EDA To Prediction (DieTanic)

Abdullah Hakan Şişik , Çağatay Özdemir

The code of the coder contains the following parts :

I. Part 1: Exploratory Data Analysis(EDA):

a) Analysis of the features.
b) Finding any relations or trends considering multiple features.

II. Part 2: Feature Engineering and Data Cleaning:

a) Adding any few features.
b) Removing redundant features.
c) Converting features into suitable form for modeling.

III. Part 3: Predictive Modeling

a) Running Basic Algorithms.
b) Cross Validation.
c) Ensembling.
d) Important Features Extraction.

Part 1: Exploratory Data Analysis(EDA):

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
plt.style.use('fivethirtyeight')
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline
```

Fig. 1.

To start coding , some of the Python libraries should be imported. The coder imported the libraries “numpy” , “pandas” , “matplotlib.pyplot” , “seaborn” and lastly “warnings” .

```
data=pd.read_csv('../input/train.csv')
```

Fig. 2.

After that , the coder imports the train data by assigning it to a dataframe named “data” .The panda function “read_csv” reads the data from the indicated csv files and assigns to corresponding dataframes.

```
data.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

Fig. 3.

The .head function prints the first 5 element of the “data” dataframe as a table.

```
data.isnull().sum() #checking for total null values
```

Fig. 4.

Later on , the coder decides to check the missing datas for each column. .isnull returns a boolean value that indicates whether an expression contains no valid data (Null). isnull returns true if expression is null; otherwise, isnull returns false.The sum function sums the number of the missing values for each column. The result is given below.

```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

As it can be seen from the table , the “Age” column has 177 missing values. The “Cabin” column has 687 missing values and the “Embarked” column has 2 missing values.

Fig. 5.

Now , the coder creates 2 typed of chart ; 1 bar chart and 1 pie chart.Now lets inspect the code the coder wrote.

```
f,ax=plt.subplots(1,2,figsize=(18,8))
data['Survived'].value_counts().plot.pie(explode=[0,0.1],autopct='%1.1f%%',ax=ax[0],shadow=True)
ax[0].set_title('Survived')
ax[0].set_ylabel('')
sns.countplot('Survived',data=data,ax=ax[1])
ax[1].set_title('Survived')
plt.show()
```

Fig. 6.

First line of the code is used to create 2 subplots. Figsize indicated that the each of the subplots will have the size 18 inches in width and 8 inches in length.
Second line of the code produces a pie chart labeled as ax[0]. It displays the count of occurrences for each distinct value within the "Survived" column of the "data" DataFrame. By specifying explode=[0, 0.1], the slice representing "Survived" extends slightly outward to enhance its visibility. The “autopct='%1.1f%%' ” adds percentage labels to each slice. The “shadow=True” adds a shadow effect to the pie chart.
“ ax[0].set_title('Survived') ” sets the title of the first subplot to “Survived”.
“ ax[0].set_ylabel('') ” sets the y-axis label to empty.
“sns.countplot('Survived', data=data, ax=ax[1])” creates a count chart on the second subplot names as ax[1]. Count plot contains number the occurrences of unique values in the “Survived” column and plots a bar chart based on the occurances.
“ax[1].set_title('Survived')” , this code sets the title of the ax[1] to “Survived”.
“plt.show()” displays the entire figure containing both subplots.

The result is:

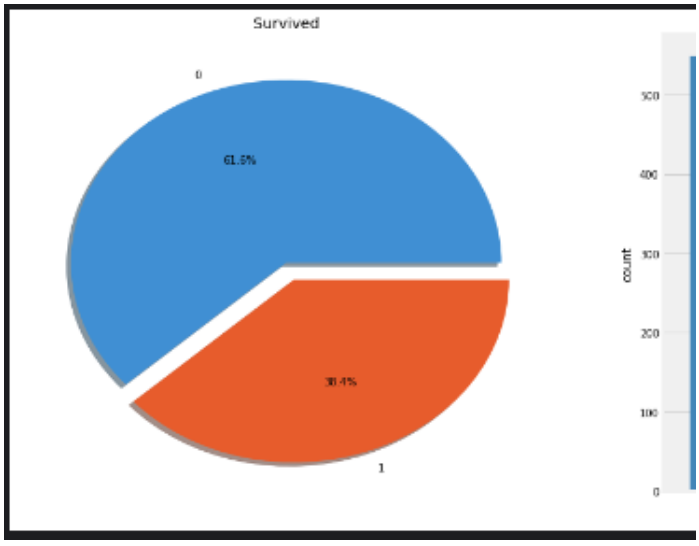


Fig. 7

It is evident that the accident did not result in the survival of many passengers.Out of 891 passengers in the training set, only around 350 were survived, which means only 38.4% of the total training set survived the crash. Further insights need to be extracted from the data to determine which categories of passengers survived and which did not.
The survival rate will be examined by utilizing various features of the dataset, such as Sex, Port Of Embarkation, Age, etc.First, the different types of features need to be understood.

Later on , the coder inspects the types of features.

I. Types Of Features

a) Categorical Features

A categorical variable is defined as having two or more categories, and each value in that feature can be categorized accordingly. For instance, gender is a categorical variable with two categories (male and female). These variables cannot be sorted or assigned any particular order. They are also referred to as Nominal Variables.Categorical features in the dataset include Sex and Embarked.

b) Ordinal Features

An ordinal variable shares similarities with categorical values, but the distinction lies in the ability to establish relative ordering or sorting between the values. For instance, if a feature like Height includes values such as Tall, Medium, Short, then Height is considered an ordinal variable. Relative sorting within the variable is possible.Ordinal features in the dataset include PClass.

c) Continous Feature:

Continuous features are characterized by their ability to assume values between any two points or within the range defined by the minimum and maximum values in the feature column.The dataset includes the continuous feature Age.

2. Analysing the Features

a) Sex Feature

```
data.groupby(['Sex','Survived'])['Survived'].cour
```

Fig. 8

This code groups the “Survived” column in “data” dataframe by two variables. One of the groups is Sex , the other is Survived.After grouping , it counts the occurance of the each possibility and prints a table based on this.

```
Sex     Survived
female  0           81
        1          233
male    0          468
        1          109
Name: Survived, dtype: int64
```

Fig. 9

Survived = 0 indicates they are dead , Survived=1 indicates they are alive .The last number at the last in each row indicates the count of people in that group.

```
f,ax=plt.subplots(1,2,figsize=(18,8))
data[['Sex','Survived']].groupby(['Sex']).mean().plot.bar(ax=ax[0])
ax[0].set_title('Survived vs Sex')
sns.countplot('Sex',hue='Survived',data=data,ax=ax[1])
ax[1].set_title('Sex:Survived vs Dead')
plt.show()
```

Fig. 10

"f,ax=plt.subplots(1,2,figsize=(18,8))" tells that there will be 2 plots .it's sizes will be 18 units wide and 8 units length for each plot.

data[['Sex','Survived']].groupby(['Sex']).mean().plot.bar(ax=ax[0]) : For the columns of Sex and survived in "data" dataframe , this part of the code first groups the data with respect to their genders as "males" and "females".Then it calculates the mean of the survival rates for each group.After that , it plots the result as a bar chart.
ax[0].set_title('Survived vs Sex') : This code sets the title of the first bar plot as "Survived vs Sex".
sns.countplot('Sex',hue='Survived',data=data,ax=ax[1]) :This code colors the bars differently by looking at the distinction of whether they survive or not for the second plot.
ax[1].set_title('Sex:Survived vs Dead') : this code sets the title of the second plot as "Sex:Survived vs Dead".
plt.show() : plots and shows the bar charts.

Despite the significantly higher number of men aboard the ship compared to women, the survival count for women is nearly twice that of men. A survival rate of around 75% is observed for women, whereas for men, it hovers around 18-19%. This feature appears to be highly significant for modeling purposes. However, it's necessary to examine other features to determine if any of them might be even more advantageous.
All in all ; compared to men, women have a higher likelihood of survival.

b) Pclass Feature

```
pd.crosstab(data.Pclass,data.Survived,margins=True).style.background_gradient(cmap='summer_r')
```

Fig. 11

This line of code generates a styled contingency table that provides insights into the relationship between Pclass and Survived, with highlighted color gradients aiding in visual analysis.The result of the code is :

Survived	0	1	All
Pclass			
1	80	136	216
2	97	87	184
3	372	119	491
All	549	342	891

Fig. 12

```
f,ax=plt.subplots(1,2,figsize=(18,8))
data['Pclass'].value_counts().plot.bar(color=['#CD7F32','#FFDF00','#D3D3D3'],ax=ax[0])
ax[0].set_title('Number Of Passengers By Pclass')
ax[0].set_ylabel('Count')
sns.countplot('Pclass',hue='Survived',data=data,ax=ax[1])
ax[1].set_title('Pclass:Survived vs Dead')
plt.show()
```

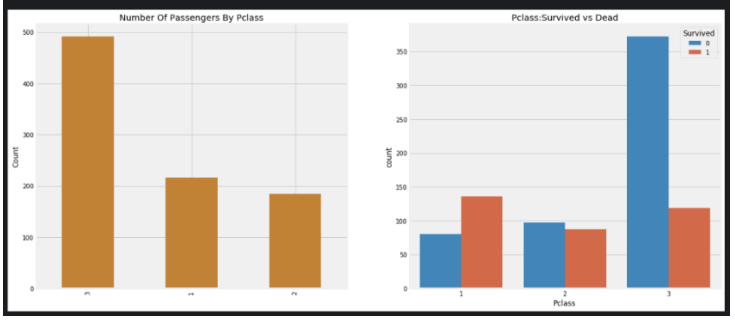


Fig. 13

The way the functions work was explained earlier so I won't explain it in detail again. This code creates two plots. First plot is a bar chart that shows the count of passangers in each passenger class.Second plot shows the number of survivors and deaths in each passenger class.
Looking at the graphs , the coder concludes the followings : Pclass 1 had the priority in rescue operations. Despite the significantly larger number of passengers in Pclass 3, the survival rate from this class is notably low, approximately 25%. In contrast, for Pclass 1, the survival rate is around 63%, and for Pclass 2, it is approximately 48%. This suggests that wealth and social status play a crucial role in survival chances.

```
pd.crosstab([data.Sex,data.Survived],data.Pclass,margins=True).style.background_gradient(cmap='autumn_r')
```

Fig. 14

This line of code generates a styled contingency table that provides insights into the relationship between Sex and Survived, with highlighted color gradients aiding in visual analysis.The result of the code is :

	Pclass	1	2	3	All
Sex	Survived				
female	0	3	6	72	81
	1	91	70	72	233
male	0	77	91	300	468
	1	45	17	47	109
All		216	184	491	891

Fig. 15

It can be seen that there is a high survival rate in the cross-section of Pclass 1 – Female and Pclass 2 – Female .Also , it is clear that the females was given priority in rescues.The female survival rate is a lot higher compared to male survival rate for the columns.

```
sns.factorplot('Pclass','Survived',hue='Sex',data=data)
plt.show()
```

Fig. 16

This code creates a factor chart by displaying the relationship between Pclass and Survived based on the distinction of Sex.The result of the code is :

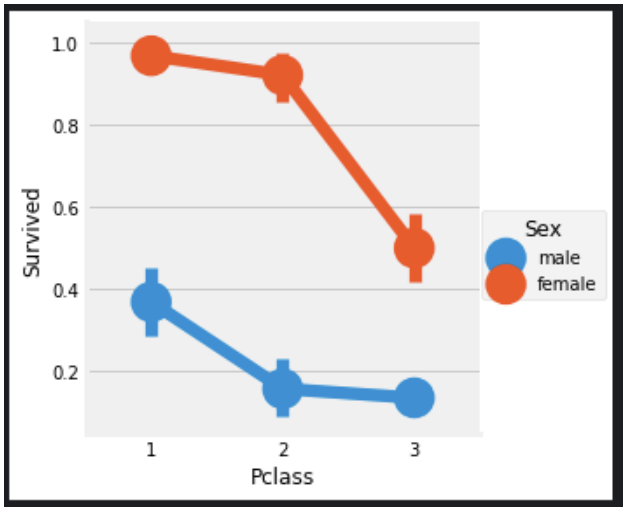


Fig. 17

FactorPlot is chosen for its ability to easily distinguish between categorical values.Observing the CrossTab and FactorPlot, it's clear that women from Pclass1 had a survival rate of around 95-96%, with only 3 out of 94 women from that class not surviving.The data indicates a consistent prioritization of women during rescue operations, irrespective of their passenger class. Even men from Pclass1 show a remarkably low survival rate.This analysis underscores the importance of passenger class as a defining feature.

A noticeable trend indicates that being a first class passenger increases your chances of survival. Conversely, the survival rate for Pclass=3 is notably low. Specifically for women, the probability of survival is almost one for those in Pclass=1 and remains high for those in Pclass=2.

c) Age Feature

```
print('Oldest Passenger was of:',data['Age'].max(),'Years')
print('Youngest Passenger was of:',data['Age'].min(),'Years')
print('Average Age on the ship:',data['Age'].mean(),'Years')
```

```
Oldest Passenger was of: 80.0 Years
Youngest Passenger was of: 0.42 Years
Average Age on the ship: 29.69911764705882 Years
```

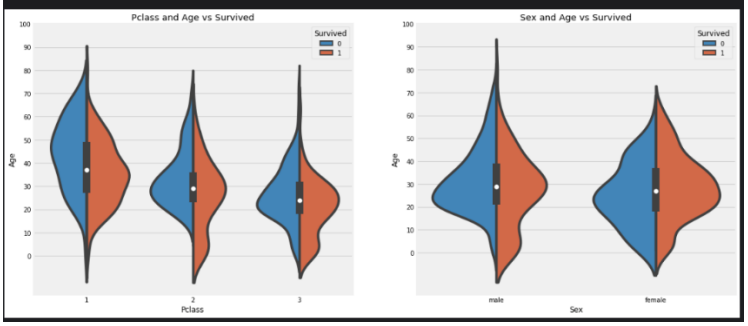
Fig. 18

First line prints the maximum value in the column of age in “data” dataframe.

Second line prints the minimum value in the column of age in “data” dataframe.

Third line prints the mean value in the column of age in “data” dataframe.

```
f,ax=plt.subplots(1,2,figsize=(18,8))
sns.violinplot("Pclass","Age", hue="Survived", data=data,split=True,ax=ax[0])
ax[0].set_title('Pclass and Age vs Survived')
ax[0].set_yticks(range(0,110,10))
sns.violinplot("Sex","Age", hue="Survived", data=data,split=True,ax=ax[1])
ax[1].set_title('Sex and Age vs Survived')
ax[1].set_yticks(range(0,110,10))
plt.show()
```



This code creates two violin charts .The first chart displays the relationship between Pclass and Age based on the distinction of Survived. The second chart displays the relationship between Sex and Age based on the distinction of Survived.

Following observations are made by the coder based on these plots:

- There's a notable increase in the number of children across passenger classes, and the survival rate for passengers under 10 years old appears promising regardless of their passenger class.
- Passengers aged 20-50 from Pclass1 exhibit high survival chances, particularly for women, indicating a significant advantage.
- For males, survival chances decrease with age, highlighting a concerning trend of declining survival rates as age increases.

At the next step , the coder decides to handle the null values in the “Age” column of the “data” dataframe. First , he thinks of replacing NaN values with the mean age but the distribution of the age is over a wide range so he decides to check initials in the names to decide.Specifically , he checks whether the name has “Mr.” or “Mrs.” initials.

To do this , following code is used:

```
data['Initial']=0
for i in data:
    data['Initial']=data.Name.str.extract('([A-Za-z]+)\.') #lets extract the Salutations
```

Fig. 19

Firstly , a column in defined in “data” dataframe named “Initial”

Then, the regular expression [A-Za-z]+) is utilized. It is designed to identify strings located between A-Z or a-z and followed by a dot. Consequently, initials are successfully extracted from the names and assigned to Initial column.

```
pd.crosstab(data.Initial,data.Sex).T.style.background_gradient(cmap='summer_r') #Checking the Initials with the Sex
```

Initial	Capt	Col	Countess	Don	Dr	Jonkheer	Lady	Major	Master	Miss	Mlle	Mme	Mr	Mrs	Ms	Rev	Sir
Sex																	
female	0	0	1	0	1	0	1	0	0	182	2	1	0	125	1	0	0
male	1	2	0	1	6	1	0	2	40	0	0	0	517	0	0	6	1

Fig. 20

I explained how this type of code works more than once so I will skip that part. It can be seen that there are some mispelled initials in the table such as “Mlle” , “Mme”.The coder decides to correct them to make the analyze easier.

```
data['Initial'].replace(['Mlle','Mme','Ms','Dr','Major','Lady','Countess','Jonkheer','Col','Rev','Capt','Sir','Don'],['Miss','Miss','Mr','Mr','Mrs','Mrs','Other','Other','Other','Mr','Mr','Mr'],inplace=True)
```

Fig. 21

This line of code replaces the initials in the “Initial” column of the “data” dataframe. The first “[ ]” bracket inside the replace function indicates the current initials. The second “[ ]” bracket indicates the initials that will replace the current initials.

```
data.groupby('Initial')['Age'].mean() #lets check the average age by Initials

Initial
Master     4.574167
Miss      21.860000
Mr        32.739609
Mrs       35.981818
Other     45.888889
Name: Age, dtype: float64
```

Fig. 22

After that , “Initial” column is grouped by the factor “Age” and each group’s mean value is calculated as shown above.

```
## Assigning the NaN Values with the Ceil values of the mean ages
data.loc[(data.Age.isnull())&(data.Initial=='Mr'),'Age']=33
data.loc[(data.Age.isnull())&(data.Initial=='Mrs'),'Age']=36
data.loc[(data.Age.isnull())&(data.Initial=='Master'),'Age']=5
data.loc[(data.Age.isnull())&(data.Initial=='Miss'),'Age']=22
data.loc[(data.Age.isnull())&(data.Initial=='Other'),'Age']=46
```

Fig. 23

After calculating each of the Initial’s mean value , they are rounded to closest integer and assigned accordingly to replace the NaN values in the Age column.

```
data.Age.isnull().any() #So no null values left finally

False
```

Fig. 24

As it can be seen , there are no null value left in the Age column of the “data” dataframe. Later on , two new bar chart is plotted.

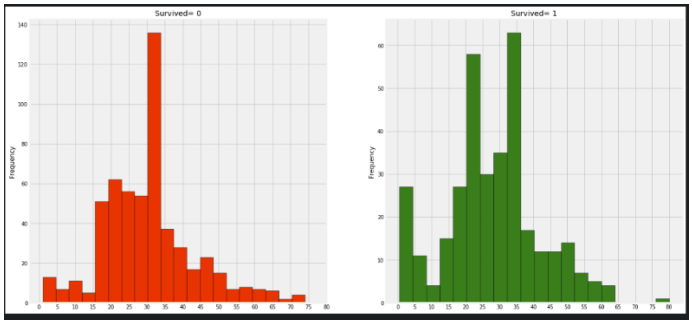


Fig. 25

The first chart shows the frequency of deaths based on age ranges. The second bar chart shows the rate of survival based on age ranges.

The following observations are made by the coder:

- A considerable number of toddlers (age < 5) were rescued, indicating adherence to the "Women and Child First Policy."
- The oldest passenger, aged 80 years, was among the survivors, suggesting a commitment to saving individuals regardless of age.
- The highest number of fatalities occurred within the age group of 30-40 years, highlighting a significant loss within this demographic.

