
**FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY**

**SEMESTER 1 SESSION 2021/2022**

**BITI 2223 MACHINE LEARNING**

# PROJECT REPORT:
# DECISION TREE ALGORITHM

**LECTURER : PROFESSOR DR AZAH KAMILAH BINTI DRAMAN @ MUDA**

**PREPARED BY:**

| NO | NAME | MATRIC NO | SECTION / GROUP |
|---|---|---|---|
| 1. | NUR FAEZAH BINTI ABDUL RAUF | B032010282 | 2 BITI S1G1 |
| 2. | NURSYAZA NISA BINTI ARFARIZAL | B032010244 | 2 BITI S1G1 |
| 3. | NUR SAFINAH BINTI ZAINAL | B032010080 | 2 BITI S1G1 |

# TABLE OF CONTENTS

**ABSTRACT**

Decision Tree is an algorithm under supervised learning and a nonlinear classification model. This algorithm is not only suitable to be used for solving regression problems, but classification problems as well. By using Decision Tree, it holds the goal of creating a training model that can be used to predict the class or value of the target variable. To achieve this, the algorithm learns simple decision rules inferred from prior data that has been trained. Basically, the Decision Tree starts from the root of the tree to predict a class label for a record. By comparing the values of the root attribute with the record's attribute, it will then follow the branch according to the value and jump to the next node.The benefit of Decision Tree is that it can be easily visualised. At each node, we know which feature is being used to decide the splits. Even though that is the case, the disadvantage of Decision Tree is that it is a greedy process, and will not perform as well as some of the other models.

## INTRODUCTION

In this section, we will be giving the scope of project description and its overview of everything included in this mini project report.

### 1. PURPOSE

The purpose of writing this report is to present a detailed explanation of applying the Decision Tree algorithm on the Titanic Dataset. In this report, we will be presenting the knowledge we gained throughout this course. This includes visualising the decision tree as a result. This report is solely for the sake of presenting our mini project for the Machine Learning course.

### 2. OVERVIEW

In this project, we will be using the training titanic dataset to explain the details of using the Decision Tree algorithm in determining whether the passenger in the Titanic Dataset will survive the crash or not. The training titanic passenger survival dataset contains 891 rows which will be used to develop a high performance predictive mode.

### 3. PROJECT BACKGROUND

This project provides information for anyone who desires to learn ways in retrieving csv file, how to drop and add features to the dataset, identifying the numerical and categorical variables, identifying missing data and replacing missing data. This project also shows methods on transforming (mapping) of categorical variables into equivalent numerical values, transforming variable ranges into discrete bins, correlation between all features and survival and correlation between each feature (Heat map). Besides, feature engineering, 10 predictive models, model evaluation and selection, parameter tuning for the selected model, KFold cross-validation for the selected model and variable importance for the proposed model will be presented as well. Last but not least, the prediction of the target value (Survival of the test data), and submitting the final results will also be included. This dataset is chosen as it is simple and thus, allowing us to focus on using classification algorithms.

## OBJECTIVE

This project embarks on the following objectives:

1. To apply the Decision Tree technique on the dataset of a machine learning model.
2. To illustrate the concept of Decision Tree algorithm on the dataset.
3. To improve the performance of the Decision Tree Algorithm on the dataset of machine learning models.

## EXPLORATORY DATA ANALYSIS

### ➔ Data Loading and Description of Data

The dataset already consists of the information about people boarding the famous Titanic ship. It consists of various variables includes data of age, sex, fare, ticket and others.This dataset comprises 891 observations of 12 columns. Below are the details about the variables that show names of all columns and their description.

- PassengerId - the unique id of the row and it does not have any effect on the target variable.
- Survived - the target variable we are trying to predict (0 or 1):
    - 1 = Survived
    - 0 = Not Survived
- Pclass (Passenger Class) - the socio-economic status of the passenger and it is a categorical ordinal feature which has three unique values (1, 2 or 3):
    - 1 = Upper Class
    - 2 = Middle Class
    - 3 = Lower Class
- Name, Sex and Age - self-explanatory
- SibSp - the total number of the passengers' siblings and spouse
- Parch - the total number of the passengers' parents and children
- Ticket - the ticket number of the passenger
- Fare - the passenger fare
- Cabin - the cabin number of the passenger
- Embarked - port of embarkation, categorical feature with three unique values (C, Q or S):
    - C = Cherbourg
    - Q = Queenstown
    - S = Southampton

Import the Packages

```
# Implements milti-dimensional array and matrices
import numpy as np
# For data manipulation and analysis
import pandas as pd

# Plotting library for Python programming language and it's numerical mathematics extension NumPy
import matplotlib.pyplot as plt
# Provides a high level interface for drawing attractive and informative statistical graphics
import seaborn as sns
%matplotlib inline
sns.set(style="darkgrid")


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
```

Import the Titanic Dataset

```
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("https://raw.githubusercontent.com/insaid2018/Term-1/master/Data/Casestudy/titanic_train.csv")
df_test = pd.read_csv("https://raw.githubusercontent.com/insaid2018/Term-1/master/Data/Casestudy/titanic_train.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)
```

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

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

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

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

Info of the Titanic dataset

```
print(df_train.info())
df_train.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   PassengerId  891 non-null    int64  
 1   Survived     891 non-null    int64  
 2   Pclass       891 non-null    int64  
 3   Name         891 non-null    object 
 4   Sex          891 non-null    object 
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64  
 7   Parch        891 non-null    int64  
 8   Ticket       891 non-null    object 
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object 
 11  Embarked     889 non-null    object 
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
None
```

| | 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 |

```
print(df_test.info())
df_test.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   PassengerId  891 non-null    int64  
 1   Survived     891 non-null    int64  
 2   Pclass       891 non-null    int64  
 3   Name         891 non-null    object 
 4   Sex          891 non-null    object 
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64  
 7   Parch        891 non-null    int64  
 8   Ticket       891 non-null    object 
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object 
 11  Embarked     889 non-null    object 
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
None
```

| | 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 |

➔ **Preprocessing of Data**

**Dealing with Missing Values**

```
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)
```

```
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
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
```

As we can see, there are some missing values present. ‘display_missing’ function shows the count of missing values in every column in both training and test set. Therefore we will,

- Dropping/Replacing missing entries of Embarked.
- Replacing missing values of Age with median values.
- Dropping the column 'Cabin' as it has too many null values.
- Replacing 0 values of fare with median values.

➔ Age

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

| | Feature 1 | Feature 2 | Correlation Coefficient |
|---|---|---|---|
| 6 | Age | Age | 1.000000 |
| 12 | Age | Pclass | 0.369226 |
| 15 | Age | SibSp | 0.308247 |
| 22 | Age | Parch | 0.189119 |
| 26 | Age | Fare | 0.096067 |
| 31 | Age | Survived | 0.077221 |
| 35 | Age | PassengerId | 0.036847 |

As we see, the median Age of Pclass feature is the best choice because of its high correlation with Age (0.369226). In order to be more accurate, Sex feature is used as the second level of group by while filling the missing Age values.

```
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[sex][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.median()))
```

```
Median age of Pclass 1 females: 35.0
Median age of Pclass 1 males: 40.0
Median age of Pclass 2 females: 28.0
Median age of Pclass 2 males: 30.0
Median age of Pclass 3 females: 21.5
Median age of Pclass 3 males: 25.0
Median age of all passengers: 28.0
```

As seen from output, Pclass and Sex groups have distinct median Age values. When passenger class increases, the median age for both males and females also increases. However, females tend to have a slightly lower median Age than males. The median ages below are used for filling the missing values in Age feature.

➔ Embarked

Embarked is a categorical feature and there are only 2 missing values in the whole data set. Both of those passengers are female, upper class and they have the same ticket number. This means that they know each other and embarked from the same port together. The mode Embarked value for an upper class female passenger is C (Cherbourg), but this doesn't necessarily mean that they embarked from that port.

```
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 | 113572 |
| **829** | 62.0 | B28 | NaN | 80.0 | Stone, Mrs. George Nelson (Martha Evelyn) | 0 | 830 | 1 | female | 0 | 1 | 113572 |
| **952** | 38.0 | B28 | NaN | 80.0 | Icard, Miss. Amelie | 0 | 62 | 1 | female | 0 | 1 | 113572 |
| **1720** | 62.0 | B28 | NaN | 80.0 | Stone, Mrs. George Nelson (Martha Evelyn) | 0 | 830 | 1 | female | 0 | 1 | 113572 |

Missing values in Embarked are filled with S with this information.

```
# Filling the missing values in Embarked with S
df_all['Embarked'] = df_all['Embarked'].fillna('S')
```

➔ Cabin

Cabin feature is a little bit tricky and it needs further exploration. The large portion of the Cabin feature is missing and the feature itself can't be ignored completely because some cabins might have higher survival rates. It turns out to be the first letter of the Cabin values are the decks in which the cabins are located. Those decks were mainly separated for one passenger class, but some of them were used by multiple passenger classes.

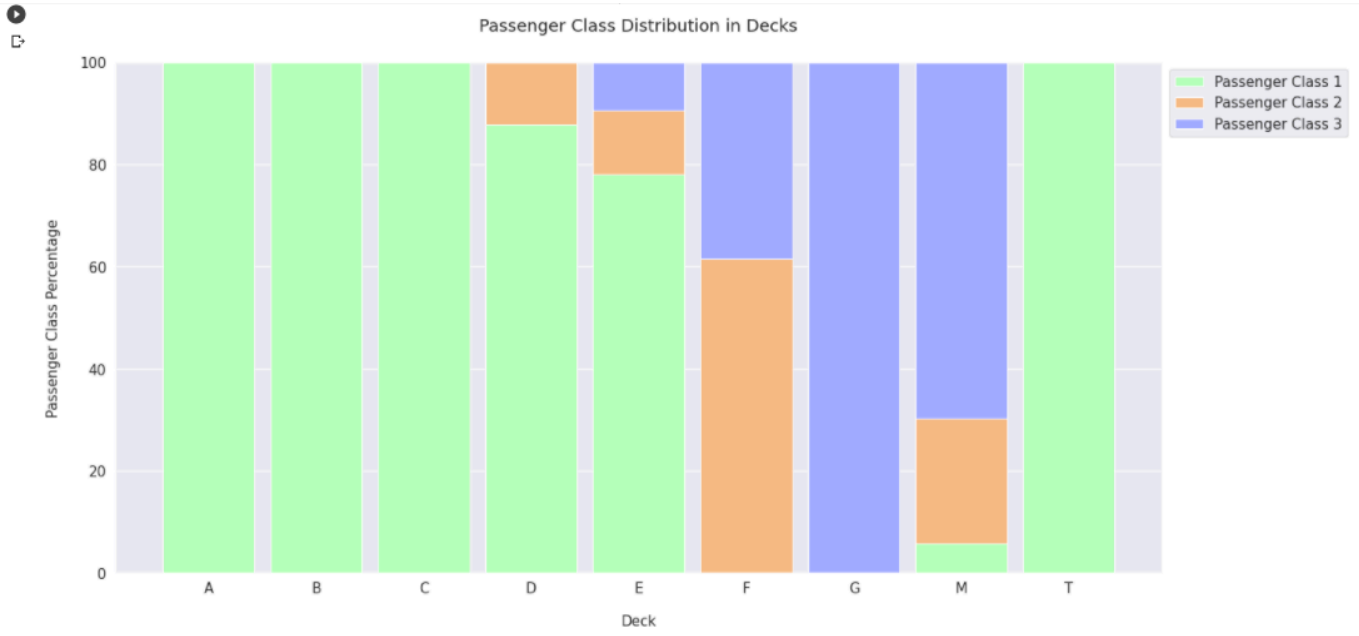
- On the Boat Deck there were 6 rooms labelled as T, U, W, X, Y, Z but only the T cabin is present in the dataset
- A, B and C decks were only for 1st class passengers
- D and E decks were for all classes
- F and G decks were for both 2nd and 3rd class passengers

From going A to G, distance to the staircase increases which might be a factor of survival.

