Data Loading and Importing the necessary libraries

```
# Data manipulation and analysis
import pandas as pd
import numpy as np

# Data visualization
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib import style

# Algorithms
from sklearn import linear_model
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import Perceptron
from sklearn.linear_model import SGDClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
```

Loading the data files

Here we import the data. For this analysis, we will be exclusively working with the Training set. We will be validating based on data from the training set as well. For our final submissions, we will make predictions based on the test set.

```
train_df = pd.read_csv('train.csv')
test_df = pd.read_csv('test.csv')

train_df['train_test'] = 1
test_df['train_test'] = 0
# test_df['Survived'] = np.NaN
all_data = pd.concat([train_df,test_df])

%matplotlib inline
all_data.columns
```

```
Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age',
'SibSp',
       'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked', 'train_test'],
      dtype='object')
```

```
train_df.head(10)
```

```
   PassengerId  Survived  Pclass  \
0            1         0       3
1            2         1       1
```

```
2            3          1       3
3            4          1       1
4            5          0       3
5            6          0       3
6            7          0       1
7            8          0       3
8            9          1       3
9           10          1       2

                                                Name     Sex   Age
SibSp  \
0                            Braund, Mr. Owen Harris    male  22.0
1
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0
1
2                             Heikkinen, Miss. Laina  female  26.0
0
3       Futrelle, Mrs. Jacques Heath (Lily May Peel)  female  35.0
1
4                           Allen, Mr. William Henry    male  35.0
0
5                                   Moran, Mr. James    male   NaN
0
6                            McCarthy, Mr. Timothy J    male  54.0
0
7                     Palsson, Master. Gosta Leonard    male   2.0
3
8  Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)  female  27.0
0
9                Nasser, Mrs. Nicholas (Adele Achem)  female  14.0
1

   Parch            Ticket     Fare Cabin Embarked  train_test
0      0         A/5 21171   7.2500   NaN        S           1
1      0          PC 17599  71.2833   C85        C           1
2      0  STON/O2. 3101282   7.9250   NaN        S           1
3      0            113803  53.1000  C123        S           1
4      0            373450   8.0500   NaN        S           1
5      0            330877   8.4583   NaN        Q           1
6      0             17463  51.8625   E46        S           1
7      1            349909  21.0750   NaN        S           1
8      2            347742  11.1333   NaN        S           1
9      0            237736  30.0708   NaN        C           1
```

```
test_df.head(10)
```

```
   PassengerId  Pclass                                          Name
Sex  \
0          892       3                              Kelly, Mr. James
male
```

```
1          893       3              Wilkes, Mrs. James (Ellen Needs)
female
2          894       2                     Myles, Mr. Thomas Francis
male
3          895       3                              Wirz, Mr. Albert
male
4          896       3  Hirvonen, Mrs. Alexander (Helga E Lindqvist)
female
5          897       3                    Svensson, Mr. Johan Cervin
male
6          898       3                          Connolly, Miss. Kate
female
7          899       2                  Caldwell, Mr. Albert Francis
male
8          900       3     Abrahim, Mrs. Joseph (Sophie Halaut Easu)
female
9          901       3                       Davies, Mr. John Samuel
male

    Age  SibSp  Parch      Ticket     Fare Cabin Embarked  train_test
0  34.5      0      0      330911   7.8292   NaN        Q           0
1  47.0      1      0      363272   7.0000   NaN        S           0
2  62.0      0      0      240276   9.6875   NaN        Q           0
3  27.0      0      0      315154   8.6625   NaN        S           0
4  22.0      1      1     3101298  12.2875   NaN        S           0
5  14.0      0      0        7538   9.2250   NaN        S           0
6  30.0      0      0      330972   7.6292   NaN        Q           0
7  26.0      1      1      248738  29.0000   NaN        S           0
8  18.0      0      0        2657   7.2292   NaN        C           0
9  21.0      2      0   A/4 48871  24.1500   NaN        S           0
```

Data understanding using Exploratory Data Analysis (EDA)

Exploratory Data Analysis refers to the critical process of performing initial investigations on data so as to discover patterns, to spot anomalies, to test hypothesis and to check assumptions with the help of summary statistics and graphical representations.

In summary, it's an approach to analyzing data sets to summarize their main characteristics, often with visual methods.

```
train_df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 13 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
```

```
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
 12  train_test   891 non-null    int64
dtypes: float64(2), int64(6), object(5)
memory usage: 90.6+ KB
```

The training-set has 891 rows and 11 features + the **target variable (survived).** 2 of the features are floats, 5 are integers and 5 are objects.

```
train_df.describe()
```

	PassengerId	Survived	Pclass	Age	SibSp	\
count	891.000000	891.000000	891.000000	714.000000	891.000000	
mean	446.000000	0.383838	2.308642	29.699118	0.523008	
std	257.353842	0.486592	0.836071	14.526497	1.102743	
min	1.000000	0.000000	1.000000	0.420000	0.000000	
25%	223.500000	0.000000	2.000000	20.125000	0.000000	
50%	446.000000	0.000000	3.000000	28.000000	0.000000	
75%	668.500000	1.000000	3.000000	38.000000	1.000000	
max	891.000000	1.000000	3.000000	80.000000	8.000000	

	Parch	Fare	train_test
count	891.000000	891.000000	891.0
mean	0.381594	32.204208	1.0
std	0.806057	49.693429	0.0
min	0.000000	0.000000	1.0
25%	0.000000	7.910400	1.0
50%	0.000000	14.454200	1.0
75%	0.000000	31.000000	1.0
max	6.000000	512.329200	1.0

Exploring missing data

```
total = train_df.isnull().sum().sort_values(ascending=False)
percent_1 = train_df.isnull().sum()/train_df.isnull().count()*100
percent_2 = (round(percent_1, 1)).sort_values(ascending=False)
missing_data = pd.concat([total, percent_2], axis=1, keys=['Total',
'%'])
missing_data.head(13)
```

```
             Total     %
Cabin          687  77.1
Age            177  19.9
Embarked         2   0.2
PassengerId      0   0.0
Survived         0   0.0
Pclass           0   0.0
Name             0   0.0
Sex              0   0.0
SibSp            0   0.0
Parch            0   0.0
Ticket           0   0.0
Fare             0   0.0
train_test       0   0.0
```

The **'Embarked'** feature has only 2 missing values, which can easily be filled or dropped. It will be much more tricky to deal with the **'Age'** feature, which has 177 missing values. The **'Cabin'** feature needs further investigation, but it looks like we might want to drop it from the dataset since 77% is missing.

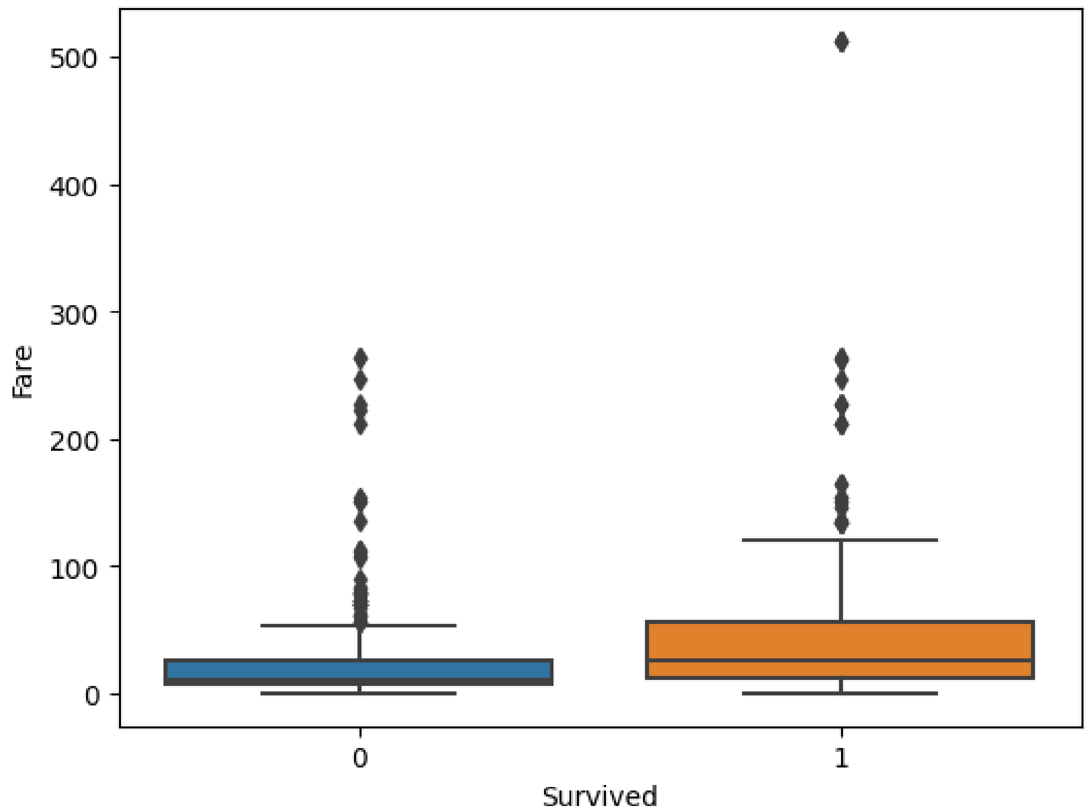
```
train_df.columns.values

array(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age',
'SibSp',
       'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked', 'train_test'],
      dtype=object)
```

Above we can see the 11 features and the target variable (survived). **What features could contribute to a high survival rate ?**

I believe it would make sense if everything except 'PassengerId', 'Name' and 'Ticket' would be high correlated with survival rate.

Dealing with the outlier

```
sns.boxplot(x='Survived',y='Fare',data=train_df);
```



Passengers who paid over 300

```
train_df[train_df['Fare']>300]
```

```
     PassengerId  Survived  Pclass                                Name  \
258          259         1       1                    Ward, Miss. Anna   
679          680         1       1  Cardeza, Mr. Thomas Drake Martinez   
737          738         1       1              Lesurer, Mr. Gustave J   

        Sex   Age  SibSp  Parch    Ticket      Fare        Cabin Embarked  \
258  female  35.0      0      0  PC 17755  512.3292          NaN        C   
679    male  36.0      0      1  PC 17755  512.3292  B51 B53 B55        C   
737    male  35.0      0      0  PC 17755  512.3292         B101        C   

     train_test  
258           1  
```

```
679            1
737            1
```

Drop the outliers

It might be beneficial to drop those outliers for the model. Further investigation needs to be done.

```
# train_df = train_df[train_df['Fare']<300]
```

The Captain went down with the ship

"The captain goes down with the ship" is a maritime tradition that a sea captain holds ultimate responsibility for both his/her ship and everyone embarked on it, and that in an emergency, he/she will either save them or die trying.