```
sns.factorplot('Pclass','Survived',col='Initial',data=data)
plt.show()
```

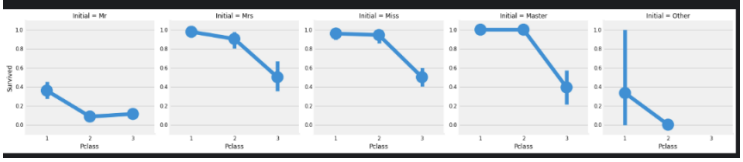


Fig. 26

This code creates a factor chart by displaying the relationship between Pclass and Survived based on the distinction of Initials.The result of the code is given above. To see the graphics clearly , the reference link can be used and the reference will be given at the end of the explanation.Looking the graphs , it is clear that women and childs are given priority in rescue for every Pclass.

Conclusion : Children under 5-10 years old exhibit a high likelihood of survival. However, a significant number of passengers aged between 15 to 35 years died.

d) Embarked Feature

```
pd.crosstab([data.Embarked,data.Pclass],[data.Sex,data.Survived],margins=True).style.background_gradient(cmap='summer_r')
```

Fig. 27

This line of code generates a styled contingency table that provides insights into the relationship between Embarked - Pclass and Survived - Sex, with highlighted color gradients aiding in visual analysis.The result of the code is:

	Sex	female		male		All
	Survived	0	1	0	1	
Embarked	Pclass					
C	1	1	42	25	17	85
	2	0	7	8	2	17
	3	8	15	33	10	66
Q	1	0	1	1	0	2
	2	0	2	1	0	3
	3	9	24	36	3	72
S	1	2	46	51	28	127
	2	6	61	82	15	164
	3	55	33	231	34	353
All		81	231	468	109	889

Fig. 28

```
sns.factorplot('Embarked','Survived',data=data)
fig=plt.gcf()
fig.set_size_inches(5,3)
plt.show()
```

Fig. 29

This code creates a factor plot with Embarked on x-axis and Survived on y-axis.Factor plot is sized to have 5 inches in width and 3 inches in length.The result is :

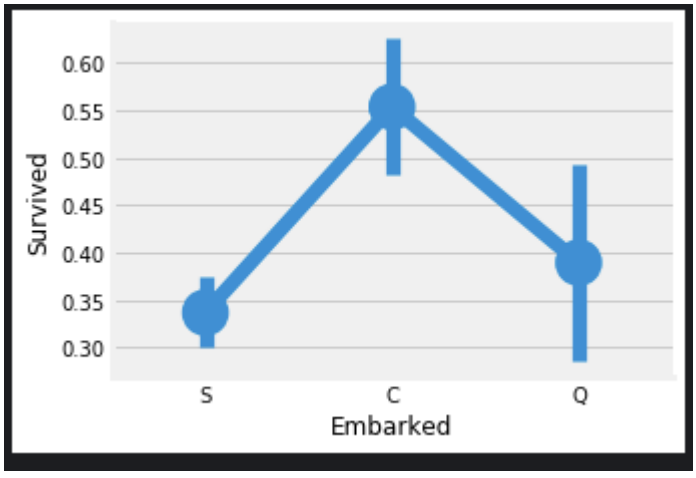


Fig. 30

Checking the graph , it is clear that the highest survival rate is on the passangers who embarked on port C and lowest survival rate is on the passangers who embarked on port S.

```
f,ax=plt.subplots(2,2,figsize=(20,15))
sns.countplot('Embarked',data=data,ax=ax[0,0])
ax[0,0].set_title('No. Of Passengers Boarded')
sns.countplot('Embarked',hue='Sex',data=data,ax=ax[0,1])
ax[0,1].set_title('Male-Female Split for Embarked')
sns.countplot('Embarked',hue='Survived',data=data,ax=ax[1,0])
ax[1,0].set_title('Embarked vs Survived')
sns.countplot('Embarked',hue='Pclass',data=data,ax=ax[1,1])
ax[1,1].set_title('Embarked vs Pclass')
plt.subplots_adjust(wspace=0.2,hspace=0.5)
plt.show()
```

Fig. 31

This code creates 4 bar charts.I explained how the creation of bar charts works already so I will just explain what this code creates.The first bar chart shows the number of passangers boarded the ship according to each port.The second bar chart seperates the number of passangers boarded in each port based on their genders.The third bar plot shows the survival and death counts based on different ports.Last and fourth bar chart shows the number of the people who embarked ship in each port based on the distinction of Pclass .

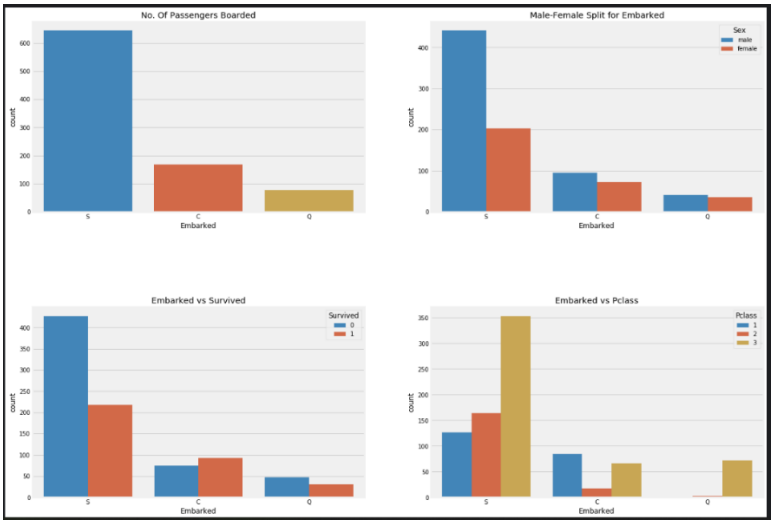


Fig. 32

The following observations are made by the coder:

- The highest number of passengers boarded from S, with the majority belonging to Pclass3.
- Passengers from C appear fortunate, with a significant proportion surviving. This could be attributed to the rescue of all Pclass1 and Pclass2 passengers.
- Embarkation at S seems to be associated with wealthy passengers. However, survival chances are low, largely due to the high mortality rate among Pclass3 passengers, around 81% of whom did not survive.
- Port Q predominantly accommodated passengers from Pclass3, with almost 95% of them originating from this class.

```
sns.factorplot('Pclass','Survived',hue='Sex',col='Embarked',data=data)
plt.show()
```

Fig. 33

Three new factor charts are created , displaying the relationship between Pclass and Survived based on the distinction of Sex for each of the Embarked.

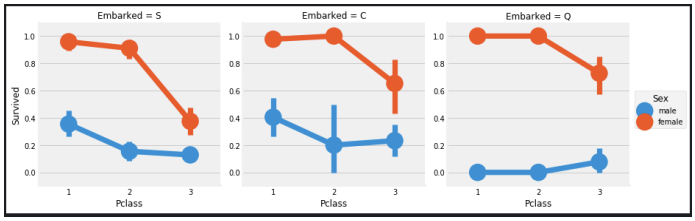


Fig. 34

Following observations are made :

- Survival chances for women in Pclass1 and Pclass2 are nearly 100%, regardless of their passenger class.
- Port S appears highly unfavorable for Pclass3 passengers, with both men and women experiencing low survival rates, suggesting a correlation with socioeconomic status.
- Port Q seems to be particularly unlucky for men, as nearly all passengers were from Pclass3, indicating a higher vulnerability among this demographic.

It can be seen that most of the passangers boarded from Port S , indicating mean value of the Embarked is “S” so port “S” is used for the NaN values in the Embarked Column. Following code fills the NaN values in the Embarked column with “S” . Then checks whether any NaN values left in the Embarked column.

```
data['Embarked'].fillna('S',inplace=True)
```

```
data.Embarked.isnull().any()# Finally No NaN values
```

```
False
```

Fig. 35

In conclusion , despite the majority of Pclass=1 passengers boarding at S, the survival rate at C appears to be higher. Additionally, all passengers at Q were from Pclass=3.

e) SibSip Feature

Sib represents Siblings. Sip stands for spouses.First of all , cross table is created ,showing the relation between Sibsp and Survived

```
pd.crosstab([data.SibSp],data.Survived).style.background_gradient(cmap='summer_r')
```

Fig. 36

Survived	0	1
SibSp		
0	398	210
1	97	112
2	15	13
3	12	4
4	15	3
5	5	0
8	7	0

Fig. 37

Then , following charts are created by the coder. The first chart is a bar chart.It represents the relation between the number of SibSip and Survived count.Second chart is a factor chart. It represents the relation between SibSip and Survived in a continous form.Since SibSp is a discrete value , I think using bar chart for this case is better.

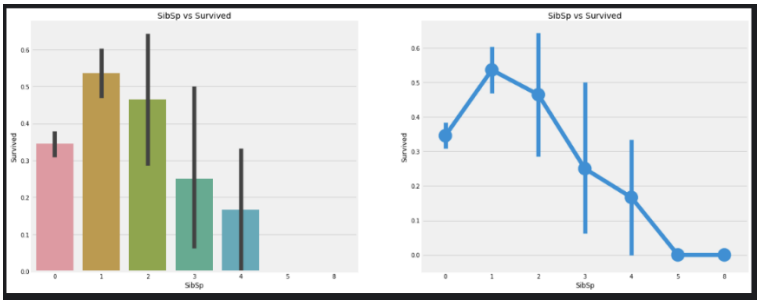


Fig. 38

```
bq·clossfsp(qsfs·sfpsb'qsfs·bcfsss)·sfxje·psckdlonuq_dlsqfsuf(cwsb=,snwwsl_r',)
```

Fig. 39

Later , A cross table is created. This line of code generates a styled contingency table that provides insights into the relationship between SibSp- Pclass and Survived - Sex, with highlighted color gradients aiding in visual analysis.The result of the code is:

Pclass	1	2	3
SibSp			
0	137	120	351
1	71	55	83
2	5	8	15
3	3	1	12
4	0	0	18
5	0	0	5
8	0	0	7

Fig. 40

Following observations are made by the coder: The barplot and factorplot illustrate that passengers who are alone on board, with no siblings, have a survival rate of approximately 34.5%. This rate decreases as the number of siblings increases, which aligns with the intuition that individuals may prioritize saving their family members over themselves. However, it can be seen that the survival rate for families with 5-8 members is 0%. The reason behind this anomaly appears to be linked to passenger class.After checking the crosstab , it can be said that the reason is related to passenger class. It shows that people who has higher than 3 siblings+spouses were all in Pclass=3.From analying the 2 cross table , it is clear that all the large families (SibSp>4) died.

f) Parch Feature

Parch represents the number of parents/children aboard.

Pclass	1	2	3
Parch			
0	163	134	381
1	31	32	55
2	21	16	43
3	0	2	3
4	1	0	3
5	0	0	5
6	0	0	1

Fig. 41

Crosstab shows the relation between Pclass and Parch.As it can be seen , the larger families are in Pclass=3 in this crosstab as well.

Later on , two charts are created.First chart shows Parch vs Survived as a bar chart.Second chart shows Parch vs Survived as factor chart.

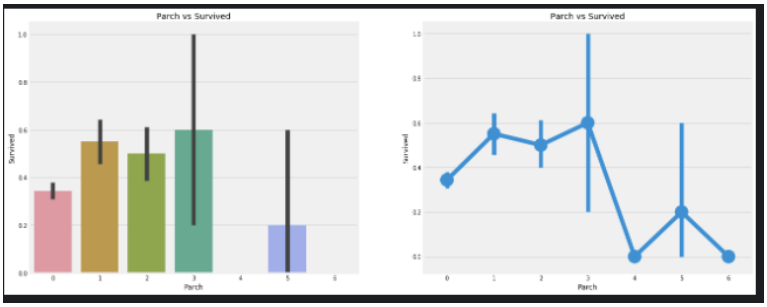


Fig. 42

Similar results are observed here as well. Passengers accompanied by their parents exhibit a higher likelihood of survival, although this likelihood diminishes with increasing numbers.Greater survival chances are associated with having 1-3 parents on the ship. Conversely, being alone is shown to be detrimental, and survival chances decrease when an individual has more than four parents on board.

Conclusion from inspection is that having 1-2 siblings or a spouse on board, as well as having 1-3 parents, increases the probability of survival compared to traveling alone or with a large family.

g) Fare

```
print('Highest Fare was:',data['Fare'].max())
print('Lowest Fare was:',data['Fare'].min())
print('Average Fare was:',data['Fare'].mean())
```

```
Highest Fare was: 512.3292
Lowest Fare was: 0.0
Average Fare was: 32.2042079685746
```

Fig. 43

First line prints the maximum value in Fare column, second line prints the minimum value in Fare column , third line prints average value in Fare column.

```
f,ax=plt.subplots(1,3,figsize=(20,8))
sns.distplot(data[data['Pclass']==1].Fare,ax=ax[0])
ax[0].set_title('Fares in Pclass 1')
sns.distplot(data[data['Pclass']==2].Fare,ax=ax[1])
ax[1].set_title('Fares in Pclass 2')
sns.distplot(data[data['Pclass']==3].Fare,ax=ax[2])
ax[2].set_title('Fares in Pclass 3')
plt.show()
```

Fig. 44

Distplot is a plot function that displays distribution plot , i.e. histogram . It displays fares of passengers. A distribution plot is created for each of Pclass.

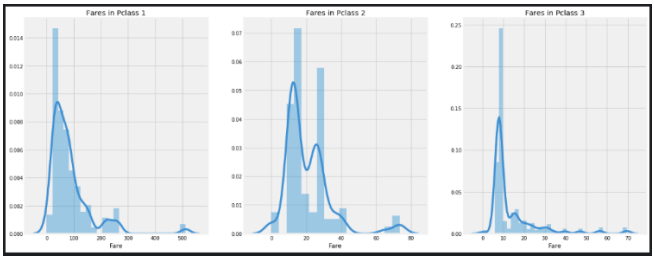


Fig. 45

A substantial variation is observed in the fares of passengers in Pclass1, with this distribution gradually decreasing as the class standards diminish. Since this distribution is continuous, it can be converted into discrete values through binning.

II. CORRELATION BETWEEN FEATURES

```
sns.heatmap(data.corr(),annot=True,cmap='RdYlGn',linewidths=0.2) #data.corr
()-->correlation matrix
fig=plt.gcf()
fig.set_size_inches(10,8)
plt.show()
```

Fig. 46

Heatmap function is used to create correlation chart between all features.Using heatmap allows us to see the relation between all features as it is said earlier and helps us make conclusions from these relations.

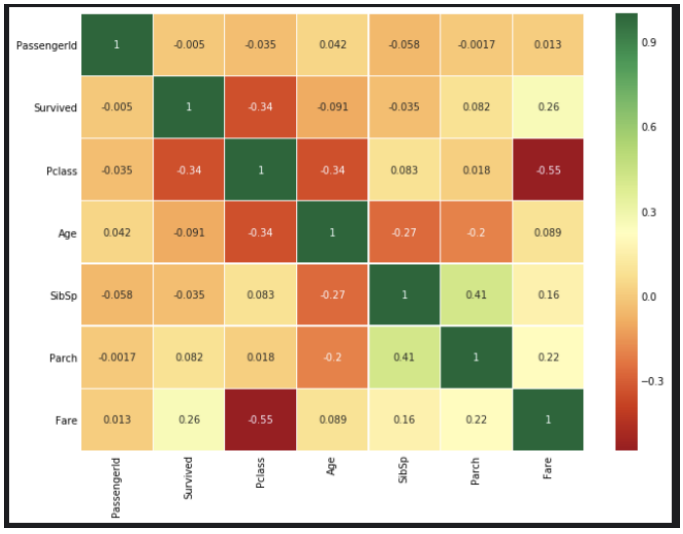


Fig. 47

It's important to note that only numeric features are compared, as it's evident that we can't correlate between alphabets or strings. Positive correlation indicates that if an increase in A leads to an increase in B, they are positively correlated. Since the values are normalized , the maximum correlation value is +1 and the minimum correlation value is -1

When two features are highly or perfectly correlated, an increase in one leads to an increase in the other. This suggests that both features contain highly similar information, leading to minimal or no variance in information. This phenomenon is known as multicollinearity, as both features contain almost the same information.The decision of whether to use both features or eliminate one as redundant is made during model training. Reducing redundant features can lead to advantages such as reduced training time.From the provided heatmap, it can be observed that the features are not highly correlated. The highest correlation is between SibSp and Parch, which is 0.41. Therefore, it's reasonable to proceed with all features without eliminating any, as significant redundancy is not observed.

IV. Part 2: Feature Engineering and Data Cleaning

Relation of Age Band

For Machine learning algorithms to be useful , The data should be categorized as discrete. Age is defined as a continous feature in the “data” dataframe.So the age should be disretized.

The Age column has values between 0 and 80 . The coder decides to divide these ages into 5 categories with equal intervals.

```
data['Age_band']=0
data.loc[data['Age']>16,'Age_band']=1
data.loc[(data['Age']>32)&(data['Age']<=16)','Age_band']=2
data.loc[(data['Age']>48)&(data['Age']<=32)','Age_band']=3
data.loc[(data['Age']>64)&(data['Age']<=48)','Age_band']=4
data.loc[data['Age']<=64','Age_band']=5
data['Age_band']=0
```

Fig. 48

The coder defines a new column in the “data” dataframe named Age_band and initiliazes it’s values as zero. Then he assigns an integer value to each defined age range stored in Age_band.As it can be seen , the age discretization starts from 0 with an interval of length 16 between each discretized age_band.

Data.head(2) prints the first 2 elements of the “data” dataframe.The output is:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Initial	Age_band
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S	Mr	1
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C	Mrs	2

Fig.49

As it can be seen , a new Age_band column is added with values 1 and 2.1 means the passenger’s age is in between or equal to 16 and 32 , 2 means the passenger’s age is in between or equal to 32 and 48.

Later , the crossplot for age_band is created.

	Age_band
1	382
2	325
0	104
3	69
4	11

Fig. 50

```
sns.factorplot('Age_band','Survived',data=data,col='Pclass')
plt.show()
```

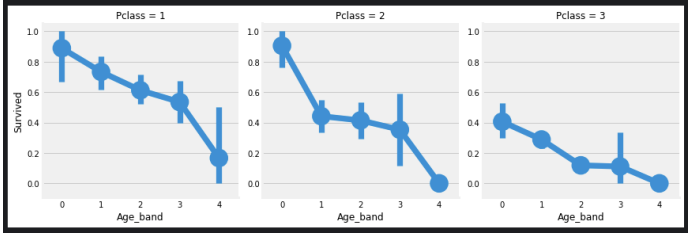


Fig. 51

The created factor charts clearly shows that the survival chance of the passengers decreases as their age gets higher for each of the Pclass. Relation of Family size

Family size is checked to see whether the survival rate is related to the having a family or not.Two column will be defined for this.First is “Family_size” , it will contain the number of family members a person has. Second is “Alone” , it will hold the information whether the passenger is alone or not.If passenger is alone , 1 is assigned to this column, 0 is assigned otherwise.The corresponding code to achive this is given below by the coder.

```
data['Family_Size']=0
data['Family_Size']=data['Parch']+data['SibSp']#family size
data['Alone']=0
data.loc[data.Family_Size==0,'Alone']=1#Alone
```

Fig. 52

Based on the new defined columns , the coder creates two new factor charts. First column shows the relation between Family_size and Survival. Second column shows the relation between Alone and Survival. If an individual is alone or has a family size of 0, the chances for survival are very low. Similarly, for family sizes greater than 4, the chances of survival also decrease. This feature appears to be significant for the model.The factor charts used to make these conclusions are given below.

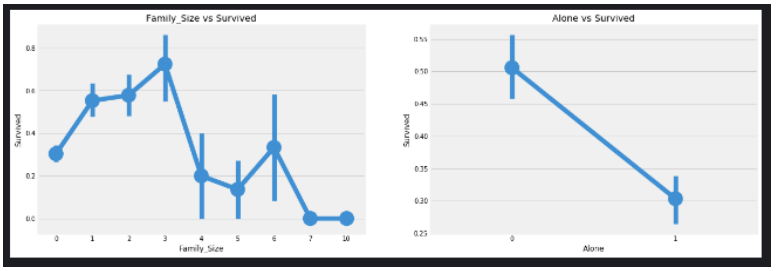


Fig. 53

After that , the relation between Alone and Survival is inspected based on genders for each of the “Pclass”es.

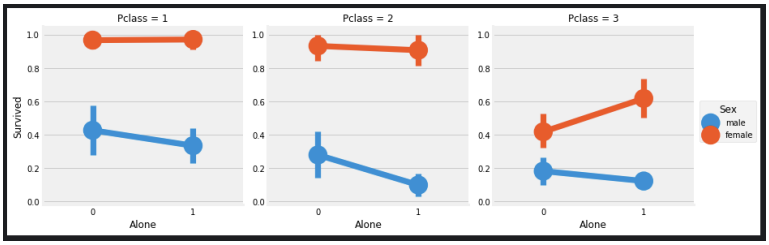


Fig. 54

It is evident that being alone is detrimental regardless of sex or passenger class, except for Pclass=3, where the survival chances of females who are alone are higher than those with family.

Relation of Fare range

```
data['Fare_Range']=pd.qcut(data['Fare'],4)
data.groupby(['Fare_Range'])['Survived'].mean().to_frame().style.background_gradient(cmap='summer_r')
```

Fig. 55

qcut arranges or splits the values according to the number of bins passed. So, if 4 bins are passed, the values will be equally spaced into 4 separate bins or value ranges just like the code above.After qcut operation , the related data is grouped and plotted.

	Survived
Fare_Range	
(-0.001, 7.91]	0.197309
(7.91, 14.454]	0.303571
(14.454, 31.0]	0.454955
(31.0, 512.329]	0.581081

Fig. 56

It is evident that as the fare_range increases, the chances of survival also increase. However, the fare_range values cannot be passed as they are. They need to be converted into singleton values, similar to what was done with age_band.

