Titanic Survival Analysis

Description of the dataset

Attribute	Description of attribute
survival	Survival (0 = No; 1 = Yes)
pclass	Passenger Class (1 = 1st; 2 = 2nd; 3 = 3rd)
name	Passenger's Name
sex	Passenger's Sex
age	Passenger's Age
sibsp	Number of Siblings/Spouses Aboard
parch	Number of Parents/Children Aboard
ticket	Ticket Number
fare	Passenger Fare
cabin	Cabin Passenger was assigned to
embarked	Port of Embarkation (C = Cherbourg; Q = Queenstown; S = Southampton)
boat	Lifeboat (if survived)
body	Body number (if did not survive and body was recovered)
home.dest	Passenger's destination

Step 1: Import all libraries

```
[2]  # Import all the necessary libaries
     import matplotlib.pyplot as plt
     import numpy as np
     import pandas as pd
     import seaborn as sns
     import tensorflow as tf
```

Step 2: Exploring the data

```
[3]  #Reading the Titanic dataset using pandas library to explore the dataset
     titanic = pd.read_csv('titanic.csv')

     # to print number of rows and columns
     print('number of rows and columns')
     print(titanic.shape)
     print() #prints empty line between results

     #to study the datatypes of the attributes
     print('datatypes of the attributes')
     print(titanic.dtypes)
     print()#prints empty line between results

     #to display the overview of the dataset
     print('Overview of the data')
     titanic.head()
```

```
[3] number of rows and columns
    (1310, 14)

    datatypes of the attributes
    pclass       float64
    survived     float64
    name          object
    sex           object
    age          float64
    sibsp        float64
    parch        float64
    ticket        object
    fare         float64
    cabin         object
    embarked      object
    boat          object
    body         float64
    home.dest     object
    dtype: object

    Overview of the data
```

	pclass	survived	name	sex	age	sibsp	parch	ticket	fare	cabin	embarked	boat	body	home.dest
0	1.0	1.0	Allen, Miss. Elisabeth Walton	female	29.0000	0.0	0.0	24160	211.3375	B5	S	2	NaN	St Louis, MO
1	1.0	1.0	Allison, Master. Hudson Trevor	male	0.9167	1.0	2.0	113781	151.5500	C22 C26	S	11	NaN	Montreal, PQ / Chesterville, ON
2	1.0	0.0	Allison, Miss. Helen Loraine	female	2.0000	1.0	2.0	113781	151.5500	C22 C26	S	NaN	NaN	Montreal, PQ / Chesterville, ON
3	1.0	0.0	Allison, Mr. Hudson Joshua Creighton	male	30.0000	1.0	2.0	113781	151.5500	C22 C26	S	NaN	135.0	Montreal, PQ / Chesterville, ON
4	1.0	0.0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	female	25.0000	1.0	2.0	113781	151.5500	C22 C26	S	NaN	NaN	Montreal, PQ / Chesterville, ON

Step 3: Function for Preprocessing the data

```
[5] #Preprocessing functions to preprocess data

    # Function to drop unused columns from the dataset
    def drop_unused_columns(titanic):
      return titanic.drop(columns=['name','ticket','cabin','boat','body','home.dest'],axis=1)

    # Function to convert categorical values to numbers
    def convert_categorical_to_numbers(titanic):
      from sklearn.preprocessing import LabelEncoder
      labelencoder= LabelEncoder()
      titanic.iloc[:,2]  =  labelencoder.fit_transform(titanic.iloc[:,2].values)
      titanic.iloc[:,7]  =  labelencoder.fit_transform(titanic.iloc[:,7].values)
      return titanic

    # Function to drop rows with na values
    def drop_rows_with_na(titanic):
      titanic = titanic.dropna(subset=['pclass','survived','sex','age','sibsp','parch','fare','embarked'])
      return titanic

    # Function to reorder the columns
    def reorder_columns(titanic):
     titanic = titanic[['survived','pclass','sex','age','sibsp','parch','fare','embarked']]
     return titanic

    #Creating an object to append all the accuracy values to visulaize at the end.
    all_accuracies = {}
```

Step 4: Overview of the data after the data is preprocessed

```
[6] #Data preprocessed - using preprocessing functions, #exploring the manipulated data

    #drop unused columns
    titanic = drop_unused_columns(titanic)

    #drop rows with na values
    titanic = drop_rows_with_na(titanic)

    #reorder the columns
    titanic = reorder_columns(titanic)

    #convert categorical values to numbers
    titanic = convert_categorical_to_numbers(titanic)

    #print the values in the data post preprocessing
    print('Dataset after preprocessing')
    print(titanic)

    #Generate descriptive statistics of the attributes
    print('Descriptive statistics')
    titanic.describe()
```

```
Dataset after preprocessing
      survived  pclass  sex      age  sibsp  parch      fare  embarked
0          1.0     1.0    0  29.0000    0.0    0.0  211.3375         2
1          1.0     1.0    1   0.9167    1.0    2.0  151.5500         2
2          0.0     1.0    0   2.0000    1.0    2.0  151.5500         2
3          0.0     1.0    1  30.0000    1.0    2.0  151.5500         2
4          0.0     1.0    0  25.0000    1.0    2.0  151.5500         2
...        ...     ...  ...      ...    ...    ...       ...       ...
1301       0.0     3.0    1  45.5000    0.0    0.0    7.2250         0
1304       0.0     3.0    0  14.5000    1.0    0.0   14.4542         0
1306       0.0     3.0    1  26.5000    0.0    0.0    7.2250         0
1307       0.0     3.0    1  27.0000    0.0    0.0    7.2250         0
1308       0.0     3.0    1  29.0000    0.0    0.0    7.8750         2

[1043 rows x 8 columns]
Descriptive statistics
```

	survived	pclass	sex	age	sibsp	parch	fare	embarked
count	1043.000000	1043.000000	1043.000000	1043.000000	1043.000000	1043.000000	1043.000000	1043.000000
mean	0.407478	2.209012	0.629914	29.813199	0.504314	0.421860	36.603024	1.545542
std	0.491601	0.840685	0.483059	14.366261	0.913080	0.840655	55.753648	0.809366
min	0.000000	1.000000	0.000000	0.166700	0.000000	0.000000	0.000000	0.000000
25%	0.000000	1.000000	0.000000	21.000000	0.000000	0.000000	8.050000	1.000000
50%	0.000000	2.000000	1.000000	28.000000	0.000000	0.000000	15.750000	2.000000
75%	1.000000	3.000000	1.000000	39.000000	1.000000	1.000000	35.077100	2.000000
max	1.000000	3.000000	1.000000	80.000000	8.000000	6.000000	512.329200	2.000000

Attributes after preprocessing and the table shows the attributes retained.

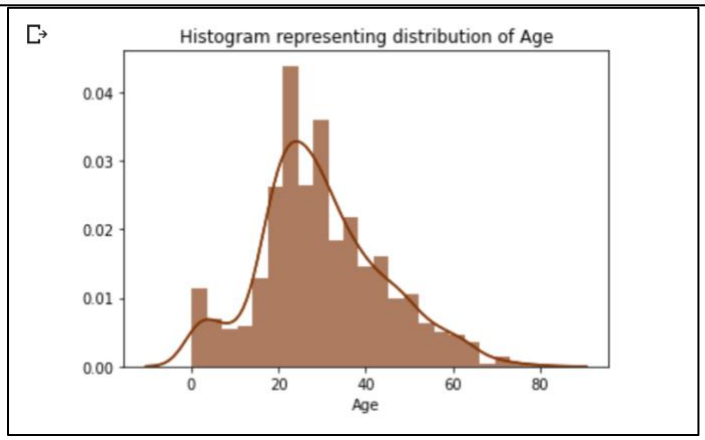
Attribute	Description of attribute

survival	Survival (0 = No; 1 = Yes)
pclass	Passenger Class (1 = 1st; 2 = 2nd; 3 = 3rd)
sex	Passenger's Sex (0 = Female, 1=Male)
age	Passenger's Age (Minimum =16months, Maximum = 80)
sibsp	Number of Siblings/Spouses Aboard (Minimum =1 count, Maximum = 8 count)
parch	Number of Parents/Children Aboard (Minimum =1 count, Maximum = 6count)
fare	Passenger Fare (Minimum =0 pounds, Maximum = 512.32 pounds)
embarked	Port of Embarkation (C - Cherbourg = 0; Q - Queenstown = 1; S - Southampton = 2)

Step 5: Exploring data with the help of visualizations:

1. The distribution of passengers based on age

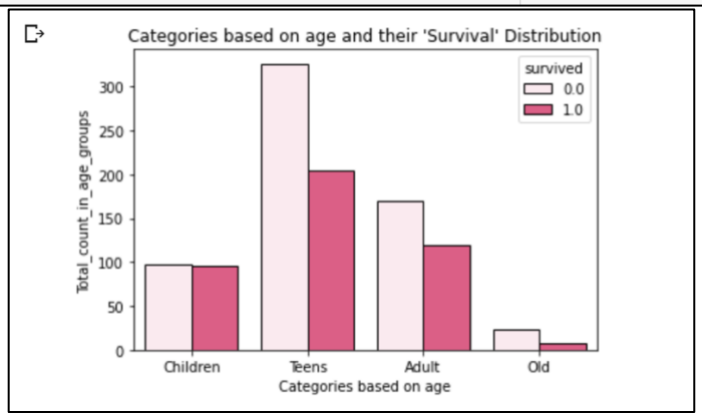
```
#to visualize the distribution of passengers by "Age"
matplotlib_Axes = sns.distplot(titanic['age'], color="#803305")
sns.distplot(titanic['age'], color="#803305").set(xlabel='Age',
                                                  title="Histogram representing distribution of Age")
plt.show()
```



Age of passengers range from 0 to 80. As per plot, there are many passengers in the range of 15 – 45 years of age.