```
# 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', 'Ticket']).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='Passenger 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)
```

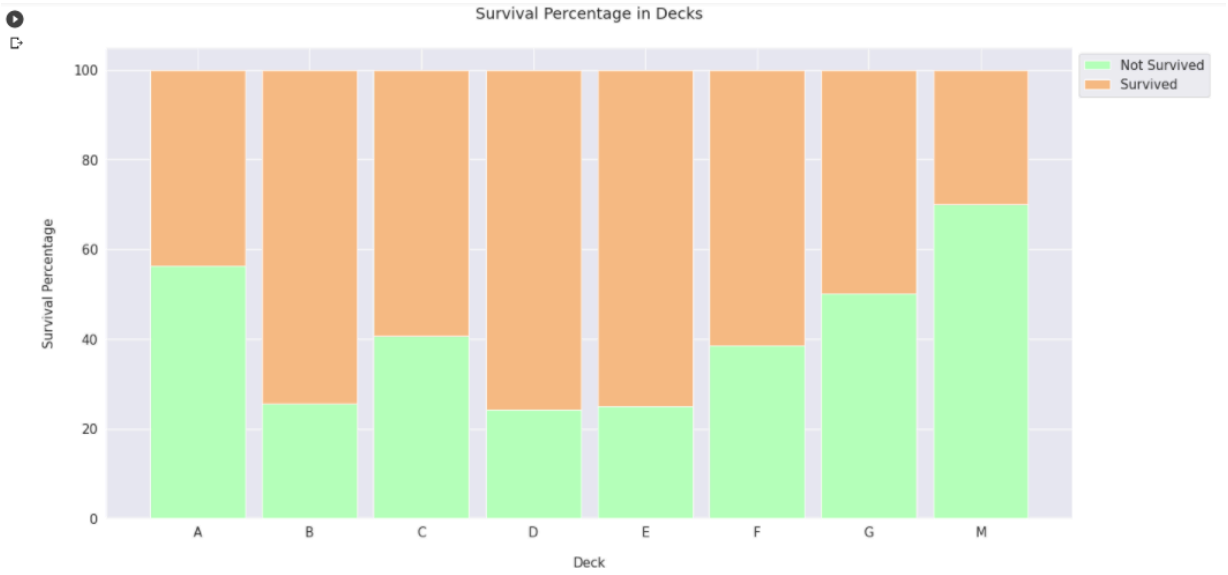



- 100% of A, B and C decks are 1st class passengers
- Deck D has 87% 1st class and 13% 2nd class passenger
- Deck E has 83% 1st class, 10% 2nd class and 7% 3rd class passengers
- Deck F has 62% 2nd class and 38% 3rd class passengers
- 100% of G deck are 3rd class passengers
- There is one person on the boat deck in T cabin and he is a 1st class passenger. The cabin passenger has the closest resemblance to A deck passengers so he is grouped with A deck.
- Passengers labelled as M are the missing values in the Cabin feature. It is impossible to find those passengers' real Deck so we use M like a deck.

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

```
df_all_decks_survived = df_all.groupby(['Deck', 'Survived']).count().drop(columns=['Sex', 'Age', 'SibSp', 'Parch', 'Fare',
                                                                                   'Embarked', 'Pclass', 'Cabin', 'PassengerId',
                                                                                   '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)
```




As we see that every deck has different survival rates and that information can't be discarded. Deck B, C, D and E have the highest survival rates.

Those decks are mostly occupied by 1st class passengers. M has the lowest survival rate which is mostly occupied by 2nd and 3rd class passengers. To conclude, cabins used by 1st class passengers have higher survival rates than cabins used by 2nd and 3rd class passengers. In my opinion M (Missing Cabin values) has the lowest survival rate because they couldn't retrieve the cabin data of the victims.

That's why,labelling that group as M is a reasonable way to handle the missing data. It is a unique group with shared characteristics. Deck feature has high-cardinality right now so some of the values are grouped with each other based on their similarities.

- A, B and C decks are labelled as ABC because all of them have only 1st class passengers
- D and E decks are labelled as DE because both of them have similar passenger class distribution and same survival rate
- F and G decks are labelled as FG because of the same reason above
- M deck doesn't need to be grouped with other decks because it is very different from others and has the lowest survival rate.

```
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()
```

```
M      1374
ABC     244
DE      130
FG       34
Name: Deck, dtype: int64
```

After filling the missing values in Age, Embarked, Fare and Deck features, there is no missing value left in both training and test set. The Cabin is dropped because the Deck feature is used instead of it.

```
# 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)
```

```
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
```

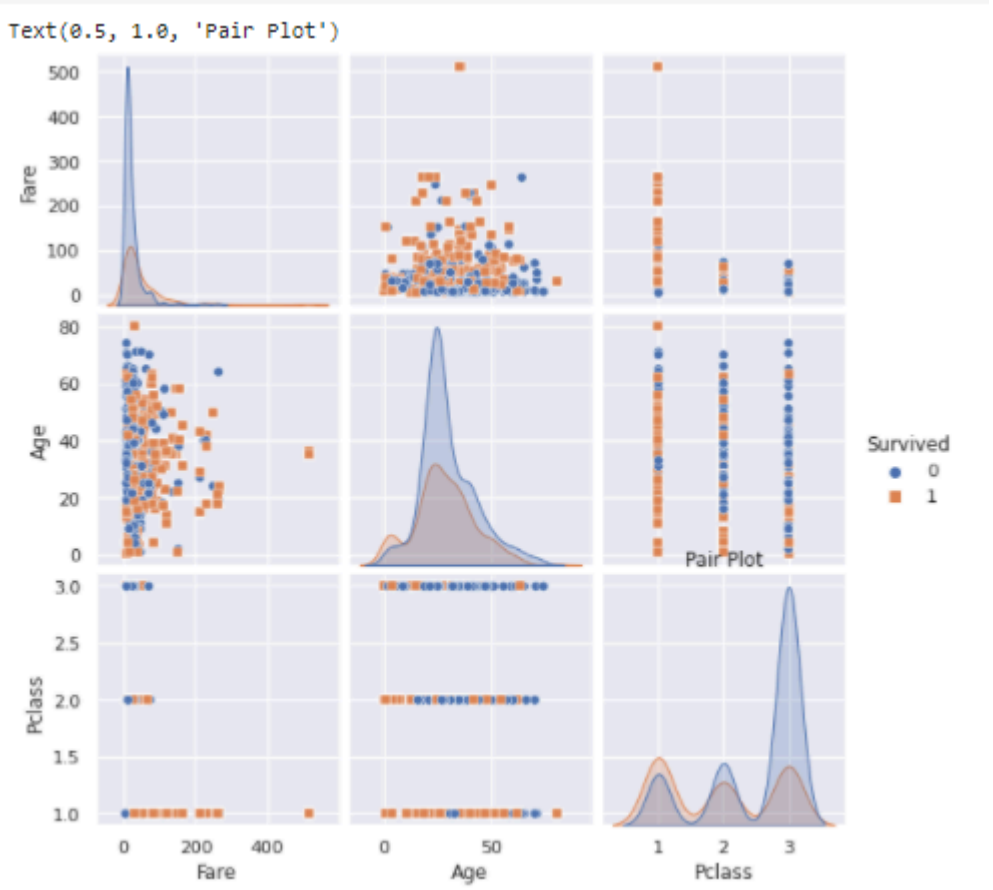
➔ Fare

```
df_all['Fare']=df_all['Fare'].replace(0,df_all['Fare'].median())
```

Draw the pair plot to know the joint relationship between 'Fare' , 'Age' , 'Pclass' & 'Survived'

```
sns.pairplot(df_all[["Fare","Age","Pclass","Survived"]],vars = ["Fare","Age","Pclass"],
             hue="Survived", dropna=True,markers=["o", "s"])
plt.title('Pair Plot')
```

```
Text(0.5, 1.0, 'Pair Plot')
```




From the observation of diagonal elements, we know that,

- More people of Pclass 1 survived than died (First peak of red is higher than blue)
- More people of Pclass 3 died than survived (Third peak of blue is higher than red)
- More people of age group 20-40 died than survived.
- Most of the people paying less fare died.

## Correlations

We establish the correlation of training and test set