In this case, **Captain Edward Gifford Crosby** went down with Titanic in a heroic gesture trying to save the passengers.

```
train_df[train_df['Name'].str.contains("Capt")]
```

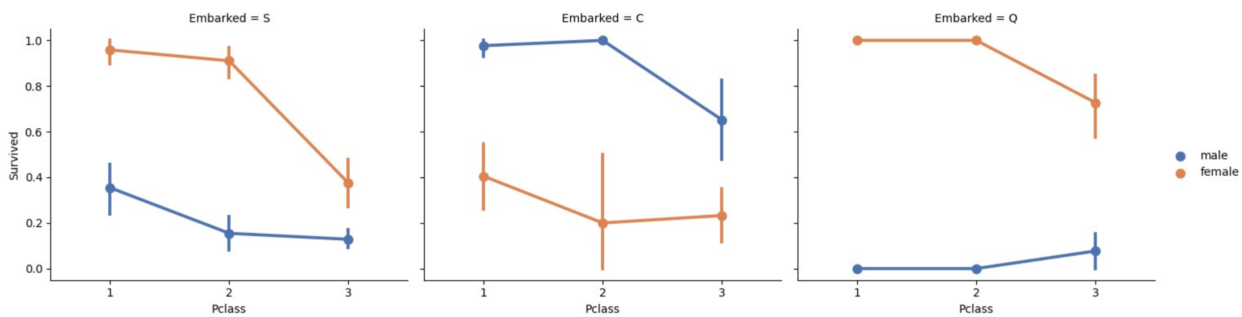
```
     PassengerId  Survived  Pclass                          Name   Sex
Age  \
745          746         0       1  Crosby, Capt. Edward Gifford  male
70.0

     SibSp  Parch     Ticket  Fare Cabin Embarked  train_test
745      1      1  WE/P 5735  71.0   B22        S           1
```

Embarked, Pclass and Sex:

```
FacetGrid = sns.FacetGrid(train_df, col='Embarked', height=4,
aspect=1.2)
FacetGrid.map(sns.pointplot, 'Pclass', 'Survived', 'Sex', ci=95.0,
palette='deep', order=None, hue_order=None)
FacetGrid.add_legend();
```

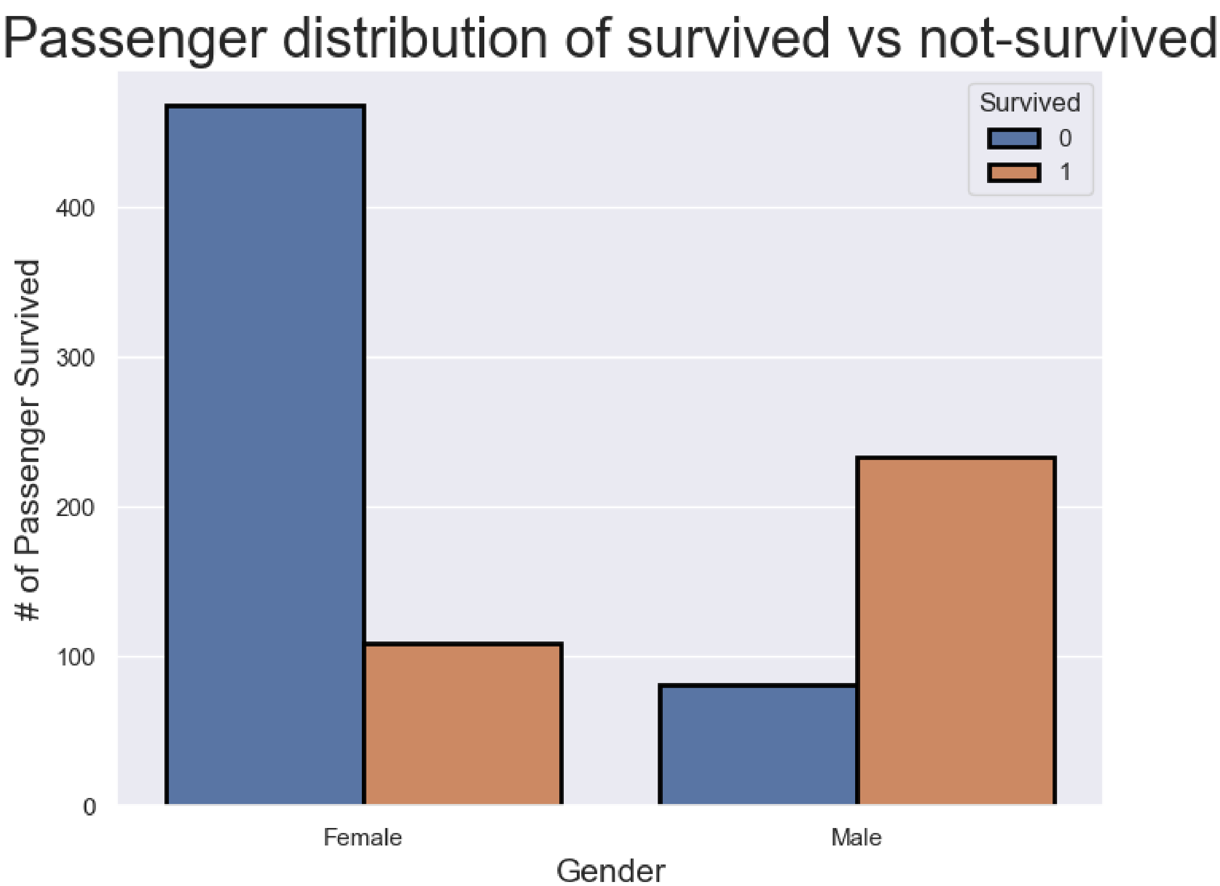
```
c:\Users\varsh\AppData\Local\Programs\Python\Python311\Lib\site-
packages\seaborn\axisgrid.py:848: FutureWarning:

The `ci` parameter is deprecated. Use `errorbar=('ci', 95.0)` for the
same effect.

  func(*plot_args, **plot_kwargs)
c:\Users\varsh\AppData\Local\Programs\Python\Python311\Lib\site-
packages\seaborn\axisgrid.py:848: FutureWarning:

The `ci` parameter is deprecated. Use `errorbar=('ci', 95.0)` for the
same effect.
```

```
  func(*plot_args, **plot_kwargs)
c:\Users\varsh\AppData\Local\Programs\Python\Python311\Lib\site-packages\seaborn\axisgrid.py:848: FutureWarning:

The `ci` parameter is deprecated. Use `errorbar=('ci', 95.0)` for the same effect.

  func(*plot_args, **plot_kwargs)
```



Distribution of Pclass and Survived

```
sns.set(style='darkgrid')
plt.subplots(figsize = (8,6))
ax=sns.countplot(x='Sex', data = train_df, hue='Survived', edgecolor=(0,0,0), linewidth=2)

# Fixing title, xlabel and ylabel
plt.title('Passenger distribution of survived vs not-survived', fontsize=25)
plt.xlabel('Gender', fontsize=15)
plt.ylabel("# of Passenger Survived", fontsize = 15)
labels = ['Female', 'Male']

# Fixing xticks.
plt.xticks(sorted(train_df.Survived.unique()),labels);
```




```
train_df.groupby(['Sex']).mean()
```

```
C:\Users\varsh\AppData\Local\Temp\ipykernel_7644\3313102057.py:1:
FutureWarning: The default value of numeric_only in
DataFrameGroupBy.mean is deprecated. In a future version, numeric_only
will default to False. Either specify numeric_only or select only
columns which should be valid for the function.
  train_df.groupby(['Sex']).mean()
```

```
        PassengerId  Survived    Pclass        Age     SibSp     Parch  \
Sex

female   431.028662  0.742038  2.159236  27.915709  0.694268  0.649682

male     454.147314  0.188908  2.389948  30.726645  0.429809  0.235702


             Fare  train_test
Sex
female  44.479818         1.0
male    25.523893         1.0
```

As previously mentioned, women are much more likely to survive than men. **74% of the women survived, while only 18% of men survived.**

Looking deeper into differences between females and males statistics

```
train_df.groupby(['Sex','Pclass']).mean()
```

```
C:\Users\varsh\AppData\Local\Temp\ipykernel_7644\4204354171.py:1: 
FutureWarning: The default value of numeric_only in 
DataFrameGroupBy.mean is deprecated. In a future version, numeric_only 
will default to False. Either specify numeric_only or select only 
columns which should be valid for the function.
  train_df.groupby(['Sex','Pclass']).mean()
```

```
               PassengerId  Survived        Age     SibSp     Parch  \
Sex    Pclass                                                          
female 1        469.212766  0.968085  34.611765  0.553191  0.457447   
       2        443.105263  0.921053  28.722973  0.486842  0.605263   
       3        399.729167  0.500000  21.750000  0.895833  0.798611   
male   1        455.729508  0.368852  41.281386  0.311475  0.278689   
       2        447.962963  0.157407  30.740707  0.342593  0.222222   
       3        455.515850  0.135447  26.507589  0.498559  0.224784   

                     Fare  train_test  
Sex    Pclass                          
female 1       106.125798         1.0  
       2        21.970121         1.0  
       3        16.118810         1.0  
male   1        67.226127         1.0  
       2        19.741782         1.0  
       3        12.661633         1.0  
```

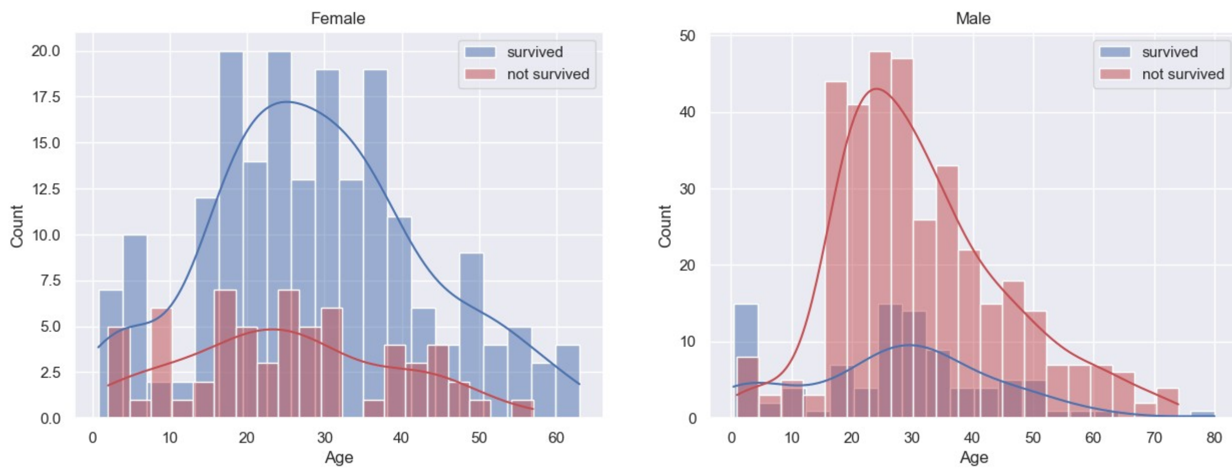
We are grouping passengers based on Sex and Ticket class (Pclass). Notice the difference between survival rates between men and women.