2. The chances of survival for passengers in different age groups

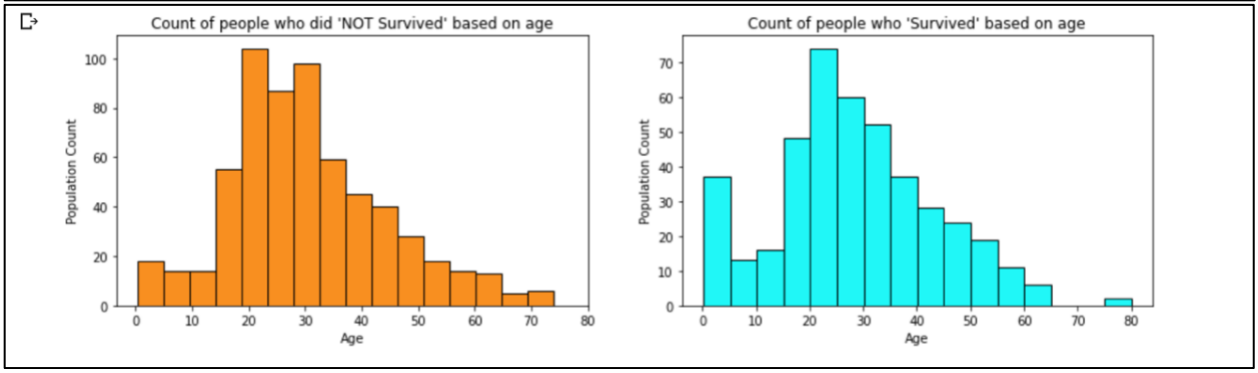
```
#to visulaize the chances of survival for passengers in different age groups - Children,adults,teens and old aged
interval = (0,18,35,60,120)
categories_based_on_age = ['Children','Teens','Adult', 'Old']
titanic['Age_categories'] = pd.cut(titanic.age, interval, labels = categories_based_on_age)
matplotlib_Axes = sns.countplot(x = 'Age_categories',  data = titanic, hue = 'survived', edgecolor = "black", color = "#f04a7f")
matplotlib_Axes.set(xlabel='Categories based on age', ylabel='Total_count_in_age_groups',
       title="Categories based on age and their 'Survival' Distribution")
plt.show()
```



Amongst all passengers, children have highest chance of survival compared to teens, adults and old ages passengers.

3. The count of passengers who survived and who did not survive based on age

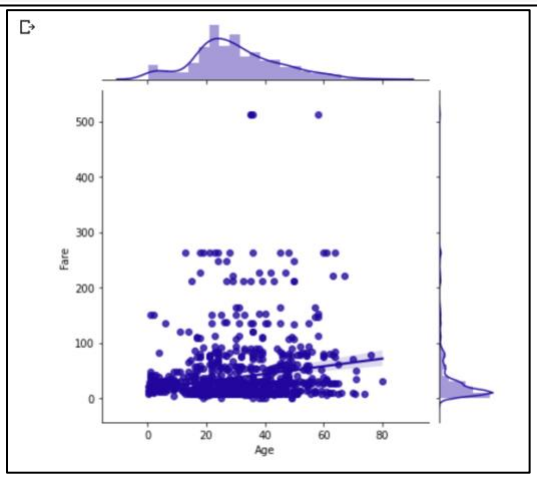
```
[9] #to visulaize count of passengers who survived and who did not survive based on age
    fig, matplotlib_Axes = plt.subplots(1, 2, figsize = (15, 4))
    titanic[titanic["survived"] == 0]["age"].plot.hist(ax = matplotlib_Axes[0], bins = 16, edgecolor = "black", color = "#f78f20")
    matplotlib_Axes[0].set_title("Count of people who did 'NOT Survived' based on age")
    matplotlib_Axes[0].set_xlabel("Age")
    matplotlib_Axes[0].set_ylabel("Population Count")
    domain_1 = list(range(0, 85, 10))
    matplotlib_Axes[0].set_xticks(domain_1)
    titanic[titanic["survived"] == 1]["age"].plot.hist(ax = matplotlib_Axes[1], bins = 16, edgecolor = "black", color = "#20f7f7")
    matplotlib_Axes[1].set_title("Count of people who 'Survived' based on age")
    matplotlib_Axes[1].set_xlabel("Age")
    matplotlib_Axes[1].set_ylabel("Population Count")
    domain_2 = list(range(0, 85, 10))
    matplotlib_Axes[1].set_xticks(domain_2)
    plt.show()
```



As per the histogram, survival count is higher for passengers between 0-5 years of age and for passengers between 15 to 25. The chances of survival decrease with increase in the age after 25.

4. The distribution of passengers with respect to age and fare

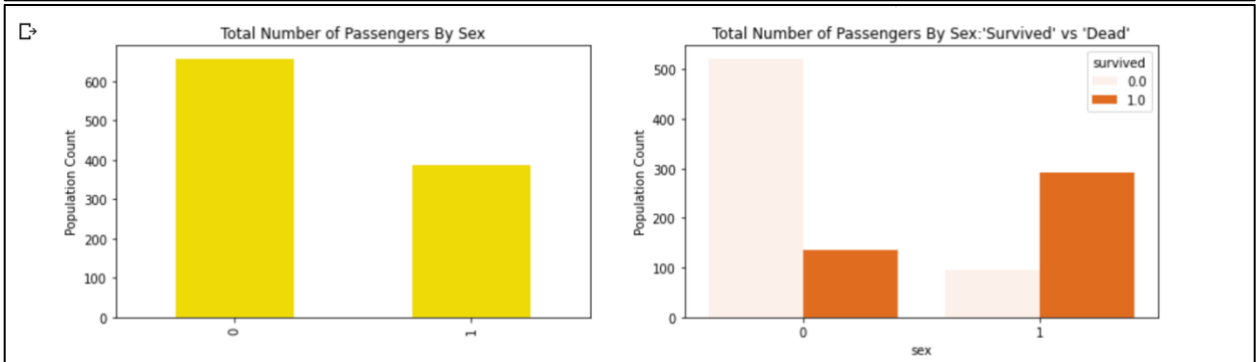
```
[10] #visualize relationship between passenger age and fare
     matplotlib_Axes = sns.jointplot(data=titanic, x='age', y='fare', kind='reg', color='#22069e')
     matplotlib_Axes.ax_joint.set_xlabel("Age")
     matplotlib_Axes.ax_joint.set_ylabel("Fare")
     plt.show()
```



As per the joint plot, maximum number of passengers are between 15-45 years of age and their fare amount is less than 100 pounds. The highest fare amount greater than 500 pounds is paid by passengers in mid 30's and late 50's.

5. The chances of survival based on passenger's sex

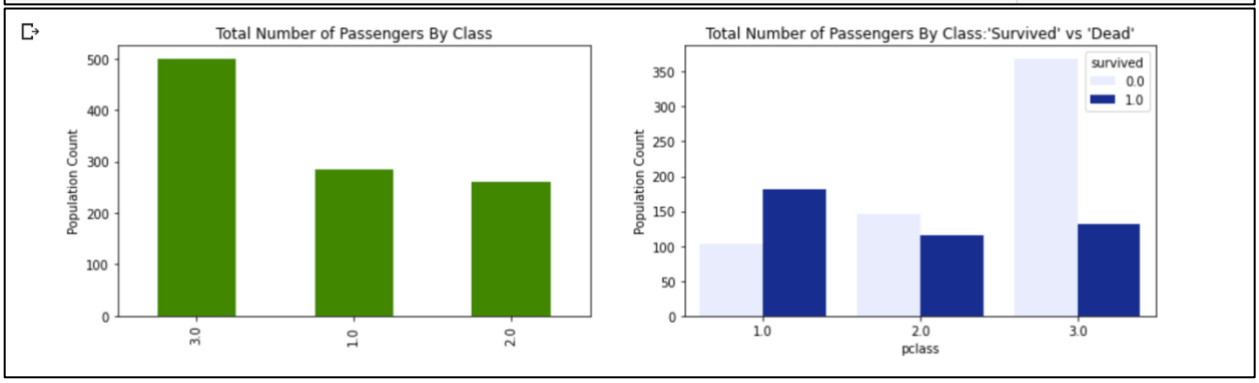
```
[11] #Visualize the count of passenger based on sex and their survival chances,
     #0- female, 1-male , 0-unsurvived, 1-survived
     fig, matplotlib_Axes = plt.subplots(1, 2, figsize = (15, 4))
     titanic["sex"].value_counts().plot.bar(color = "#edda05", ax = matplotlib_Axes[0])
     matplotlib_Axes[0].set_title("Total Number of Passengers By Sex")
     matplotlib_Axes[0].set_ylabel("Population Count")
     sns.countplot("sex", hue = "survived", data = titanic, ax = matplotlib_Axes[1],color='#ff6600')
     matplotlib_Axes[1].set_title("Total Number of Passengers By Sex:'Survived' vs 'Dead'")
     matplotlib_Axes[1].set_ylabel("Population Count")
     plt.show()
```



It can be observed that the count of female passengers is higher than male passengers and the bar plot in the right indicates that females have higher chances of survival compared to males.