```
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)
```

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

| | Feature 1 | Feature 2 | Correlation Coefficient |
|---|---|---|---|
| 6 | Pclass | Fare | 0.553071 |
| 8 | SibSp | Parch | 0.414838 |
| 10 | Pclass | Age | 0.413583 |
| 12 | Survived | Pclass | 0.338481 |
| 14 | Survived | Fare | 0.254743 |
| 16 | SibSp | Age | 0.249854 |
| 18 | Parch | Fare | 0.214436 |
| 20 | Age | Parch | 0.175526 |
| 22 | SibSp | Fare | 0.157717 |
| 24 | Age | Fare | 0.124704 |

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

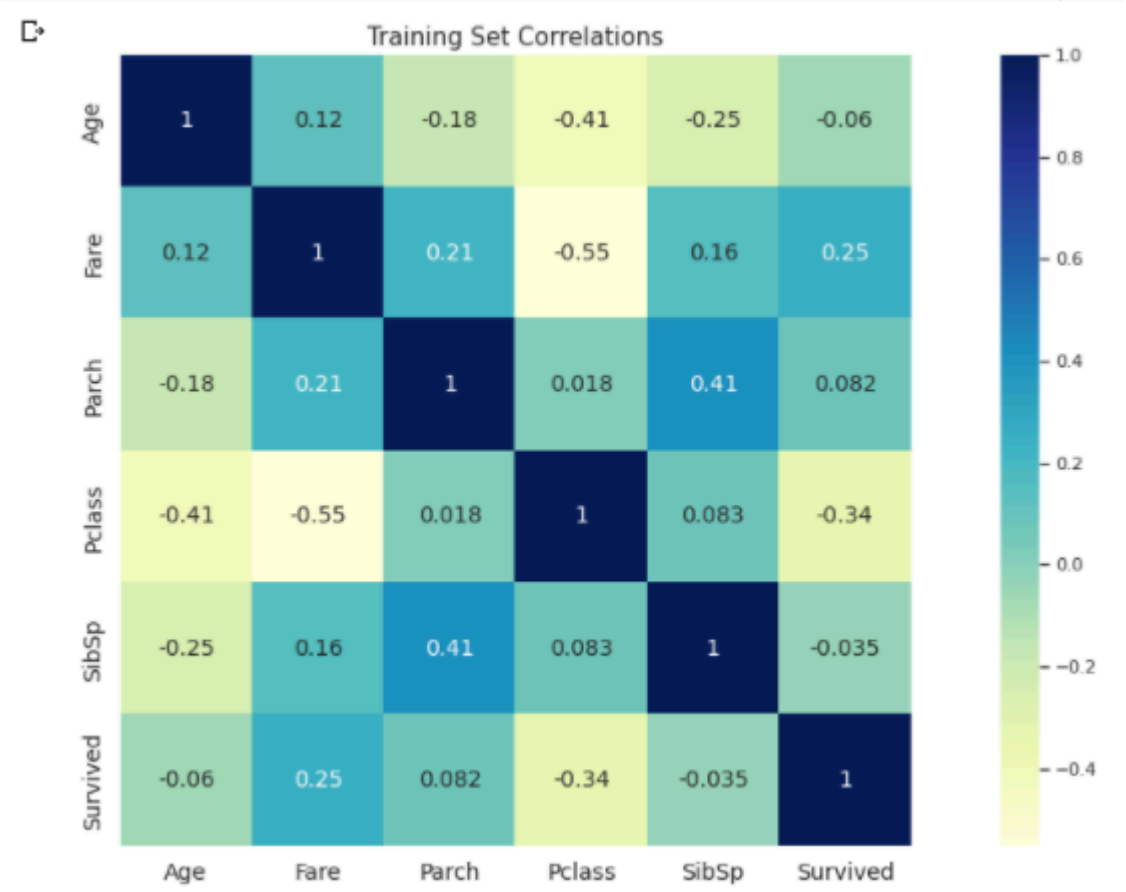
| | Feature 1 | Feature 2 | Correlation Coefficient |
|---|---|---|---|
| 6 | Pclass | Fare | 0.549500 |
| 8 | SibSp | Parch | 0.414838 |
| 10 | Age | Pclass | 0.413583 |
| 12 | Age | SibSp | 0.249854 |
| 14 | Parch | Fare | 0.216225 |
| 16 | Parch | Age | 0.175526 |
| 18 | SibSp | Fare | 0.159651 |
| 20 | Age | Fare | 0.122692 |

Observing the correlation

- Features are highly correlated with each other and dependent on each other.
- The highest correlation between features is 0.549500 in the training set and test set (between Fare and Pclass).
- The other features are also highly correlated. There are 9 correlations in the training set and 6 correlations in the test set that are higher than 0.1.
- There are 9 correlations in the training set and 6 correlations in the test set that are higher than 0.1.

Then, we establish them using heatmap

```
fig, axs = plt.subplots(nrows=2, figsize=(20, 20))

sns.heatmap(df_train.drop(['PassengerId'], axis=1).corr(), ax=axs[0], annot=True,
            square=True, cmap='YlGnBu', linecolor ='black', annot_kws={'size': 14})
sns.heatmap(df_test.drop(['PassengerId'], axis=1).corr(), ax=axs[1], annot=True,
            square=True, cmap='YlGnBu', linecolor ='black', annot_kws={'size': 14})

for i in range(2):
    axs[i].tick_params(axis='x', labelsize=14)
    axs[i].tick_params(axis='y', labelsize=14)

axs[0].set_title('Training Set Correlations', size=15)
axs[1].set_title('Test Set Correlations', size=15)

plt.show()
```

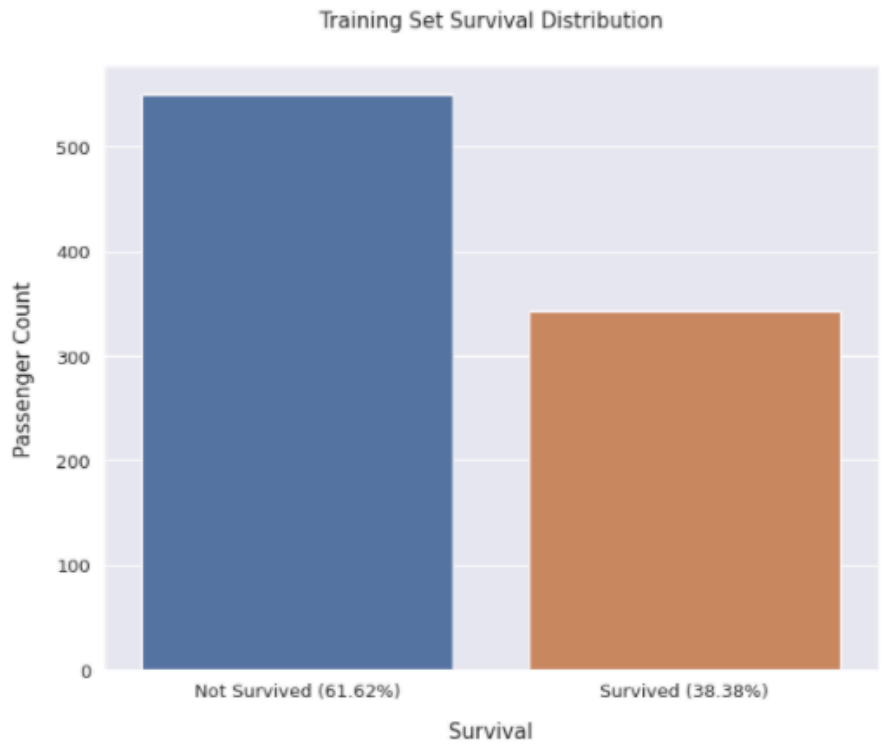

Training Set Correlations
Age
Fare
Parch
Pclass
SibSp
Survived
1
0.12
-0.18
-0.41
-0.25
-0.06
0.12
1
0.21
-0.55
0.16
0.25
-0.18
0.21
1
0.018
0.41
0.082
-0.41
-0.55
0.018
1
0.083
-0.34
-0.25
0.16
0.41
0.083
1
-0.035
-0.06
0.25
0.082
-0.34
-0.035
1
Age
Fare
Parch
Pclass
SibSp
Survived
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4

**Target Distribution**

- 38.38% (342/891) of training set is Class 1
- 61.62% (549/891) of training set is Class 0