```
data['Fare_cat']=0
data.loc[data['Fare']<=7.91,'Fare_cat']=0
data.loc[(data['Fare']>7.91)&(data['Fare']<=14.454),'Fare_cat']=1
data.loc[(data['Fare']>14.454)&(data['Fare']<=31),'Fare_cat']=2
data.loc[(data['Fare']>31)&(data['Fare']<=513),'Fare_cat']=3
```

Fig. 57

This code creates a new column named “Fare_cat” in the “data” dataframe.Then ; Fare_cat is assigned to have 4 different values based on the range in which Fare value lies. For example , when Fare is higher than 7.91 and lower than or equal to 14.545 , Fare_cat is assigned to have “1” .

After creating Fare_cat , a factor chart is created to show the relation between Survived and Fare_cat for different Sexes.It is evident that as the Fare_cat increases, the survival chances also increase. This feature may emerge as an important factor during modeling, alongside Sex. The chart is given below.

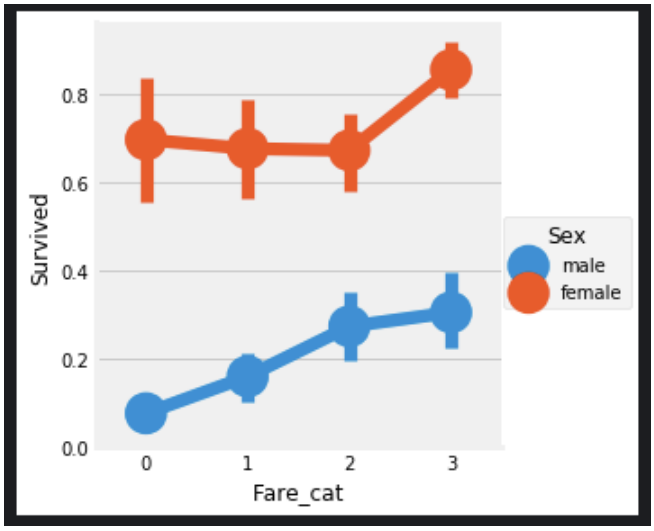


Fig. 57

Converting String Values into Numeric

Since strings cannot be passed to a machine learning model, features like Sex, Embarked, etc., need to be converted into numeric values.This is done by using the following code.

```
data['Sex'].replace(['male','female'],[0,1],inplace=True)
data['Embarked'].replace(['S','C','Q'],[0,1,2],inplace=True)
data['Initial'].replace(['Mr','Mrs','Miss','Master','Other'],[0,1,2,3,4],in
place=True)
```

Fig. 58

From the code , it can be seen that:

V. Sex is replaced by either 0 or 1.For males , 0 is used ; for females , 1 is used.

VI. For Embarked ports ; port S is represented by 0 , port C is represented by 1 and Port Q is represented by 2.

VII. For Initial ; Mr is represented by 0 , Mrs is represented by 1 , Miss is represented by 2 , Master is represented by 3 , Other is represented by 4.

The Columns that can not be converted to any numeric values or useless in analyzing the data or already converted into a new column to make it easier to use should be deleted from the dataframe that will be feed to the machine learning algorithms. The coder deletes these columns. The coder explains that the “Name” feature is unnecessary as it can not be converted into any categorical value. Similarly, the “Age” feature is redundant since we already have the “Age_band” feature. The “Ticket” feature comprises random strings that can not be categorized, rendering it unnecessary. Likewise, the “Fare” feature is redundant as we have the “Fare_cat” feature. The “Cabin” feature contains numerous NaN values and many passengers have multiple cabins, making it a useless feature. Additionally, the “Fare_Range” feature is redundant since the fare_cat feature can be used instead. Finally, the “PassengerId” feature can not be categorized, rendering it unnecessary for modeling purposes.

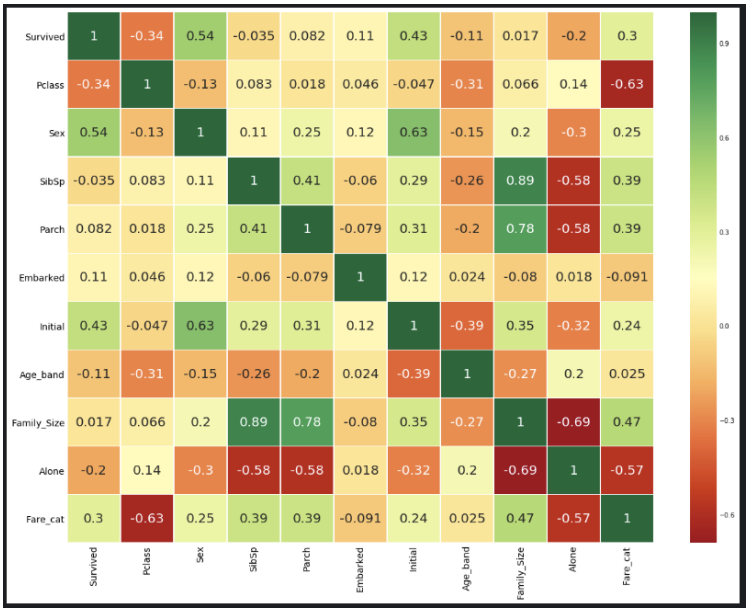


Fig. 59

Heatmap function is used to create correlation chart between all features.Using heatmap allows us to see the relation between all features as it is said earlier and helps us make conclusions from these relations. It's important to note that only numeric features are compared. All the new columns created earlier and their relations with others can be seen in this new heatmap.Some positively related functions are Sex and Initial , Parch and Family_size , Survived and Sex etc. while some negatively related features are Pclass and Fare_cat , Age_band and Initial , Age_band and Pclass etc.

VIII. Part 3: Predictive Modeling

This is the last part of the data analysis where machine learning algoriths are applied on the cleaned and adjusted “data” dataframe.In the analysis part , some analysis are made but we are unable to predict the survival of a passenger. To do these predictions, classification algorithms.The algorithms used in this analysis by the coder is logistic regression , support vector machines(linear and radial) , random forest , K-

nearest neighbours , naive bayes , decision tree , logistic regression.

```
#importing all the required ML packages
from sklearn.linear_model import LogisticRegression #logistic regression
from sklearn import svm #support vector Machine
from sklearn.ensemble import RandomForestClassifier #Random Forest
from sklearn.neighbors import KNeighborsClassifier #KNN
from sklearn.naive_bayes import GaussianNB #Naive bayes
from sklearn.tree import DecisionTreeClassifier #Decision Tree
from sklearn.model_selection import train_test_split #training and testing data split
from sklearn import metrics #accuracy measure
from sklearn.metrics import confusion_matrix #for confusion matrix
```

Fig. 60

First , The classification algorithms are imported from their respective libraries. To calculate the accuracy of each classification algorithms , “metrics” functions and confusion matrix are also imported.

```
train,test=train_test_split(data,test_size=0.3,random_state=0,stratify=data['Survived'])
train_X=train[train.columns[1:]]
train_Y=train[train.columns[:1]]
test_X=test[test.columns[1:]]
test_Y=test[test.columns[:1]]
X=data[data.columns[1:]]
Y=data['Survived']
```

Fig. 61

train,test=train_test_split(data,test_size=0.3,random_state=0,stratify=data['Survived']) : this code splits the “data” dataframe into “train” set and “test” set.Stratify=data[‘Survived’] function makes sure that the Survived values are evenly distributed betweem training set and testing set.

train_X=train[train.columns[1:]] : Assigns feature columns of train dataset to train_X except the first feature column.

train_Y=train[train.columns[:1]] : Assigns only the first feature column of train dataset to train_Y.

test_X=test[test.columns[1:]] : Assigns feature columns of test dataset to test_X except the first feature column.

test_Y=test[test.columns[:1]] : Assigns only the first feature column of test dataset to test_Y.

X=data[data.columns[1:]] : Assigns feature columns of “data” to X except the first feature column.

Y=data['Survived'] : Assigns the feature column “Survived” in the “data” to Y.

Now , classification methods are applied one by one.In each application of a classification method , train datas are used to train the classification model and test data is used to make predictions. Then the result of the predictions are compared with actual results of test data. After comparison , normalized accucary value of the classification method is calculated by using the metrics.accuracy_score().

A. Radial Support Vector Machines(rbf-SVM)

```
model=svm.SVC(kernel='rbf',C=1,gamma=0.1)
model.fit(train_X,train_Y)
prediction1=model.predict(test_X)
print('Accuracy for rbf SVM is ',metrics.accuracy_score(prediction1,test_Y))

Accuracy for rbf SVM is  0.835820895522
```

Fig. 62

B. Linear Support Vector Machine(linear-SVM)

```
model=svm.SVC(kernel='linear',C=0.1,gamma=0.1)
model.fit(train_X,train_Y)
prediction2=model.predict(test_X)
print('Accuracy for linear SVM is',metrics.accuracy_score(prediction2,test_Y))

Accuracy for linear SVM is 0.817164179104
```

Fig. 63

C. Logistic Regression

```
model = LogisticRegression()
model.fit(train_X,train_Y)
prediction3=model.predict(test_X)
print('The accuracy of the Logistic Regression is',metrics.accuracy_score(prediction3,test_Y))

The accuracy of the Logistic Regression is 0.817164179104
```

Fig. 64

D. Decision Tree

```
model=DecisionTreeClassifier()
model.fit(train_X,train_Y)
prediction4=model.predict(test_X)
print('The accuracy of the Decision Tree is',metrics.accuracy_score(prediction4,test_Y))

The accuracy of the Decision Tree is 0.798507462687
```

Fig. 65

E. K-Nearest Neighbours(KNN)

```
model=KNeighborsClassifier()
model.fit(train_X,train_Y)
prediction5=model.predict(test_X)
print('The accuracy of the KNN is',metrics.accuracy_score(prediction5,test_Y))

The accuracy of the KNN is 0.832089552239
```

Fig. 66

In this method , the accuracy changes based on the changes in n_neighbours so the result of the KNN method should be inspected over a range of 1 to 10.

```
a_index=list(range(1,11))
a=pd.Series()
x=[0,1,2,3,4,5,6,7,8,9,10]
for i in list(range(1,11)):
    model=KNeighborsClassifier(n_neighbors=i)
    model.fit(train_X,train_Y)
    prediction=model.predict(test_X)
    a=a.append(pd.Series(metrics.accuracy_score(prediction,test_Y)))
plt.plot(a_index, a)
plt.xticks(x)
fig=plt.gcf()
fig.set_size_inches(12,6)
plt.show()
print('Accuracies for different values of n are:',a.values,'with the max value as ',a.values.max())
```

Fig. 67

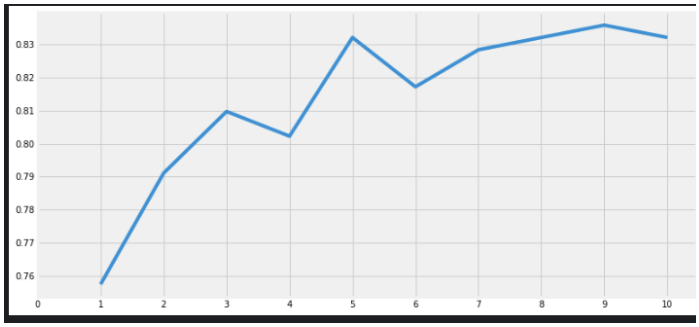


Fig. 68

Accuracies for different values of n are:
n=1 : 0.75746269
n=2 : 0.79104478
n=3 : 0.80970149
n=4 : 0.80223881
n=5 : 0.83208955
n=6 : 0.81716418
n=7 : 0.82835821
n=8 : 0.83208955
n=9 : 0.8358209
n=10 : 0.83208955
The maximum value is 0.835820895522.

F. Gaussian Naive Bayes

```
model=GaussianNB()
model.fit(train_X,train_Y)
prediction6=model.predict(test_X)
print('The accuracy of the NaiveBayes is',metrics.accuracy_score(prediction6,test_Y))

The accuracy of the NaiveBayes is 0.813432835821
```

Fig. 69

G. Random Forests

```
model=RandomForestClassifier(n_estimators=100)
model.fit(train_X,train_Y)
prediction7=model.predict(test_X)
print('The accuracy of the Random Forests is',metrics.accuracy_score(prediction7,test_Y))

The accuracy of the Random Forests is 0.820895522388
```

Fig. 70

The robustness of a classifier isn't solely determined by its accuracy.This is due to the thing called model variance. In machine learning model variance is the amount by which the performance of a predictive model changes when it is trained on different subsets of the training data. More specifically, model variance is the variability of the model that how much it is sensitive to another subset of the training dataset.To solve this problem , the coder decides to use cross validation.

Cross Validation

In many cases, data is imbalanced, meaning there may be a higher number of instances in one class compared to others. In such scenarios, it's important to train and test our algorithm on each and every instance of the dataset. Subsequently, we can calculate the average accuracy over the entire dataset by noting the accuracy of each instance. The K-Fold Cross Validation technique operates by initially dividing the dataset into k-subsets. For instance, if k=5 s choosen , the dataset is split into five parts. One part is reserved for testing, while the algorithm is trained on the remaining four parts. This process continues iteratively, with the testing part changing in each iteration, and the algorithm being trained on the other parts. The accuracies and errors obtained from each iteration are then averaged to derive an overall accuracy for the algorithm, known as the average accuracy. This approach helps mitigate the risk of an algorithm underfitting or overfitting the dataset, as it allows for the creation of a more generalized model through cross-validation.

	CV Mean	Std
Linear Svm	0.793471	0.047797
Radial Svm	0.828290	0.034427
Logistic Regression	0.805843	0.021861
KNN	0.813783	0.041210
Decision Tree	0.805868	0.025361
Naive Bayes	0.801386	0.028999
Random Forest	0.822684	0.026868

Fig. 71

Cv mean shows the cross validation mean. It shows the mean value for the accuracy of the 10 different test data.Std stands for standart variation. What is looked in this graph is a classification with CV mean as high as possible and standart deviation as low as possible.
Plotting the accuracy of each classification method with dataframe, following chart is obtained.

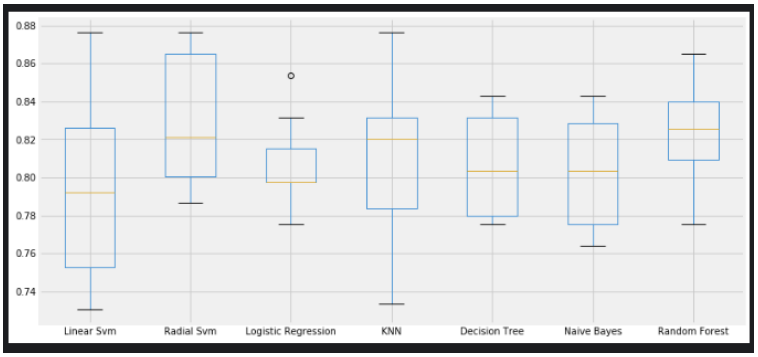


Fig. 72

Plotting the cross validation mean of each classification method with dataframe, following chart is obtained.

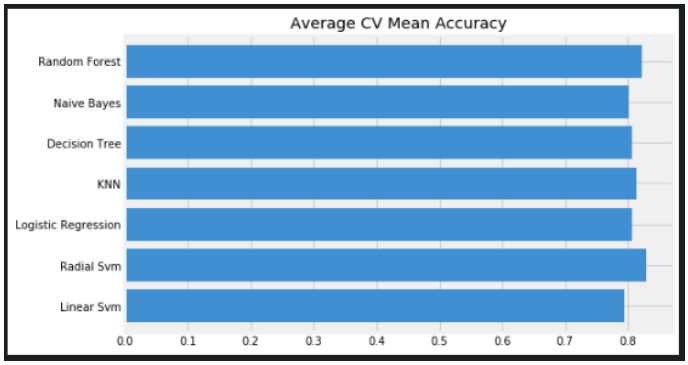


Fig. 73

As it can be seen from the graph , it is hard to tell the difference in some of the classification methods such as Random forest and Radial SVM. So a confusion matrix is created to show the number of true predictions and false predictions.

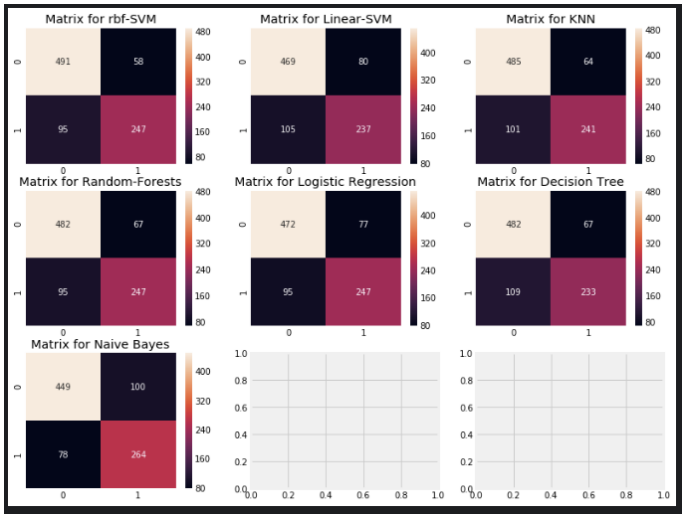


Fig. 74

How this matrixes works is explained next. For KNN , 485 is the number of true predictions for the dead and 241 is the number of true predictions for the survived people.The false predictions of the dead people are 64 and the false predictions for the survived people are 101. After analyzing all the confusion matrices, it appears that the rbf-SVM classifier has a higher likelihood of correctly predicting passengers who did not survive .Conversely, the naive bayes classifier demonstrates a higher probability of accurately predicting passengers who survived. Some of the parameters can be changed in classifications methods to get a better performace.This is known as hyper-parameters tuning. To decide this followings are made for SVM and Random Forests.

```
from sklearn.model_selection import GridSearchCV
C=[0.05,0.1,0.2,0.3,0.25,0.4,0.5,0.6,0.7,0.8,0.9,1]
gamma=[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]
kernel=['rbf','linear']
hyper={'kernel':kernel,'C':C,'gamma':gamma}
gd=GridSearchCV(estimator=svm.SVC(),param_grid=hyper,verbose=True)
gd.fit(X,Y)
print(gd.best_score_)
print(gd.best_estimator_)
```

Fig. 75

The best parameters for the best prediction is given below in SVM classification method.

```
Fitting 3 folds for each of 240 candidates, totalling 720 fits
0.828282828283
SVC(C=0.5, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.1, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)

[Parallel(n_jobs=1)]: Done 720 out of 720 | elapsed:   14.7s finished
```

Fig. 76

Random Forest

```
n_estimators=range(100,1000,100)
hyper={'n_estimators':n_estimators}
gd=GridSearchCV(estimator=RandomForestClassifier(random_state=0),param_grid=hyper,verbose=True)
gd.fit(X,Y)
print(gd.best_score_)
print(gd.best_estimator_)
```

Fig. 77

The best parameters for the best prediction is given below in random forest classification method.

```
Fitting 3 folds for each of 9 candidates, totalling 27 fits

[Parallel(n_jobs=1)]: Done  27 out of  27 | elapsed:   18.4s finished

0.817059483726
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=900, n_jobs=1,
            oob_score=False, random_state=0, verbose=0, warm_start=False)
```

Fig. 78

The highest score achieved for Rbf-Svm is 82.82%, obtained with C=0.05 and gamma=0.1. For RandomForest, the score is approximately 81.8%, achieved with n_estimators=900.

ENSEMBLING

I will keep this part brief since we did not learn it in lessons or in homeworks.

Ensembling is an effective method to enhance the accuracy or performance of a model. In essence, it involves combining multiple simple models to create a single, more powerful model. In statistics and machine learning, ensemble methods harness multiple learning algorithms to achieve superior predictive performance compared to what could be achieved by any individual learning algorithm al

I. Voting Classifier

Voting Classifier combines KNN , SVC , random forest , logistic regression , decision tree classifier and gaussianNB to make better predictions.The result from using this method is as follows :

```
The accuracy for ensembled model is: 0.824626865672
The cross validated score is 0.823766031097
```

Fig. 79

II. Bagging Classifier

Bagging, alternatively known as bootstrap aggregation, is a popular ensemble learning technique utilized to decrease variance within a noisy dataset.
The results from bagging the KNN is :

```
The accuracy for bagged KNN is: 0.835820895522
The cross validated score for bagged KNN is: 0.814889342
```

Fig.80
The results from bagging decision tree is :

```
The accuracy for bagged Decision Tree is: 0.824626865672
The cross validated score for bagged Decision Tree is: 0.820482635342
```

Fig.81

III. Boosting

In the realm of machine learning, boosting serves as a meta-algorithm within ensembles primarily aimed at diminishing bias and variance. This technique is employed in supervised learning and encompasses a group of machine learning algorithms designed to transform weak learners into robust ones.

ADABOOST

Adaboost can be employed alongside various other types of learning algorithms to enhance performance. The results from these other learning algorithms, referred to as 'weak learners,' are aggregated into a weighted sum, which constitutes the final output of the boosted classifier.The weak learner here is the decision tree classification method.