6. The count of passenger by class and their survival chances by class

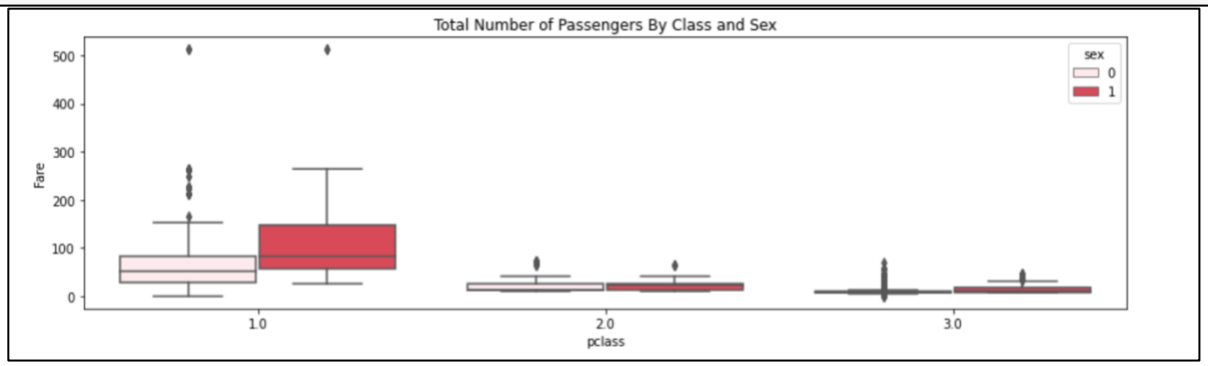
```
[12] #Visualize the count of passenger based on class and their survival chances,
     #1-firstClass, 2-SecondClass,3-ThirdClass , 0-unsurvived, 1-survived
     fig, matplotlib_Axes = plt.subplots(1, 2, figsize = (15, 4))
     titanic["pclass"].value_counts().plot.bar(color = "#408701", ax = matplotlib_Axes[0])
     matplotlib_Axes[0].set_title("Total Number of Passengers By Class")
     matplotlib_Axes[0].set_ylabel("Population Count")
     sns.countplot("pclass", hue = "survived", data = titanic, ax = matplotlib_Axes[1],color='#0321a3')
     matplotlib_Axes[1].set_title("Total Number of Passengers By Class:'Survived' vs 'Dead'")
     matplotlib_Axes[1].set_ylabel("Population Count")
     plt.show()
```



As per the bar plot, there count of passenger in the third class is higher compared to passengers in second and the first class. In the bar plot, at the right, indicates that the survival chances of patients are very low for third class. Passengers in first class have highest chances of survival compared to other two classes.

7. Variation in fare with respect to the class passenger belonged to

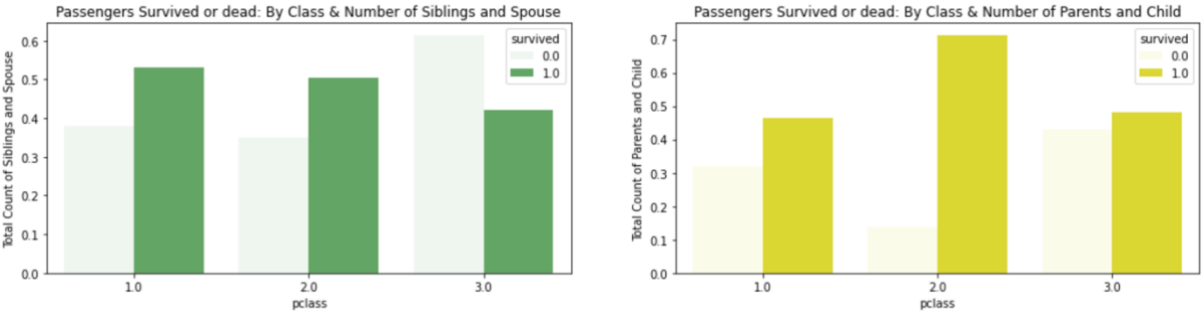
```
[13] #visualize passenger fare with respect to their clas
     fig, matplotlib_Axes = plt.subplots(1, figsize = (15, 4))
     sns.boxplot(x='pclass',y='fare',hue='sex',data=titanic,color='#f03245')
     matplotlib_Axes.set_title("Total Number of Passengers By Class and Sex")
     matplotlib_Axes.set_ylabel("Fare")
     plt.show()
```



As per the boxplot, the fare amount for both males and females are almost the same for passengers in second class and third class and their fare amount is fairly lower than 100. Whereas the fare amount is significantly higher for first class passengers. Within the first class, men seem to have paid high fare amount compared to females.

8. The survival chances for passengers accompanied by siblings, spouse, parents or children with respect to the class passenger belonged to

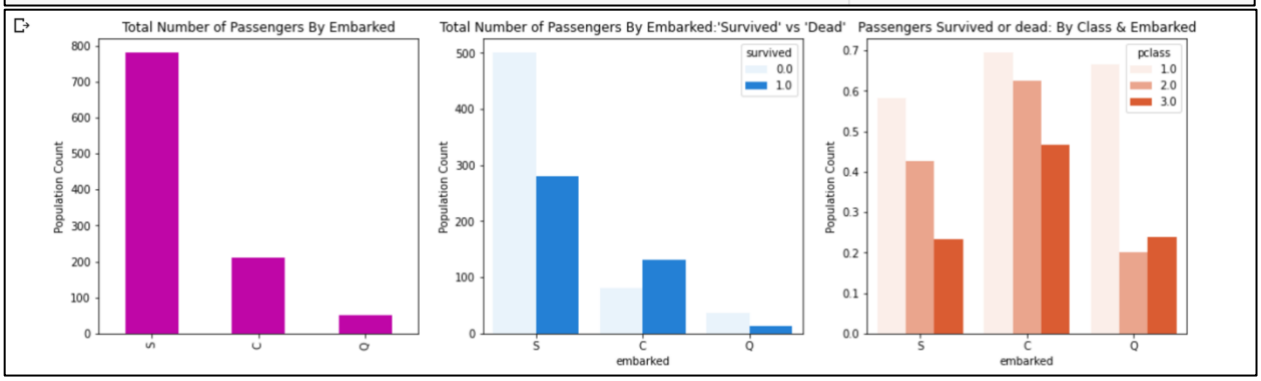
```
[14] #Visualize the survival chnaces for passengers travelling with siblings, spouse, parents and children
     #1-firstClass, 2-SecondClass,3-ThirdClass , 0-unsurvived, 1-survived
     fig, matplotlib_Axes = plt.subplots(1, 2, figsize = (18, 4))
     sns.barplot(x="pclass", y='sibsp',hue = "survived", data = titanic, ax = matplotlib_Axes[0],color='#58b059',ci=None)
     matplotlib_Axes[0].set_title("Passengers Survived or dead: By Class & Number of Siblings and Spouse")
     matplotlib_Axes[0].set_ylabel("Total Count of Siblings and Spouse")

     sns.barplot(x="pclass", y='parch',hue = "survived", data = titanic, ax = matplotlib_Axes[1],color='#f5f116',ci=None)
     matplotlib_Axes[1].set_title("Passengers Survived or dead: By Class & Number of Parents and Child")
     matplotlib_Axes[1].set_ylabel("Total Count of Parents and Child")

     plt.show()
```



As per the bar plot, at the left, the chances of survival for passenger is highest in first class and second class provided the passengers are with their siblings or spouse. But on the other hand, this is not true for passengers in the third class. In the bar plot, at the right, the chances of survival are better for passengers accompanied by children or parents. Amongst all the classes, second class passengers travelling with children or parents have highest chances of survival.