```
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()
```

```
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.
```

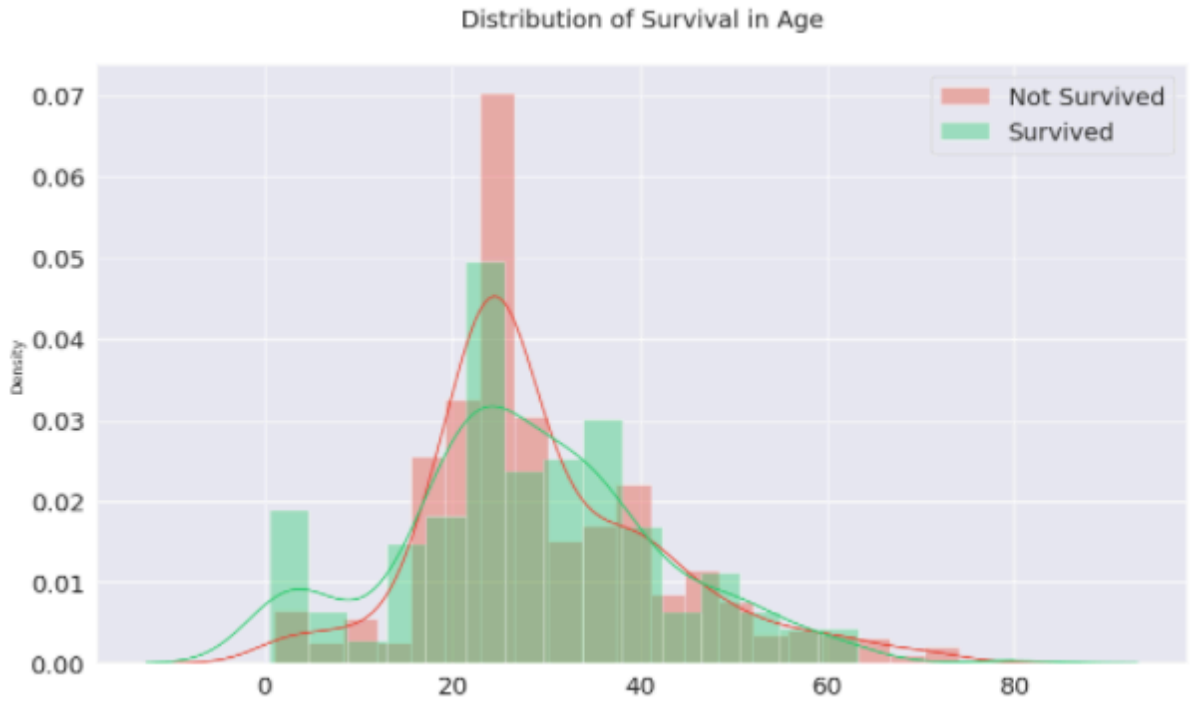


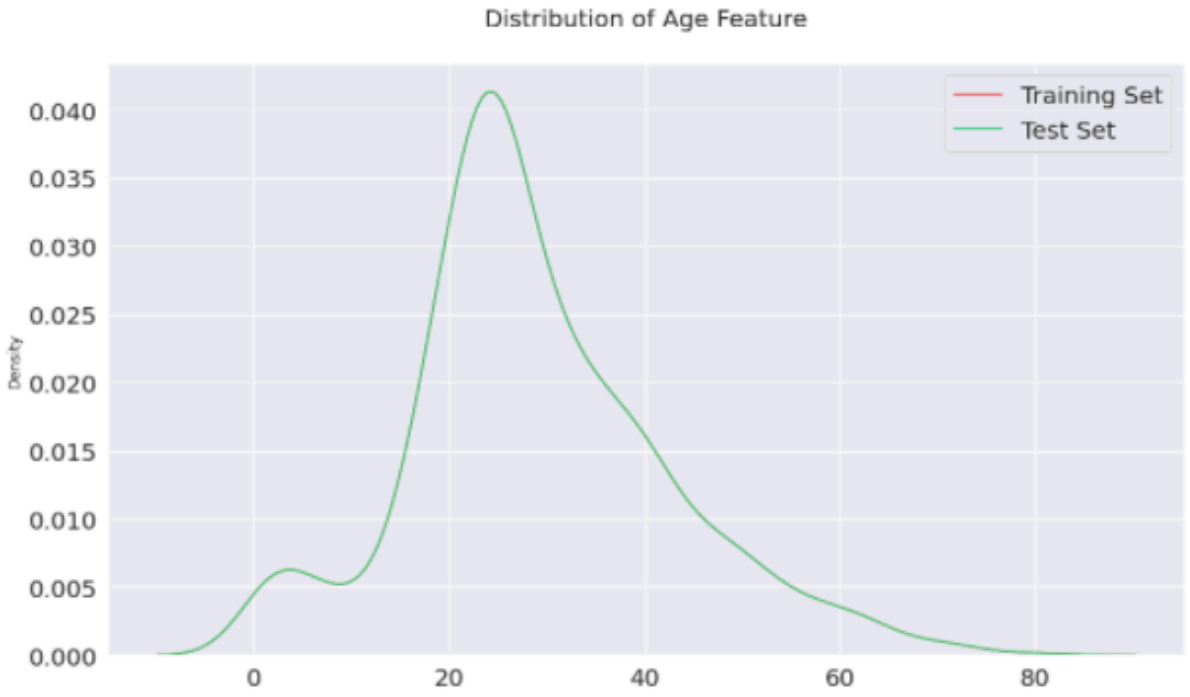
**Target Distribution in Features**

- **Continuous Features**

Both of the continuous features (Age and Fare) have good split points and spikes for a decision tree to learn. One potential problem for both features is, the distribution has more spikes and bumps in the training set, but it is smoother in the test set. Model may not be able to generalise to the test set because of this reason.

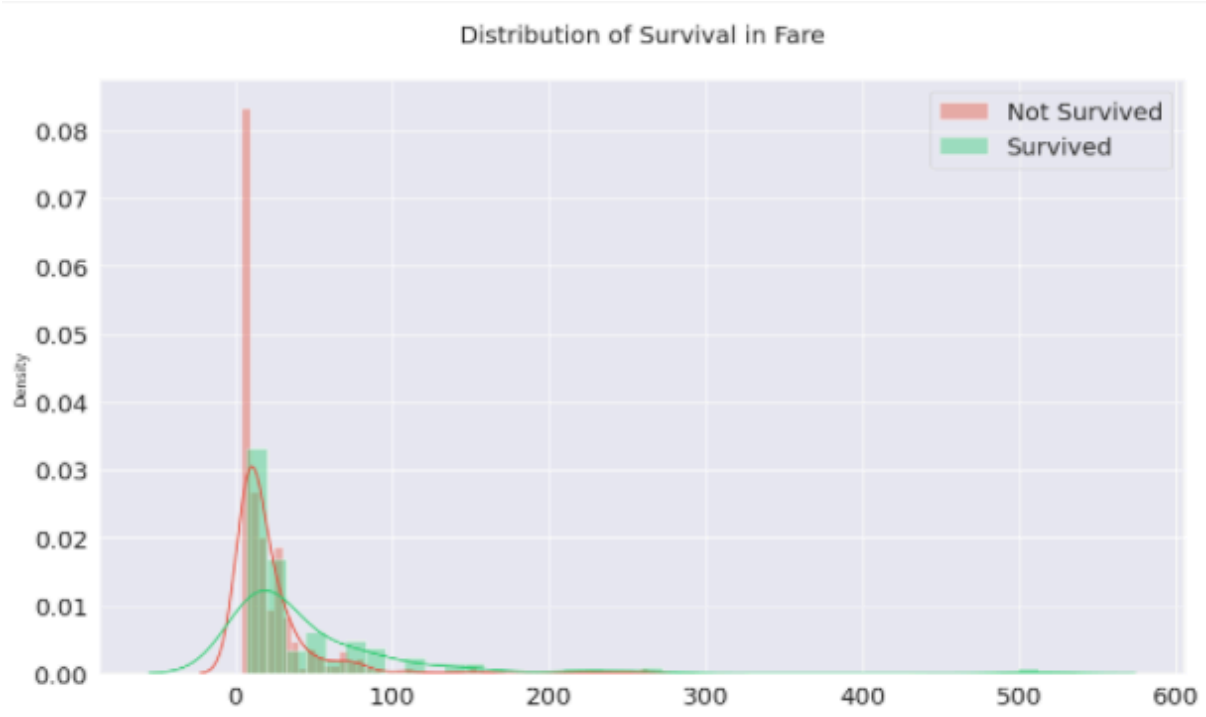
```
cont_features = ['Age', 'Fare']
surv = df_train['Survived'] == 1

fig, axs = plt.subplots(ncols=2, nrows=2, figsize=(20, 20))
plt.subplots_adjust(right=1.5)

for i, feature in enumerate(cont_features):
    # Distribution of survival in feature
    sns.distplot(df_train[~surv][feature], label='Not Survived', hist=True, color='#e74c3c', ax=axs[0][i])
    sns.distplot(df_train[surv][feature], label='Survived', hist=True, color='#2ecc71', ax=axs[0][i])

    # Distribution of feature in dataset
    sns.distplot(df_train[feature], label='Training Set', hist=False, color='#e74c3c', ax=axs[1][i])
    sns.distplot(df_test[feature], label='Test Set', hist=False, color='#2ecc71', ax=axs[1][i])

    axs[0][i].set_xlabel('')
    axs[1][i].set_xlabel('')

    for j in range(2):
        axs[i][j].tick_params(axis='x', labelsize=20)
        axs[i][j].tick_params(axis='y', labelsize=20)

    axs[0][i].legend(loc='upper right', prop={'size': 20})
    axs[1][i].legend(loc='upper right', prop={'size': 20})
    axs[0][i].set_title('Distribution of Survival in {}'.format(feature), size=20, y=1.05)

axs[1][0].set_title('Distribution of {} Feature'.format('Age'), size=20, y=1.05)
axs[1][1].set_title('Distribution of {} Feature'.format('Fare'), size=20, y=1.05)

plt.show()
```

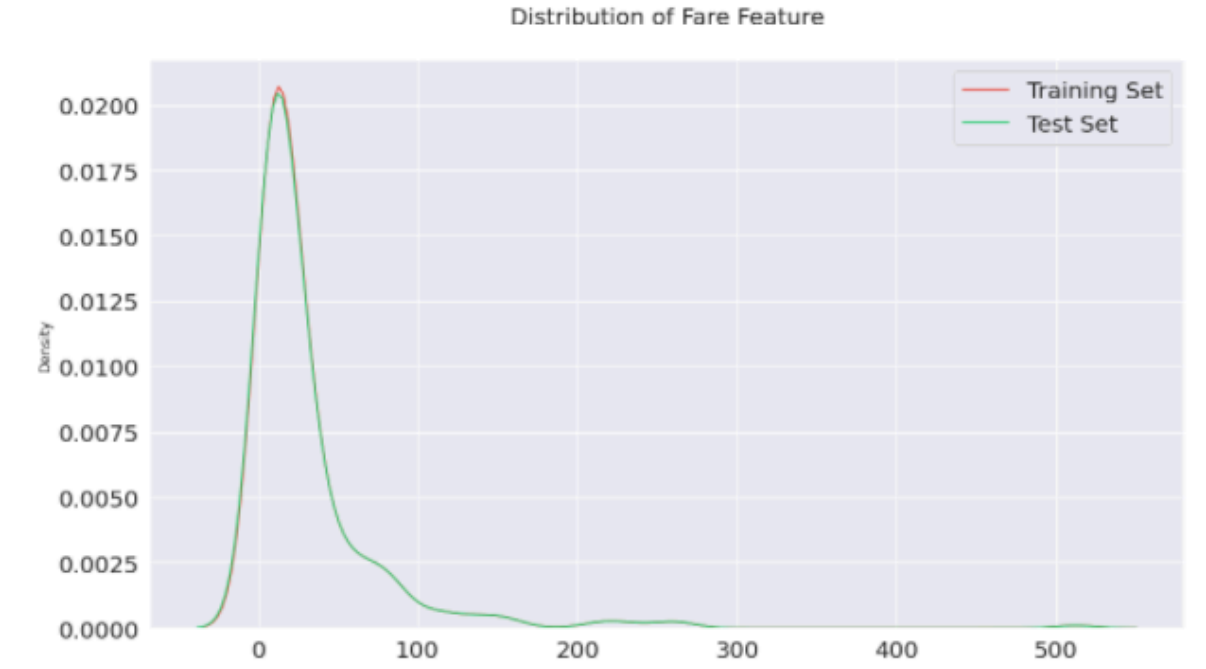



Observation from the continuous features graph

- Distribution of Age feature clearly shows that among 25 years old has a higher survival rate than any of the other age groups



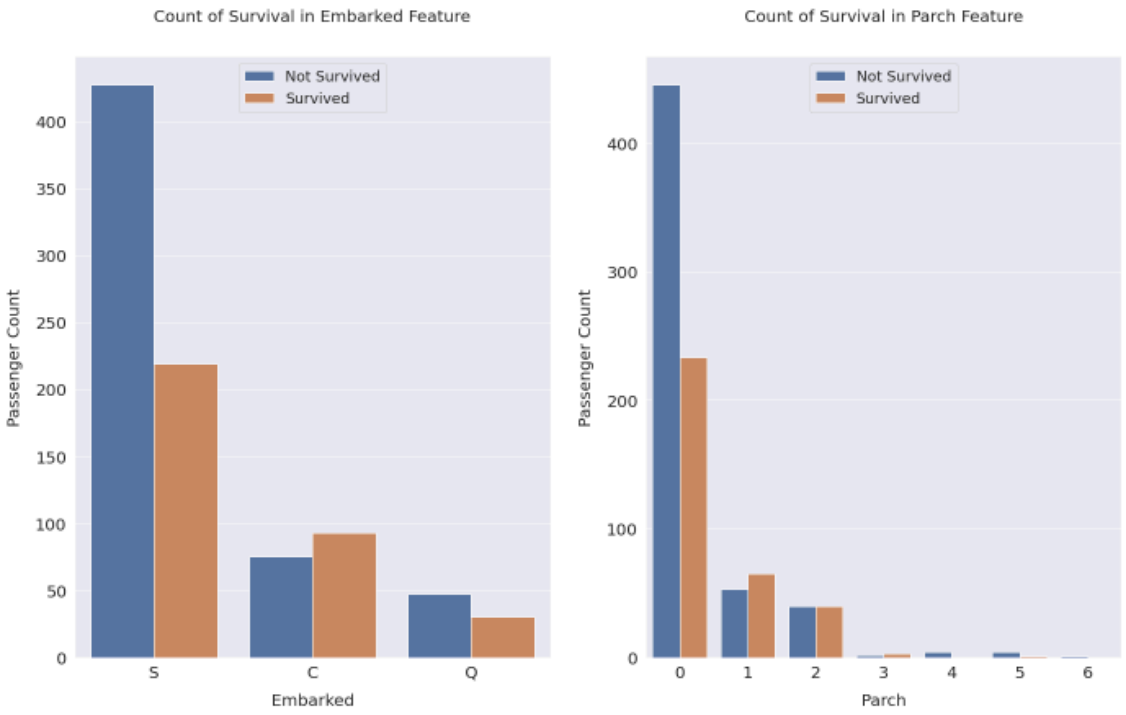

Observation from the continuous features graph

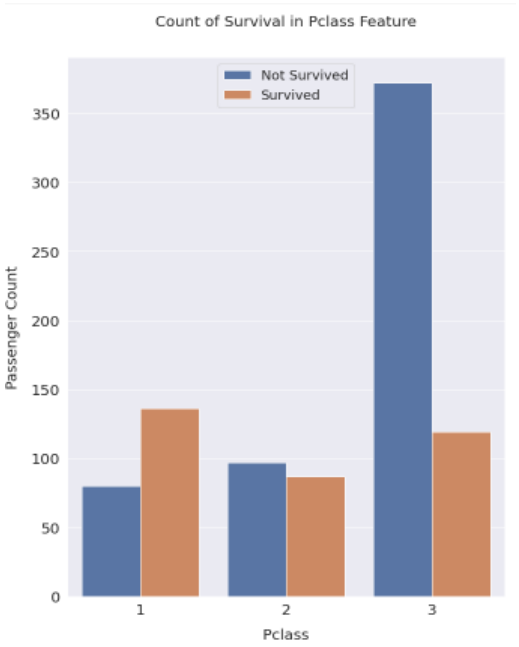
- Distribution of Fare feature, the survival rate is higher on distribution tails. The distribution also has a positive skew because of the extremely large outliers.

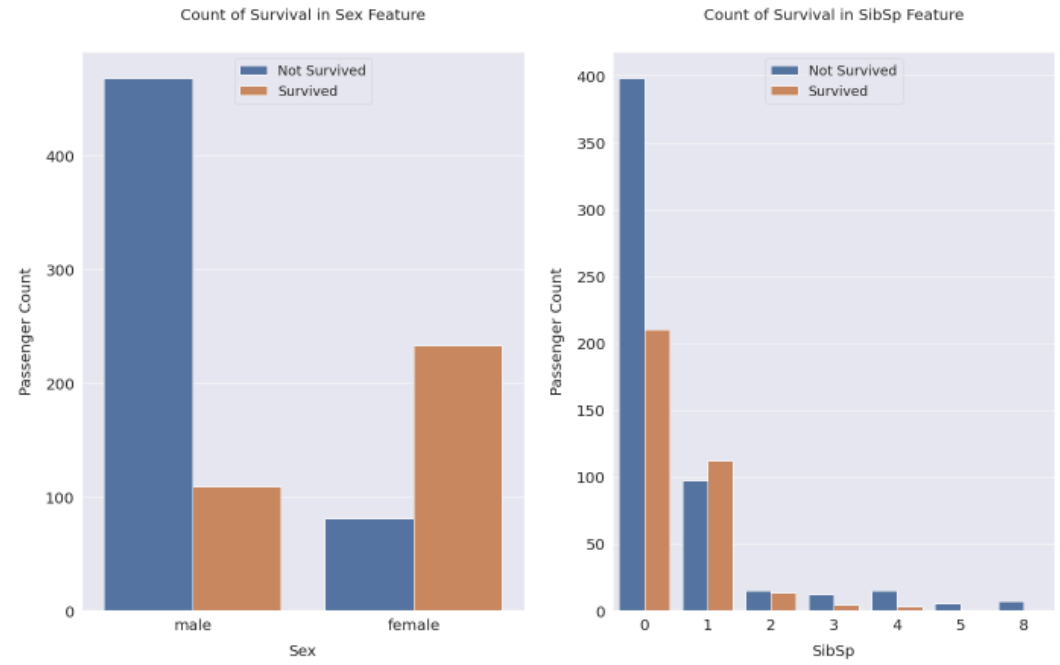
- **Categorical Features**

  Every categorical feature has at least one class with a high mortality rate. Those classes are very helpful to predict whether the passenger is a survivor or victim.