Women are much more likely to survive than men, **specially women in the first and second class.** It also shows that men in the first class are almost **3-times more likely to survive** than men in the third class.

Age and Sex distributions

```
survived = 'survived'
not_survived = 'not survived'

fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(15, 5))

women = train_df[train_df['Sex']=='female']
men = train_df[train_df['Sex']=='male']

# Plot Female Survived vs Not-Survived distribution
ax = sns.histplot(women[women['Survived']==1].Age.dropna(), bins=20,
```

```
label = survived, ax = axes[0],color='b', kde=True)
ax = sns.histplot(women[women['Survived']==0].Age.dropna(), bins=20,
label = not_survived, ax = axes[0],color='r', kde=True)
ax.legend()
ax.set_title('Female')

# Plot Male Survived vs Not-Survived distribution
ax = sns.histplot(men[men['Survived']==1].Age.dropna(), bins=20, label
= survived, ax = axes[1],color='b', kde=True)
ax = sns.histplot(men[men['Survived']==0].Age.dropna(), bins=20, label
= not_survived, ax = axes[1],color='r', kde=True)
ax.legend()
ax.set_title('Male');
```



We can see that **men** have a higher probability of survival when they are between **18 and 35 years old.** For **women,** the survival chances are higher between **15 and 40 years old.**

For men the probability of survival is very low between the **ages of 5 and 18**, and **after 35**, but that isn't true for women. Another thing to note is that **infants have a higher probability of survival.**

Saving children first

```
train_df[train_df['Age']<18].groupby(['Sex','Pclass']).mean()
```

```
C:\Users\varsh\AppData\Local\Temp\ipykernel_7644\1113519119.py:1:
FutureWarning: The default value of numeric_only in
DataFrameGroupBy.mean is deprecated. In a future version, numeric_only
will default to False. Either specify numeric_only or select only
columns which should be valid for the function.
  train_df[train_df['Age']<18].groupby(['Sex','Pclass']).mean()
```

```
                PassengerId  Survived        Age     SibSp     Parch  \
Sex    Pclass
female 1         525.375000  0.875000  14.125000  0.500000  0.875000
       2         369.250000  1.000000   8.333333  0.583333  1.083333
```

```
       3         374.942857  0.542857   8.428571  1.571429  1.057143
male   1         526.500000  1.000000   8.230000  0.500000  2.000000
       2         527.818182  0.818182   4.757273  0.727273  1.000000
       3         437.953488  0.232558   9.963256  2.069767  1.000000

                      Fare  train_test
Sex    Pclass                         
female 1        104.083337         1.0
       2         26.241667         1.0
       3         18.727977         1.0
male   1        116.072900         1.0
       2         25.659473         1.0
       3         22.752523         1.0
```

Children below 18 years of age have higher chances of surviving, proven they saved childen first

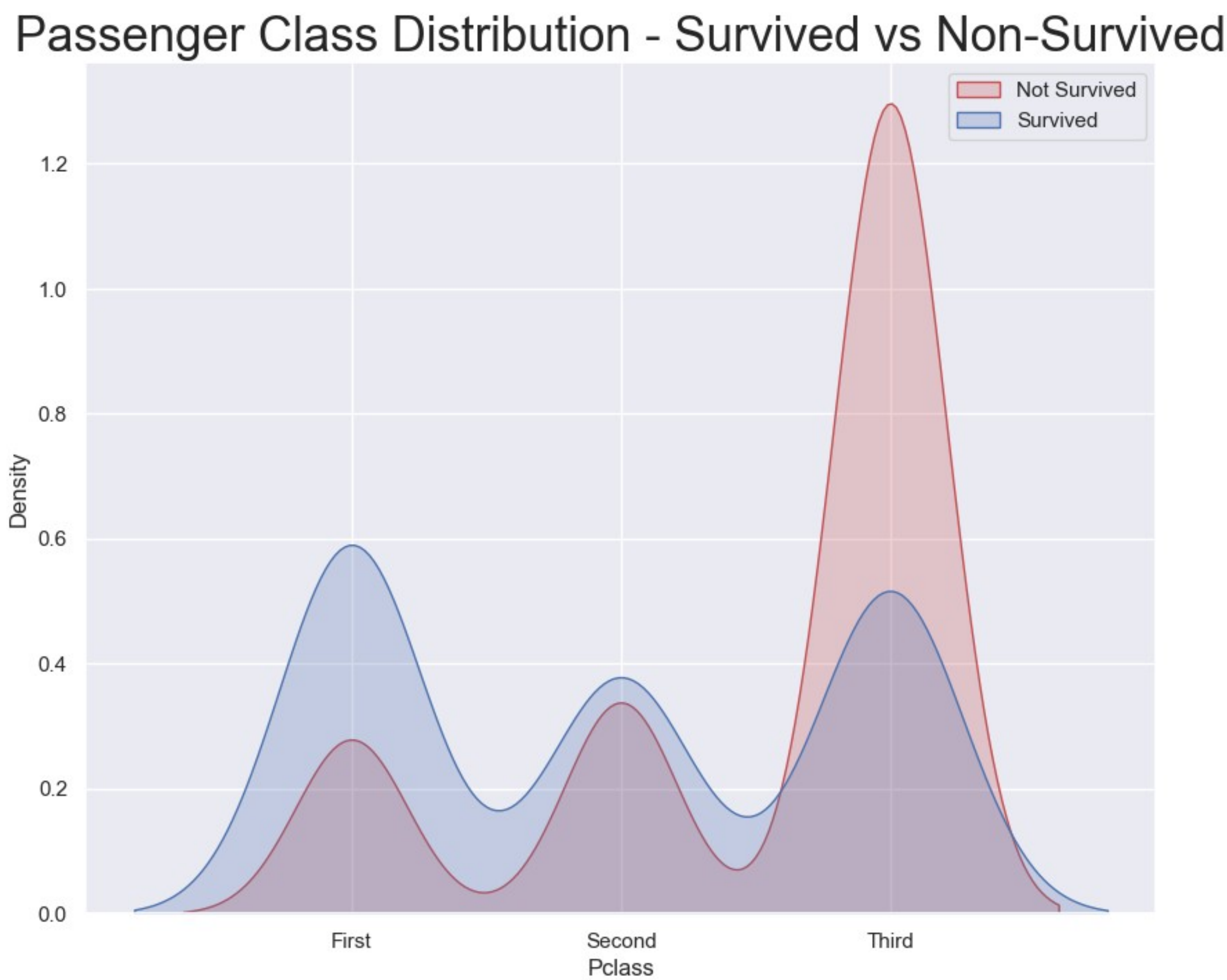
Passenger class distribution; Survived vs Non-Survived

```
plt.subplots(figsize = (8,8))
ax=sns.countplot(x='Pclass',hue='Survived',data=train_df)
plt.title("Passenger Class Distribution - Survived vs Non-Survived",
fontsize = 25);
```