9. Influence of embarked on the chances of survival

```
[15] #Visualize the count of passenger based on embarked and their survival chances,
     #1-firstClass, 2-SecondClass,3-ThirdClass , 0-unsurvived, 1-survived
     #embarked -> C-Cherbourg = 0, Q- Queenstown=1, S-Southhampton=2
     fig, matplotlib_Axes = plt.subplots(1, 3, figsize = (18, 5))
     titanic["embarked"].value_counts().plot.bar(color = "#bf06a7", ax = matplotlib_Axes[0])
     matplotlib_Axes[0].set_title("Total Number of Passengers By Embarked")
     matplotlib_Axes[0].set_ylabel("Population Count")

     sns.countplot("embarked", hue = "survived", data = titanic, ax = matplotlib_Axes[1],color='#0781f2')
     matplotlib_Axes[1].set_title("Total Number of Passengers By Embarked:'Survived' vs 'Dead'")
     matplotlib_Axes[1].set_ylabel("Population Count")

     sns.barplot(x="embarked", y='survived',hue = "pclass", data = titanic, ax = matplotlib_Axes[2],color='#f54e16',ci=None)
     matplotlib_Axes[2].set_title("Passengers Survived or dead: By Class & Embarked")
     matplotlib_Axes[2].set_ylabel("Population Count")
     plt.show()
```



As per the leftmost bar plot, significantly amount of people have boarded from ‘S’ compared to ‘C’ and ‘Q’. The middle bar plot summarizes that passengers who have embarked from ‘C’ have higher chances of survival and that is because most there are higher number of passengers who belong to first class as shown in the rightmost bar plot.

Step 6: Skikit-Learn for predicting survival of the passenger.

Skikit-learn library is used to create the models to measure accuracy. The data is first preprocessed and then split into training and the test dataset. Models used are Logistic regression, K nearest neighbors, Gaussian Naïve Bayes, Decision tree and Random Forest.

```
# Classification models using Skikit learn

#Reading the titanic dataset
titanic = pd.read_csv('titanic.csv')

#Drop unwanted attributes
titanic = drop_unused_columns(titanic)
#drop rows with NA
titanic = drop_rows_with_na(titanic)
#re order columsn to split into depedent and independent variables
titanic = reorder_columns(titanic)
#convert categorical attributes to numeric datatype
titanic = convert_categorical_to_numbers(titanic)
Xvars= titanic.iloc[:,1:8].values
Yvars= titanic.iloc[:,0].values

#Splitting the dataset into 80% training dataset and 20% testing dataset
from sklearn.model_selection import train_test_split
train_Xvars, test_Xvars, train_Yvars, test_Yvars= train_test_split(Xvars,Yvars,test_size=0.2,random_state=0)

#Scale the data so that it is normalized
from sklearn.preprocessing import StandardScaler
sc=StandardScaler()
train_Xvars=sc.fit_transform(train_Xvars)
test_Xvars=sc.fit_transform(test_Xvars)

#Creating a function for classifier models.
#Models using machine learning from skikit learn library
def classification_models(train_Xvars,train_Yvars):

  #Logistic Regression classifier
  from sklearn.linear_model import LogisticRegression
  LogisticRegression= LogisticRegression(random_state=0)
  LogisticRegression.fit(train_Xvars,train_Yvars)
```

```
  #K nearest Neighbors classifier
  from sklearn.neighbors import KNeighborsClassifier
  KNeighborsClassifier=KNeighborsClassifier(n_neighbors=5, metric='minkowski',p=2)
  KNeighborsClassifier.fit(train_Xvars,train_Yvars)

  #Gaussian Naive Bayes classifier
  from sklearn.naive_bayes import GaussianNB
  GaussianNB = GaussianNB()
  GaussianNB.fit(train_Xvars,train_Yvars)

  #Decision tree classifier
  from sklearn.tree import DecisionTreeClassifier
  DecisionTreeClassifier = DecisionTreeClassifier(criterion='entropy',random_state=0)
  DecisionTreeClassifier.fit(train_Xvars,train_Yvars)

  #Random Forest Classifier
  from sklearn.ensemble import RandomForestClassifier
  RandomForestClassifier = RandomForestClassifier(n_estimators=10, criterion='entropy', random_state=0)
  RandomForestClassifier.fit(train_Xvars,train_Yvars)

  #Printing the training acccurcy for each model
  print('[0]Logistic Regression, Accuracy for training data:', LogisticRegression.score(train_Xvars,train_Yvars))
  print('[1]K Neighbors Training Accusracy, Accuracy for training data:', KNeighborsClassifier.score(train_Xvars,train_Yvars))
  print('[2]Gaussian NB Training Accusracy, Accuracy for training data:', GaussianNB.score(train_Xvars,train_Yvars))
  print('[3]Decision Tree Training Accusracy, Accuracy for training data:', DecisionTreeClassifier.score(train_Xvars,train_Yvars))
  print('[4]Random Forest Training Accusracy, Accuracy for training data:', RandomForestClassifier.score(train_Xvars,train_Yvars))

  #appending accuracy values into an object created above
  all_accuracies['Logistic Regression'] = float(LogisticRegression.score(train_Xvars,train_Yvars))
  all_accuracies['K Neighbors Classifier'] = float(KNeighborsClassifier.score(train_Xvars,train_Yvars))
  all_accuracies['Gaussian Naive Bayes'] = float(GaussianNB.score(train_Xvars,train_Yvars))
  all_accuracies['Decision Tree Classifier'] = float(DecisionTreeClassifier.score(train_Xvars,train_Yvars))
  all_accuracies['Random Forest Classifier'] = float(RandomForestClassifier.score(train_Xvars,train_Yvars))

  return LogisticRegression, KNeighborsClassifier, GaussianNB, DecisionTreeClassifier, RandomForestClassifier
```

```
#Calling the function for all the training data
classifier_accuracy_results = classification_models(train_Xvars,train_Yvars)
```

```
[0]Logistic Regression, Accuracy for training data: 0.7805755395683454
[1]K Neighbors Training Accusracy, Accuracy for training data: 0.8453237410071942
[2]Gaussian NB Training Accusracy, Accuracy for training data: 0.7793764988009593
[3]Decision Tree Training Accusracy, Accuracy for training data: 0.9784172661870504
[4]Random Forest Training Accusracy, Accuracy for training data: 0.9592326139088729
```

Random Forest and Decision tree classifier models have higher accuracy for training data.

Performance of Classification models on the testing data using confusion matrix

```
[17] #Performance of classification models on test data using confusion matrix for accuracy
     from sklearn.metrics import confusion_matrix
     for i in range( len(classifier_accuracy_results) ):
       confusionMatrix=confusion_matrix(test_Yvars,classifier_accuracy_results[i].predict(test_Xvars))

       #True negative,True positive,False positive and False negative
       TrueNegative,FalsePositive,FalseNegative,TruePositive = confusion_matrix(test_Yvars,classifier_accuracy_results[i].predict(test_Xvars)).ravel()

      #Accuracy for testing data
       Accuracy_for_testing_data = (TruePositive+TrueNegative) / (TruePositive+TrueNegative+FalseNegative+FalsePositive)

       print(confusionMatrix)
       print('-----------------------------------------------------------')
       print("Classifier Model ID[{}] Testing Accuracy = '{}'".format(i, Accuracy_for_testing_data ))
       print()
```

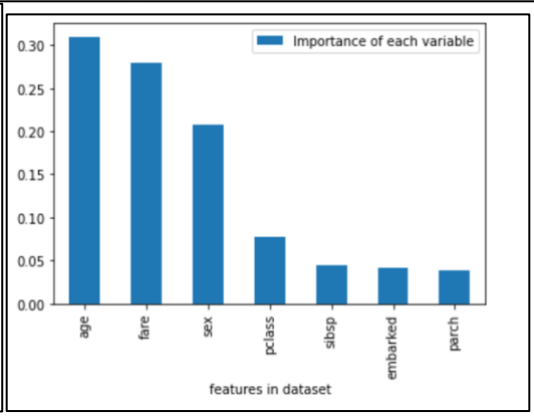
```
[[103  23]
 [ 19  64]]
-----------------------------------------------------------
Classifier Model ID[0] Testing Accuracy = '0.7990430622009569'

[[117   9]
 [ 28  55]]
-----------------------------------------------------------
Classifier Model ID[1] Testing Accuracy = '0.8229665071770335'

[[97 29]
 [19 64]]
-----------------------------------------------------------
Classifier Model ID[2] Testing Accuracy = '0.7703349282296651'

[[104  22]
 [ 28  55]]
-----------------------------------------------------------
Classifier Model ID[3] Testing Accuracy = '0.7607655502392344'

[[115  11]
 [ 32  51]]
-----------------------------------------------------------
Classifier Model ID[4] Testing Accuracy = '0.7942583732057417'
```

Testing accuracy is greater for K nearest neighbors compared to other classifier models

Feature importances of Random Forest Classifier

```
[18] #Feature importances of Random Forest Classifier to study the impact of each attribute in the prediction of survival
     RandomForestClassifier = classifier_accuracy_results[4]
     feature_importances = pd.DataFrame({'features in dataset':titanic.iloc[:,1:8].columns,
                                         'Importance of each variable': np.round(RandomForestClassifier.feature_importances_,3)})
     feature_importances = feature_importances.sort_values('Importance of each variable',ascending=False).set_index('features in dataset')
     feature_importances.plot.bar()
     feature_importances
```

features in dataset	Importance of each variable
age	0.310
fare	0.280
sex	0.208
pclass	0.078
sibsp	0.044
embarked	0.041
parch	0.039



Random Forest importances are considered because it has higher accuracy for training data. Age, fare and sex have a significant influence on passenger being classified as survived and not survived.