```
cat_features = ['Embarked', 'Parch', 'Pclass', 'Sex', 'SibSp']

fig, axs = plt.subplots(ncols=2, nrows=3, figsize=(20, 20))
plt.subplots_adjust(right=1.5, top=1.25)

for i, feature in enumerate(cat_features, 1):
    plt.subplot(2, 3, i)
    sns.countplot(x=feature, hue='Survived', data=df_train)

    plt.xlabel('{}'.format(feature), size=20, labelpad=15)
    plt.ylabel('Passenger Count', size=20, labelpad=15)
    plt.tick_params(axis='x', labelsize=20)
    plt.tick_params(axis='y', labelsize=20)

    plt.legend(['Not Survived', 'Survived'], loc='upper center', prop={'size': 18})
    plt.title('Count of Survival in {} Feature'.format(feature), size=20, y=1.05)

plt.show()
```






Observation from Categorical Features graph

- Best categorical features are Pclass and Sex because they have the most homogenous distributions.
- Passengers boarded from Southampton has a lower survival rate unlike other ports. More than half of the passengers boarded from Cherbourg had survived. This observation could be related to Pclass feature.
- Parch and SibSp features show that passengers with only one family member have a higher survival rate.

## FEATURE ENGINEERING

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

| | 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 |

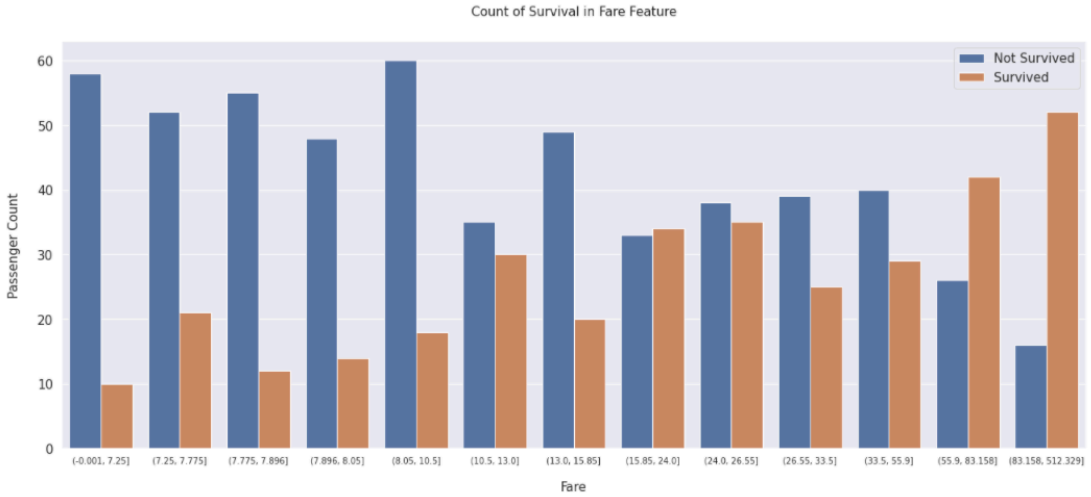
### ➔ Binning Continuous Features

i. Fare

Fare feature is positively skewed and survival rate is extremely high on the right end. 13 quantile based bins are used for Fare feature. Even though the bins are too much, they provide a decent amount of information gain.

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

```
fig, axs = plt.subplots(figsize=(22, 9))
sns.countplot(x='Fare', hue='Survived', data=df_all)

plt.xlabel('Fare', size=15, labelpad=20)
plt.ylabel('Passenger Count', size=15, labelpad=20)
plt.tick_params(axis='x', labelsize=10)
plt.tick_params(axis='y', labelsize=15)

plt.legend(['Not Survived', 'Survived'], loc='upper right', prop={'size': 15})
plt.title('Count of Survival in {} Feature'.format('Fare'), size=15, y=1.05)

plt.show()
```



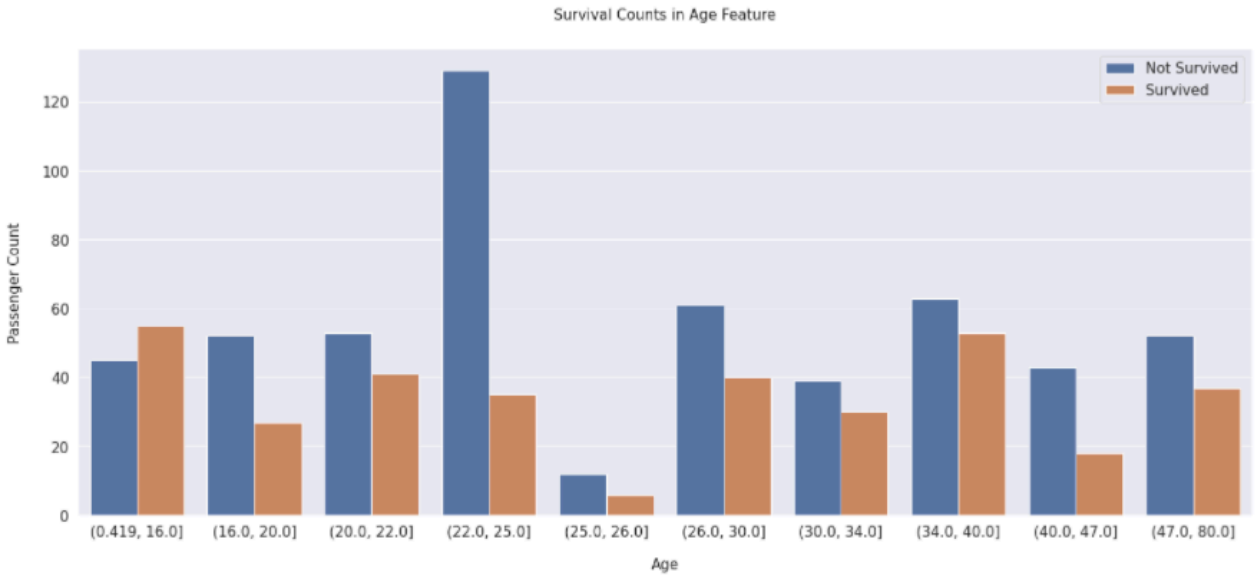

From the plot graph, The groups at the left side of the graph have the lowest survival rate and the groups at the right side of the graph have the highest survival rate. This high survival rate was not visible in the distribution graph.

ii. Age

Age feature has a normal distribution with some spikes and bumps and 10 quantile based bins are used for Age.

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