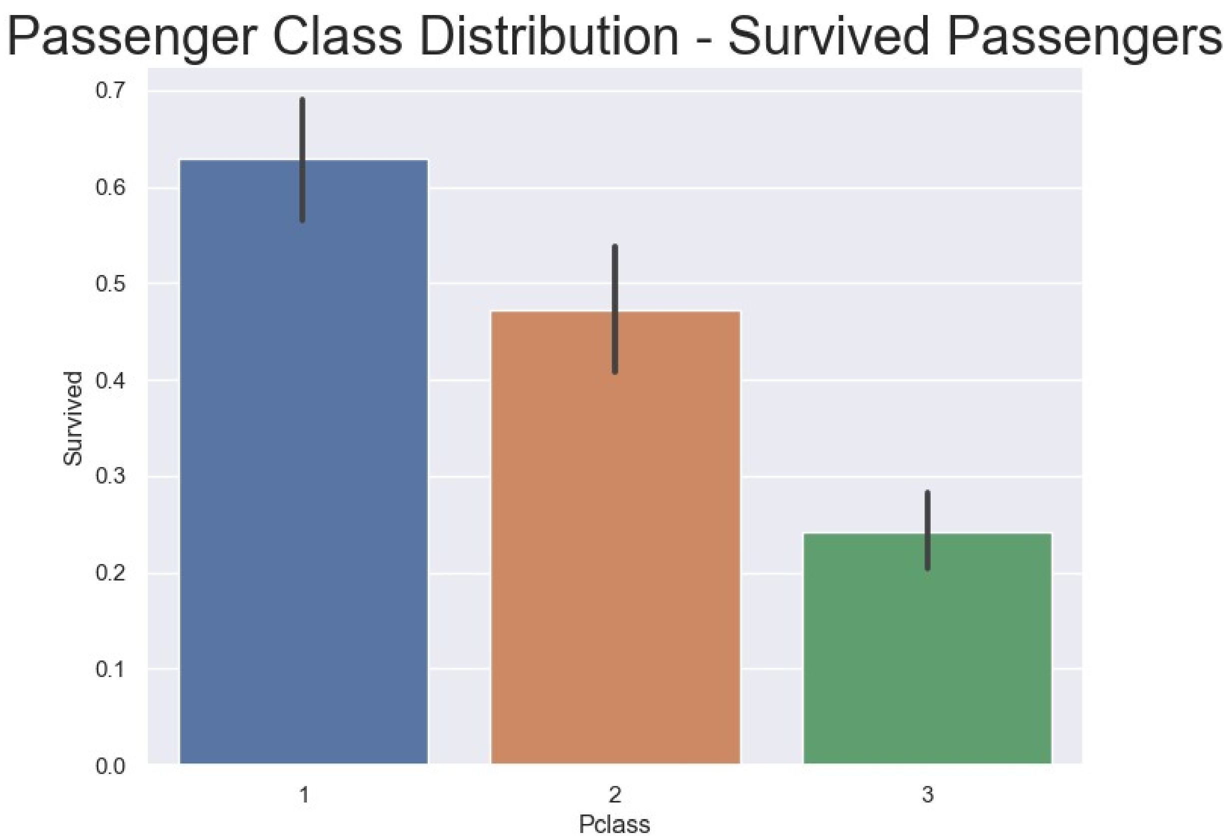

```
plt.subplots(figsize=(10,8))
ax=sns.kdeplot(train_df.loc[(train_df['Survived'] ==
0),'Pclass'],shade=True,color='r',label='Not Survived')
ax.legend()
ax=sns.kdeplot(train_df.loc[(train_df['Survived'] ==
1),'Pclass'],shade=True,color='b',label='Survived')
ax.legend()

plt.title("Passenger Class Distribution - Survived vs Non-Survived",
fontsize = 25)
labels = ['First', 'Second', 'Third']
plt.xticks(sorted(train_df.Pclass.unique()),labels);
```

```
C:\Users\varsh\AppData\Local\Temp\ipykernel_7644\942285626.py:2:
FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  ax=sns.kdeplot(train_df.loc[(train_df['Survived'] ==
```

```
0),'Pclass'],shade=True,color='r',label='Not Survived')
C:\Users\varsh\AppData\Local\Temp\ipykernel_7644\942285626.py:4:
FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  ax=sns.kdeplot(train_df.loc[(train_df['Survived'] ==
1),'Pclass'],shade=True,color='b',label='Survived')
```




```
plt.subplots(figsize = (8,6))
sns.barplot(x='Pclass', y='Survived', data=train_df);
plt.title("Passenger Class Distribution - Survived Passengers",
fontsize = 25);
```

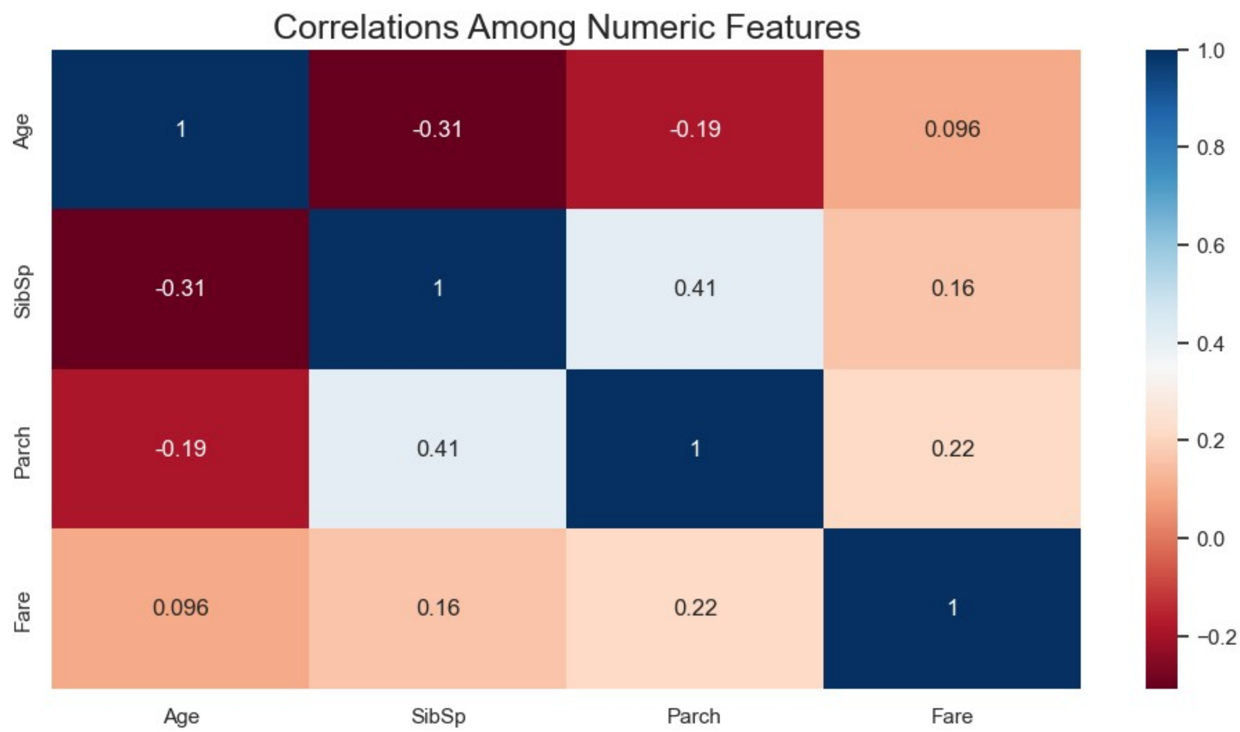


The graphs above clearly shows that **economic status (Pclass)** played an important role regarding the potential survival of the Titanic passengers. First class passengers had a much higher chance of survival than passengers in the 3rd class. We note that:

- 63% of the 1st class passengers survived the Titanic wreck
- 48% of the 2nd class passengers survived
- Only 24% of the 3rd class passengers survived

Correlation Matrix and Heatmap

```
# Look at numeric and categorical values separately
df_num = train_df[['Age','SibSp','Parch','Fare']]
df_cat =
train_df[['Survived','Pclass','Sex','Ticket','Cabin','Embarked']]

plt.subplots(figsize = (12,6))
sns.heatmap(df_num.corr(), annot=True,cmap="RdBu")
plt.title("Correlations Among Numeric Features", fontsize = 18);
```



We notice from the heatmap above that:

- **Parents and sibling like to travel together (light blue squares)**
- **Age has a high negative correlation with number of siblings**

Feature Engineering and Data Processing

Feature Engineering is the process of using raw data to create features that will be used for predictive modeling. Using, transforming, and combining existing features to define new features are also considered to be feature engineering.

Drop 'PassengerId'

First, I will drop ‘PassengerId’ from the train set, because it does not contribute to a persons' survival probability. I will not drop it from the test set, since it is required for the submission.

```
train_df = train_df.drop(['PassengerId'], axis=1)
train_df.head()
```

```
   Survived  Pclass                                               Name  \
0         0       3                            Braund, Mr. Owen Harris

1         1       1  Cumings, Mrs. John Bradley (Florence Briggs Th...

2         1       3                             Heikkinen, Miss. Laina

3         1       1       Futrelle, Mrs. Jacques Heath (Lily May Peel)
```

```
4          0       3                          Allen, Mr. William Henry

      Sex   Age  SibSp  Parch            Ticket     Fare Cabin
Embarked  \
0    male  22.0      1      0         A/5 21171   7.2500   NaN
S
1  female  38.0      1      0          PC 17599  71.2833   C85
C
2  female  26.0      0      0  STON/O2. 3101282   7.9250   NaN
S
3  female  35.0      1      0            113803  53.1000  C123
S
4    male  35.0      0      0            373450   8.0500   NaN
S

   train_test
0           1
1           1
2           1
3           1
4           1
```

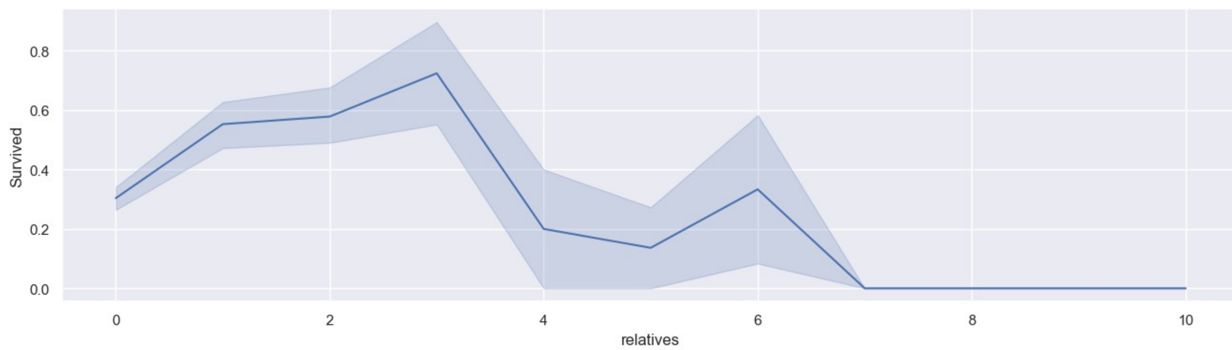
Combining SibSp and Parch

SibSp and Parch would make more sense as a combined feature that shows the total number of relatives a person has on the Titanic. I will create the new feature 'relative' below, and also a value that shows if someone is not alone.

```python
data = [train_df, test_df]
for dataset in data:
    dataset['relatives'] = dataset['SibSp'] + dataset['Parch']
    dataset.loc[dataset['relatives'] > 0, 'not_alone'] = 0
    dataset.loc[dataset['relatives'] == 0, 'not_alone'] = 1
    dataset['not_alone'] = dataset['not_alone'].astype(int)
train_df['not_alone'].value_counts()
```

```
1    537
0    354
Name: not_alone, dtype: int64
```

```python
plt.subplots(figsize = (16,4))
ax = sns.lineplot(x='relatives',y='Survived', data=train_df)
```