Testing the model for Rose and Jack Survival

name	pclass	sex	age	sibsp	parch	fare	embarked
Rose	1	0	20	1	0	100	0
Jack	3	1	23	1	0	5	0

```
[20] #Predicting chances of survival for 'Rose'
     Predict_Rose_survival=[[1,0,20,1,0,100,0]]

     prediction_val=classifier_accuracy_results[1].predict(Predict_Rose_survival)
     print(prediction_val)

     if prediction_val==0:
       print('Sorry! You did not survive!!!')
     else:
       print('Great News! You survived!')

     [1.]
     Great News! You survived!
```

```
     #Predicting chances of survival for 'Jack'
     Predict_Jack_survival=[[3,1,23,1,0,5,0]]

     prediction_val=classifier_accuracy_results[1].predict(Predict_Jack_survival)
     print(prediction_val)

     if prediction_val==0:
       print('Sorry! You did not survive!!!')
     else:
       print('Great News! You survived!')

     [0.]
     Sorry! You did not survive!!!
```

The model can be further used to predict the survival chances of Jack and Rose. As per the prediction results, Rose survives and Jack does not. The table represents the inputs for Rose and Jack that was used for prediction.

Step 6: TensorFlow for predicting survival of the passenger using the TensorFlow's high level API with keras.

Building a model by assembling layers, the most common type of such model is tf.keras.Sequential. The first step is to create a fully connected network to train this model with the training data. The data is fed to the model after breaking up the entire training dataset into batches, here the batch size is 10. When all the batches are fed exactly once, then it completes an epoch. By repeating this process multiple times increases the success of training the model. With multiple epoch the accuracy increases and with a certain number of epochs the accuracy remains almost the same.

```
[21] # Predict survival using Tensorflow's high level API with keras
     #using tf.keras

     #Creating independent and dependent data variables
     def create_Xvar_Yvar(titanic, total_size):
       Yvars = []
       for i in range(0, total_size):
         if i in titanic.index:
           Yvar = np.zeros(2)
           if titanic.at[i, 'survived'] == 0:
             Yvar[0] = 1.0
           elif titanic.at[i, 'survived'] == 1:
             Yvar[1] = 1.0
           Yvars.append(Yvar)
       Yvars = np.array(Yvars)

       Xvar = titanic.drop(columns=['survived'])
       Xvar = np.array(Xvar)

       return Xvar, Yvars
```

```
#generate training and test data after preprocessing the data
#preprocessing donw with the help of preprocessing functions declared above
def generate_dataset(input):
  total_size = len(input)
  #dropping unwanted attributes
  input = drop_unused_columns(input)
  #dropping rows with 'na'
  input = drop_rows_with_na(input)
  #convert categorical attributes to numeric datatype
  input = convert_categorical_to_numbers(input)
  return create_Xvar_Yvar(input, total_size)

#read titanic daatset
titanic = pd.read_csv('titanic.csv')

#randomly select 80% for training
indices = np.random.rand(len(titanic)) <= 0.8

#creating training dataset
training_data = titanic[indices]
training_data = training_data.reset_index(drop=True)

#creating testing dataset
test_data = titanic[~indices]
test_data = test_data.reset_index(drop=True)

train_Xvars, train_Yvars = generate_dataset(training_data)
test_Xvars, test_Yvars = generate_dataset(test_data)
```

```
#creating a sequential model using tf.keras
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(500, activation='relu', input_shape=[7,]))
model.add(tf.keras.layers.Dense(500, activation='relu'))
model.add(tf.keras.layers.Dense(500, activation='relu'))
model.add(tf.keras.layers.Dense(2, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='Adam', metrics=['accuracy'])

#initializing metric variables
total_training_accuracy = 0.0
total_validation_accuracy = 0.0
total_test_accuracy = 0.0

#using the sequential model created above
for _ in range(0, 1):
  history = model.fit(train_Xvars, train_Yvars, epochs=70, batch_size=10, validation_split=0.2)
  dictionary = pd.DataFrame(history.history)

  #Calculating mean accuracy and validation of mean accuracy
  total_training_accuracy += dictionary['accuracy'].mean()
  total_validation_accuracy += dictionary['val_accuracy'].mean()

  #evaluate the model
  testing_loss, testing_accuracy = model.evaluate(test_Xvars, test_Yvars)

  total_test_accuracy += testing_accuracy

#Priting model metrics obtained
print('Training mean accuracy: ' + str(total_training_accuracy))
print('Validation mean accuracy: ' + str(total_validation_accuracy))
print('Testing mean accuracy: ' + str(total_test_accuracy))

#append the accuracy for the model for furture visulaization
all_accuracies['Sequential Neural Network'] = total_training_accuracy
```

Results for Epoch=50 and Epoch=70

Below the results are shown for both epoch = 50 and epoch =70, showing that for the training data the mean accuracy increases with increasing epoch. With epoch =50 the accuracy is about 75% and it increases close to 80% with epoch=70 and anything greater than epoch of 70 the accuracy remains the same.

Results for epoch = 50

```
Epoch 1/30
68/68 [==============================] - 0s 6ms/step - loss: 1.1420 - accuracy: 0.5911 - val_loss: 0.5707 - val_accuracy: 0.7396
Epoch 2/30
68/68 [==============================] - 0s 5ms/step - loss: 0.7315 - accuracy: 0.6015 - val_loss: 0.7151 - val_accuracy: 0.6036
Epoch 3/30
68/68 [==============================] - 0s 5ms/step - loss: 0.6829 - accuracy: 0.6474 - val_loss: 0.6286 - val_accuracy: 0.5858
Epoch 4/30
68/68 [==============================] - 0s 5ms/step - loss: 0.6531 - accuracy: 0.6563 - val_loss: 0.6224 - val_accuracy: 0.5976
Epoch 5/30
68/68 [==============================] - 0s 5ms/step - loss: 0.6190 - accuracy: 0.6681 - val_loss: 0.6988 - val_accuracy: 0.5858
Epoch 6/30
68/68 [==============================] - 0s 5ms/step - loss: 0.6276 - accuracy: 0.6770 - val_loss: 0.6019 - val_accuracy: 0.6154
Epoch 7/30
68/68 [==============================] - 0s 5ms/step - loss: 0.5985 - accuracy: 0.6874 - val_loss: 0.7694 - val_accuracy: 0.6095
Epoch 8/30
68/68 [==============================] - 0s 5ms/step - loss: 0.6156 - accuracy: 0.7274 - val_loss: 0.6091 - val_accuracy: 0.6982
Epoch 9/30
68/68 [==============================] - 0s 5ms/step - loss: 0.5686 - accuracy: 0.7363 - val_loss: 0.5691 - val_accuracy: 0.7041
Epoch 10/30
68/68 [==============================] - 0s 5ms/step - loss: 0.5377 - accuracy: 0.7733 - val_loss: 0.6262 - val_accuracy: 0.7278
Epoch 11/30
68/68 [==============================] - 0s 5ms/step - loss: 0.5931 - accuracy: 0.7644 - val_loss: 0.6160 - val_accuracy: 0.6864
Epoch 12/30
68/68 [==============================] - 0s 5ms/step - loss: 0.5183 - accuracy: 0.7600 - val_loss: 0.5524 - val_accuracy: 0.7160
Epoch 13/30
68/68 [==============================] - 0s 5ms/step - loss: 0.4814 - accuracy: 0.7867 - val_loss: 0.5896 - val_accuracy: 0.6982
Epoch 14/30
68/68 [==============================] - 0s 5ms/step - loss: 0.4915 - accuracy: 0.7941 - val_loss: 0.6025 - val_accuracy: 0.6982
Epoch 15/30
68/68 [==============================] - 0s 5ms/step - loss: 0.4792 - accuracy: 0.8044 - val_loss: 0.5812 - val_accuracy: 0.7041
```

```
Epoch 16/30
68/68 [==============================] - 0s 5ms/step - loss: 0.5007 - accuracy: 0.7896 - val_loss: 0.6197 - val_accuracy: 0.6864
Epoch 17/30
68/68 [==============================] - 0s 5ms/step - loss: 0.4573 - accuracy: 0.8059 - val_loss: 0.6082 - val_accuracy: 0.7219
Epoch 18/30
68/68 [==============================] - 0s 5ms/step - loss: 0.4577 - accuracy: 0.7941 - val_loss: 0.5885 - val_accuracy: 0.7041
Epoch 19/30
68/68 [==============================] - 0s 5ms/step - loss: 0.4430 - accuracy: 0.8133 - val_loss: 0.5962 - val_accuracy: 0.7041
Epoch 20/30
68/68 [==============================] - 0s 5ms/step - loss: 0.4534 - accuracy: 0.7985 - val_loss: 0.6105 - val_accuracy: 0.6746
Epoch 21/30
68/68 [==============================] - 0s 5ms/step - loss: 0.4773 - accuracy: 0.8044 - val_loss: 0.5859 - val_accuracy: 0.6982
Epoch 22/30
68/68 [==============================] - 0s 5ms/step - loss: 0.4841 - accuracy: 0.8030 - val_loss: 0.5551 - val_accuracy: 0.7337
Epoch 23/30
68/68 [==============================] - 0s 5ms/step - loss: 0.4424 - accuracy: 0.7911 - val_loss: 0.5624 - val_accuracy: 0.7456
Epoch 24/30
68/68 [==============================] - 0s 5ms/step - loss: 0.4627 - accuracy: 0.7985 - val_loss: 0.5814 - val_accuracy: 0.7160
Epoch 25/30
68/68 [==============================] - 0s 5ms/step - loss: 0.4340 - accuracy: 0.8104 - val_loss: 0.6077 - val_accuracy: 0.7278
Epoch 26/30
68/68 [==============================] - 0s 5ms/step - loss: 0.4227 - accuracy: 0.8222 - val_loss: 0.5668 - val_accuracy: 0.7160
Epoch 27/30
68/68 [==============================] - 0s 5ms/step - loss: 0.4263 - accuracy: 0.8252 - val_loss: 0.6024 - val_accuracy: 0.6982
Epoch 28/30
68/68 [==============================] - 0s 5ms/step - loss: 0.4374 - accuracy: 0.8178 - val_loss: 0.6668 - val_accuracy: 0.6272
Epoch 29/30
68/68 [==============================] - 0s 5ms/step - loss: 0.4330 - accuracy: 0.8193 - val_loss: 0.6291 - val_accuracy: 0.6746
Epoch 30/30
68/68 [==============================] - 0s 5ms/step - loss: 0.4404 - accuracy: 0.8044 - val_loss: 0.5737 - val_accuracy: 0.7278
7/7 [==============================] - 0s 2ms/step - loss: 0.3957 - accuracy: 0.8458
Training mean accuracy: 0.7591111143430074
Validation mean accuracy: 0.6842209060986837
Testing mean accuracy: 0.8457711338996887
```