```
# Plot graph
fig, axs = plt.subplots(figsize=(22, 9))
sns.countplot(x='Age', hue='Survived', data=df_all)

plt.xlabel('Age', size=15, labelpad=20)
plt.ylabel('Passenger Count', size=15, labelpad=20)
plt.tick_params(axis='x', labelsize=15)
plt.tick_params(axis='y', labelsize=15)

plt.legend(['Not Survived', 'Survived'], loc='upper right', prop={'size': 15})
plt.title('Survival Counts in {} Feature'.format('Age'), size=15, y=1.05)

plt.show()
```



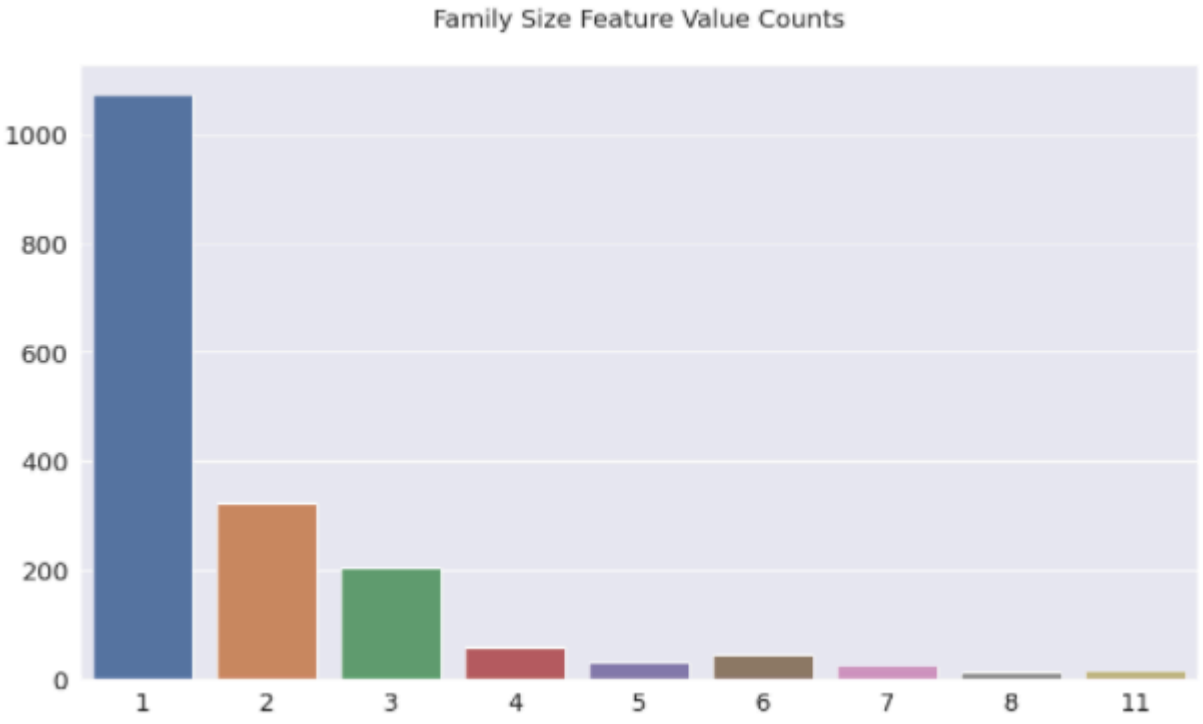

From the graph, the first bin has the highest survival rate and the 4th bin has the lowest survival rate. Those were the biggest spikes in the distribution.

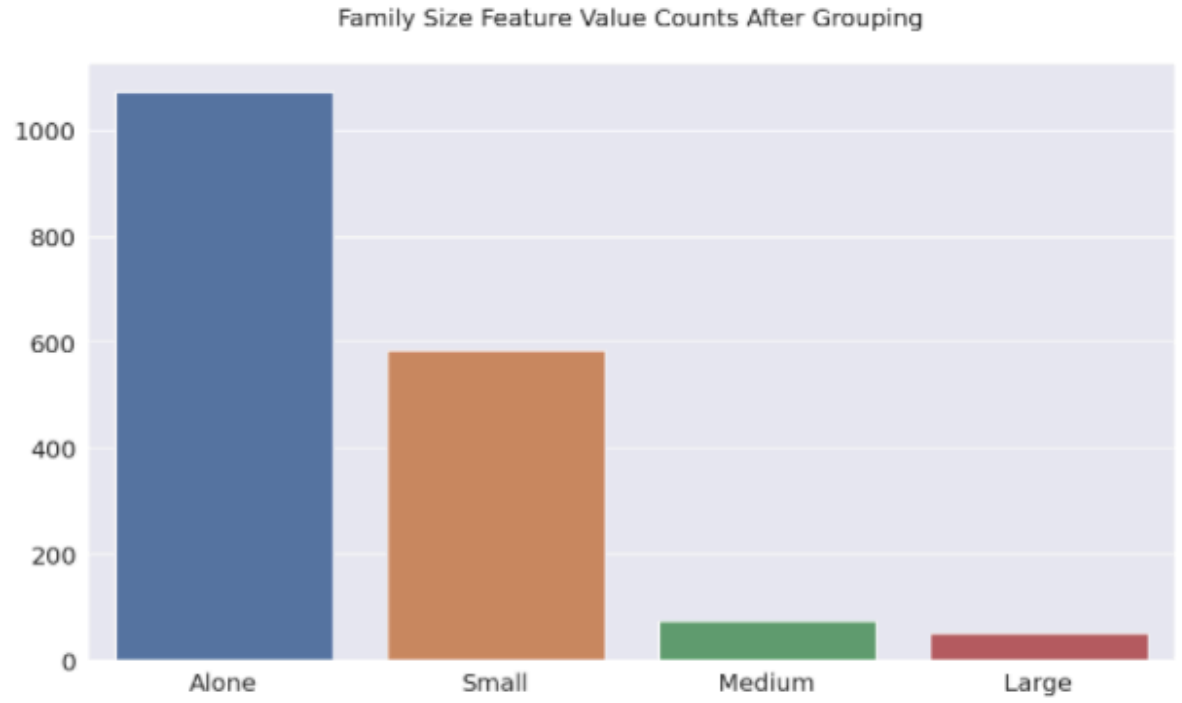
➔ **Frequency Encoding**

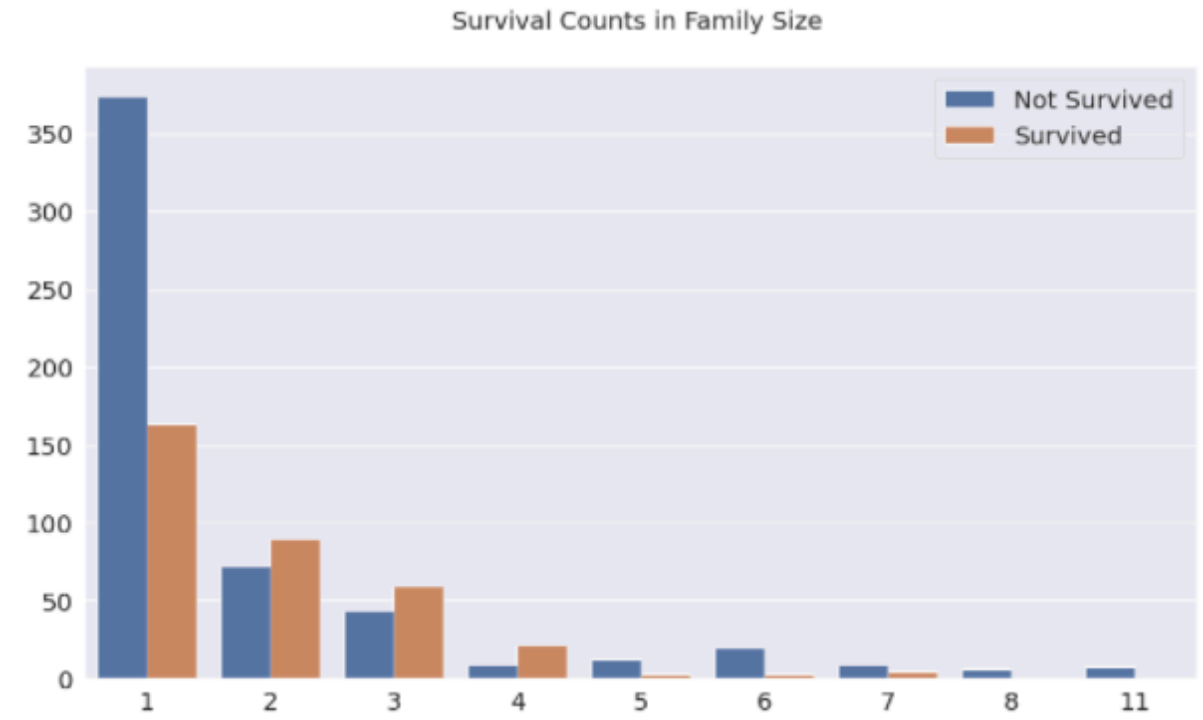
Family_Size is created by adding SibSp, Parch and 1. SibSp is the count of siblings and spouse, and Parch is the count of parents and children. Those columns are added in order to find the total size of families. Adding 1 at the end, is the current passenger. Graphs have clearly shown that family size is a predictor of survival because different values have different survival rates.

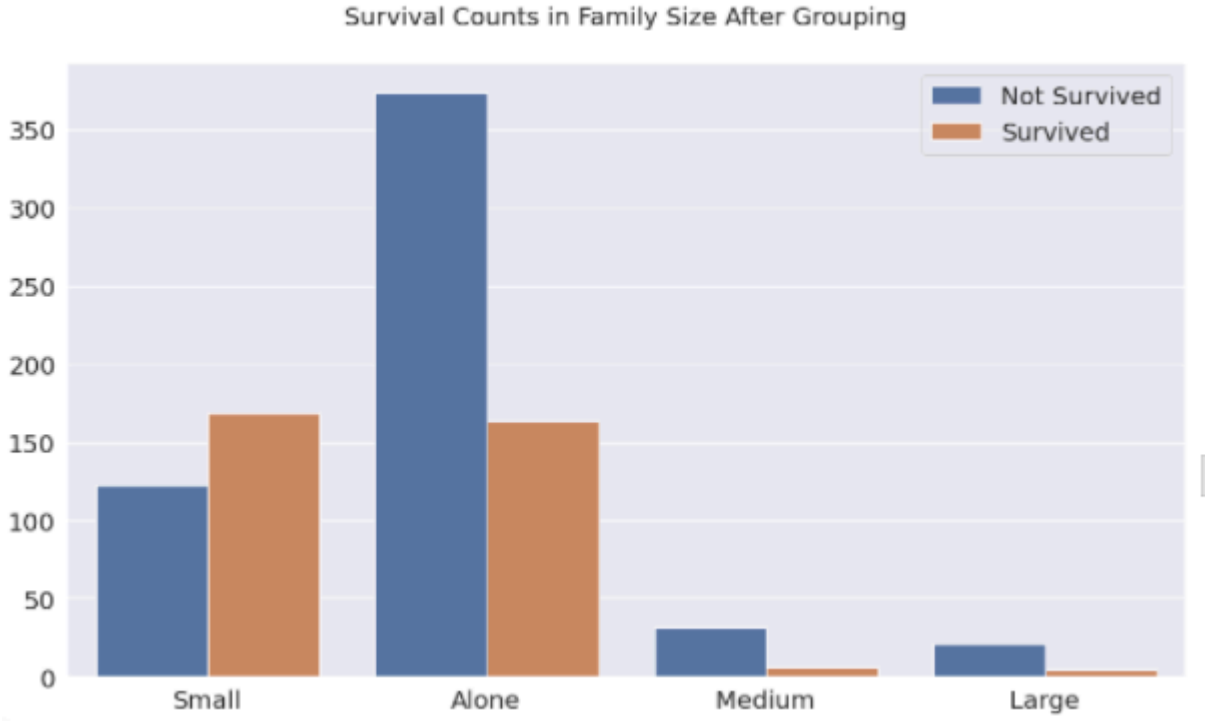
- Family Size with 1 are labelled as Alone
- Family Size with 2, 3 and 4 are labelled as Small
- Family Size with 5 and 6 are labelled as Medium
- Family Size with 7, 8 and 11 are labelled as Large