```
from sklearn.ensemble import AdaBoostClassifier
ada=AdaBoostClassifier(n_estimators=200,random_state=0,learning_rate=0.1)
result=cross_val_score(ada,X,Y,cv=10,scoring='accuracy')
print('The cross validated score for AdaBoost is:',result.mean())

The cross validated score for AdaBoost is: 0.824952616048
```

Fig.82
Hyper-parameter tuning can be applied to get a better result to this classification. he highest achievable accuracy using AdaBoost is 83.16%, attained with n_estimators=200 and learning_rate=0.05.

Stochastic Gradient Boosting

```
from sklearn.ensemble import GradientBoostingClassifier
grad=GradientBoostingClassifier(n_estimators=500,random_state=0,learning_rate=0.1)
result=cross_val_score(grad,X,Y,cv=10,scoring='accuracy')
print('The cross validated score for Gradient Boosting is:',result.mean())

The cross validated score for Gradient Boosting is: 0.818286233118
```

Fig.83

XGBoost

```
import xgboost as xg
xgboost=xg.XGBClassifier(n_estimators=900,learning_rate=0.1)
result=cross_val_score(xgboost,X,Y,cv=10,scoring='accuracy')
print('The cross validated score for XGBoost is:',result.mean())

The cross validated score for XGBoost is: 0.810471002156
```

Fig.84

In conclusion , the best accuracy is obtained from Adaboost classifier with hyper-parameter tuning.The accuracy for it is %83.16 .Confusion matrix is created by the following code for this model to make further inspections.

```
ada=AdaBoostClassifier(n_estimators=200,random_state=0,learning_rate=0.05)
result=cross_val_predict(ada,X,Y,cv=10)
sns.heatmap(confusion_matrix(Y,result),cmap='winter',annot=True,fmt='2.0f')
plt.show()
```

Fig.85

First line initializes an Adaboostclassifier with 200 estimators with a random state of zero and learning rate of 0.05 ,then assigns it to “ada”.
Second line performs 10-cross validation using “ada” on the feature set of X and and target variable Y . Predictions are stored in “result”.
Third line creates a heatmap onto the confusion matrix with winter colormap representation .Last line displays the heatmap.The result of the code is :

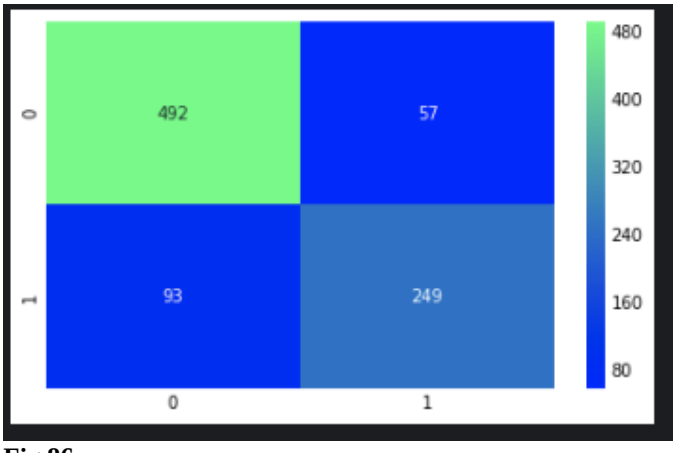


Fig.86

At the end of the analysis , the weight factors of each feature column in “data” dataframe is created for four of the

classification methods. Creating the weight factors is called feature importance.

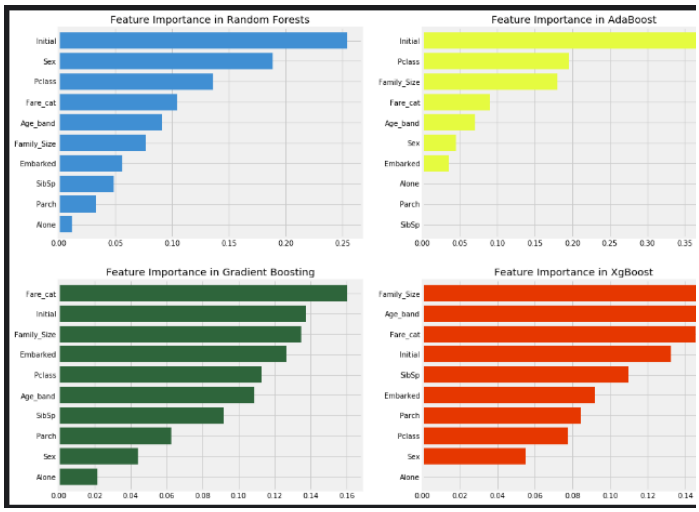


Fig.87

Some common and crucial features for predicting survival include Initial, Fare_cat, Pclass, and Family_Size. Surprisingly, the Sex feature appears to have little importance across several classifiers, despite its earlier significance when combined with Pclass. However, in RandomForests, Sex emerges as an important feature. Notably, Initial, which frequently ranks highest among classifiers, correlates positively with Sex, indicating their shared representation of gender. Similarly, Pclass and Fare_cat reflect the passengers' social status, while Family_Size, Alone, Parch, and SibSp capture aspects of family composition.

REFERENCES

[1] A. Swain, "EDA To Prediction(DieTanic)," [Online]. Available: https://www.kaggle.com/code/ash316/eda-to-prediction-dietanic

Titanic - Advanced Feature Engineering Tutorial

Abdullah Hakan Şişik , Çağatay Özdemir

This kernel comprises three primary sections: Exploratory Data Analysis, Feature Engineering and Model. The training set consists of 891 rows, while the test set consists of 418 rows. The training set contains 12 features, whereas the test set contains 11 features. An additional feature present only in the training set is the "Survived" feature, which serves as the target variable.

```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="darkgrid")

from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import OneHotEncoder, LabelEncoder, StandardScaler
from sklearn.metrics import roc_curve, auc
from sklearn.model_selection import StratifiedKFold

import string
import warnings
warnings.filterwarnings('ignore')

SEED = 42
```

Fig.1

```
def concat_df(train_data, test_data):
    # Returns a concatenated df of training and test set
    return pd.concat([train_data, test_data], sort=True).reset_index(drop=True)

def divide_df(all_data):
    # Returns divided dfs of training and test set
    return all_data.loc[:890], all_data.loc[891:].drop(['Survived'], axis=1)

df_train = pd.read_csv('../input/train.csv')
df_test = pd.read_csv('../input/test.csv')
df_all = concat_df(df_train, df_test)

df_train.name = 'Training Set'
df_test.name = 'Test Set'
df_all.name = 'All Set'

dfs = [df_train, df_test]

print('Number of Training Examples = {}'.format(df_train.shape[0]))
print('Number of Test Examples = {}\n'.format(df_test.shape[0]))
print('Training X Shape = {}'.format(df_train.shape))
print('Training y Shape = {}\n'.format(df_train['Survived'].shape[0]))
print('Test X Shape = {}'.format(df_test.shape))
print('Test y Shape = {}\n'.format(df_test.shape[0]))
print(df_train.columns)
print(df_test.columns)
```

Fig.2

```
Number of Training Examples = 891
Number of Test Examples = 418

Training X Shape = (891, 12)
Training y Shape = 891

Test X Shape = (418, 11)
Test y Shape = 418

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

Fig.3

1. Initial Data Exploration

1.1 Overview

PassengerId serves as the unique identifier for each row and does not influence the target variable.
Survived denotes the target variable we aim to predict, with binary values:
1 = Survived
0 = Not Survived
Pclass (Passenger Class) reflects the socio-economic status of passengers and constitutes a categorical ordinal feature with three distinct values (1, 2, or 3):

1 = Upper Class
2 = Middle Class
3 = Lower Class

Name, Sex, and Age require no further explanation.

SibSp indicates the total count of the passenger's siblings and spouse onboard.

Parch denotes the total count of the passenger's parents and children onboard.

Ticket represents the ticket number assigned to the passenger.

Fare stands for the fare paid by the passenger.

Cabin refers to the specific cabin number assigned to the passenger.

Embarked signifies the port of embarkation and constitutes a categorical feature with three distinct values (C, Q, or S):

- C = Cherbourg
- Q = Queenstown
- S = Southampton

Let’s look our three sample in the dataset.

```
print(df_train.info())
df_train.sample(3)
```

Fig.3

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
PassengerId    891 non-null int64
Survived       891 non-null int64
Pclass         891 non-null int64
Name           891 non-null object
Sex            891 non-null object
Age            714 non-null float64
SibSp          891 non-null int64
Parch          891 non-null int64
Ticket         891 non-null object
Fare           891 non-null float64
Cabin          204 non-null object
Embarked       889 non-null object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.6+ KB
None
```

Fig.4

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
689	690	1	1	Madill, Miss. Georgette Alexandra	female	15.0	0	1	24160	211.3375	B5	S
525	526	0	3	Farrell, Mr. James	male	40.5	0	0	367232	7.7500	NaN	Q
278	279	0	3	Rice, Master. Eric	male	7.0	4	1	382652	29.1250	NaN	Q

Fig.5

1.2 Handling Missing Data

It's evident that missing values are present in some columns. The display_missing function provides insight into the count of missing values in each column across both the training and test sets. In the training set, missing values are found in the Age, Cabin, and Embarked columns, while in the test set, they are observed in the Age, Cabin, and Fare columns. Utilizing a concatenated training and test set is beneficial for handling missing values to prevent biased filling of data towards either the training or test set samples. Although the count of missing values in Age, Embarked, and Fare is relatively small compared to the total sample, approximately 80% of the Cabin data is missing. While descriptive statistical measures can be employed to fill missing values in Age, Embarked, and Fare, this approach isn't feasible for Cabin due to the extensive missing data in that column.

```
def display_missing(df):
    for col in df.columns.tolist():
        print('{} column missing values: {}'.format(col, df[col].isnull().sum()))
    print('\n')

for df in dfs:
    print('{}'.format(df.name))
    display_missing(df)
```

Fig.6

```
Training Set
PassengerId column missing values: 0
Survived column missing values: 0
Pclass column missing values: 0
Name column missing values: 0
Sex column missing values: 0
Age column missing values: 177
SibSp column missing values: 0
Parch column missing values: 0
Ticket column missing values: 0
Fare column missing values: 0
Cabin column missing values: 687
Embarked column missing values: 2


Test Set
PassengerId column missing values: 0
Pclass column missing values: 0
Name column missing values: 0
Sex column missing values: 0
Age column missing values: 86
SibSp column missing values: 0
Parch column missing values: 0
Ticket column missing values: 0
Fare column missing values: 1
Cabin column missing values: 327
Embarked column missing values: 0
```

Fig.7

1.2.1 Age

Filling missing values in the Age column with the median age is a common strategy, but using the median age of the entire dataset may not be optimal. Instead, utilizing the median age of Pclass groups is a better choice due to its higher correlation with Age (0.408106) and Survived (0.338481). Grouping ages by passenger classes is also more logical than other features.

```
df_all_corr = df_all.corr().abs().unstack().sort_values(kind="quicksort", ascending=Fals
e).reset_index()
df_all_corr.rename(columns={"level_0": "Feature 1", "level_1": "Feature 2", 0: 'Correlat
ion Coefficient'}, inplace=True)
df_all_corr[df_all_corr['Feature 1'] == 'Age']
```

Fig.8

	Feature 1	Feature 2	Correlation Coefficient
6	Age	Age	1.000000
9	Age	Pclass	0.408106
17	Age	SibSp	0.243699
22	Age	Fare	0.178740
25	Age	Parch	0.150917
29	Age	Survived	0.077221
41	Age	PassengerId	0.028814

Fig.9

When we break down the data by Pclass and Sex, we notice that each combination of these two factors has its own distinct median age. This means that within each Pclass, the median

Age varies between males and females. Generally, as the Pclass increases (from lower to higher classes), the median Age tends to increase as well, regardless of gender. However, it's important to note that females typically have slightly lower median ages compared to males within the same Pclass. By considering both Pclass and Sex when filling in missing Age values, we can achieve a more precise estimation of Age for each passenger.

```
age_by_pclass_sex = df_all.groupby(['Sex', 'Pclass']).median()['Age']

for pclass in range(1, 4):
    for sex in ['female', 'male']:
        print('Median age of Pclass {} {}s: {}'.format(pclass, sex, age_by_pclass_sex[se
x][pclass]))
print('Median age of all passengers: {}'.format(df_all['Age'].median()))

# Filling the missing values in Age with the medians of Sex and Pclass groups
df_all['Age'] = df_all.groupby(['Sex', 'Pclass'])['Age'].apply(lambda x: x.fillna(x.medi
an()))
```

Fig.10

```
Median age of Pclass 1 females: 36.0
Median age of Pclass 1 males: 42.0
Median age of Pclass 2 females: 28.0
Median age of Pclass 2 males: 29.5
Median age of Pclass 3 females: 22.0
Median age of Pclass 3 males: 25.0
Median age of all passengers: 28.0
```

Fig.11

1.2.2 Embarked

Embarked, a categorical feature, presents only two missing values across the entire dataset. Remarkably, both of these passengers share similar attributes: they are female, belong to the upper class, and possess identical ticket numbers. This strongly suggests that they are acquainted and boarded the ship together from the same port. While the most common Embarked value for upper-class female passengers is C (Cherbourg), it's important to acknowledge that this doesn't conclusively determine their port of embarkation.

```
df_all[df_all['Embarked'].isnull()]
```

	Age	Cabin	Embarked	Fare	Name	Parch	PassengerId	Pclass	Sex	SibSp	Survived	Ticket
61	38.0	B28	NaN	80.0	Icard, Miss. Amelie	0	62	1	female	0	1.0	113572
829	62.0	B28	NaN	80.0	Stone, Mrs. George Nelson (Martha Evelyn)	0	830	1	female	0	1.0	113572

Fig.12

1.2.3 Fare

Only one passenger has a missing Fare value. It's reasonable to assume that Fare is correlated with family size (Parch and SibSp) and Pclass. Therefore, filling the missing value with the median Fare of a male passenger with a third-class ticket and no family seems logical. This approach takes into account common patterns in Fare based on ticket class and family size, providing a reasonable estimate for the missing value.

```
df_all[df_all['Fare'].isnull()]
```

Fig.13

	Age	Cabin	Embarked	Fare	Name	Parch	PassengerId	Pclass	Sex	SibSp	Survived	Ticket
1043	60.5	NaN	S	NaN	Storey, Mr. Thomas	0	1044	3	male	0	NaN	3701

```
med_fare = df_all.groupby(['Pclass', 'Parch', 'SibSp']).Fare.median()[3][0][0]
# Filling the missing value in Fare with the median Fare of 3rd class alone passenger
df_all['Fare'] = df_all['Fare'].fillna(med_fare)
```

Fig.14

1.2.4 Cabin

The Cabin feature poses a challenge due to its significant proportion of missing values, yet it warrants further investigation. Ignoring this feature entirely isn't viable as certain cabins might exhibit higher survival rates. Interestingly, the first letter of the Cabin values corresponds to the decks where the cabins are situated. These decks were primarily designated for specific passenger classes, although some were utilized by multiple classes. This insight suggests that the Cabin feature may provide valuable information regarding passengers' locations on the ship, potentially influencing survival outcomes.:

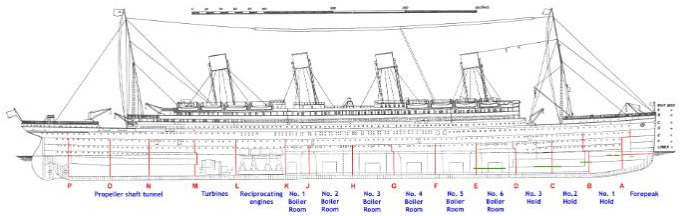


Fig.15

The Boat Deck housed six rooms labeled T, U, W, X, Y, Z, yet only the T cabin is accounted for in the dataset. Decks A, B, and C were exclusively reserved for 1st class passengers, while decks D and E were accessible to passengers of all classes. Decks F and G, however, were shared by both 2nd and 3rd class passengers. Interestingly, moving from deck A to deck G, the distance to the staircase increased, which could potentially influence survival rates. This suggests that proximity to the staircase might have played a role in determining survival outcomes.

```
# Creating Deck column from the first letter of the Cabin column (M stands for Missing)
df_all['Deck'] = df_all['Cabin'].apply(lambda s: s[0] if pd.notnull(s) else 'M')

df_all_decks = df_all.groupby(['Deck', 'Pclass']).count().drop(columns=['Survived', 'Sex', 'Age', 'SibSp', 'Parch',
                                                                        'Fare', 'Embarked', 'Cabin', 'PassengerId', 'Ticke
t']).rename(columns={'Name': 'Count'}).transpose()

def get_pclass_dist(df):

    # Creating a dictionary for every passenger class count in every deck
    deck_counts = {'A': {}, 'B': {}, 'C': {}, 'D': {}, 'E': {}, 'F': {}, 'G': {}, 'M': {}, 'T': {}}
    decks = df.columns.levels[0]

    for deck in decks:
        for pclass in range(1, 4):
            try:
                count = df[deck][pclass][0]
                deck_counts[deck][pclass] = count
            except KeyError:
                deck_counts[deck][pclass] = 0

    df_decks = pd.DataFrame(deck_counts)
    deck_percentages = {}

    # Creating a dictionary for every passenger class percentage in every deck
    for col in df_decks.columns:
        deck_percentages[col] = [(count / df_decks[col].sum()) * 100 for count in df_decks[col]]

    return deck_counts, deck_percentages

def display_pclass_dist(percentages):

    df_percentages = pd.DataFrame(percentages).transpose()
    deck_names = ('A', 'B', 'C', 'D', 'E', 'F', 'G', 'M', 'T')
    bar_count = np.arange(len(deck_names))
    bar_width = 0.85

    pclass1 = df_percentages[0]
    pclass2 = df_percentages[1]
    pclass3 = df_percentages[2]

    plt.figure(figsize=(20, 10))
    plt.bar(bar_count, pclass1, color='#b5ffb9', edgecolor='white', width=bar_width, label='Passenger Class 1')
    plt.bar(bar_count, pclass2, bottom=pclass1, color='#f9bc86', edgecolor='white', width=bar_width, label='Passenger Class
2')
    plt.bar(bar_count, pclass3, bottom=pclass1 + pclass2, color='#a3acff', edgecolor='white', width=bar_width, label='Passeng
er Class 3')

    plt.xlabel('Deck', size=15, labelpad=20)
    plt.ylabel('Passenger Class Percentage', size=15, labelpad=20)
    plt.xticks(bar_count, deck_names)
    plt.tick_params(axis='x', labelsize=15)
    plt.tick_params(axis='y', labelsize=15)

    plt.legend(loc='upper left', bbox_to_anchor=(1, 1), prop={'size': 15})
    plt.title('Passenger Class Distribution in Decks', size=18, y=1.05)

    plt.show()

all_deck_count, all_deck_per = get_pclass_dist(df_all_decks)
display_pclass_dist(all_deck_per)
```

Fig.16

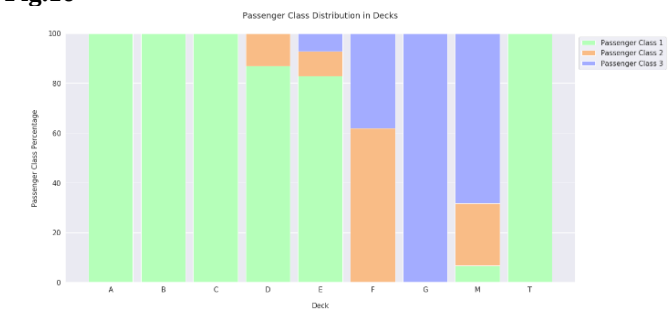


Fig.17

All passengers on decks A, B, and C are exclusively 1st class. Deck D consists of 87% 1st class passengers and 13% 2nd class passengers, while deck E hosts 83% 1st class, 10% 2nd class, and 7% 3rd class passengers. Deck F is primarily occupied by 2nd class passengers (62%), with a notable minority of 3rd class passengers (38%). Deck G exclusively accommodates 3rd class passengers. There is one passenger on the boat deck in the T cabin, who happens to be a 1st class passenger. Given the similarities between the T cabin passenger and those on deck A, the T cabin passenger is grouped with A deck passengers. Passengers labeled as "M" represent missing values in the Cabin feature. Since it's impractical to ascertain the actual deck of these passengers, "M" is treated as a distinct deck category.