Results for epoch = 70

```
Epoch 1/60
69/69 [==============================] - 0s 6ms/step - loss: 1.0506 - accuracy: 0.6003 - val_loss: 0.5736 - val_accuracy: 0.7457
Epoch 2/60
69/69 [==============================] - 1s 9ms/step - loss: 0.6714 - accuracy: 0.6672 - val_loss: 0.6484 - val_accuracy: 0.6012
Epoch 3/60
69/69 [==============================] - 0s 5ms/step - loss: 0.6421 - accuracy: 0.6715 - val_loss: 0.6014 - val_accuracy: 0.6705
Epoch 4/60
69/69 [==============================] - 0s 5ms/step - loss: 0.6099 - accuracy: 0.6860 - val_loss: 0.6271 - val_accuracy: 0.6647
Epoch 5/60
69/69 [==============================] - 0s 5ms/step - loss: 0.5971 - accuracy: 0.7006 - val_loss: 0.6362 - val_accuracy: 0.6821
Epoch 6/60
69/69 [==============================] - 0s 5ms/step - loss: 0.5898 - accuracy: 0.6875 - val_loss: 0.6296 - val_accuracy: 0.6705
Epoch 7/60
69/69 [==============================] - 0s 5ms/step - loss: 0.5647 - accuracy: 0.7311 - val_loss: 0.5881 - val_accuracy: 0.6590
Epoch 8/60
69/69 [==============================] - 0s 5ms/step - loss: 0.5514 - accuracy: 0.7384 - val_loss: 0.6025 - val_accuracy: 0.6821
Epoch 9/60
69/69 [==============================] - 0s 5ms/step - loss: 0.5665 - accuracy: 0.7209 - val_loss: 0.6532 - val_accuracy: 0.6763
Epoch 10/60
69/69 [==============================] - 0s 5ms/step - loss: 0.5460 - accuracy: 0.7602 - val_loss: 0.6955 - val_accuracy: 0.6474
Epoch 11/60
69/69 [==============================] - 0s 5ms/step - loss: 0.5284 - accuracy: 0.7820 - val_loss: 0.6943 - val_accuracy: 0.6763
Epoch 12/60
69/69 [==============================] - 0s 5ms/step - loss: 0.5309 - accuracy: 0.7471 - val_loss: 0.6489 - val_accuracy: 0.6936
Epoch 13/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4989 - accuracy: 0.7733 - val_loss: 0.6891 - val_accuracy: 0.6821
Epoch 14/60
69/69 [==============================] - 0s 5ms/step - loss: 0.5079 - accuracy: 0.7907 - val_loss: 0.6059 - val_accuracy: 0.7168
Epoch 15/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4845 - accuracy: 0.7863 - val_loss: 0.7182 - val_accuracy: 0.6936
Epoch 16/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4705 - accuracy: 0.8096 - val_loss: 0.6361 - val_accuracy: 0.6936
Epoch 17/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4578 - accuracy: 0.8009 - val_loss: 0.7865 - val_accuracy: 0.6763
Epoch 18/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4529 - accuracy: 0.8096 - val_loss: 0.6375 - val_accuracy: 0.6994
Epoch 19/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4612 - accuracy: 0.8052 - val_loss: 0.5839 - val_accuracy: 0.7225
Epoch 20/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4756 - accuracy: 0.7892 - val_loss: 0.7107 - val_accuracy: 0.6821
Epoch 21/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4480 - accuracy: 0.8096 - val_loss: 0.7297 - val_accuracy: 0.6763
Epoch 22/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4482 - accuracy: 0.8140 - val_loss: 0.6738 - val_accuracy: 0.6936
Epoch 23/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4175 - accuracy: 0.8183 - val_loss: 0.5703 - val_accuracy: 0.7052
Epoch 24/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4307 - accuracy: 0.8227 - val_loss: 0.6856 - val_accuracy: 0.6879
Epoch 25/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4134 - accuracy: 0.8299 - val_loss: 0.6346 - val_accuracy: 0.7399
Epoch 26/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4325 - accuracy: 0.8169 - val_loss: 0.7804 - val_accuracy: 0.6705
Epoch 27/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4554 - accuracy: 0.8169 - val_loss: 0.5650 - val_accuracy: 0.7052
Epoch 28/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4174 - accuracy: 0.8183 - val_loss: 0.7080 - val_accuracy: 0.6647
Epoch 29/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4139 - accuracy: 0.8241 - val_loss: 0.6613 - val_accuracy: 0.6994
Epoch 30/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4355 - accuracy: 0.8256 - val_loss: 0.6409 - val_accuracy: 0.7052
Epoch 31/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4072 - accuracy: 0.8328 - val_loss: 0.6837 - val_accuracy: 0.6705
Epoch 32/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4309 - accuracy: 0.8183 - val_loss: 0.5877 - val_accuracy: 0.7283
Epoch 33/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4125 - accuracy: 0.8328 - val_loss: 0.6703 - val_accuracy: 0.6879
Epoch 34/60
69/69 [==============================] - 0s 5ms/step - loss: 0.3965 - accuracy: 0.8358 - val_loss: 0.6474 - val_accuracy: 0.7225
Epoch 35/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4091 - accuracy: 0.8343 - val_loss: 0.8069 - val_accuracy: 0.6705
Epoch 36/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4241 - accuracy: 0.8314 - val_loss: 0.6436 - val_accuracy: 0.6763
Epoch 37/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4129 - accuracy: 0.8285 - val_loss: 0.7138 - val_accuracy: 0.6705
Epoch 38/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4252 - accuracy: 0.8096 - val_loss: 0.6515 - val_accuracy: 0.7052
Epoch 39/60
69/69 [==============================] - 0s 5ms/step - loss: 0.3904 - accuracy: 0.8314 - val_loss: 0.5863 - val_accuracy: 0.7283
Epoch 40/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4193 - accuracy: 0.8110 - val_loss: 0.6711 - val_accuracy: 0.7110
Epoch 41/60
69/69 [==============================] - 0s 5ms/step - loss: 0.3930 - accuracy: 0.8372 - val_loss: 0.7438 - val_accuracy: 0.6705
Epoch 42/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4033 - accuracy: 0.8285 - val_loss: 0.7327 - val_accuracy: 0.6994
Epoch 43/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4001 - accuracy: 0.8387 - val_loss: 0.7548 - val_accuracy: 0.6994
Epoch 44/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4275 - accuracy: 0.8125 - val_loss: 0.6461 - val_accuracy: 0.6936
Epoch 45/60
69/69 [==============================] - 0s 5ms/step - loss: 0.3947 - accuracy: 0.8387 - val_loss: 0.6665 - val_accuracy: 0.7052
```

```
Epoch 46/60
69/69 [==============================] - 0s 5ms/step - loss: 0.3946 - accuracy: 0.8328 - val_loss: 0.6688 - val_accuracy: 0.6936
Epoch 47/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4103 - accuracy: 0.8198 - val_loss: 0.7998 - val_accuracy: 0.6590
Epoch 48/60
69/69 [==============================] - 0s 5ms/step - loss: 0.3996 - accuracy: 0.8285 - val_loss: 0.6350 - val_accuracy: 0.7168
Epoch 49/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4034 - accuracy: 0.8198 - val_loss: 0.6408 - val_accuracy: 0.7225
Epoch 50/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4022 - accuracy: 0.8343 - val_loss: 0.6939 - val_accuracy: 0.6763
Epoch 51/60
69/69 [==============================] - 0s 5ms/step - loss: 0.3865 - accuracy: 0.8343 - val_loss: 0.7083 - val_accuracy: 0.6590
Epoch 52/60
69/69 [==============================] - 0s 5ms/step - loss: 0.3849 - accuracy: 0.8328 - val_loss: 0.6751 - val_accuracy: 0.6936
Epoch 53/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4101 - accuracy: 0.8314 - val_loss: 0.8581 - val_accuracy: 0.6936
Epoch 54/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4415 - accuracy: 0.8038 - val_loss: 0.6830 - val_accuracy: 0.6821
Epoch 55/60
69/69 [==============================] - 0s 5ms/step - loss: 0.4271 - accuracy: 0.8285 - val_loss: 0.6909 - val_accuracy: 0.7225
Epoch 56/60
69/69 [==============================] - 0s 5ms/step - loss: 0.3876 - accuracy: 0.8372 - val_loss: 0.7384 - val_accuracy: 0.6994
Epoch 57/60
69/69 [==============================] - 0s 5ms/step - loss: 0.3798 - accuracy: 0.8503 - val_loss: 0.6827 - val_accuracy: 0.7052
Epoch 58/60
69/69 [==============================] - 0s 5ms/step - loss: 0.3951 - accuracy: 0.8358 - val_loss: 0.6476 - val_accuracy: 0.7110
Epoch 59/60
69/69 [==============================] - 0s 5ms/step - loss: 0.3759 - accuracy: 0.8459 - val_loss: 0.7689 - val_accuracy: 0.6647
Epoch 60/60
69/69 [==============================] - 0s 5ms/step - loss: 0.3752 - accuracy: 0.8401 - val_loss: 0.7386 - val_accuracy: 0.6879
6/6 [==============================] - 0s 2ms/step - loss: 0.6607 - accuracy: 0.7857
Training mean accuracy: 0.7991763522227605
Validation mean accuracy: 0.6901734113693238
Testing mean accuracy: 0.7857142686843872
```