Missing Data

As a reminder, we have to deal with **Cabin (687 missing values), Embarked (2 missing values)** and **Age (177 missing values).**

```
import re
deck = {"A": 1, "B": 2, "C": 3, "D": 4, "E": 5, "F": 6, "G": 7, "U": 8}
data = [train_df, test_df]

for dataset in data:
    dataset['Cabin'] = dataset['Cabin'].fillna("U0")
    dataset['Deck'] = dataset['Cabin'].map(lambda x: re.compile("([a-zA-Z]+)").search(x).group())
    dataset['Deck'] = dataset['Deck'].map(deck)
    dataset['Deck'] = dataset['Deck'].fillna(0)
    dataset['Deck'] = dataset['Deck'].astype(int)

# We can now drop the Cabin feature
train_df = train_df.drop(['Cabin'], axis=1)
test_df = test_df.drop(['Cabin'], axis=1)
```

Age

As seen previously on **"3.1 Dealing with Missing Values"**, there are a lot of missing 'Age' values (177 data points). We can normalize the 'Age' feature by creating an array that contains random numbers, which are computed based on the mean age value in regards to the standard deviation and is_null.

```
data = [train_df, test_df]

for dataset in data:
    mean = train_df["Age"].mean()
    std = test_df["Age"].std()
    is_null = dataset["Age"].isnull().sum()

    # Compute random numbers between the mean, std and is_null
    rand_age = np.random.randint(mean - std, mean + std, size =
```

```
is_null)

    # Fill NaN values in Age column with random values generated
    age_slice = dataset["Age"].copy()
    age_slice[np.isnan(age_slice)] = rand_age
    dataset["Age"] = age_slice
    dataset["Age"] = train_df["Age"].astype(int)

train_df["Age"].isnull().sum()

0
```

Embarked

Since the Embarked feature has only 2 missing values, we will fill these with the most common one.

```
train_df['Embarked'].describe()

count     889
unique      3
top         S
freq      644
Name: Embarked, dtype: object
```

We notice the most popular embark location is **Southampton (S).**

```
common_value = 'S'
data = [train_df, test_df]

for dataset in data:
    dataset['Embarked'] = dataset['Embarked'].fillna(common_value)

train_df['Embarked'].isnull().sum()

0
```

Converting Features

```
train_df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 14 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   Survived    891 non-null    int64
 1   Pclass      891 non-null    int64
 2   Name        891 non-null    object
 3   Sex         891 non-null    object
```

```
 4   Age          891 non-null    int32
 5   SibSp        891 non-null    int64
 6   Parch        891 non-null    int64
 7   Ticket       891 non-null    object
 8   Fare         891 non-null    float64
 9   Embarked     891 non-null    object
 10  train_test   891 non-null    int64
 11  relatives    891 non-null    int64
 12  not_alone    891 non-null    int32
 13  Deck         891 non-null    int32
dtypes: float64(1), int32(3), int64(6), object(4)
memory usage: 87.1+ KB
```

We can see that **'Fare'** is a float data-type. Also, we need to deal with 4 categorical features: **Name, Sex, Ticket, and Embarked**

Fare

Converting 'Fare' from **float64** to **int64** using the **astype()** function provided by pandas

```
data = [train_df, test_df]

for dataset in data:
    dataset['Fare'] = dataset['Fare'].fillna(0)
    dataset['Fare'] = dataset['Fare'].astype(int)

train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 14 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Survived     891 non-null    int64
 1   Pclass       891 non-null    int64
 2   Name         891 non-null    object
 3   Sex          891 non-null    object
 4   Age          891 non-null    int32
 5   SibSp        891 non-null    int64
 6   Parch        891 non-null    int64
 7   Ticket       891 non-null    object
 8   Fare         891 non-null    int32
 9   Embarked     891 non-null    object
 10  train_test   891 non-null    int64
 11  relatives    891 non-null    int64
 12  not_alone    891 non-null    int32
 13  Deck         891 non-null    int32
dtypes: int32(4), int64(6), object(4)
memory usage: 83.7+ KB
```

Name

Feature Engineering the name of passengers to extract a person's title (Mr, Miss, Master, and Other), so we can build another feature called **'Title'** out of it.

```
data = [train_df, test_df]
titles = {"Mr": 1, "Miss": 2, "Mrs": 3, "Master": 4, "Other": 5}

for dataset in data:
    # Extract titles
    dataset['Title'] = dataset.Name.str.extract('([A-Za-z]+)\.', 
expand=False)

    # Replace titles with a more common title or as Other
    dataset['Title'] = dataset['Title'].replace(['Lady', 
'Countess','Capt', 'Col','Don', 'Dr','Major', 'Rev', 'Sir', 
'Jonkheer', 'Dona'], 'Other')
    dataset['Title'] = dataset['Title'].replace('Mlle', 'Miss')
    dataset['Title'] = dataset['Title'].replace('Ms', 'Miss')
    dataset['Title'] = dataset['Title'].replace('Mme', 'Mrs')

    # Convert titles into numbers
    dataset['Title'] = dataset['Title'].map(titles)

    # Filling NaN with 0 just to be safe
    dataset['Title'] = dataset['Title'].fillna(0)

train_df = train_df.drop(['Name'], axis=1)
test_df = test_df.drop(['Name'], axis=1)

# Checking results
train_df.head()
```

```
   Survived  Pclass     Sex  Age  SibSp  Parch            Ticket  Fare  
\
0         0       3    male   22      1      0         A/5 21171     7   

1         1       1  female   38      1      0          PC 17599    71   

2         1       3  female   26      0      0  STON/O2. 3101282     7   

3         1       1  female   35      1      0            113803    53   

4         0       3    male   35      0      0            373450     8   


  Embarked  train_test  relatives  not_alone  Deck  Title  
0        S           1          1          0     8      1  
1        C           1          1          0     3      3  
2        S           1          0          1     8      2  
```

```
3         S           1          1          0     3      3
4         S           1          0          1     8      1
```

Sex

Convert feature 'Sex' into numeric values

- male = 0
- female = 1

```
genders = {"male": 0, "female": 1}
data = [train_df, test_df]

for dataset in data:
    dataset['Sex'] = dataset['Sex'].map(genders)

train_df.head()
```

```
   Survived  Pclass  Sex  Age  SibSp  Parch            Ticket  Fare
Embarked  \
0         0       3    0   22      1      0         A/5 21171     7
S
1         1       1    1   38      1      0          PC 17599    71
C
2         1       3    1   26      0      0  STON/O2. 3101282     7
S
3         1       1    1   35      1      0            113803    53
S
4         0       3    0   35      0      0            373450     8
S

   train_test  relatives  not_alone  Deck  Title
0           1          1          0     8      1
1           1          1          0     3      3
2           1          0          1     8      2
3           1          1          0     3      3
4           1          0          1     8      1
```

Ticket

```
train_df['Ticket'].describe()
```

```
count        891
unique       681
top       347082
freq           7
Name: Ticket, dtype: object
```

Since the **'Ticket'** feature has 681 unique values, it would be very hard to convert them into an useful feature. **Hence, we will drop it from the DataFrame.**

```
train_df = train_df.drop(['Ticket'], axis=1)
test_df = test_df.drop(['Ticket'], axis=1)

train_df.head()
```

```
   Survived  Pclass  Sex  Age  SibSp  Parch  Fare Embarked  train_test
\
0         0       3    0   22      1      0     7        S           1

1         1       1    1   38      1      0    71        C           1

2         1       3    1   26      0      0     7        S           1

3         1       1    1   35      1      0    53        S           1

4         0       3    0   35      0      0     8        S           1


   relatives  not_alone  Deck  Title
0          1          0     8      1
1          1          0     3      3
2          0          1     8      2
3          1          0     3      3
4          0          1     8      1
```

Convert 'Embarked' feature into numeric values

```
ports = {"S": 0, "C": 1, "Q": 2}
data = [train_df, test_df]

for dataset in data:
    dataset['Embarked'] = dataset['Embarked'].map(ports)

train_df.head()
```

```
   Survived  Pclass  Sex  Age  SibSp  Parch  Fare  Embarked
train_test  \
0         0       3    0   22      1      0     7         0
1
1         1       1    1   38      1      0    71         1
1
2         1       3    1   26      0      0     7         0
1
3         1       1    1   35      1      0    53         0
1
4         0       3    0   35      0      0     8         0
1

   relatives  not_alone  Deck  Title
0          1          0     8      1
1          1          0     3      3
```

```
2           0           1      8       2
3           1           0      3       3
4           0           1      8       1
```

Creating new Categories

```
data = [train_df, test_df]
for dataset in data:
    dataset['Age'] = dataset['Age'].astype(int)
    dataset.loc[ dataset['Age'] <= 11, 'Age'] = 0
    dataset.loc[(dataset['Age'] > 11) & (dataset['Age'] <= 18), 'Age']
= 1
    dataset.loc[(dataset['Age'] > 18) & (dataset['Age'] <= 22), 'Age']
= 2
    dataset.loc[(dataset['Age'] > 22) & (dataset['Age'] <= 27), 'Age']
= 3
    dataset.loc[(dataset['Age'] > 27) & (dataset['Age'] <= 33), 'Age']
= 4
    dataset.loc[(dataset['Age'] > 33) & (dataset['Age'] <= 40), 'Age']
= 5
    dataset.loc[(dataset['Age'] > 40) & (dataset['Age'] <= 66), 'Age']
= 6
    dataset.loc[ dataset['Age'] > 66, 'Age'] = 6

# Checking the distribution
train_df['Age'].value_counts()

6    165
4    163
5    153
3    136
2    112
1     94
0     68
Name: Age, dtype: int64
```