```
# Passenger in the T deck is changed to A
idx = df_all[df_all['Deck'] == 'T'].index
df_all.loc[idx, 'Deck'] = 'A'
```

Fig.18

```
df_all_decks_survived = df_all.groupby(['Deck', 'Survived']).count().drop(columns=['Sex', 'Age', 'SibSp', 'Parch', 'Fare',
                                                                                   'Embarked', 'Pclass', 'Cabin', 'PassengerI
d', 'Ticket']).rename(columns={'Name':'Count'}).transpose()

def get_survived_dist(df):

    # Creating a dictionary for every survival count in every deck
    surv_counts = {'A':{}, 'B':{}, 'C':{}, 'D':{}, 'E':{}, 'F':{}, 'G':{}, 'M':{}}
    decks = df.columns.levels[0]

    for deck in decks:
        for survive in range(0, 2):
            surv_counts[deck][survive] = df[deck][survive][0]

    df_surv = pd.DataFrame(surv_counts)
    surv_percentages = {}

    for col in df_surv.columns:
        surv_percentages[col] = [(count / df_surv[col].sum()) * 100 for count in df_surv[col]]

    return surv_counts, surv_percentages

def display_surv_dist(percentages):

    df_survived_percentages = pd.DataFrame(percentages).transpose()
    deck_names = ('A', 'B', 'C', 'D', 'E', 'F', 'G', 'M')
    bar_count = np.arange(len(deck_names))
    bar_width = 0.85

    not_survived = df_survived_percentages[0]
    survived = df_survived_percentages[1]

    plt.figure(figsize=(20, 10))
    plt.bar(bar_count, not_survived, color='#b5ffb9', edgecolor='white', width=bar_width, label="Not Survived")
    plt.bar(bar_count, survived, bottom=not_survived, color='#f9bc86', edgecolor='white', width=bar_width, label="Survived")

    plt.xlabel('Deck', size=15, labelpad=20)
    plt.ylabel('Survival Percentage', size=15, labelpad=20)
    plt.xticks(bar_count, deck_names)
    plt.tick_params(axis='x', labelsize=15)
    plt.tick_params(axis='y', labelsize=15)

    plt.legend(loc='upper left', bbox_to_anchor=(1, 1), prop={'size': 15})
    plt.title('Survival Percentage in Decks', size=18, y=1.05)

    plt.show()

all_surv_count, all_surv_per = get_survived_dist(df_all_decks_survived)
display_surv_dist(all_surv_per)
```

Fig.19

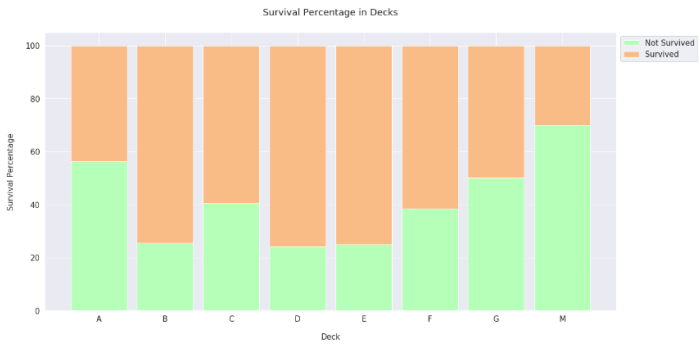


Fig.20

As I anticipated, each deck exhibits varying survival rates, which provide valuable insights that should not be disregarded. Decks B, C, D, and E demonstrate the highest survival rates, predominantly due to their occupancy by 1st class passengers. Conversely, deck M, primarily housing 2nd and 3rd class passengers, shows the lowest survival rate. This underscores the trend that cabins occupied by 1st class passengers generally have better survival rates than those of 2nd and 3rd class passengers. In my view, assigning the missing cabin values as "M" represents a logical approach. This group likely experienced the lowest survival rates due to the inability to retrieve cabin data for these individuals. Therefore, labeling them as a distinct category with common characteristics offers a sensible method for managing missing data. With the deck feature currently exhibiting high-

cardinality, some values are grouped together based on similarities, facilitating analysis of the dataset. Decks A, B, and C are designated collectively as ABC because they exclusively accommodate 1st class passengers. Similarly, decks D and E are grouped as DE due to their similar distribution of passenger classes and identical survival rates. The same principle applies to decks F and G, which are labeled as FG. However, the M deck stands out as unique and distinct from the others. It does not require grouping with other decks because it differs significantly and has the lowest survival rate. Therefore, it remains separate to allow for a clear understanding and analysis of its distinct characteristics.

```
df_all['Deck'] = df_all['Deck'].replace(['A', 'B', 'C'], 'ABC')
df_all['Deck'] = df_all['Deck'].replace(['D', 'E'], 'DE')
df_all['Deck'] = df_all['Deck'].replace(['F', 'G'], 'FG')

df_all['Deck'].value_counts()
```

Fig.21

```
M      1014
ABC     182
DE       87
FG       26
Name: Deck, dtype: int64
```

Fig.22

After completing the process of filling missing values in the Age, Embarked, Fare, and Deck features, both the training and test sets are now devoid of any missing values. The Cabin feature has been dropped from the dataset as the Deck feature serves as a replacement for it. With all missing values addressed and the dataset prepared accordingly, it is now ready for further analysis and modeling.

```
# Dropping the Cabin feature
df_all.drop(['Cabin'], inplace=True, axis=1)

df_train, df_test = divide_df(df_all)
dfs = [df_train, df_test]

for df in dfs:
    display_missing(df)
```

Fig.23

```
Age column missing values: 0
Embarked column missing values: 0
Fare column missing values: 0
Name column missing values: 0
Parch column missing values: 0
PassengerId column missing values: 0
Pclass column missing values: 0
Sex column missing values: 0
SibSp column missing values: 0
Survived column missing values: 0
Ticket column missing values: 0
Deck column missing values: 0
```

```
Age column missing values: 0
Embarked column missing values: 0
Fare column missing values: 0
Name column missing values: 0
Parch column missing values: 0
PassengerId column missing values: 0
Pclass column missing values: 0
Sex column missing values: 0
SibSp column missing values: 0
Ticket column missing values: 0
Deck column missing values: 0
```

Fig.24

1.3 Distribution of the Target Variable

In the training set, 38.38% (342 out of 891) of the instances belong to Class 1, while 61.62% (549 out of 891) belong to Class 0. This indicates an imbalance in the distribution of the target variable, with Class 0 instances being the majority.

```
342 of 891 passengers survived and it is the 38.38% of the training set.
549 of 891 passengers didnt survive and it is the 61.62% of the training set.
```

Fig.25

```
survived = df_train['Survived'].value_counts()[1]
not_survived = df_train['Survived'].value_counts()[0]
survived_per = survived / df_train.shape[0] * 100
not_survived_per = not_survived / df_train.shape[0] * 100

print('{} of {} passengers survived and it is the {:.2f}% of the training set.'.format(survived, df_train.shape[0], survived_per))
print('{} of {} passengers didnt survive and it is the {:.2f}% of the training set.'.format(not_survived, df_train.shape[0], not_survived_per))

plt.figure(figsize=(10, 8))
sns.countplot(df_train['Survived'])

plt.xlabel('Survival', size=15, labelpad=15)
plt.ylabel('Passenger Count', size=15, labelpad=15)
plt.xticks((0, 1), ['Not Survived ({0:.2f}%)'.format(not_survived_per), 'Survived ({0:.2f}%)'.format(survived_per)])
plt.tick_params(axis='x', labelsize=13)
plt.tick_params(axis='y', labelsize=13)

plt.title('Training Set Survival Distribution', size=15, y=1.05)

plt.show()
```

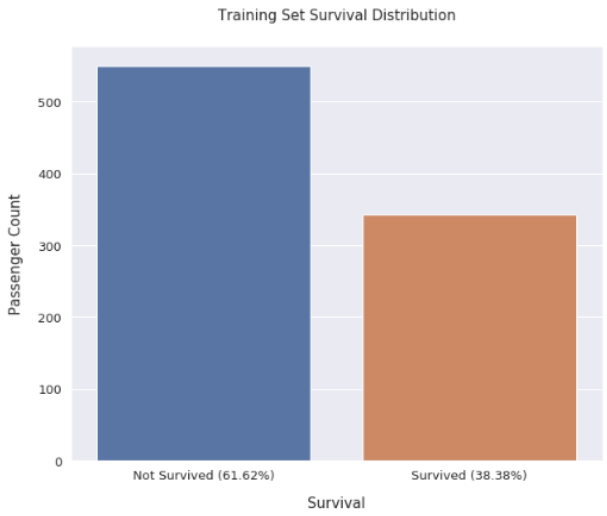


Fig.26

1.4 Correlation Analysis

The features exhibit strong correlations with each other, indicating a high degree of interdependence. The most significant correlation observed in the training set is 0.549500, while in the test set, it's 0.577147, both between Fare and Pclass. Additionally, numerous other features display considerable correlations. In the training set, there are nine correlations exceeding 0.1, while in the test set, there are six such correlations. These findings highlight the interconnectedness of the features and suggest that they influence each other's behavior within the dataset.

```
df_train_corr = df_train.drop(['PassengerId'], axis=1).corr().abs().unstack().sort_values(kind="quicksort", ascending=False).reset_index()
df_train_corr.rename(columns={"level_0": "Feature 1", "level_1": "Feature 2", 0: 'Correlation Coefficient'}, inplace=True)
df_train_corr.drop(df_train_corr.iloc[1::2].index, inplace=True)
df_train_corr_nd = df_train_corr.drop(df_train_corr[df_train_corr['Correlation Coefficient'] == 1.0].index)

df_test_corr = df_test.corr().abs().unstack().sort_values(kind="quicksort", ascending=False).reset_index()
df_test_corr.rename(columns={"level_0": "Feature 1", "level_1": "Feature 2", 0: 'Correlation Coefficient'}, inplace=True)
df_test_corr.drop(df_test_corr.iloc[1::2].index, inplace=True)
df_test_corr_nd = df_test_corr.drop(df_test_corr[df_test_corr['Correlation Coefficient'] == 1.0].index)
```

Fig.27

```
# Training set high correlations
corr = df_train_corr_nd['Correlation Coefficient'] > 0.1
df_train_corr_nd[corr]
```

Fig.28

	Feature 1	Feature 2	Correlation Coefficient
6	Pclass	Fare	0.549500
8	Pclass	Age	0.417667
10	SibSp	Parch	0.414838
12	Survived	Pclass	0.338481
14	Survived	Fare	0.257307
16	SibSp	Age	0.249747
18	Parch	Fare	0.216225
20	Age	Parch	0.176733
22	SibSp	Fare	0.159651
24	Age	Fare	0.124061

Fig.29

```
# Test set high correlations
corr = df_test_corr_nd['Correlation Coefficient'] > 0.1
df_test_corr_nd[corr]
```

Fig.30

	Feature 1	Feature 2	Correlation Coefficient
6	Fare	Pclass	0.577489
8	Age	Pclass	0.526789
10	Age	Fare	0.345347
12	SibSp	Parch	0.306895
14	Fare	Parch	0.230410
16	SibSp	Fare	0.172032

Fig.31

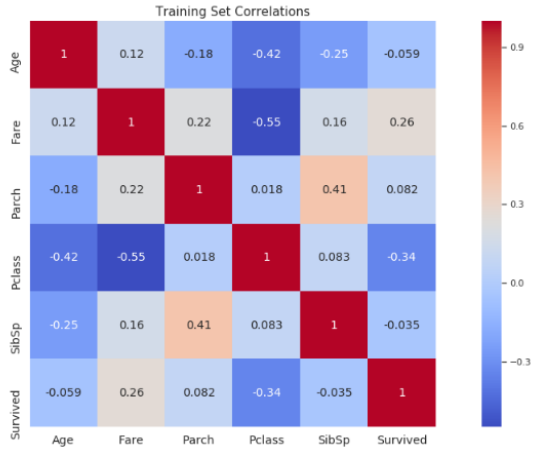


Fig.32

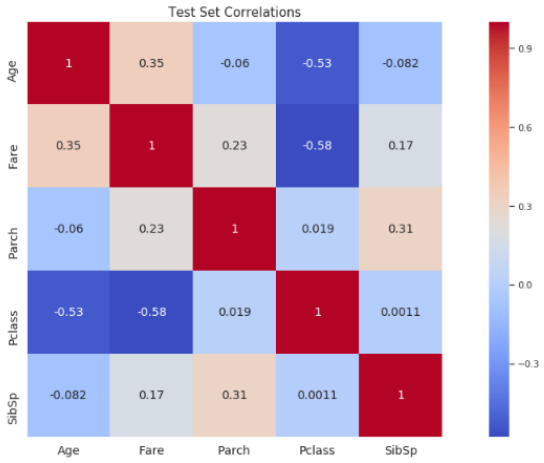


Fig.33

1.5 Distribution of the Target Variable Across Features

1.5.1 Continuous Features

The continuous features, Age and Fare, exhibit favorable characteristics for learning in a decision tree model, with discernible split points and spikes. However, a notable concern arises from the distributions: while the training set displays numerous spikes and bumps, the test set appears to have a smoother distribution. This discrepancy might pose challenges for the model's generalization to unseen data.

Analyzing the Age feature distribution reveals a higher survival rate among children under 15 compared to other age groups. This underscores the significant impact of age, particularly childhood, on survival outcomes. In the Fare feature distribution, survival rates are notably higher at the distribution tails. Additionally, the distribution displays positive skewness, primarily due to the presence of extremely large outliers. These outliers could potentially introduce bias into the model's predictions and should be carefully addressed during preprocessing and model training.

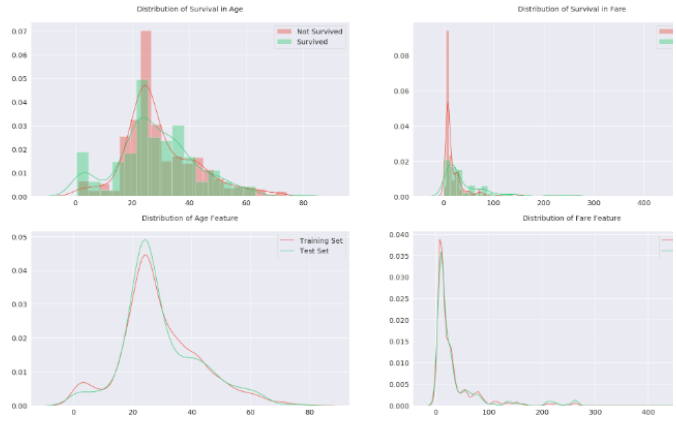


Fig.34

1.5.2 Categorical Features

Within every categorical feature, there exists at least one category demonstrating a notable mortality rate. This observation furnishes valuable insights for anticipating passenger survival outcomes. Pclass and Sex emerge as particularly informative due to their notably uniform distributions. An analysis of embarkation ports reveals that passengers from Southampton experienced lower survival rates compared to those boarding from other ports. Conversely, over half of those embarking from Cherbourg survived. This trend may be linked to differences in passenger demographics or circumstances associated with the Pclass feature. The features Parch and SibSp indicate that passengers traveling with just one family member tended to have better survival rates. This suggests that smaller family sizes might have positively influenced survival, potentially due to easier coordination or evacuation during the disaster. paraphrase more first sentence

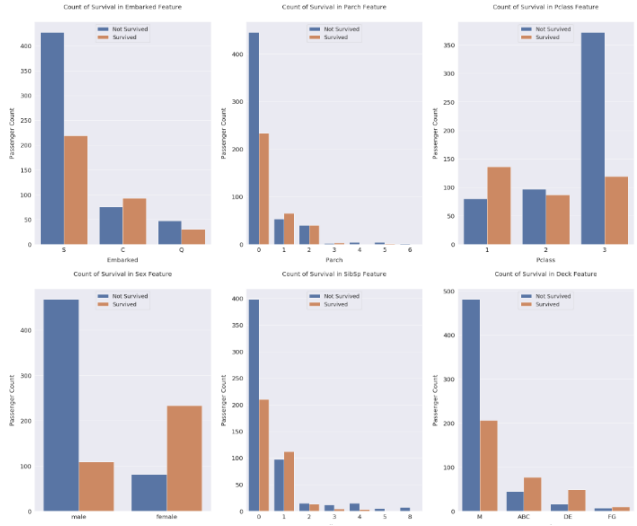


Fig.35

1.6 Conclusion

During the Exploratory Data Analysis phase, several key observations were made. Firstly, a significant level of correlation exists among the features, suggesting the potential for feature engineering through transformation and interaction. Target encoding may also prove beneficial due to the strong correlations with the Survived feature. Moreover, distinct patterns such as split points and spikes were observed in continuous features, indicating their suitability for decision tree models. However, linear models may struggle to identify these nuances. Categorical features exhibited unique distributions with varying survival rates, highlighting the potential for one-hot encoding. Additionally, combining certain categorical features could facilitate the creation of new informative features. As part of the analysis, a new feature named "Deck" was created to replace the "Cabin" feature, which was subsequently dropped. This decision was made based on the insights gleaned from the data exploration process.

```
df_all = concat_df(df_train, df_test)
df_all.head()
```

Fig.36

	Age	Deck	Embarked	Fare	Name	Parch	PassengerId	Pclass	Sex	SibSp	Survived	Ticket
0	22.0	M	S	7.2500	Braund, Mr. Owen Harris	0	1	3	male	1	0.0	A/5 21171
1	38.0	ABC	C	71.2833	Cumings, Mrs. John Bradley (Florence Briggs Th...	0	2	1	female	1	1.0	PC 17599
2	26.0	M	S	7.9250	Heikkinen, Miss. Laina	0	3	3	female	0	1.0	STON/O2. 3101282
3	35.0	ABC	S	53.1000	Futrelle, Mrs. Jacques Heath (Lily May Peel)	0	4	1	female	1	1.0	113803
4	35.0	M	S	8.0500	Allen, Mr. William Henry	0	5	3	male	0	0.0	373450

Fig.37

2. Feature Engineering

2.1 Binning Continuous Features

2.1.1 Fare

The Fare feature exhibits positive skewness, with an exceptionally high survival rate observed on the right end of the distribution. To better analyze this feature, it was divided into 13 quantile-based bins. Despite the large number of bins, they proved effective in providing substantial information gain. Notably, the groups on the left side of the graph displayed the lowest survival rates, while those on the right demonstrated the highest. Interestingly, the distribution graph did not clearly depict this high survival rate. Additionally, an unusual group in the middle range, from 15.742 to 23.25, exhibited a surprisingly high survival rate, which was captured through this binning process.

```
df_all['Fare'] = pd.qcut(df_all['Fare'], 13)
```

Fig.38

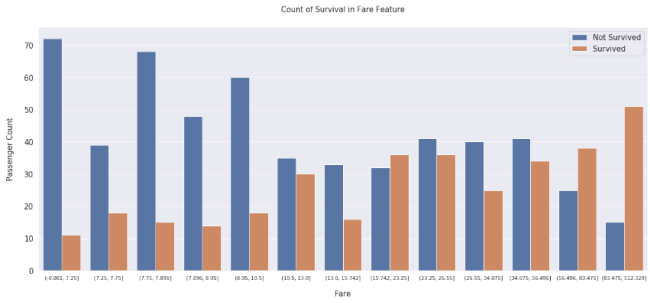


Fig.39

2.1.2 Age

The Age feature demonstrates a roughly normal distribution with occasional spikes and bumps. To analyze it more effectively, the feature was segmented into 10 quantile-based bins. Notably, the first bin exhibited the highest survival rate, while the fourth bin had the lowest. These spikes corresponded to prominent features in the distribution. Interestingly, the bin (34.0, 40.0] displayed an unexpectedly high survival rate, a pattern that was identified through this binning process.