Testing the model for Rose and Jack Survival

name	pclass	sex	age	sibsp	parch	fare	embarked
Rose	1	0	20	1	0	100	0
Jack	3	1	23	1	0	5	0

```
#Predicting chances of survival for 'Rose' using tf.keras.Sequential model
Predict_Rose_survival=[[1,0,20,1,0,100,0]]
predictions = model.predict(Predict_Rose_survival)
print('predictions',predictions)
```

```
predictions [[0.1630439 0.8369561]]
```

```
#Predicting chances of survival for 'Jack' using tf.keras.Sequential model
Predict_Jack_survival=[[3,1,23,1,0,5,0]]
predictions = model.predict(Predict_Jack_survival)
print('predictions',predictions)
```

```
predictions [[0.8870739  0.11292607]]
```

The sequential model can be further used to predict the survival chances of Jack and Rose. As per the prediction results, Rose survives(83.69% chances) and Jack does not(11.2% chances). The table represents the inputs for Rose and Jack that was used for prediction.

Step 7: TensorFlow for predicting survival of the passenger using the TensorFlow's high level API with Estimators.

The data is preprocessed and further divided into training and test datasets to measure accuracy of the models for both the datasets. The models used are Linear classifier estimator and Boosted tree estimator.

Creating the code for the model

```
[24] # Predict using Tensflow estimator for Boostedtrees
     from IPython.display import clear_output

     #Read the titanic dataset
     titanic = pd.read_csv('titanic.csv')

     #drop unwanted attributes
     titanic = drop_unused_columns(titanic)

     #drop rows with 'na'
     titanic = drop_rows_with_na(titanic)

     # reorder the columns
     titanic = reorder_columns(titanic)

     #Divide the dataset for 80% training and 20% testing
     indices = np.random.rand(len(titanic)) <= 0.8

     #process of creating training and testing dataset
     train_Xvars = titanic[indices]
     train_Xvars = train_Xvars.reset_index(drop=True)
     train_Xvars = train_Xvars.astype({'pclass': 'int32','sibsp': 'int32','parch': 'int32'})

     test_Xvars = titanic[~indices]
     test_Xvars = test_Xvars.reset_index(drop=True)
     test_Xvars = test_Xvars.astype({'pclass': 'int32','sibsp': 'int32', 'parch': 'int32'})

     train_Yvars = train_Xvars.pop('survived')
     test_Yvars = test_Xvars.pop('survived')
```

```
   #Feature extraction - using only what is required and manipulating the data
   categorical_features = ['pclass', 'sex', 'sibsp', 'parch','embarked']
   numeric_features = ['age', 'fare']

   features = []
   for feature in categorical_features:
     vocabulary = train_Xvars[feature].unique()
     features.append(tf.feature_column.indicator_column(
         tf.feature_column.categorical_column_with_vocabulary_list(feature,
                                                   vocabulary)))

   for feature in numeric_features:
     features.append(tf.feature_column.numeric_column(feature,
                                             dtype=tf.float32))

   #Processing the dataset using methods, shuffle, repeat and batch
   def process_dataset(X, Y, number_of_epochs=None, shuffle=True):
     def process():
       dataset = tf.data.Dataset.from_tensor_slices((dict(X), Y))
       if shuffle:
         dataset = dataset.shuffle(len(train_Yvars))
       dataset = dataset.repeat(number_of_epochs)
       dataset = dataset.batch(len(train_Yvars))
       return dataset
     return process

   #obtaining training and testing data
   training = process_dataset(train_Xvars, train_Yvars)
   testing = process_dataset(test_Xvars, test_Yvars,number_of_epochs=1, shuffle=False)
```

```
   #Using Linear Classifier estimator from TensorFlow estimator API
   linear_estimator = tf.estimator.LinearClassifier(features)
   linear_estimator.train(training, max_steps=10)

   #evaluate the results for linear estimator
   linear_result = linear_estimator.evaluate(testing)

   #Using Boosted Tree Classifier estimator from TensorFlow estimator API
   params = {'n_trees': 50, 'max_depth': 3,'n_batches_per_layer': 1,'center_bias': True}

   boosted_trees_estimator = tf.estimator.BoostedTreesClassifier(features,
                                           **params)
   boosted_trees_estimator.train(training, max_steps=100)

   #evaluate the results for boosted tree estimator
   boosted_tree_result = boosted_trees_estimator.evaluate(testing)
   clear_output()
```

Results for Linear and Boosted Tree Estimators

```
[25] #Printing the results
     print('Linear estimator results')
     print(pd.Series(linear_result))
     print('---------------------------------------')
     print('Boosted Tree estimator results')
     print(pd.Series(boosted_tree_result))

     #append the accuracy to the object for furture visualization
     all_accuracies['Linear estimator']=linear_result['accuracy']
     all_accuracies['Boosted tree estimator']=boosted_tree_result['accuracy']
```

