step -
accuracy: 0.8045 - loss: 0.4624
```

22.6 Plotting Loss Graph

[93]:
```python
plt.plot(history.history['loss'], label = 'Training Loss')
plt.plot(history.history['val_loss'], label = 'Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



22.7 Predicting using test data

[94]:
```
ANN_prediction = model.predict(test)
ANN_prediction = np.where(ANN_prediction >= threshold, 1, 0)
ANN_prediction
```

```
1/1              0s 28ms/step
```

[94]:
```
array([[0],
       [1],
       [0],
       [0],
       [1],
       [0],
       [1],
       [0],
       [1],
       [0],
       [0],
       [0],
       [1],
```

```
[0],
[1],
[1],
[0],
[0],
[1]])
```