Fare

For the 'Fare' feature, we need to do the same as with the 'Age' feature. But it isn't that easy, because if we cut the range of the fare values into a few equally big categories, 80% of the values would fall into the first category. Fortunately, we can use pandas "qcut()" function, that we can use to see, how we can form the categories.

```
train_df.head()

   Survived  Pclass  Sex  Age  SibSp  Parch  Fare  Embarked
train_test  \
0         0       3    0    2      1      0     7         0
1
1         1       1    1    5      1      0    71         1
```

```
1
2          1        3     1     3       0       0      7          0
1
3          1        1     1     5       1       0     53          0
1
4          0        3     0     5       0       0      8          0
1

   relatives  not_alone  Deck  Title
0          1          0     8      1
1          1          0     3      3
2          0          1     8      2
3          1          0     3      3
4          0          1     8      1
```

```
pd.qcut(train_df['Fare'], q=6)
```

```
0       (-0.001, 7.0]
1       (52.0, 512.0]
2       (-0.001, 7.0]
3       (52.0, 512.0]
4          (7.0, 8.0]
            ...
886       (8.0, 14.0]
887      (26.0, 52.0]
888      (14.0, 26.0]
889      (26.0, 52.0]
890     (-0.001, 7.0]
Name: Fare, Length: 891, dtype: category
Categories (6, interval[float64, right]): [(-0.001, 7.0] < (7.0, 8.0]
< (8.0, 14.0] < (14.0, 26.0] < (26.0, 52.0] < (52.0, 512.0]]
```

Using the values from **pd.qcut()** to create bins for Fare

```
data = [train_df, test_df]

for dataset in data:
    dataset.loc[ dataset['Fare'] <= 7, 'Fare'] = 0
    dataset.loc[(dataset['Fare'] > 7) & (dataset['Fare'] <= 8),
'Fare'] = 1
    dataset.loc[(dataset['Fare'] > 8) & (dataset['Fare'] <= 14),
'Fare']   = 2
    dataset.loc[(dataset['Fare'] > 14) & (dataset['Fare'] <= 26),
'Fare']   = 3
    dataset.loc[(dataset['Fare'] > 26) & (dataset['Fare'] <= 52),
'Fare']   = 4
    dataset.loc[dataset['Fare'] > 52, 'Fare'] = 5
    dataset['Fare'] = dataset['Fare'].astype(int)
```

```
# Checking the dataset
train_df.head(10)
```

```
   Survived  Pclass  Sex  Age  SibSp  Parch  Fare  Embarked
train_test  \
0         0       3    0    2      1      0     0         0
1
1         1       1    1    5      1      0     5         1
1
2         1       3    1    3      0      0     0         0
1
3         1       1    1    5      1      0     5         0
1
4         0       3    0    5      0      0     1         0
1
5         0       3    0    4      0      0     1         2
1
6         0       1    0    6      0      0     4         0
1
7         0       3    0    0      3      1     3         0
1
8         1       3    1    3      0      2     2         0
1
9         1       2    1    1      1      0     4         1
1

   relatives  not_alone  Deck  Title
0          1          0     8      1
1          1          0     3      3
2          0          1     8      2
3          1          0     3      3
4          0          1     8      1
5          0          1     8      1
6          0          1     5      1
7          4          0     8      4
8          2          0     8      3
9          1          0     8      3
```

Model building

```
X_train = train_df.drop("Survived", axis=1)
Y_train = train_df["Survived"]
X_test  = test_df.drop("PassengerId", axis=1).copy()
```

Stochastic Gradient Descent (SGD)

```
sgd = linear_model.SGDClassifier(max_iter=5, tol=None)
sgd.fit(X_train, Y_train)
```

```
Y_pred = sgd.predict(X_test)

sgd.score(X_train, Y_train)
acc_sgd = round(sgd.score(X_train, Y_train) * 100, 2)

# Print score
print(round(acc_sgd,2,), "%")
```

```
58.14 %
```

Decision Tree

```
decision_tree = DecisionTreeClassifier()
decision_tree.fit(X_train, Y_train)

Y_pred = decision_tree.predict(X_test)

acc_decision_tree = round(decision_tree.score(X_train, Y_train) * 100,
2)

# Print score
print(round(acc_decision_tree,2,), "%")
```

```
93.04 %
```

Random Forest

```
random_forest = RandomForestClassifier(n_estimators=100)
random_forest.fit(X_train, Y_train)

Y_prediction = random_forest.predict(X_test)

random_forest.score(X_train, Y_train)
acc_random_forest = round(random_forest.score(X_train, Y_train) * 100,
2)

# Print score
print(round(acc_random_forest,2,), "%")
```

```
93.04 %
```

Logistic Regression

```
logreg = LogisticRegression()
logreg.fit(X_train, Y_train)

Y_pred = logreg.predict(X_test)

acc_log = round(logreg.score(X_train, Y_train) * 100, 2)
```

```
# Print score
print(round(acc_log,2,), "%")
```

```
81.82 %
```

KNN

```
knn = KNeighborsClassifier(n_neighbors = 3)
knn.fit(X_train, Y_train)

Y_pred = knn.predict(X_test)

acc_knn = round(knn.score(X_train, Y_train) * 100, 2)

# Print score
print(round(acc_knn,2,), "%")
```

```
85.63 %
```

Gaussian Naive Bayes

```
gaussian = GaussianNB()
gaussian.fit(X_train, Y_train)

Y_pred = gaussian.predict(X_test)

acc_gaussian = round(gaussian.score(X_train, Y_train) * 100, 2)

# Print score
print(round(acc_gaussian,2,), "%")
```

```
78.45 %
```

Perceptron

```
perceptron = Perceptron(max_iter=1000)
perceptron.fit(X_train, Y_train)

Y_pred = perceptron.predict(X_test)

acc_perceptron = round(perceptron.score(X_train, Y_train) * 100, 2)

# Print score
print(round(acc_perceptron,2,), "%")
```

```
72.5 %
```

Model evaluation

Which one is the best model?

```
results = pd.DataFrame({
    'Model': ['KNN', 'Logistic Regression',
              'Random Forest', 'Naive Bayes', 'Perceptron',
              'Stochastic Gradient Decent',
              'Decision Tree'],
    'Score': [acc_knn, acc_log,
              acc_random_forest, acc_gaussian, acc_perceptron,
              acc_sgd, acc_decision_tree]})

result_df = results.sort_values(by='Score', ascending=False)
result_df = result_df.set_index('Score')
result_df.head(9)
```

Score	Model
93.04	Random Forest
93.04	Decision Tree
85.63	KNN
81.82	Logistic Regression
78.45	Naive Bayes
72.50	Perceptron
58.14	Stochastic Gradient Decent

The **Random Forest classifier** goes on top of the Machine Learning models, followed by **Decision Tree** and **KNN** respectfully. Now we need to check how the Random Forest performs by using cross validation.

K-Fold Cross Validation

K-Fold Cross Validation randomly splits the training data into **K subsets called folds**. Image we split our data into 4 folds (K = 4). The random forest model would be trained and validated 4 times, using a different fold for validation every time, while it would be trained on the remaining 3 folds.

The image below shows the process, using 4 folds (K = 4). Every row represents one training + validation process. In the first row, the model is trained on the second, third and fourth subsets and validated on the first subset. In the second row, the model is trained on the first, third and fourth subsets and validated on the second subset. K-Fold Cross Validation repeats this process until every fold acted once as an evaluation fold.

```
from sklearn.model_selection import cross_val_score

rf = RandomForestClassifier(n_estimators=100)
scores = cross_val_score(rf, X_train, Y_train, cv=10, scoring =
"accuracy")
```

```python
print("Scores:", scores)
print("Mean:", scores.mean())
print("Standard Deviation:", scores.std())
```

```
Scores: [0.77777778 0.83146067 0.75280899 0.85393258 0.87640449
0.83146067
 0.79775281 0.76404494 0.84269663 0.83146067]
Mean: 0.8159800249687891
Standard Deviation: 0.038734632131379475
```

This looks much more realistic than before. The **Random Forest classifier** model has an average **accuracy of 81%** with a **standard deviation of 3.9%**. The standard deviation tell us how precise the estimates are.

- This means the accuracy of our model can differ **± 3.9%**

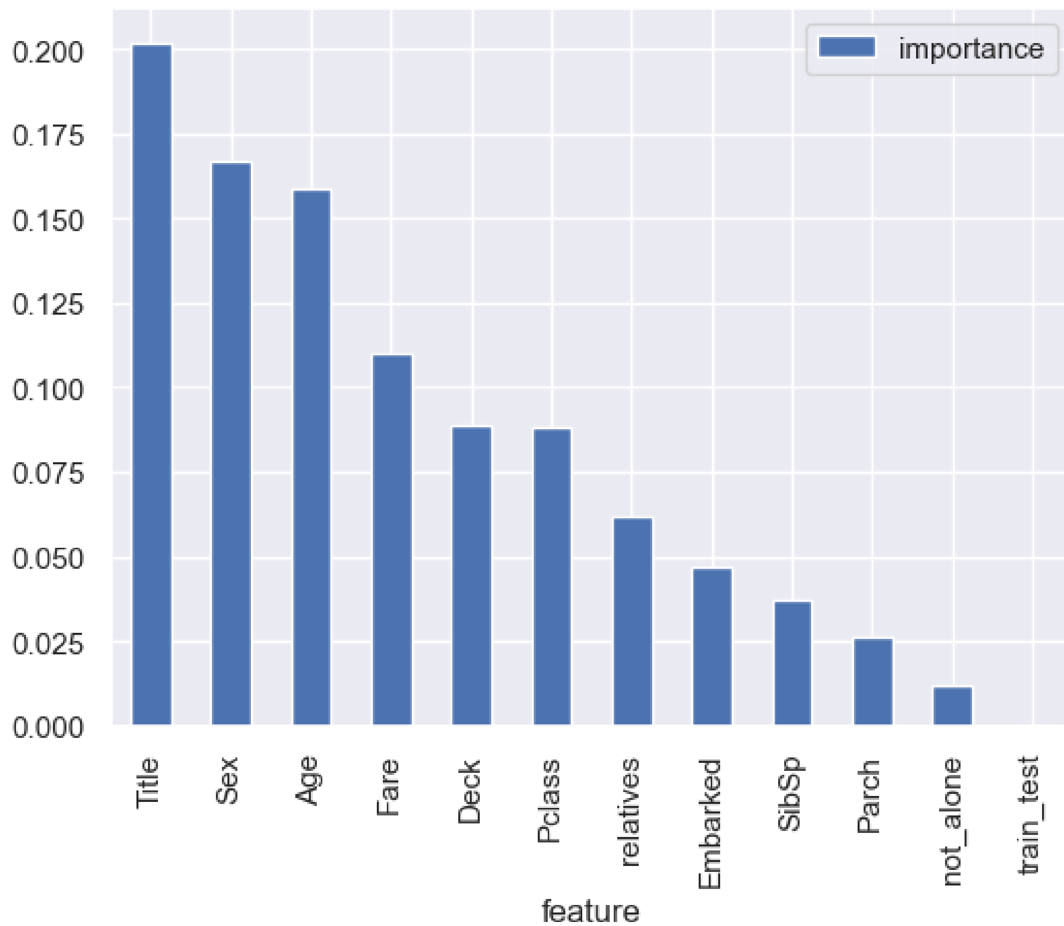
I believe the accuracy looks good. Since Random Forest is a model easy to use, we will try to increase its performance even further in the following section.