```
df_all['Family_Size'] = df_all['SibSp'] + df_all['Parch'] + 1

fig, axs = plt.subplots(figsize=(20, 20), ncols=2, nrows=2)
plt.subplots_adjust(right=1.5)

sns.barplot(x=df_all['Family_Size'].value_counts().index, y=df_all['Family_Size']
            .value_counts().values, ax=axs[0][0])
sns.countplot(x='Family_Size', hue='Survived', data=df_all, ax=axs[0][1])

axs[0][0].set_title('Family Size Feature Value Counts', size=20, y=1.05)
axs[0][1].set_title('Survival Counts in Family Size ', size=20, y=1.05)

family_map = {1: 'Alone', 2: 'Small', 3: 'Small', 4: 'Small', 5: 'Medium', 6: 'Medium',
              7: 'Large', 8: 'Large', 11: 'Large'}
df_all['Family_Size_Grouped'] = df_all['Family_Size'].map(family_map)

sns.barplot(x=df_all['Family_Size_Grouped'].value_counts().index, y=df_all['Family_Size_Grouped']
            .value_counts().values, ax=axs[1][0])
sns.countplot(x='Family_Size_Grouped', hue='Survived', data=df_all, ax=axs[1][1])

axs[1][0].set_title('Family Size Feature Value Counts After Grouping', size=20, y=1.05)
axs[1][1].set_title('Survival Counts in Family Size After Grouping', size=20, y=1.05)

for i in range(2):
    axs[i][1].legend(['Not Survived', 'Survived'], loc='upper right', prop={'size': 20})
    for j in range(2):
        axs[i][j].tick_params(axis='x', labelsize=20)
        axs[i][j].tick_params(axis='y', labelsize=20)
        axs[i][j].set_xlabel('')
        axs[i][j].set_ylabel('')
plt.show()
```


Family Size Feature Value Counts
1000
800
600
400
200
0
1
2
3
4
5
6
7
8
11


Family Size Feature Value Counts After Grouping
1000
800
600
400
200
0
Alone
Small
Medium
Large

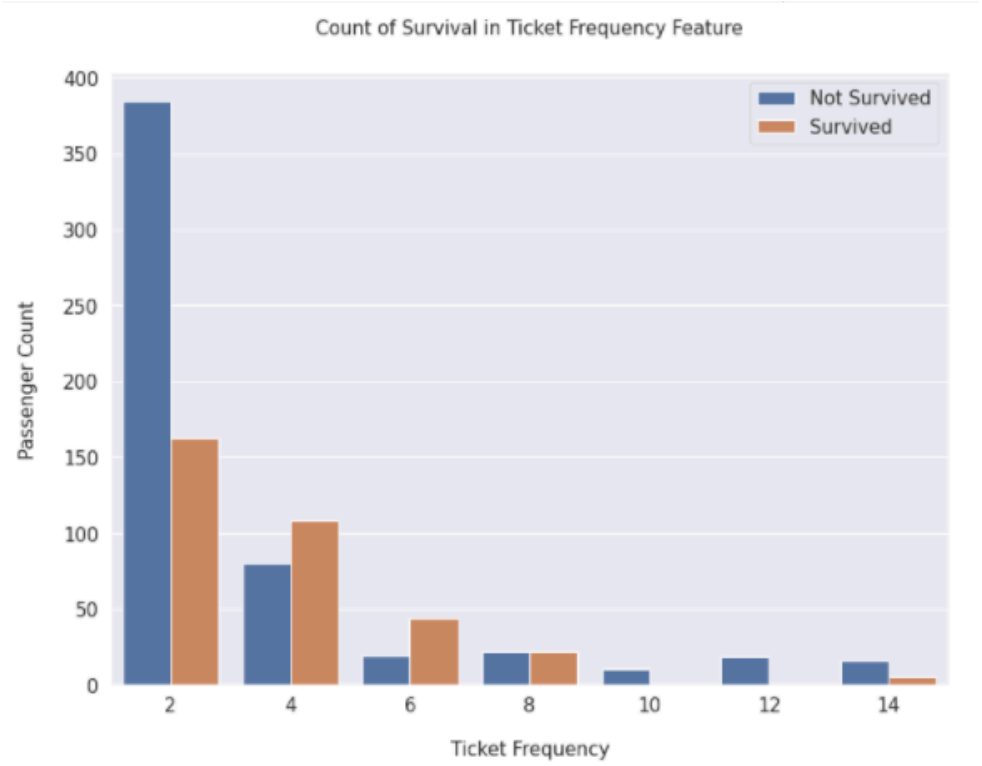

Survival Counts in Family Size
Not Survived
Survived
350
300
250
200
150
100
50
0
1
2
3
4
5
6
7
8
11


Survival Counts in Family Size After Grouping
Not Survived
Survived
350
300
250
200
150
100
50
0
Small
Alone
Medium
Large

There are too many unique Ticket values to analyse, so grouping them up by their frequencies makes things easier. It differs from Family_Size as there are many passengers travelling along with groups. Those groups consist of friends, nannies, maids and others. They were not counted as family, but they used the same ticket.

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

```
fig, axs = plt.subplots(figsize=(12, 9))
sns.countplot(x='Ticket_Frequency', hue='Survived', data=df_all)

plt.xlabel('Ticket Frequency', size=15, labelpad=20)
plt.ylabel('Passenger Count', size=15, labelpad=20)
plt.tick_params(axis='x', labelsize=15)
plt.tick_params(axis='y', labelsize=15)

plt.legend(['Not Survived', 'Survived'], loc='upper right', prop={'size': 15})
plt.title('Count of Survival in {} Feature'.format('Ticket Frequency'), size=15, y=1.05)

plt.show()
```




According to the graph below, groups with **2**,**4** and **6** members had a higher survival rate. Passengers who travel alone have the lowest survival rate. 8th group members have an equal survival rate. After **8** group members, survival rate decreases drastically. This pattern is very similar to the Family_Size feature but there are minor differences. Ticket_Frequency values are not grouped like Family_Size because that would basically create the same feature with perfect correlation. This kind of feature wouldn't provide any additional information gain.

➔ **Title**

Title is created by extracting the prefix before the Name feature.

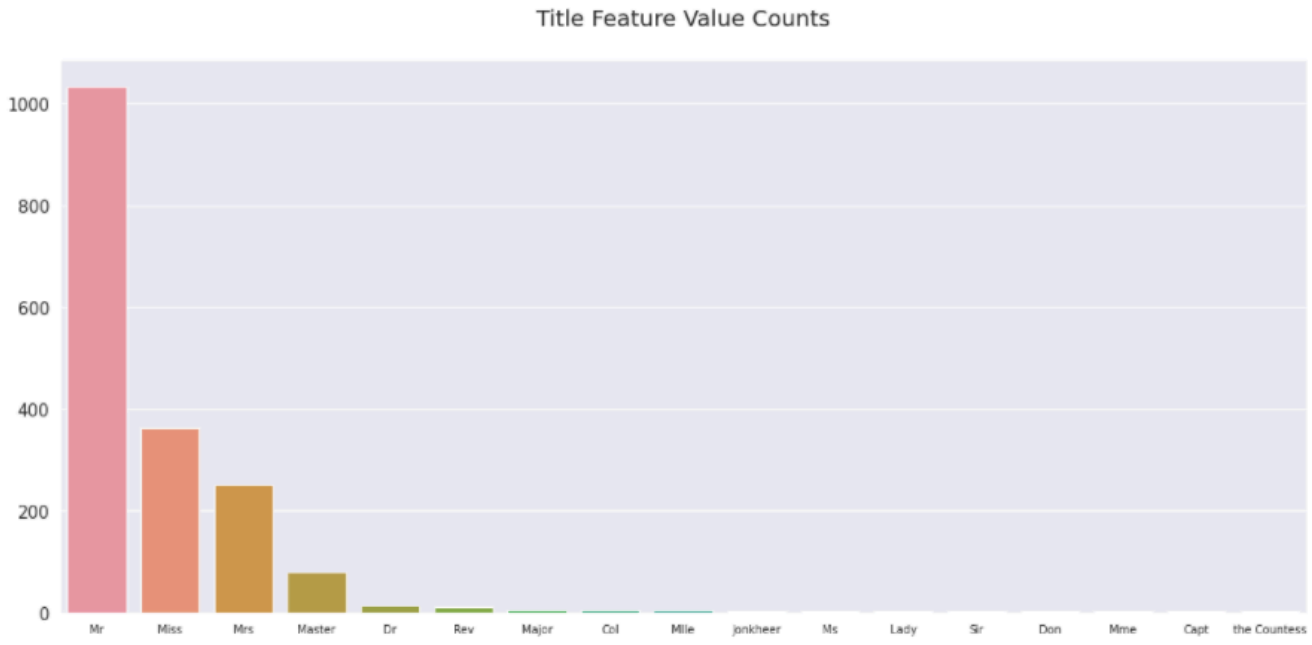
```
df_all['Title'] = df_all['Name'].str.split(', ', expand=True)[1].str.split('.', expand=True)[0]
df_all['Is_Married'] = 0
df_all['Is_Married'].loc[df_all['Title'] == 'Mrs'] = 1
```

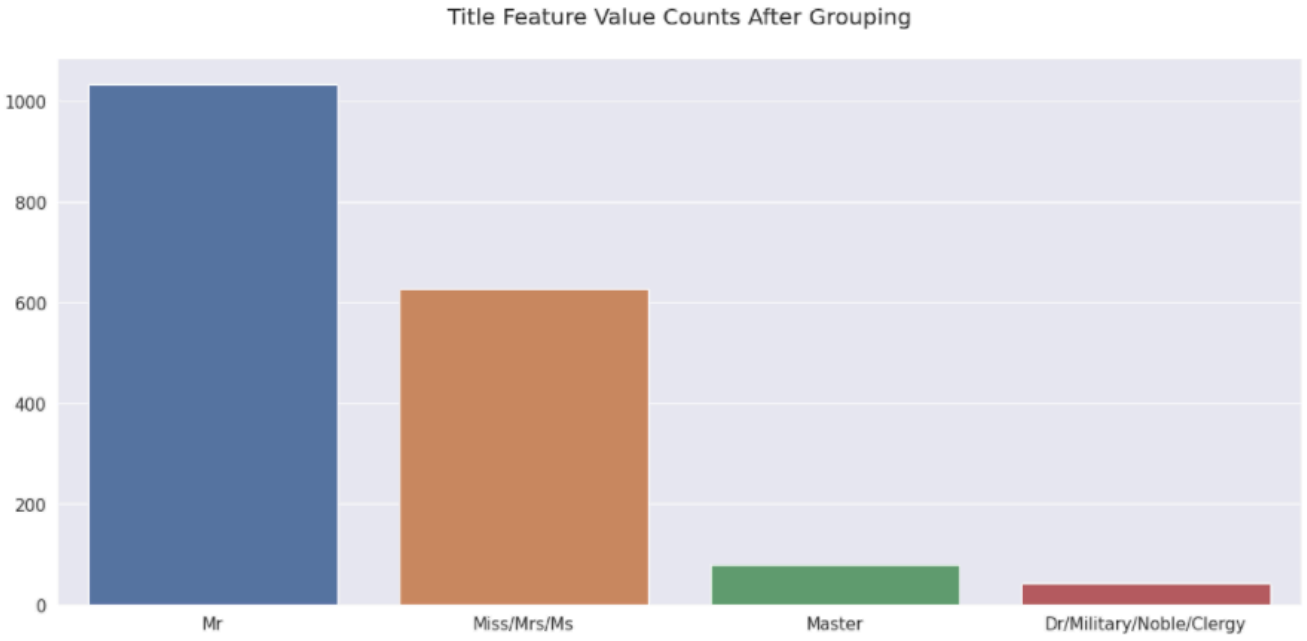
```
fig, axs = plt.subplots(nrows=2, figsize=(20, 20))
sns.barplot(x=df_all['Title'].value_counts().index, y=df_all['Title'].value_counts().values, ax=axs[0])