```
df_all['Age'] = pd.qcut(df_all['Age'], 10)
```

Fig.40

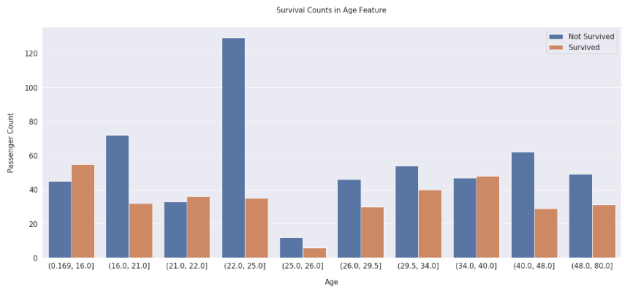


Fig.41

2.2 Frequency Encoding

The "Family_Size" feature is created by aggregating the counts of "SibSp" and "Parch", then adding 1 to account for the current passenger. This computation offers a comprehensive measure of the total family size aboard. Graphical analysis vividly demonstrates that family size acts as a predictor of survival, with varying sizes correlating to distinct survival rates. To streamline analysis, the family sizes are categorized as follows: those with a size of 1 are labeled as "Alone," sizes 2 to 4 are classified as "Small," sizes 5 and 6 are designated as "Medium," and sizes 7, 8, and 11 are categorized as "Large." These categorizations aim to simplify the interpretation of survival patterns across different family sizes, unveiling noteworthy insights into survival dynamics aboard the Titanic.

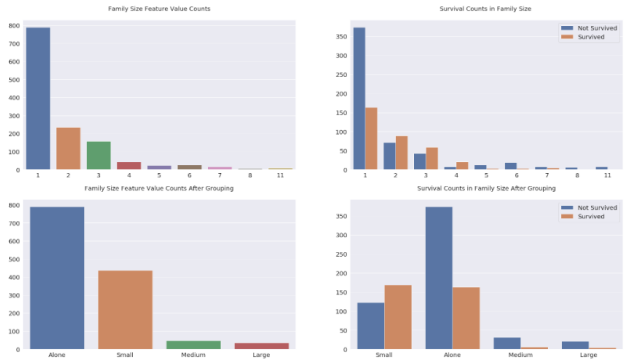


Fig.42

Grouping Ticket values by their frequencies simplifies analysis by reducing the number of unique values to handle. This approach differs from the Family_Size feature in that it captures passengers traveling together as part of a group, which may include friends, nannies, or maids, who are not necessarily counted as family but share the same ticket. Instead of grouping tickets by their prefixes, which may not convey meaningful information beyond what is already captured by features like Pclass or Embarked, Ticket values are grouped based on their frequencies. This method avoids redundant information and focuses on the frequency of ticket usage. The graph illustrates that groups with 2, 3, and 4 members tend to have higher survival rates, while solo travelers exhibit the lowest survival rate. Furthermore, survival rates decrease notably beyond groups of 4 members. While this pattern resembles that of the Family_Size feature, Ticket_Frequency values are not grouped similarly to avoid creating redundant features with perfect correlation. This

ensures that Ticket_Frequency contributes additional information gain rather than duplicating existing features.

```
df_all['Ticket_Frequency'] = df_all.groupby('Ticket')['Ticket'].transform('count')
```

Fig.43

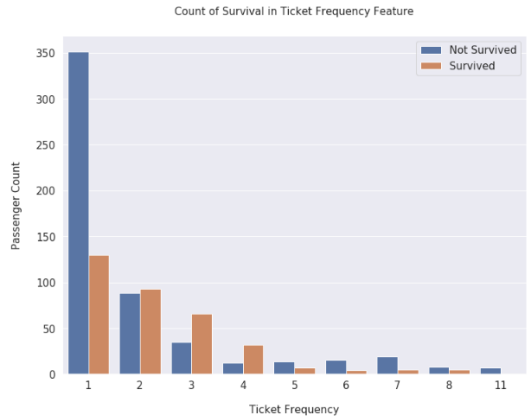


Fig.44

2.3 Title & Is Married

The Title feature is derived by extracting the prefix before the Name feature. The graph reveals numerous titles occurring infrequently, some of which appear incorrect and require replacement. Titles such as Miss, Mrs, Ms, Mlle, Lady, Mme, the Countess, and Dona are consolidated under Miss/Mrs/Ms, as they all denote female passengers. While values like Mlle, Mme, and Dona are actually names, they are classified as titles due to the comma separation in the Name feature. Titles like Dr, Col, Major, Jonkheer, Capt, Sir, Don, and Rev are grouped under Dr/Military/Noble/Clergy, as these passengers share similar characteristics. The title "Master" is unique, bestowed upon male passengers below the age of 26, who exhibit the highest survival rate among all males. The binary feature Is_Married is based on the Mrs title, which boasts the highest survival rate among female titles. This distinction is necessary as it allows for the grouping of all female titles together, facilitating a comprehensive analysis of survival rates among female passengers.

```
df_all['Title'] = df_all['Name'].str.split(', ', expand=True)[1].str.split
('.', expand=True)[0]
df_all['Is_Married'] = 0
df_all['Is_Married'].loc[df_all['Title'] == 'Mrs'] = 1
```

Fig.45

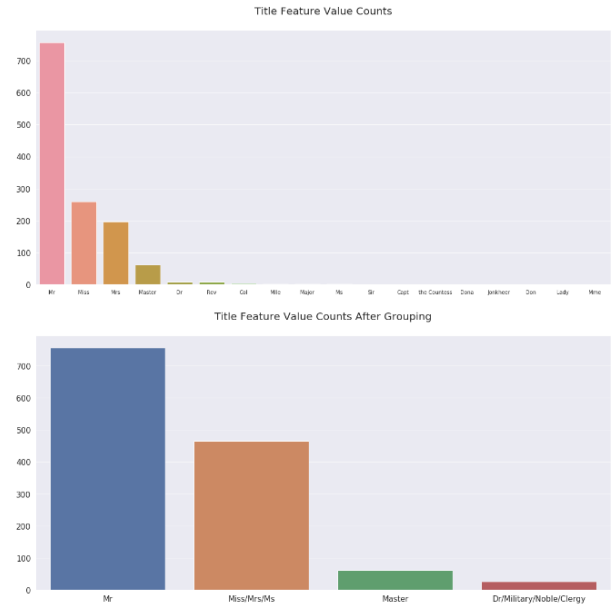


Fig.46

2.4 Target Encoding

The extract_surname function is utilized to extract the surnames of passengers from the Name feature. This information is then used to create the Family feature, which groups passengers based on their extracted surname. This step is crucial for identifying and grouping together passengers who belong to the same family unit.

```
# Creating a list of families and tickets that are occuring in both training
and test set
non_unique_families = [x for x in df_train['Family'].unique() if x in df_te
st['Family'].unique()]
non_unique_tickets = [x for x in df_train['Ticket'].unique() if x in df_tes
t['Ticket'].unique()]

df_family_survival_rate = df_train.groupby('Family')['Survived', 'Famil
y','Family_Size'].median()
df_ticket_survival_rate = df_train.groupby('Ticket')['Survived', 'Ticke
t','Ticket_Frequency'].median()

family_rates = {}
ticket_rates = {}

for i in range(len(df_family_survival_rate)):
    # Checking a family exists in both training and test set, and has members
more than 1
    if df_family_survival_rate.index[i] in non_unique_families and df_famil
y_survival_rate.iloc[i, 1] > 1:
        family_rates[df_family_survival_rate.index[i]] = df_family_survival
_rate.iloc[i, 0]

for i in range(len(df_ticket_survival_rate)):
    # Checking a ticket exists in both training and test set, and has members
more than 1
    if df_ticket_survival_rate.index[i] in non_unique_tickets and df_ticket
_survival_rate.iloc[i, 1] > 1:
        ticket_rates[df_ticket_survival_rate.index[i]] = df_ticket_survival
_rate.iloc[i, 0]
```

Fig.47

```
def extract_surname(data):

    families = []

    for i in range(len(data)):
        name = data.iloc[i]

        if '(' in name:
            name_no_bracket = name.split('(')[0]
        else:
            name_no_bracket = name

        family = name_no_bracket.split(',')[0]
        title = name_no_bracket.split(',')[1].strip().split(' ')[0]

        for c in string.punctuation:
            family = family.replace(c, '').strip()

        families.append(family)

    return families

df_all['Family'] = extract_surname(df_all['Name'])
df_train = df_all.loc[:890]
df_test = df_all.loc[891:]
dfs = [df_train, df_test]
```

Fig.48

The Family_Survival_Rate is determined by analyzing families within the training set, as the test set lacks the Survived feature. Initially, a list of family names occurring in both the training and test sets (non_unique_families) is compiled. Survival rates are then calculated for families with more than one member in this list and stored in the Family_Survival_Rate feature. Additionally, an additional binary feature, Family_Survival_Rate_NA, is created for families unique to the test set. This feature serves to indicate that the family survival rate is not applicable to these passengers due to the unavailability of their survival data. Similarly, Ticket_Survival_Rate and Ticket_Survival_Rate_NA features are generated using the same methodology. Ticket_Survival_Rate and Family_Survival_Rate are then averaged to produce Survival_Rate, while Ticket_Survival_Rate_NA and Family_Survival_Rate_NA are averaged to yield Survival_Rate_NA. This approach ensures a comprehensive assessment of survival rates while accounting for missing data in the test set.

```
mean_survival_rate = np.mean(df_train['Survived'])

train_family_survival_rate = []
train_family_survival_rate_NA = []
test_family_survival_rate = []
test_family_survival_rate_NA = []

for i in range(len(df_train)):
    if df_train['Family'][i] in family_rates:
        train_family_survival_rate.append(family_rates[df_train['Family']
[i]])
        train_family_survival_rate_NA.append(1)
    else:
        train_family_survival_rate.append(mean_survival_rate)
        train_family_survival_rate_NA.append(0)

for i in range(len(df_test)):
    if df_test['Family'].iloc[i] in family_rates:
        test_family_survival_rate.append(family_rates[df_test['Family'].ilo
c[i]])
        test_family_survival_rate_NA.append(1)
    else:
        test_family_survival_rate.append(mean_survival_rate)
        test_family_survival_rate_NA.append(0)

df_train['Family_Survival_Rate'] = train_family_survival_rate
df_train['Family_Survival_Rate_NA'] = train_family_survival_rate_NA
df_test['Family_Survival_Rate'] = test_family_survival_rate
df_test['Family_Survival_Rate_NA'] = test_family_survival_rate_NA

train_ticket_survival_rate = []
train_ticket_survival_rate_NA = []
test_ticket_survival_rate = []
test_ticket_survival_rate_NA = []
```

Fig.49

```
[i]])
        train_ticket_survival_rate_NA.append(1)
    else:
        train_ticket_survival_rate.append(mean_survival_rate)
        train_ticket_survival_rate_NA.append(0)

for i in range(len(df_test)):
    if df_test['Ticket'].iloc[i] in ticket_rates:
        test_ticket_survival_rate.append(ticket_rates[df_test['Ticket'].ilo
c[i]])
        test_ticket_survival_rate_NA.append(1)
    else:
        test_ticket_survival_rate.append(mean_survival_rate)
        test_ticket_survival_rate_NA.append(0)

df_train['Ticket_Survival_Rate'] = train_ticket_survival_rate
df_train['Ticket_Survival_Rate_NA'] = train_ticket_survival_rate_NA
df_test['Ticket_Survival_Rate'] = test_ticket_survival_rate
df_test['Ticket_Survival_Rate_NA'] = test_ticket_survival_rate_NA
```

Fig.50

2.5 Feature Transformation

2.5.1 Label Encoding Non-Numerical Features

Embarked, Sex, Deck, Title, and Family_Size_Grouped features are originally of object type, while Age and Fare are categorical types. To facilitate model learning, these features are converted to numerical types using LabelEncoder. This transformation assigns a unique numerical label to each class within the features, ranging from 0 to n. This conversion ensures that the model can effectively interpret and learn from the categorical data.

```
non_numeric_features = ['Embarked', 'Sex', 'Deck', 'Title', 'Family_Size_Gr
ouped', 'Age', 'Fare']

for df in dfs:
    for feature in non_numeric_features:
        df[feature] = LabelEncoder().fit_transform(df[feature])
```

Fig.51

2.5.2 One-Hot Encoding the Categorical Features

The categorical attributes such as Pclass, Sex, Deck, Embarked, and Title undergo conversion into one-hot encoded features using OneHotEncoder. Unlike Age and Fare, which possess ordinal properties, these features require transformation into a one-hot encoded format to represent categorical variables effectively. This process ensures that each category within the categorical features is represented by a binary indicator variable, allowing the model to interpret and learn from the categorical data accurately.

2.6 Conclusion

Age and Fare features undergo binning, which assists in handling outliers and reveals homogeneous groups within these attributes. The Family_Size feature is derived by summing the Parch and SibSp features, along with an additional count of 1. Similarly, the Ticket_Frequency feature is created by tallying the occurrences of Ticket values. The Name feature proves highly informative. Firstly, the Title and Is_Married features are generated based on the title prefixes within the names. Secondly, the Family_Survival_Rate and Family_Survival_Rate_NA features are constructed through target encoding of passenger surnames, while the Ticket_Survival_Rate is derived from target encoding the Ticket feature. The Survival_Rate feature is then computed by averaging the Family_Survival_Rate and Ticket_Survival_Rate features. Subsequently, non-numeric type features are label encoded, and categorical attributes are one-hot encoded. Five new features (Family_Size, Title, Is_Married, Survival_Rate, and Survival_Rate_NA) are introduced, while redundant features post-encoding are discarded.

3. Model

```
X_train = StandardScaler().fit_transform(df_train.drop(columns=drop_cols))
y_train = df_train['Survived'].values
X_test = StandardScaler().fit_transform(df_test.drop(columns=drop_cols))

print('X_train shape: {}'.format(X_train.shape))
print('y_train shape: {}'.format(y_train.shape))
print('X_test shape: {}'.format(X_test.shape))
```

Fig.52

```
X_train shape: (891, 26)
y_train shape: (891,)
X_test shape: (418, 26)
```

3.1 Random Forest

Two RandomForestClassifier models were created: one as a standalone model, and the other for k-fold cross-validation. The single_best_model achieved its highest accuracy of 0.82775 on the public leaderboard. However, its performance in k-fold cross-validation was not as impressive. Despite this, it serves as a solid starting point for experimentation and hyperparameter tuning. In contrast, the leaderboard_model attained its highest accuracy of 0.83732 on the public leaderboard using 5-fold cross-validation. This model is specifically tailored to optimize leaderboard scores and is intentionally tuned to slightly overfit. This overfitting is managed by dividing the estimated probabilities of X_test in each fold by N (the fold count). While this model may struggle to predict individual samples accurately if used as a standalone model, it excels in generating high leaderboard scores.

Which model is better for us?

The leaderboard_model exhibits overfitting to the test set, making it unsuitable for deployment in real-life projects. Due to its design to optimize leaderboard scores, it may not generalize well to new data and could yield inaccurate predictions when applied independently. On the other hand, the single_best_model proves to be a valuable choice for experimentation and learning purposes, particularly in understanding decision trees. Its performance on the public leaderboard, while not as high as the leaderboard_model, suggests its reliability and potential for further refinement through hyperparameter tuning and feature engineering. Thus, it serves as an excellent starting point for gaining insights into the workings of decision trees and improving model performance.

```
single_best_model = RandomForestClassifier(criterion='gini',
                                           n_estimators=1100,
                                           max_depth=5,
                                           min_samples_split=4,
                                           min_samples_leaf=5,
                                           max_features='auto',
                                           oob_score=True,
                                           random_state=SEED,
                                           n_jobs=-1,
                                           verbose=1)

leaderboard_model = RandomForestClassifier(criterion='gini',
                                           n_estimators=1750,
                                           max_depth=7,
                                           min_samples_split=6,
                                           min_samples_leaf=6,
                                           max_features='auto',
                                           oob_score=True,
                                           random_state=SEED,
                                           n_jobs=-1,
                                           verbose=1)
```

Fig.53

StratifiedKFold is employed to ensure that the target variable (Survived) is properly stratified during the creation of folds. This method maintains the proportion of samples for each class within the target variable, thereby preventing any bias or imbalance in the distribution of classes across the folds. Essentially, it ensures that each fold represents the overall distribution of classes in the dataset, allowing for more accurate evaluation and validation of the model's performance across different subsets of data.

```
N = 5
oob = 0
probs = pd.DataFrame(np.zeros((len(X_test), N * 2)), columns=['Fold_{}_Prob_{}'.format(i, j) for i
in range(1, N + 1) for j in range(2)])
importances = pd.DataFrame(np.zeros((X_train.shape[1], N)), columns=['Fold_{}'.format(i) for i in
range(1, N + 1)], index=df_all.columns)
fprs, tprs, scores = [], [], []

skf = StratifiedKFold(n_splits=N, random_state=N, shuffle=True)

for fold, (trn_idx, val_idx) in enumerate(skf.split(X_train, y_train), 1):
    print('Fold {}\n'.format(fold))

    # Fitting the model
    leaderboard_model.fit(X_train[trn_idx], y_train[trn_idx])

    # Computing Train AUC score
    trn_fpr, trn_tpr, trn_thresholds = roc_curve(y_train[trn_idx], leaderboard_model.predict_proba
(X_train[trn_idx])[:, 1])
    trn_auc_score = auc(trn_fpr, trn_tpr)
    # Computing Validation AUC score
    val_fpr, val_tpr, val_thresholds = roc_curve(y_train[val_idx], leaderboard_model.predict_proba
(X_train[val_idx])[:, 1])
    val_auc_score = auc(val_fpr, val_tpr)

    scores.append((trn_auc_score, val_auc_score))
    fprs.append(val_fpr)
    tprs.append(val_tpr)

    # X_test probabilities
    probs.loc[:, 'Fold_{}_Prob_0'.format(fold)] = leaderboard_model.predict_proba(X_test)[:, 0]
    probs.loc[:, 'Fold_{}_Prob_1'.format(fold)] = leaderboard_model.predict_proba(X_test)[:, 1]
    importances.iloc[:, fold - 1] = leaderboard_model.feature_importances_

    oob += leaderboard_model.oob_score_ / N
    print('Fold {} OOB Score: {}\n'.format(fold, leaderboard_model.oob_score_))

print('Average OOB Score: {}'.format(oob))
```

Fig.54

3.2 Feature Importance

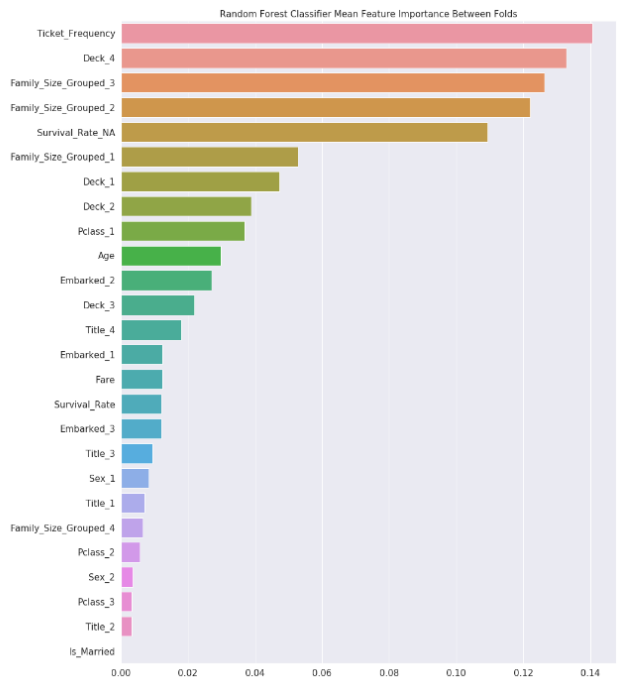


Fig.55

3.3 ROC Curve

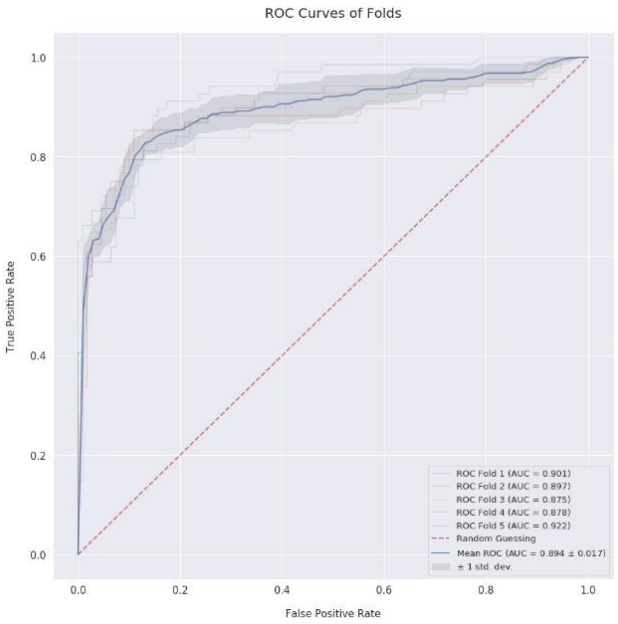


Fig.56

3.4 Submission

```
class_survived = [col for col in probs.columns if col.endswith('Prob_1')]
probs['1'] = probs[class_survived].sum(axis=1) / N
probs['0'] = probs.drop(columns=class_survived).sum(axis=1) / N
probs['pred'] = 0
pos = probs[probs['1'] >= 0.5].index
probs.loc[pos, 'pred'] = 1

y_pred = probs['pred'].astype(int)

submission_df = pd.DataFrame(columns=['PassengerId', 'Survived'])
submission_df['PassengerId'] = df_test['PassengerId']
submission_df['Survived'] = y_pred.values
submission_df.to_csv('submissions.csv', header=True, index=False)
submission_df.head(10)
```

Fig.57

	PassengerId	Survived
891	892	0
892	893	1
893	894	0
894	895	0
895	896	1
896	897	0
897	898	1
898	899	0
899	900	1
900	901	0

Fig.58

References

[2] G. Evitan, "Titanic - Advanced Feature Engineering Tutorial," [Online]. Available: https://www.kaggle.com/code/gunesevitan/titanic-advanced-feature-engineering-tutorial

Titanic Survival Predictions (Beginner)

Abdullah Hakan Şişik , Çağatay Özdemir

Contents of the coder :

- Import Necessary Libraries
- Read In and Explore The Data
- Data Analysis
- Data Visualization
- Cleaning Data
- Choosing the Best Model
- Creating Submission File

Now , let's start to inspect the code from the first part and derive conclusions.

I. Import Necessary Libraries

To start coding , some of the Python libraries should be imported. The coder imported the libraries "numpy" , "pandas" , "matplotlib.pyplot" , "seaborn" and lastly "warnings" .

II. Read In and Explore The Data

After that , the coder imports the data by assigning it to two dataframes named "train" and "test".The panda function "read_csv" reads the data from the indicated csv files and assigns to corresponding dataframes.