Random Forest

```python
importances =
pd.DataFrame({'feature':X_train.columns,'importance':np.round(random_forest.feature_importances_,3)})
importances =
importances.sort_values('importance',ascending=False).set_index('feature')

importances.head(12)
```

```
            importance
feature
Title            0.202
Sex              0.167
Age              0.159
Fare             0.110
Deck             0.089
Pclass           0.088
relatives        0.062
Embarked         0.047
SibSp            0.037
Parch            0.026
not_alone        0.012
train_test       0.000
```

```python
importances.plot.bar();
```



Results

'not_alone' and 'Parch' don't play a significant role in the Random Forest classifiers prediction process. Thus, I will drop them from the DataFrame and train the classifier once again. We could also remove more features, however, this would inquire more investigations of the feature's effect on our model. For now, I will only remove 'not_alone' and 'Parch' from the DataFrame.

```
# Dropping not_alone
train_df  = train_df.drop("not_alone", axis=1)
test_df  = test_df.drop("not_alone", axis=1)

# Dropping Parch
train_df  = train_df.drop("Parch", axis=1)
test_df  = test_df.drop("Parch", axis=1)

# # Reassigning features
X_train = train_df.drop("Survived", axis=1)
Y_train = train_df["Survived"]
X_test  = test_df.drop("PassengerId", axis=1).copy()
```

Training the Random Forest classifier once again

```
random_forest = RandomForestClassifier(n_estimators=100, oob_score = 
True)
random_forest.fit(X_train, Y_train)

Y_prediction = random_forest.predict(X_test)

random_forest.score(X_train, Y_train)
acc_random_forest = round(random_forest.score(X_train, Y_train) * 100, 
2)

# Print scores
print(round(acc_random_forest,2,), "%")
```

```
93.04 %
```

Feature importance without 'not_alone' and 'Parch' features

```
importances = 
pd.DataFrame({'feature':X_train.columns,'importance':np.round(random_f
orest.feature_importances_,3)})
importances = 
importances.sort_values('importance',ascending=False).set_index('featu
re')

importances.head(12)
```

	importance
feature	
Title	0.205
Sex	0.179
Age	0.154
Fare	0.105
Pclass	0.089
Deck	0.089
relatives	0.088
Embarked	0.047
SibSp	0.044
train_test	0.000

The **Random Forest** model predicts as good as it did before. A general rule is that, the more features you have, the more likely your model will suffer from overfitting and vice versa. But I think our data looks fine for now and hasn't too much features.

Moreover, there is another way to validate the Random Forest classifier, which is as accurate as the score used before. We can use something called **Out of Bag (OOB) score** to estimate the generalization accuracy. **Basically, the OOB score is computed as the number of correctly predicted rows from the out of the bag sample**.

```
print("oob score:", round(random_forest.oob_score_, 4)*100, "%")
```

```
oob score: 80.92 %
```

Now we can start tuning the **hyperameters** of random forest.

Hyperparameter Tuning

```
# Simple performance reporting function
def clf_performance(classifier, model_name):
    print(model_name)
    print('Best Score: ' + str(classifier.best_score_))
    print('Best Parameters: ' + str(classifier.best_params_))

from sklearn.model_selection import GridSearchCV

rf = RandomForestClassifier(random_state = 1)
param_grid =  {'n_estimators': [400,450,500,550],
               'criterion':['gini','entropy'],
               'bootstrap': [True],
               'max_depth': [15, 20, 25],
               'max_features': ['auto','sqrt', 10],
               'min_samples_leaf': [2,3],
               'min_samples_split': [2,3]}

clf_rf = GridSearchCV(rf, param_grid = param_grid, cv = 5, verbose =
True, n_jobs = -1)
best_clf_rf = clf_rf.fit(X_train,Y_train)

# Print score
clf_performance(best_clf_rf,'Random Forest')

Fitting 5 folds for each of 288 candidates, totalling 1440 fits

c:\Users\varsh\AppData\Local\Programs\Python\Python311\Lib\site-
packages\sklearn\ensemble\_forest.py:424: FutureWarning:
`max_features='auto'` has been deprecated in 1.1 and will be removed
in 1.3. To keep the past behaviour, explicitly set
`max_features='sqrt'` or remove this parameter as it is also the
default value for RandomForestClassifiers and ExtraTreesClassifiers.
  warn(

Random Forest
Best Score: 0.8316489862532169
Best Parameters: {'bootstrap': True, 'criterion': 'entropy',
'max_depth': 15, 'max_features': 'auto', 'min_samples_leaf': 3,
'min_samples_split': 2, 'n_estimators': 400}
```

Testing new parameters

```
random_forest = RandomForestClassifier(criterion = "gini",
                                       max_depth = 20,
                                       max_features='auto',
```

```
                                        min_samples_leaf = 3,
                                        min_samples_split = 2,
                                        n_estimators=450,
                                        oob_score=True,
                                        random_state=1,
                                        n_jobs=-1)

random_forest.fit(X_train, Y_train)
Y_prediction = random_forest.predict(X_test)

random_forest.score(X_train, Y_train)

print("oob score:", round(random_forest.oob_score_, 4)*100, "%")
```

```
c:\Users\varsh\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\ensemble\_forest.py:424: FutureWarning:
`max_features='auto'` has been deprecated in 1.1 and will be removed in 1.3. To keep the past behaviour, explicitly set
`max_features='sqrt'` or remove this parameter as it is also the default value for RandomForestClassifiers and ExtraTreesClassifiers.
  warn(

oob score: 82.38 %
```

Further evaluation

Confusion Matrix

```
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import confusion_matrix

predictions = cross_val_predict(random_forest, X_train, Y_train, cv=3)
confusion_matrix(Y_train, predictions)
```

```
c:\Users\varsh\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\ensemble\_forest.py:424: FutureWarning:
`max_features='auto'` has been deprecated in 1.1 and will be removed in 1.3. To keep the past behaviour, explicitly set
`max_features='sqrt'` or remove this parameter as it is also the default value for RandomForestClassifiers and ExtraTreesClassifiers.
  warn(
c:\Users\varsh\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\ensemble\_forest.py:424: FutureWarning:
`max_features='auto'` has been deprecated in 1.1 and will be removed in 1.3. To keep the past behaviour, explicitly set
`max_features='sqrt'` or remove this parameter as it is also the default value for RandomForestClassifiers and ExtraTreesClassifiers.
  warn(
c:\Users\varsh\AppData\Local\Programs\Python\Python311\Lib\site-packages\sklearn\ensemble\_forest.py:424: FutureWarning:
```

```
`max_features='auto'` has been deprecated in 1.1 and will be removed
in 1.3. To keep the past behaviour, explicitly set
`max_features='sqrt'` or remove this parameter as it is also the
default value for RandomForestClassifiers and ExtraTreesClassifiers.
  warn(

array([[493,  56],
       [100, 242]], dtype=int64)
```

The first row is about the not-survived-predictions: **494 passengers were correctly classified as not survived** (called true negatives) and **55 where wrongly classified as not survived** (false positives).

The second row is about the survived-predictions: **98 passengers where wrongly classified as survived** (false negatives) and **244 where correctly classified as survived** (true positives).

A confusion matrix produces an idea of how accurate the model is.

Precision and Recall

```
from sklearn.metrics import precision_score, recall_score

print("Precision:", precision_score(Y_train, predictions))
print("Recall:",recall_score(Y_train, predictions))

Precision: 0.8120805369127517
Recall: 0.7076023391812866
```

Our model predicts correctly that **a passenger survived 81% of the time** (precision). The **recall** tells us that **71% of the passengers tested actually survived.**

F-score

It is possible to combine precision and recall into one score, which is called the F-score. The F-score is computed with the harmonic mean of precision and recall. Note that it assigns more weight to low values. As a result, the classifier will only get a high F-score if both recall and precision are high.

```
from sklearn.metrics import f1_score
f1_score(Y_train, predictions)

0.7562500000000001
```

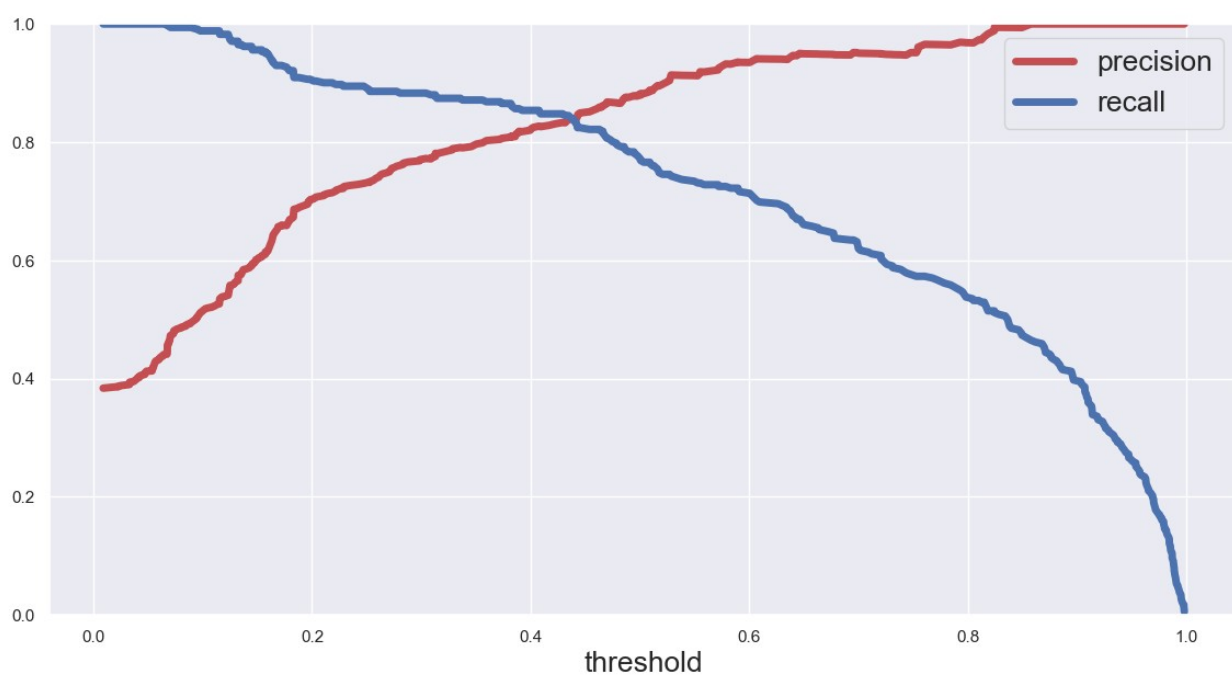
There we have it, a **76% F-score.** The score is not high because we have a recall of 71%. Unfortunately, the F-score is not perfect, because it favors classifiers that have a similar precision and recall. This can be a problem because often times we are searching for a high precision and other times a high recall. An increase of precision can result in a decrease of recall, and vice versa (depending on the threshold). This is called the **precision/recall trade-off.**