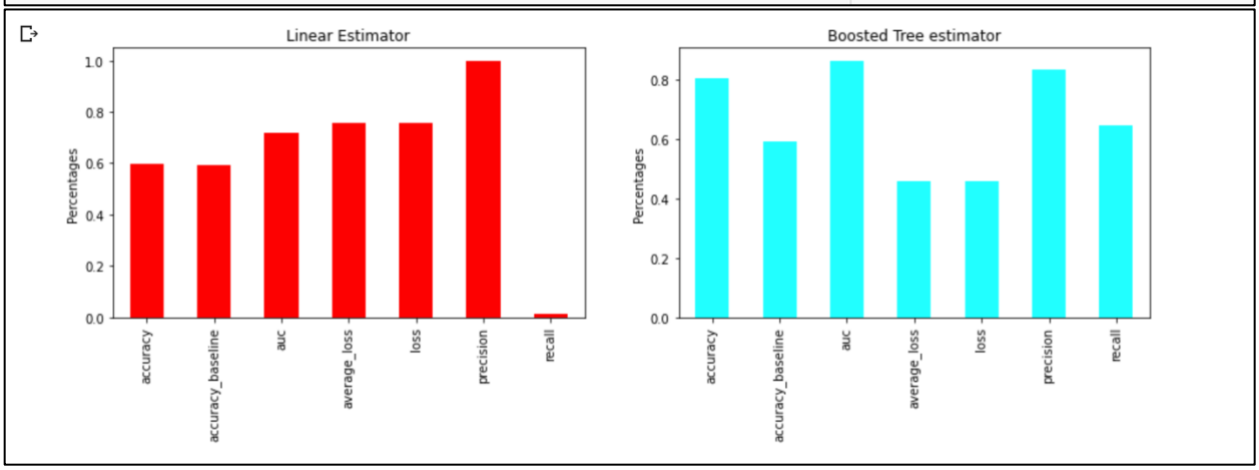
```
Linear estimator results
accuracy                0.597938
accuracy_baseline       0.592784
auc                     0.719262
auc_precision_recall    0.623428
average_loss            0.758718
label/mean              0.407216
loss                    0.758718
precision               1.000000
prediction/mean         0.178567
recall                  0.012658
global_step            10.000000
dtype: float64
```

```
Boosted Tree estimator results
accuracy                0.804124
accuracy_baseline       0.592784
auc                     0.865603
auc_precision_recall    0.800464
average_loss            0.457556
label/mean              0.407216
loss                    0.457556
precision               0.836066
prediction/mean         0.405058
recall                  0.645570
global_step           100.000000
dtype: float64
```

Boosted tree estimator outperforms in all the performance metrics compared to Linear estimator
The tabular results obtained above for both Linear estimator and Boosted trees estimator are also plotted in a form of bar plot to provide clear insights for comparison.

Visualization of the results for Linear estimator and Boosted trees estimator

```
[ ] #Plotting linear regression estimator vs Boosted Tree estimator results
    fig, matplotlib_Axes = plt.subplots(1, 2, figsize = (15, 4))

    linear_result = pd.Series(linear_result)
    linear_result=linear_result.drop(['global_step','auc_precision_recall','label/mean','prediction/mean'])
    linear_result.plot.bar(color = "#FF0000", ax = matplotlib_Axes[0])
    matplotlib_Axes[0].set_title("Linear Estimator")
    matplotlib_Axes[0].set_ylabel("Percentages")

    boosted_tree_result = pd.Series(boosted_tree_result)
    boosted_tree_result = boosted_tree_result.drop(['global_step','auc_precision_recall','label/mean','prediction/mean'])
    boosted_tree_result.plot.bar(color = "#00FFFF", ax = matplotlib_Axes[1])
    matplotlib_Axes[1].set_title("Boosted Tree estimator")
    matplotlib_Axes[1].set_ylabel("Percentages")
    plt.show()
```

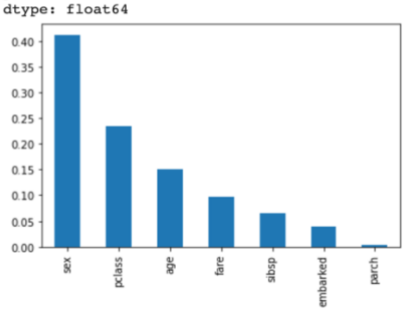


Boosted tree estimator outperforms in all the performance metrics compared to Linear estimator

To evaluate feature importances for Boosted tree classifier

```
[ ] #Feature importances of Boosted Trees Classifier
    feature_importances = boosted_trees_estimator.experimental_feature_importances(normalize=True)
    feature_importances = pd.Series(feature_importances)
    feature_importances.plot.bar()
    feature_importances
```

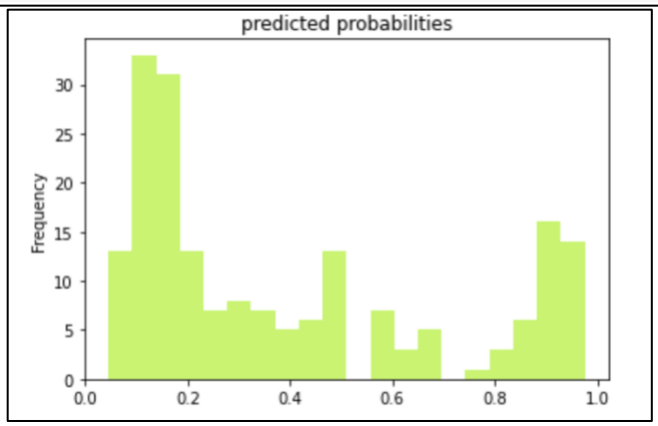
```
sex         0.412189
pclass      0.234229
age         0.149804
fare        0.097714
sibsp       0.064269
embarked    0.038617
parch       0.003177
dtype: float64
```



Boosted tree estimator importances are considered because it has higher accuracy. Sex, pclass and age have a significant influence on passenger being classified as survived and not survived.

Visualization of the predicted probabilities on survival for the testing dataset

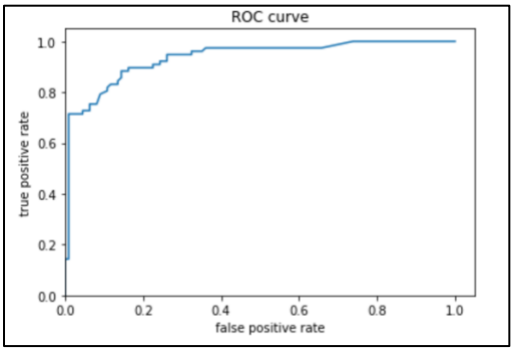
```
[ ] #Prediction based on testing dataset and plotting the predicted probabilities
    predictions = list(boosted_trees_estimator.predict(testing))
    probabilities = pd.Series([pred['probabilities'][1] for pred in predictions])
    probabilities.plot(kind='hist', bins=20, title='predicted probabilities',color='#C9F371')
    plt.show()
```



The predicted probabilities for most passengers for the testing data are on the lower scale. Very few people have higher chances of survival.

The Receiver Operator Characteristics(ROC) curve visualizes performance

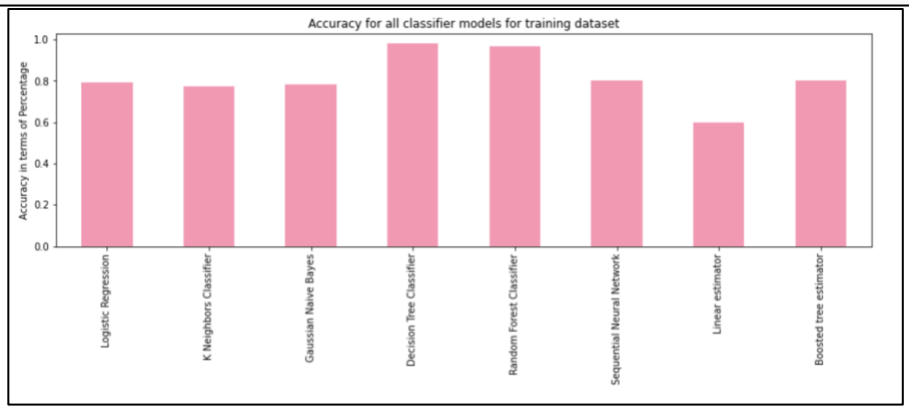
```
[29] #ROC curve for Boosted Tree Classifier to evaluate performance
     from sklearn.metrics import roc_curve
     fpr, tpr, _ = roc_curve(test_Yvars, probabilities)
     plt.plot(fpr, tpr)
     plt.title('ROC curve')
     plt.xlabel('false positive rate')
     plt.ylabel('true positive rate')
     plt.xlim(0,)
     plt.ylim(0,)
     plt.show()
```



The Receiver Operator Characteristics(ROC) curve, helps to understand the performance of the model. Closer the curve is to the top left corner better is the performance of the classifier model. In this case the accuracy of the model is 80% which is why the curve is closer to the left corner.

Visualization of all accuracies obtained from the models used above

```
#summarizing all the accuracies obtained from the models above for better comparisons
fig, matplotlib_Axes = plt.subplots(1, figsize = (15, 4))
all_accuracies = pd.Series(all_accuracies)
all_accuracies.plot.bar(color = "#f199B3")
matplotlib_Axes.set_title('Accuracy for all classifier models for training dataset')
matplotlib_Axes.set_ylabel('Accuracy in terms of Percentage')
plt.show()
```



Decision tress and Random Forest Classifier have higher accuracy results. Logistic regression, K nearest Neighbors , Gaussian Naïve Bayes, Boosted tress estimator and Sequential neural network models have almost the same accuracy results and the lowest value for accuracy is for a Linear estimator model.