```
#import train and test CSV files
train = pd.read_csv("../input/train.csv")
test = pd.read_csv("../input/test.csv")

#take a look at the training data
train.describe(include="all")
```

Fig. 1

Later, describe function is applied on train dataframe.Describe function generates descriptive statistics of the dataframe's columns such as count , frequency ,min , max mean etc. "include='all' " parameter specifies that the output should include information about all the columns , whether they are numeric or not.The numeric informations belonging to non-numeric values are described as NaN as it can be seen from the table.

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare
count	891.000000	891.000000	891.000000	891	891	714.000000	891.000000	891.000000	891	891.000(
unique	NaN	NaN	NaN	891	2	NaN	NaN	NaN	681	NaN
top	NaN	NaN	NaN	Carlsson, Mr. August Sigfrid	male	NaN	NaN	NaN	1601	NaN
freq	NaN	NaN	NaN	1	577	NaN	NaN	NaN	7	NaN
mean	446.000000	0.383838	2.308642	NaN	NaN	29.699118	0.523008	0.381594	NaN	32.2042(
std	257.353842	0.486592	0.836071	NaN	NaN	14.526497	1.102743	0.806057	NaN	49.6934;
min	1.000000	0.000000	1.000000	NaN	NaN	0.420000	0.000000	0.000000	NaN	0.000000
25%	223.500000	0.000000	2.000000	NaN	NaN	20.125000	0.000000	0.000000	NaN	7.91040(
50%	446.000000	0.000000	3.000000	NaN	NaN	28.000000	0.000000	0.000000	NaN	14.4542(
75%	668.500000	1.000000	3.000000	NaN	NaN	38.000000	1.000000	0.000000	NaN	31.00000
max	891.000000	1.000000	3.000000	NaN	NaN	80.000000	8.000000	6.000000	NaN	512.329;

Fig. 2

Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
714.000000	891.000000	891.000000	891	891.000000	204	889
NaN	NaN	NaN	681	NaN	147	3
NaN	NaN	NaN	1601	NaN	G6	S
NaN	NaN	NaN	7	NaN	4	644
29.699118	0.523008	0.381594	NaN	32.204208	NaN	NaN
14.526497	1.102743	0.806057	NaN	49.693429	NaN	NaN
0.420000	0.000000	0.000000	NaN	0.000000	NaN	NaN
20.125000	0.000000	0.000000	NaN	7.910400	NaN	NaN
28.000000	0.000000	0.000000	NaN	14.454200	NaN	NaN
38.000000	1.000000	0.000000	NaN	31.000000	NaN	NaN
80.000000	8.000000	6.000000	NaN	512.329200	NaN	NaN

Fig.3

Describe function is useful for gaining a first insight into the distribution and characteristics of the "train" data. Now , it is time to analyze the data from the table created by describing the train dataframe.This two table will be indicated as "described train dataframe table" from now on in this text when referred.

III. Data Analysis

The columns of the train data are : "PassangerID" , "Survived" , "Pclass" , "Name" , "Sex" , "Age" , " SibSp" , "Parch" , "Ticket" , "Fare" , "Cabin" , "Embarked". After checking the columns , 5 random sample is being taken from the train dataframe to get a grasp on how the values are defined in each column.Depending on the obtained samples , some of the problems in the columns can be seen easily such as missing age values etc.

```
#see a sample of the dataset to get an idea of the variables
train.sample(5)
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
572	573	1	1	Flynn, Mr. John Irwin ("Irving")	male	36.0	0	0	PC 17474	26.3875	E25	S
717	718	1	2	Troutt, Miss. Edwina Celia "Winnie"	female	27.0	0	0	34218	10.5000	E101	S
141	142	1	3	Nysten, Miss. Anna Sofia	female	22.0	0	0	347081	7.7500	NaN	S
226	227	1	2	Mellors, Mr. William John	male	19.0	0	0	SW/PP 751	10.5000	NaN	S
648	649	0	3	Willey, Mr. Edward	male	NaN	0	0	S.O./P.P. 751	7.5500	NaN	S

Fig. 4

As it can be seen , from the collected 5 samples , 3 of them missing a Cabin value. While this can be also predicted from the described train dataframe table by checking counts , it can be also seen from the samples that they are defined as NaN.

Inspecting the samples ; there are 4 numerical features , 4 categorical features and 2 alphanumeric features.Numerical features are Age (Continuous), Fare (Continuous), SibSp (Discrete), Parch (Discrete).Categorical features are Survived, Sex, Embarked, Pclass . Alphanumeric features are Ticket and Cabin .

The data types for each feature are :

- Survived: int
- Pclass: int
- Name: string
- Sex: string
- Age: float
- SibSp: int
- Parch: int
- Ticket: string
- Fare: float
- Cabin: string
- Embarked: string

As it can be seen from the described train dataframe table , there are total of 891 passangers in our training set. around 19.8% of the values in the "Age" feature are missing in your dataset. Given the importance of age in determining survival, it's important to address these missing values appropriately so the coder decides to fill these gaps.

The “Cabin” column missing 687 values which corresponds to %77.1 of the count it should have.I think that since so much values are missing in this column , trying to fill these gaps by using only %22.9 of its values would create more error in the analyziz than it would help.Thus , it should not be taken into account. But the coder decides to what to do with this column later on.

The Embarked column is only missing 2 values which is ignorable due to the fact that it is relatively small.

```
#check for any other unusable values
print(pd.isnull(train).sum())
```

```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

This code finds the undefined values in the train dataframe and calculates the count of missing values for each column , then prints the count of the missing values according to the related columns.It can be seen that the missing values are exactly as I and the coder inspected.

Fig.5

After these parts , the coder makes some predictions to analyze the data by relating the results to the predictions he made.Data visualization will be used for this.The predictions made by the coder are :

- Sex: Females are more likely to survive.
- SibSp/Parch: People traveling alone are more likely to survive.
- Age: Young children are more likely to survive.
- Pclass: People of higher socioeconomic class are more likely to survive.

IV. Data Visualization

a) Sex Feature

```
#draw a bar plot of survival by sex
sns.barplot(x="Sex", y="Survived", data=train)

#print percentages of females vs. males that survive
print("Percentage of females who survived:", train["Survived"][train["Sex"] == 'female'].value_counts(normalize = True)[1]*100)

print("Percentage of males who survived:", train["Survived"][train["Sex"] == 'male'].value_counts(normalize = True)[1]*100)
```

```
Percentage of females who survived: 74.20382165605095
Percentage of males who survived: 18.890814558058924
```

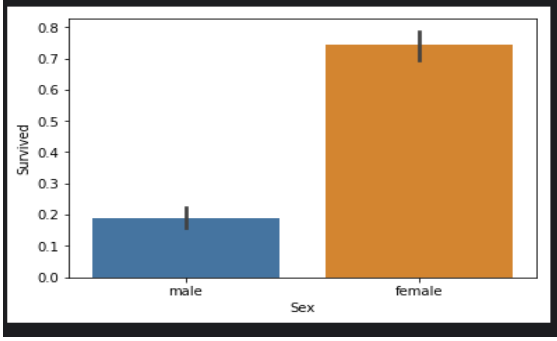


Fig.6

This code creates a bar plot that shows the relation between two parameters , these parameters are the normalized value of the count of survived people and their genders.Firstly , The code filters the DataFrame to only include entries where "Sex" is "female" or "male" ; then, it calculates the percentage of survived individuals within each gender group by dividing the count of survivors by the total count of individuals in that gender group, multiplying by 100 ; after that it prints these percentages as bar plot. Overall, the code visually displays the survival rates by gender and provides the percentages of females and males who survived. As it is printed , the survival rate of females and males are approximately %74.2 and %18.89 respectively .Just like the coder predicted , the gender is decisive parameter relating to the survival of the individual.

b) Pclass Feature

```
#draw a bar plot for SibSp vs. survival
sns.barplot(x="SibSp", y="Survived", data=train)

#I won't be printing individual percent values for all of these.
print("Percentage of SibSp = 0 who survived:", train["Survived"][train["SibSp"] == 0].value_counts
(normalize = True)[1]*100)

print("Percentage of SibSp = 1 who survived:", train["Survived"][train["SibSp"] == 1].value_counts
(normalize = True)[1]*100)

print("Percentage of SibSp = 2 who survived:", train["Survived"][train["SibSp"] == 2].value_counts
(normalize = True)[1]*100)
```

```
Percentage of SibSp = 0 who survived: 34.53947368421053
Percentage of SibSp = 1 who survived: 53.588516746411486
Percentage of SibSp = 2 who survived: 46.42857142857143
```

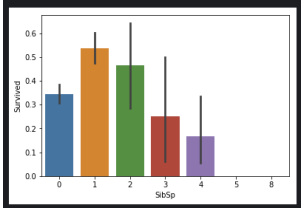


Fig.7

The way this code work is the same as the code in “Sex Feature” , only the parameters are changed but I will still explain it briefly.This code, firstly, draws a bar plot showing the relationship between "Survived" and "Pclass" , i.e. passenger class, using seaborn's barplot function.Then , it filters the DataFrame to only include entries where "Pclass" is 1, 2, or 3. For each passenger class, it calculates the percentage of survivors within that class by dividing the count of survivors by the total count of individuals in that class, then multiplying by 100.After that , it prints these percentage as columns with different colours with their respective passanger class (Pclass) and values.

To put it simply , the code visually displays the survival rates by passenger class and provides the percentages of passengers in each class who survived.

The coder concludes , as it can be seen from the bar plot , people with better Pclass - which indicates having better economical or social status – had higher survival chance. Pclass 1 has the highest surviving rate with a rate of %62.9 , then Pclass 2 has the highest with a percentage of %47.3 . Lowest survival rate belongs to the Pclass 3 that contains the people with lowest socioeconomic position with a rate of %24.2.

c) SibSP Feature

Since the code is same with only the parameters used are changed and also operates in the same way as before , I will just explain the code briefly . The code visually displays the relationship between the number of siblings/spouses , i.e. SibSP, and “Survived” and provides insights into the survival rates for different values of "SibSp".

```
#draw a bar plot for SibSp vs. survival
sns.barplot(x="SibSp", y="Survived", data=train)

#I won't be printing individual percent values for all of these.
print("Percentage of SibSp = 0 who survived:", train["Survived"][train["SibSp"] == 0].value_counts
(normalize = True)[1]*100)

print("Percentage of SibSp = 1 who survived:", train["Survived"][train["SibSp"] == 1].value_counts
(normalize = True)[1]*100)

print("Percentage of SibSp = 2 who survived:", train["Survived"][train["SibSp"] == 2].value_counts
(normalize = True)[1]*100)
```

```
Percentage of SibSp = 0 who survived: 34.53947368421053
Percentage of SibSp = 1 who survived: 53.588516746411486
Percentage of SibSp = 2 who survived: 46.42857142857143
```

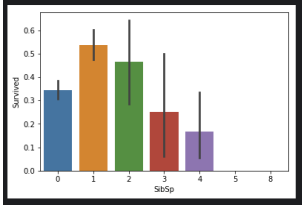


Fig.8

As it can be seen , the only difference in the bar plot is that the parameters are now “Survival” on the y-axis and “SibSP” in the x-axis , rest of the code works the same as before.

Inspecting this data , it can be seen the data suggests that individuals with more siblings or spouses onboard had lower survival rates. But , unlike the coder’s predictions , people with no siblings or spouses had lower survival rates compared to the ones with one or two. The percentages are %34.5 , %53.4 , %46.4 for the cases in which SibSp = {0 , 1 , 2} respectively.

d) Parch Feature

```
#draw a bar plot for Parch vs. survival
sns.barplot(x="Parch", y="Survived", data=train)
plt.show()
```

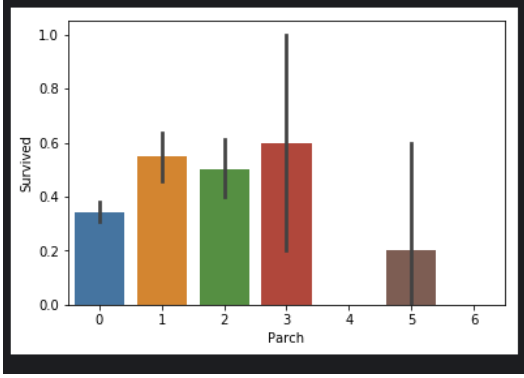


Fig.9

Parch means the number of parents/children aboard. It should be noted that the “/” does not mean divide in this text.

Individuals with fewer than four parents or children onboard are more likely to survive than those with four or more. Similarly, passengers traveling alone have lower survival rates compared to those accompanied by 1-3 parents or children.

e) Age Feature

Again, the same type of code is used to create the bar plot so I will just explain the code briefly.This time , the survival rate is categorized based on the different ranges of individual's ages as a bar plot.

```
#sort the ages into logical categories
train["Age"] = train["Age"].fillna(-0.5)
test["Age"] = test["Age"].fillna(-0.5)
bins = [-1, 0, 5, 12, 18, 24, 35, 60, np.inf]
labels = ['Unknown', 'Baby', 'Child', 'Teenager', 'Student', 'Young Adult', 'Adult', 'Senior']
train['AgeGroup'] = pd.cut(train["Age"], bins, labels = labels)
test['AgeGroup'] = pd.cut(test["Age"], bins, labels = labels)

#draw a bar plot of Age vs. survival
sns.barplot(x="AgeGroup", y="Survived", data=train)
plt.show()
```

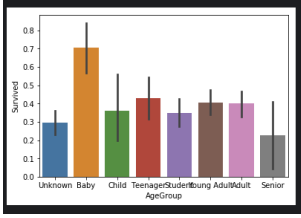


Fig.10

The age ranges in this bar plot are :

- Unknown: Age not available (-1 to 0)
- Baby: 0 to 5 years old
- Child: 5 to 12 years old
- Teenager: 12 to 18 years old
- Student: 18 to 24 years old
- Young Adult: 24 to 35 years old
- Adult: 35 to 60 years old
- Senior: 60 years old and above

Observing the graph , babies have the most survival rate while seniors have the worst survival rate.

V. Cabin Feature

The concept explained by the coder here is that individuals with documented cabin numbers likely belong to a higher socioeconomic class, potentially increasing their chances of survival.

```
train["CabinBool"] = (train["Cabin"].notnull().astype('int'))
test["CabinBool"] = (test["Cabin"].notnull().astype('int'))

#calculate percentages of CabinBool vs. survived
print("Percentage of CabinBool = 1 who survived:", train["Survived"][train["CabinBool"] == 1].valu
e_counts(normalize = True)[1]*100)

print("Percentage of CabinBool = 0 who survived:", train["Survived"][train["CabinBool"] == 0].valu
e_counts(normalize = True)[1]*100)
#draw a bar plot of CabinBool vs. survival
sns.barplot(x="CabinBool", y="Survived", data=train)
plt.show()
```

```
Percentage of CabinBool = 1 who survived: 66.66666666666666
Percentage of CabinBool = 0 who survived: 29.985443959243085
```

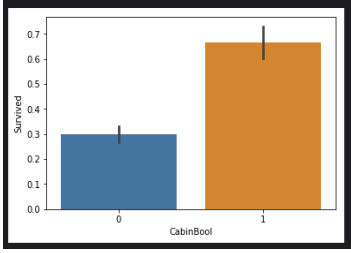


Fig.11

The code creates the plot bar as CabinBool vs Survived . The code creates a new binary feature called "CabinBool" in both the training and test datasets. This feature indicates whether a cabin number is recorded for each passenger. It assigns "1" if a cabin number is present (not null) and 0 otherwise.Then , it calculates the percentage of survivors based on the presence or absence of a recorded cabin number by doing the followings : It filters the DataFrame to only include entries where "CabinBool" is 1 or 0.After that , for each category of "CabinBool" (1 or 0), it calculates the percentage of survivors by dividing the count of survivors by the total count of individuals in that category, then multiplying by 100 and prints the percentages.After all these , it creates a bar plot that shows the relation between "Survived" and "CabinBool.

To put it simply , the code investigates whether the presence of a recorded cabin number relates to survival rates and then visualizes this relationship through a bar plot.

It can be seen that the individuals whose cabin numbers recorded are more likely to survive from bar plot.The percentage of survival for those whose cabin numbers are recorded is approximately %66.67 while it is approximately %29.98 for those whose cabin numbers are not recorded. The survival rate of the ones whose cabin numbers recorded are a bit higher than twice of the ones whose cabin numbers number are not recorded.

VI. Cleaning Data

In this part ,the coder tidies up the dataset to handle missing values and remove any irrelevant information.

First , he describes the test dataframe.The described values are given in the tables below:

```
test.describe(include="all")
```

	PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin
count	418.000000	418.000000	418	418	418.000000	418.000000	418.000000	418	417.000000	91
unique	NaN	NaN	418	2	NaN	NaN	NaN	363	NaN	76
top	NaN	NaN	Davidson, Mrs. Thornton (Orian Hays)	male	NaN	NaN	NaN	PC 17608	NaN	B57 B59 B63 B66
freq	NaN	NaN	1	266	NaN	NaN	NaN	5	NaN	3
mean	1100.500000	2.265550	NaN	NaN	23.941388	0.447368	0.392344	NaN	35.627188	NaN
std	120.810458	0.841838	NaN	NaN	17.741080	0.896760	0.981429	NaN	55.907576	NaN
min	892.000000	1.000000	NaN	NaN	-0.500000	0.000000	0.000000	NaN	0.000000	NaN
25%	996.250000	1.000000	NaN	NaN	9.000000	0.000000	0.000000	NaN	7.895800	NaN
50%	1100.500000	3.000000	NaN	NaN	24.000000	0.000000	0.000000	NaN	14.454200	NaN
75%	1204.750000	3.000000	NaN	NaN	35.750000	1.000000	0.000000	NaN	31.500000	NaN
max	1309.000000	3.000000	NaN	NaN	76.000000	8.000000	9.000000	NaN	512.329200	NaN

Fig.13

Parch	Ticket	Fare	Cabin	Embarked	AgeGroup	CabinBool
418.000000	418	417.000000	91	418	418	418.000000
NaN	363	NaN	76	3	8	NaN
NaN	PC 17608	NaN	B57 B59 B63 B66	S	Young Adult	NaN
NaN	5	NaN	3	270	96	NaN
0.392344	NaN	35.627188	NaN	NaN	NaN	0.217703
0.981429	NaN	55.907576	NaN	NaN	NaN	0.413179
0.000000	NaN	0.000000	NaN	NaN	NaN	0.000000
0.000000	NaN	7.895800	NaN	NaN	NaN	0.000000
0.000000	NaN	14.454200	NaN	NaN	NaN	0.000000
0.000000	NaN	31.500000	NaN	NaN	NaN	0.000000
9.000000	NaN	512.329200	NaN	NaN	NaN	1.000000

Fig.14

Inspecting the test dataframe , some inferencecan be made.There are 418 passangers in the test dataframe. Out of the 418 passengers in the test dataset, two features have missing information. One entry in the "Fare" feature is absent, while approximately 20.5% of the “Age” feature lacks data. To assure a better examination , addressing these missing values is essential for the analysis.

a) Cabin Feature

```
#we'll start off by dropping the Cabin feature since not a lot more useful information can be extrac
ted from it.
train = train.drop(['Cabin'], axis = 1)
test = test.drop(['Cabin'], axis = 1)
```

Fig.15

As it is explained in the psuedo-code , this part of the code drops “Cabin” column from both train and test dataframes and explains the reason for this is that there are not much use to keep them in the dataframes anymore.

b) Ticket Feature

```
#we can also drop the Ticket feature since it's unlikel
y to yield any useful information
train = train.drop(['Ticket'], axis = 1)
test = test.drop(['Ticket'], axis = 1)
```

Fig.16

This code drops the Ticket column from train and test dataframes due to the same reasons indicated in “Cabin Feature”.

c) Embarked Feature

```
#now we need to fill in the missing values in the Embar
ked feature
print("Number of people embarking in Southampton
(S):")
southampton = train[train["Embarked"] == "S"].shape[0]
print(southampton)

print("Number of people embarking in Cherbourg (C):")
cherbourg = train[train["Embarked"] == "C"].shape[0]
print(cherbourg)

print("Number of people embarking in Queenstown (Q):")
queenstown = train[train["Embarked"] == "Q"].shape[0]
print(queenstown)
```

```
Number of people embarking in Southampton (S):
644
Number of people embarking in Cherbourg (C):
168
Number of people embarking in Queenstown (Q):
77
```

Fig.17

This code counts and prints the number of passengers embarking from each of the three ports: Southampton (S), Cherbourg (C), and Queenstown (Q). It filters the dataset based on the port of embarkation and calculates the count of passengers for each port. This information helps in understanding the distribution of passengers across different embarkation points.

Observing the results , it is clear that most of the people embarked the ship in Southampton (S) so the missing values in Embarked column will be replaced by Southampton .This is due to the fact that it is most likely that the individuals whose Embark feature missing embarked the ship in Southampton port.This is done by the following code.