Precision Recall Curve

For each person the Random Forest algorithm has to classify, it computes a probability based on a function and it classifies the person as **survived** (when the score is bigger the than threshold) or as **not survived** (when the score is smaller than the threshold). That's why the threshold plays an important part in this process.

Let's plot the precision and recall with the threshold using matplotlib.

```
from sklearn.metrics import precision_recall_curve

# Getting the probabilities of our predictions
y_scores = random_forest.predict_proba(X_train)
y_scores = y_scores[:,1]

precision, recall, threshold = precision_recall_curve(Y_train,
y_scores)
def plot_precision_and_recall(precision, recall, threshold):
    plt.plot(threshold, precision[:-1], "r", label="precision",
linewidth=5)
    plt.plot(threshold, recall[:-1], "b", label="recall", linewidth=5)
    plt.xlabel("threshold", fontsize=19)
    plt.legend(loc="upper right", fontsize=19)
    plt.ylim([0, 1])

plt.figure(figsize=(14, 7))
plot_precision_and_recall(precision, recall, threshold)
plt.show()
```

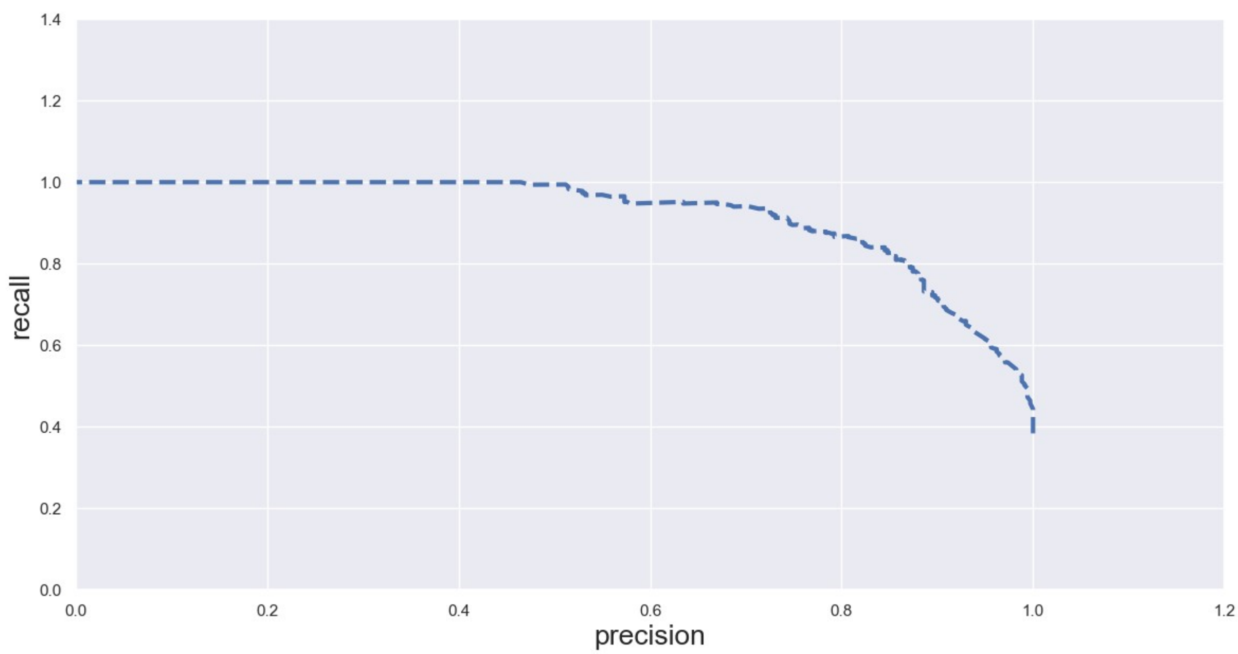


We can see in the graph above that the recall is falling of rapidly when the precision reaches around 85%. Thus, we may want to select the precision/recall trade-off before this point (maybe at around 75%).

Now we are able to choose a threshold, that gives the best precision/recall trade-off for the current problem. For example, if a precision of 80% is required, we can easily look at the plot and identify the threshold needed, which is around 0.4. Then we could train the model with exactly that threshold and expect the desired accuracy.

Another way is to plot the precision and recall against each other:

```
def plot_precision_vs_recall(precision, recall):
    plt.plot(recall, precision, "b--", linewidth=3)
    plt.xlabel("precision", fontsize=19)
    plt.ylabel("recall", fontsize=19)
    plt.axis([0, 1.2, 0, 1.4])

plt.figure(figsize=(14, 7))
plot_precision_vs_recall(precision, recall)
plt.show()
```

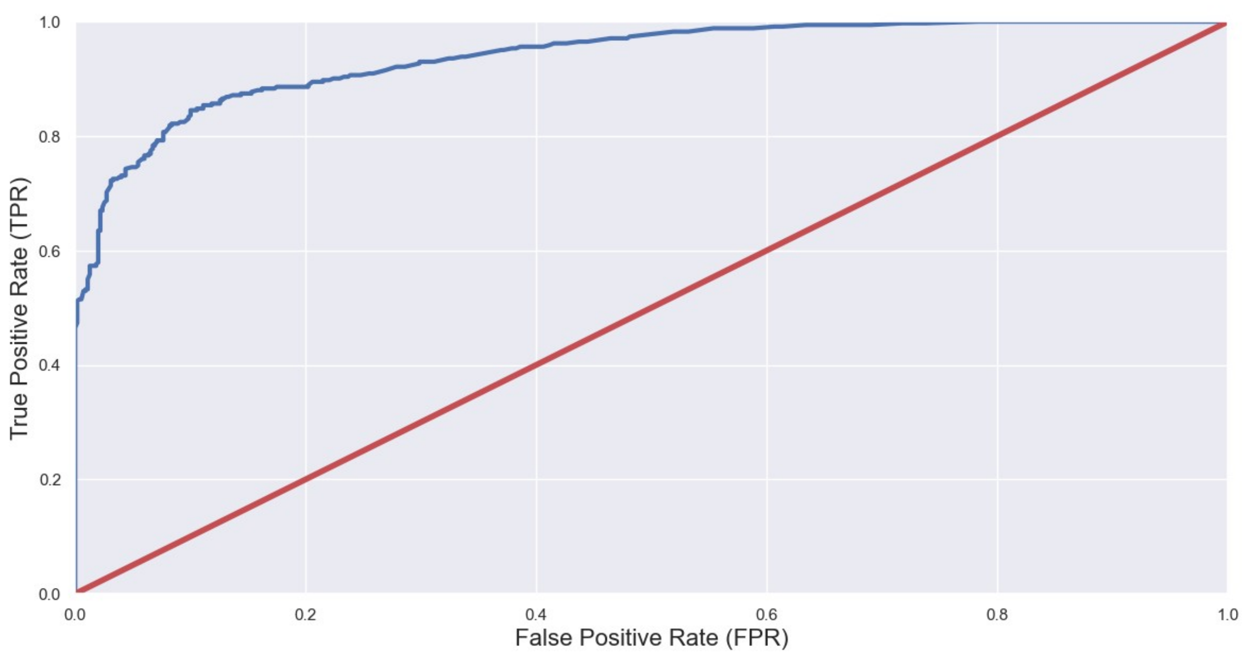


ROC AUC Curve

Another way to evaluate and compare binary classifiers is the ROC AUC Curve. This curve plots the true positive rate (also called recall) against the false positive rate (ratio of incorrectly classified negative instances), instead of plotting the precision versus the recall values.

```
from sklearn.metrics import roc_curve

# Compute true positive rate and false positive rate
```

```python
false_positive_rate, true_positive_rate, thresholds = 
roc_curve(Y_train, y_scores)

# Plotting them against each other
def plot_roc_curve(false_positive_rate, true_positive_rate, 
label=None):
    plt.plot(false_positive_rate, true_positive_rate, linewidth=3, 
label=label)
    plt.plot([0, 1], [0, 1], 'r', linewidth=4)
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate (FPR)', fontsize=16)
    plt.ylabel('True Positive Rate (TPR)', fontsize=16)

plt.figure(figsize=(14, 7))
plot_roc_curve(false_positive_rate, true_positive_rate)
plt.show()
```



The red line represents a purely random classifier (e.g. a coin flip). Thus, the classifier should be as far away from it as possible. The Random Forest model looks good.

There's a tradeoff here because the classifier produces more false positives the higher the true positive rate is.

ROC AUC Score

The ROC AUC Score is the corresponding score to the ROC AUC Curve. It is simply computed by measuring the area under the curve, which is called AUC.

A classifier that is 100% correct would have a ROC AUC Score of 1, and a completely random classifier would have a score of 0.5.

```python
from sklearn.metrics import roc_auc_score
r_a_score = roc_auc_score(Y_train, y_scores)
print("ROC-AUC-Score:", r_a_score)
```

```
ROC-AUC-Score: 0.9386604032850797
```

We got a **93% ROC AUC Score**