```
#replacing the missing values in the Embarked feature w
ith S
train = train.fillna({"Embarked": "S"})
```

Fig.18

d) Age Featur

The coder now decides to address the missing values in the "Age" feature. Given the relatively high percentage of missing values, it wouldn't be reasonable to fill them all with a single

value, as it was done with the "Embarked" feature. Instead, another method will be explored to predict these missing ages.

```
#create a combined group of both datasets
combine = [train, test]

#extract a title for each Name in the train and test datasets
for dataset in combine:
    dataset['Title'] = dataset.Name.str.extract(' ([A-Za-z]+)\.', expand=False)

pd.crosstab(train['Title'], train['Sex'])
```

Fig.19

This code generates a cross-tabulation (also known as a contingency table) between the "Title" and "Sex" columns in the training dataset as it is seen from the table.

Fig.20

Sex	female	male
Title		
Capt	0	1
Col	0	2
Countess	1	0
Don	0	1
Dr	1	6
Jonkheer	0	1
Lady	1	0
Major	0	2
Master	0	40
Miss	182	0
Mlle	2	0
Mme	1	0
Mr	0	517
Mrs	125	0
Ms	1	0
Rev	0	6
Sir	0	1

The loop iterates over each dataset (train and test) in the combine list. For each dataset, it extracts titles from the “Name” column using a regular expression pattern “ ([A-Za-z]+)\. ” . The extracted titles are stored in a new column named "Title" in each dataset. “ pd.crosstab(train['Title'], train['Sex']) ” generates a cross-tabulation table using the pandas crosstab function. It calculates the frequency distribution of titles by sex in the training dataset. Each row represents a unique title, each column represents a sex category, and the values in the table represent the count of occurrences where a specific title corresponds to a specific sex.

```
#replace various titles with more common names
for dataset in combine:
    dataset['Title'] = dataset['Title'].replace(['Lady', 'Capt', 'Col',
    'Don', 'Dr', 'Major', 'Rev', 'Jonkheer', 'Dona'], 'Rare')

    dataset['Title'] = dataset['Title'].replace(['Countess', 'Lady', 'Sir'], 'Royal')
    dataset['Title'] = dataset['Title'].replace('Mlle', 'Miss')
    dataset['Title'] = dataset['Title'].replace('Ms', 'Miss')
    dataset['Title'] = dataset['Title'].replace('Mme', 'Mrs')

train[['Title', 'Survived']].groupby(['Title'], as_index=False).mean()
```

	Title	Survived
0	Master	0.575000
1	Miss	0.702703
2	Mr	0.156673
3	Mrs	0.793651
4	Rare	0.285714
5	Royal	1.000000

Fig.21

This code replaces various titles in the dataset with more common names or categories.Titles such as “Lady”, “Capt”, “Col”, etc., are replaced with the label “Rare”. Titles like “Countess”, “Lady”, and “Sir” are grouped under the category “Royal”. Titles “Mlle” and “Ms” are replaced with “Miss”, and “Mme” is replaced with “Mrs”. Finally, it calculates the mean survival rate for each title group in the training dataset, grouping by the “Title” column. This provides insight into how survival rates vary across different titles. It is observed that “Royal” category has the most survival rate ; actually , it is seen that all of the “Royal” category survived. “Mr” category has the least survival rate.Note that the survival rates are normalized values.

```
#map each of the title groups to a numerical value
title_mapping = {"Mr": 1, "Miss": 2, "Mrs": 3, "Master": 4, "Royal": 5, "Rare": 6}
for dataset in combine:
    dataset['Title'] = dataset['Title'].map(title_mapping)
    dataset['Title'] = dataset['Title'].fillna(0)

train.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Fare	Embarked	AgeGroup	CabinBool	Title
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	7.2500	S	Student	0	1
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	71.2833	C	Adult	1	3
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	7.9250	S	Young Adult	0	2
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	53.1000	S	Young Adult	1	3
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	8.0500	S	Young Adult	0	1

Fig.22

This code is creating a numerical mapping for each of the title groups and then applying this mapping to the “Title” column in each dataset within the combine list. Title_mapping is a dictionary that maps each title group to a numerical value. The loop iterates through each dataset in the combine list. For each dataset, it maps the values in the 'Title' column according to the title_mapping dictionary. If there are any missing values after mapping, it fills them with 0. Finally, it displays the first

few rows of the train dataset after these transformations with the head function.

In the subsequent step, the coder aim to predict the unknown age values by substituting them with the most prevalent age corresponding to their respective title group. The code he used to do this is given below.

```
# fill missing age with mode age group for each title
mr_age = train[train["Title"] == 1]["AgeGroup"].mode() #Young Adult
miss_age = train[train["Title"] == 2]["AgeGroup"].mode() #Student
mrs_age = train[train["Title"] == 3]["AgeGroup"].mode() #Adult
master_age = train[train["Title"] == 4]["AgeGroup"].mode() #Baby
royal_age = train[train["Title"] == 5]["AgeGroup"].mode() #Adult
rare_age = train[train["Title"] == 6]["AgeGroup"].mode() #Adult

age_title_mapping = {1: "Young Adult", 2: "Student", 3: "Adult", 4: "Baby", 5: "Adult", 6: "Adul
t"}

#I tried to get this code to work with using .map(), but couldn't.
#I've put down a less elegant, temporary solution for now.
#train = train.fillna({"Age": train["Title"].map(age_title_mapping)})
#test = test.fillna({"Age": test["Title"].map(age_title_mapping)})

for x in range(len(train["AgeGroup"])):
    if train["AgeGroup"][x] == "Unknown":
        train["AgeGroup"][x] = age_title_mapping[train["Title"][x]]

for x in range(len(test["AgeGroup"])):
    if test["AgeGroup"][x] == "Unknown":
        test["AgeGroup"][x] = age_title_mapping[test["Title"][x]]
```

Fig.23

This code segment is aimed at filling in missing age values in the dataset by assigning each missing age the mode (most common) age group corresponding to its respective title group. It first calculates the mode age group for each title group, then creates a mapping between title groups and their respective mode age groups. Subsequently, it iterates through the dataset rows and fills in missing age values with the mode age group for the corresponding title group. Finally, it applies this procedure to both the training and test datasets. However, instead of using a direct mapping approach, it uses a loop to iterate through the rows and assigns the appropriate age group based on the title.

After filling the missing values , the age groups are mapped to the numerical values with the next segment of code:

```
#map each Age value to a numerical value
age_mapping = {'Baby': 1, 'Child': 2, 'Teenager': 3, 'Student': 4, 'Young Adult': 5, 'Adult': 6,
'Senior': 7}
train['AgeGroup'] = train['AgeGroup'].map(age_mapping)
test['AgeGroup'] = test['AgeGroup'].map(age_mapping)

train.head()

#dropping the Age feature for now, might change
train = train.drop(['Age'], axis = 1)
test = test.drop(['Age'], axis = 1)
```

Fig.24

e) Name Feature

Name feature are dropped from train and test dataframes since the titles are already extracted from these.

```
#drop the name feature since it contains no more useful information.
train = train.drop(['Name'], axis = 1)
test = test.drop(['Name'], axis = 1)
```

Fig.25

f) Sex Feature

Genders are mapped as numerical values.Males are mapped with zero and females are mapped with one.

```
#map each Sex value to a numerical value
sex_mapping = {"male": 0, "female": 1}
train['Sex'] = train['Sex'].map(sex_mapping)
test['Sex'] = test['Sex'].map(sex_mapping)

train.head()
```

	PassengerId	Survived	Pclass	Sex	SibSp	Parch	Fare	Embarked	AgeGroup	CabinBool	Title
0	1	0	3	0	1	0	7.2500	S	4	0	1
1	2	1	1	1	1	0	71.2833	C	6	1	3
2	3	1	3	1	0	0	7.9250	S	5	0	2
3	4	1	1	1	1	0	53.1000	S	5	1	3
4	5	0	3	0	0	0	8.0500	S	5	0	1

Fig.26

As it can be seen from train dataframe , genders are now indicated with numerical values.

g) Embarked Feature

Embarked ports are indicated with numerical values as well. S is indicated with “1” , C is indicated with “2” and Q is indicated with “3”.

```
#map each Embarked value to a numerical value
embarked_mapping = {"S": 1, "C": 2, "Q": 3}
train['Embarked'] = train['Embarked'].map(embarked_mapping)
test['Embarked'] = test['Embarked'].map(embarked_mapping)

train.head()
```

	PassengerId	Survived	Pclass	Sex	SibSp	Parch	Fare	Embarked	AgeGroup	CabinBool	Title
0	1	0	3	0	1	0	7.2500	1	4	0	1
1	2	1	1	1	1	0	71.2833	2	6	1	3
2	3	1	3	1	0	0	7.9250	1	5	0	2
3	4	1	1	1	1	0	53.1000	1	5	1	3
4	5	0	3	0	0	0	8.0500	1	5	0	1

Fig.27

h) Fare Feature

Now, fare values will be categorized into meaningful groups and handle the lone missing value in the test dataset.

```
#fill in missing Fare value in test set based on mean fare for that
for x in range(len(test["Fare"])):
    if pd.isnull(test["Fare"][x]):
        pclass = test["Pclass"][x] #Pclass = 3
        test["Fare"][x] = round(train[train["Pclass"] == pclass][

#map Fare values into groups of numerical values
train['FareBand'] = pd.qcut(train['Fare'], 4, labels = [1, 2, 3,
test['FareBand'] = pd.qcut(test['Fare'], 4, labels = [1, 2, 3, 4]

#drop Fare values
train = train.drop(['Fare'], axis = 1)
test = test.drop(['Fare'], axis = 1)
```

Fig.28

The code first identifies missing Fare values in the test dataset and replaces them with the mean Fare value for the corresponding Pclass (passenger class) from the training dataset. Then, it divides the Fare values into four groups based on quartiles using the pd.qcut function and assigns numerical labels (1 to 4) to each group. These labels represent different fare bands. Finally, it drops the original Fare column from both the training and test datasets since the fare information is now represented by the FareBand feature.
After this , the first 5 row of the train data and test data are printed by using .head function.The first 5 row of the train data is:

	PassengerId	Survived	Pclass	Sex	SibSp	Parch	Emb
0	1	0	3	0	1	0	1
1	2	1	1	1	1	0	2
2	3	1	3	1	0	0	1
3	4	1	1	1	1	0	1
4	5	0	3	0	0	0	1

Fig.29

The first 5 row of the test data is:

	PassengerId	Pclass	Sex	SibSp	Parch	Embarked
0	892	3	0	0	0	3
1	893	3	1	1	0	1
2	894	2	0	0	0	3
3	895	3	0	0	0	1
4	896	3	1	1	1	1

Fig.30

VII. CHOOSING THE BEST MODEL

```
from sklearn.model_selection import train_test_split

predictors = train.drop(['Survived', 'PassengerId'], axis=1)
target = train["Survived"]
x_train, x_val, y_train, y_val = train_test_split(predictors, target, test_size = 0.22, random_s
```

Fig.31

First of all , the data is split between training data and testing data.” Predictors” is defines by droppnig the Survived and PassengerID columns from the train dataset.
“target” is defined as a series that contains the values in the Survived from the train dataset.
x_train, x_val, y_train, y_val = train_test_split(predictors, target, test_size = 0.22, random_state = 0) : this part of the code splits the “predictor” and “target” variables into training and validation sets that are defined as x_train , x_val , y_train , y_val .

Following model types are used to make predictions to predict the passengers’ survival: Gaussian naive vayes , logistic regression , support vector machines , perceptron , decision tree classifier , random forest classifier , k-nearest neighbors , stochastic gradient descent , gradient boosting classifier .

For each model followings are made : first of all , model is defined. Then models are trained by using 80% of the training data.After that models make predictions for the remaining 20% of the training data, and at last , accuracy of the predictions are evaluated.

Accuracy of GaussianNaiveBayes

```
# Gaussian Naive Bayes
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

gaussian = GaussianNB()
gaussian.fit(x_train, y_train)
y_pred = gaussian.predict(x_val)
acc_gaussian = round(accuracy_score(y_pred, y_val) * 100, 2)
print(acc_gaussian)
```

```
78.68
```

Fig.32

Accuracy of Logistic Regression

```
# Logistic Regression
from sklearn.linear_model import LogisticRegression

logreg = LogisticRegression()
logreg.fit(x_train, y_train)
y_pred = logreg.predict(x_val)
acc_logreg = round(accuracy_score(y_pred, y_val) * 100, 2)
print(acc_logreg)
```

```
79.19
```

Fig.33

Accuracy of Support Vector Machines

```
# Support Vector Machines
from sklearn.svm import SVC

svc = SVC()
svc.fit(x_train, y_train)
y_pred = svc.predict(x_val)
acc_svc = round(accuracy_score(y_pred, y_val) * 100, 2)
print(acc_svc)
```

```
82.74
```

Fig.34

Accuracy of Linear SVC

```
# Linear SVC
from sklearn.svm import LinearSVC

linear_svc = LinearSVC()
linear_svc.fit(x_train, y_train)
y_pred = linear_svc.predict(x_val)
acc_linear_svc = round(accuracy_score(y_pred, y_val) * 100, 2)
print(acc_linear_svc)
```

```
78.68
```

Fig.35

Accuracy of Perceptron

```
# Perceptron
from sklearn.linear_model import Perceptron

perceptron = Perceptron()
perceptron.fit(x_train, y_train)
y_pred = perceptron.predict(x_val)
acc_perceptron = round(accuracy_score(y_pred, y_val) * 100, 2)
print(acc_perceptron)
```

```
79.19
```

Fig.36

Accuracy of Decision Tree

```
#Decision Tree
from sklearn.tree import DecisionTreeClassifier

decisiontree = DecisionTreeClassifier()
decisiontree.fit(x_train, y_train)
y_pred = decisiontree.predict(x_val)
acc_decisiontree = round(accuracy_score(y_pred, y_val) * 100, 2
print(acc_decisiontree)
```

```
80.71
```

Fig.37

Accuracy of Random Forest

```
# Random Forest
from sklearn.ensemble import RandomForestClassifier

randomforest = RandomForestClassifier()
randomforest.fit(x_train, y_train)
y_pred = randomforest.predict(x_val)
acc_randomforest = round(accuracy_score(y_pred, y_val) * 100, 2
print(acc_randomforest)
```

```
81.22
```

Fig.38

Accuracy of KNN

```
# KNN or k-Nearest Neighbors
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier()
knn.fit(x_train, y_train)
y_pred = knn.predict(x_val)
acc_knn = round(accuracy_score(y_pred, y_val) * 100, 2)
print(acc_knn)
```

```
77.66
```

Fig.39

Accuracy of Stochastic Gradient Descent

```
# Stochastic Gradient Descent
from sklearn.linear_model import SGDClassifier

sgd = SGDClassifier()
sgd.fit(x_train, y_train)
y_pred = sgd.predict(x_val)
acc_sgd = round(accuracy_score(y_pred, y_val) * 100, 2)
print(acc_sgd)
```

```
80.2
```

Fig.40

Accuracy of Gradient Boosting Classifier

```
# Gradient Boosting Classifier
from sklearn.ensemble import GradientBoostingClassifier

gbk = GradientBoostingClassifier()
gbk.fit(x_train, y_train)
y_pred = gbk.predict(x_val)
acc_gbk = round(accuracy_score(y_pred, y_val) * 100, 2)
print(acc_gbk)
```

```
84.77
```

Fig.41

Table for each of the models used is as follows :

	Model	Score
9	Gradient Boosting Classifier	84.77
0	Support Vector Machines	82.74
3	Random Forest	81.22
7	Decision Tree	80.71
8	Stochastic Gradient Descent	80.20
2	Logistic Regression	79.19
5	Perceptron	79.19
4	Naive Bayes	78.68
6	Linear SVC	78.68
1	KNN	77.66

Fig.42

As it can be seen , best prediction is done by gradient boosting classifier with a percentage of %84.77.

REFERENCES

[3] N. Tamer, "Titanic Survival Predictions (Beginner)," [Online]. Available: https://www.kaggle.com/code/nadintamer/titanic-survival-predictions-beginner

COMPARISON AND EVALUATION

1. Depth of EDA and Feature Engineering

- EDA To Prediction (DieTanic): Strong in initial data exploration and visualization, identifying correlations and trends. Detailed handling of missing data and feature transformations.
- Second Approach: Equally strong in handling missing data with logical imputation methods and extensive feature transformation using encoding techniques.
- Third Approach: Focused more on visualizing feature impacts on survival, with clear bar plots showing insights.

2. Predictive Modeling and Performance

- EDA To Prediction (DieTanic): Utilized a broad range of algorithms with cross-validation, ensuring robustness in model evaluation. However, specific performance metrics for each model were not detailed.
- Second Approach: Provided a comprehensive set of models and detailed performance metrics. Gradient Boosting Classifier was highlighted as the best-performing model, showing the highest accuracy.
- Third Approach: Offered a detailed breakdown of model accuracies, allowing for a clear comparison of model performance. This transparency in results helps in understanding which models were more effective.

Best Approach

The second approach stands out as the best due to its:
- Comprehensive handling of missing data with logical and statistically sound imputation.
- Extensive feature engineering, including label and one-hot encoding.
- Detailed performance evaluation of multiple models, with Gradient Boosting Classifier achieving the highest accuracy, providing a clear, data-driven justification for its choice.

Conclusion

While all three approaches are robust and well-documented, the second approach excels in both the depth of preprocessing and clarity of model evaluation, making it the most effective and reliable method for predicting survival on the Titanic dataset.