axs[0].tick_params(axis='x', labelsize=10)
axs[1].tick_params(axis='x', labelsize=15)

for i in range(2):
    axs[i].tick_params(axis='y', labelsize=15)

axs[0].set_title('Title Feature Value Counts', size=20, y=1.05)

df_all['Title'] = df_all['Title'].replace(['Miss', 'Mrs','Ms', 'Mlle', 'Lady', 'Mme',
                                           'the Countess', 'Dona'], 'Miss/Mrs/Ms')
df_all['Title'] = df_all['Title'].replace(['Dr', 'Col', 'Major', 'Jonkheer', 'Capt',
                                           'Sir', 'Don', 'Rev'], 'Dr/Military/Noble/Clergy')

sns.barplot(x=df_all['Title'].value_counts().index, y=df_all['Title'].value_counts().values, ax=axs[1])
axs[1].set_title('Title Feature Value Counts After Grouping', size=20, y=1.05)

plt.show()
```

According to the graph, there are many titles that are occuring very few times and need to be replaced.

- **Miss**, **Mrs**, **Ms**, **Mlle**, **Lady**, **Mme**, **the Countess**, **Dona** titles are replaced with **Miss/Mrs/Ms** because all of them are female.
- Values like **Mlle**, **Mme** and **Dona** are actually the name of the passengers, but they are classified as titles because the Name feature is split by comma.
- **Dr**, **Col**, **Major**, **Jonkheer**, **Capt**, **Sir**, **Don** and **Rev** titles are replaced with **Dr/Military/Noble/Clergy** because those passengers have similar characteristics.
- **Master** is a unique title. It is given to male passengers below age **26**. They have the highest survival rate among all males.

Is_Married is a binary feature based on the **Mrs** title. **Mrs** title has the highest survival rate among other female titles. This title needs to be a feature because all female titles are grouped with each other.


Title Feature Value Counts
1000
800
600
400
200
0
Mr
Miss
Mrs
Master
Dr
Rev
Major
Col
Mlle
jonkheer
Ms
Lady
Sir
Don
Mme
Capt
the Countess


Title Feature Value Counts After Grouping
1000
800
600
400
200
0
Mr
Miss/Mrs/Ms
Master
Dr/Military/Noble/Clergy

➔ **Target Encoding**

- extract_surname function is used for extracting surnames of passengers from the Name feature. Family features are created with the extracted surname. This is necessary for grouping passengers in the same family.

```
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]
```

- Family_Survival_Rate is calculated from families in the training set since there is no Survived feature in the test set. A list of family names that are occuring in both training and test set (non_unique_families), is created. The survival rate is calculated for families with more than 1 member in that list, and stored in the Family_Survival_Rate feature.

```
# 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_test['Family'].unique()]
non_unique_tickets = [x for x in df_train['Ticket'].unique() if x in df_test['Ticket'].unique()]

df_family_survival_rate = df_train.groupby('Family')['Survived', 'Family','Family_Size'].median()
df_ticket_survival_rate = df_train.groupby('Ticket')['Survived', 'Ticket','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_family_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]
```

- An extra binary feature Family_Survival_Rate_NA is created for families that are unique to the test set. This feature is also necessary because there is no way to calculate those families' survival rate. This feature implies that family survival rate is not applicable to those passengers because there is no way to retrieve their survival rate.
- Ticket_Survival_Rate and Ticket_Survival_Rate_NA features are also created with the same method. Ticket_Survival_Rate and Family_Survival_Rate are averaged and become Survival_Rate, and Ticket_Survival_Rate_NA and Family_Survival_Rate_NA are also averaged and become Survival_Rate_NA.

```
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'].iloc[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 = []
```

```
for i in range(len(df_train)):
    if df_train['Ticket'][i] in ticket_rates:
        train_ticket_survival_rate.append(ticket_rates[df_train['Ticket'][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'].iloc[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
```

```
for df in [df_train, df_test]:
    df['Survival_Rate'] = (df['Ticket_Survival_Rate'] + df['Family_Survival_Rate']) / 2
    df['Survival_Rate_NA'] = (df['Ticket_Survival_Rate_NA'] + df['Family_Survival_Rate_NA']) / 2
```

➔ **Feature Transformation**

**i. Label Encoding Non-Numerical Features**

Embarked, Sex, Deck , Title and Family_Size_Grouped are object type, and Age and Fare features are category type. They are converted to numerical type with LabelEncoder. LabelEncoder basically labels the classes from 0 to n. This process is necessary for models to learn from those features.

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

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

**ii. One-Hot Encoding the Categorical Features**

The categorical features (Pclass, Sex, Deck, Embarked, Title) are converted to one-hot encoded features with OneHotEncoder. Age and Fare features are not converted because they are ordinal unlike the previous ones.

```
cat_features = ['Pclass', 'Sex', 'Deck', 'Embarked', 'Title', 'Family_Size_Grouped']
encoded_features = []

for df in dfs:
    for feature in cat_features:
        encoded_feat = OneHotEncoder().fit_transform(df[feature].values.reshape(-1, 1)).toarray()
        n = df[feature].nunique()
        cols = ['{}_{}'.format(feature, n) for n in range(1, n + 1)]
        encoded_df = pd.DataFrame(encoded_feat, columns=cols)
        encoded_df.index = df.index
        encoded_features.append(encoded_df)

df_train = pd.concat([df_train, *encoded_features[:6]], axis=1)
df_test = pd.concat([df_test, *encoded_features[6:]], axis=1)
```

```
df_all = concat_df(df_train, df_test)
drop_cols = ['Deck', 'Embarked', 'Family', 'Family_Size', 'Family_Size_Grouped', 'Survived',
             'Name', 'Parch', 'PassengerId', 'Pclass', 'Sex', 'SibSp', 'Ticket', 'Title',
            'Ticket_Survival_Rate', 'Family_Survival_Rate', 'Ticket_Survival_Rate_NA', 'Family_Survival_Rate_NA']

df_all.drop(columns=drop_cols, inplace=True)

df_all.head()
```

| | Age | Deck_1 | Deck_2 | Deck_3 | Deck_4 | Embarked_1 | Embarked_2 | Embarked_3 | Family_Size_Grouped_1 | Family_Size_Grouped_2 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 1 | 7 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 2 | 4 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 1.0 | 0.0 |
| 3 | 7 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 4 | 7 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 1.0 | 0.0 |

| Family_Size_Grouped_3 | Family_Size_Grouped_4 | Fare | Is_Married | Pclass_1 | Pclass_2 | Pclass_3 | Sex_1 |
|---|---|---|---|---|---|---|---|
| 0.0 | 1.0 | 0 | 0 | 0.0 | 0.0 | 1.0 | 0.0 |
| 0.0 | 1.0 | 11 | 1 | 1.0 | 0.0 | 0.0 | 1.0 |
| 0.0 | 0.0 | 3 | 0 | 0.0 | 0.0 | 1.0 | 1.0 |
| 0.0 | 1.0 | 10 | 1 | 1.0 | 0.0 | 0.0 | 1.0 |
| 0.0 | 0.0 | 3 | 0 | 0.0 | 0.0 | 1.0 | 0.0 |

**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))
y_test = df_test['Survival_Rate'].values

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

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

## DECISION TREE

Decision tree is one of the most frequently and widely used supervised machine learning algorithms that can perform both regression and classification tasks. The intuition behind the decision tree algorithm is simple, yet also very powerful.

Important Terminology Related to Decision Tree

- **Root Node**: It represents the entire population or sample and this further gets divided into two or more homogeneous sets.
- **Splitting**: It is a process of dividing a node into two or more sub-nodes.
- **Decision Node**: When a sub-node splits into further sub-nodes, then it is called decision node.
- **Leaf/ Terminal Node**: Nodes that do not split are called Leaf or Terminal nodes.
- **Pruning**: When we remove sub-nodes of a decision node, this process is called pruning. It is the opposite process of splitting.
- **Branch:** A subsection of the entire tree is called branch or sub-tree.
- **Parent and Child Node**: A node, which is divided into sub-nodes is called parent node of sub-nodes whereas sub-nodes are the child of parent node.

### ➔ Gini Impurity

When we want to build a Decision Tree, the best split for each node of the tree needs to be found first. The commonly used metric for Decision Tree to find the best split is *Information Gain (IG)*. In the *sklearn* library, there is Gini Impurity that uses the common measure as IG. We use Gini Impurity in our code.

```
[254] # Define function to calculate Gini Impurity
      def get_gini_impurity(survived_count, total_count):
          survival_prob = survived_count/total_count
          not_survival_prob = (1 - survival_prob)
          random_observation_survived_prob = survival_prob
          random_observation_not_survived_prob = (1 - random_observation_survived_prob)
          mislabelling_survided_prob = not_survival_prob * random_observation_survived_prob
          mislabelling_not_survided_prob = survival_prob * random_observation_not_survived_prob
          gini_impurity = mislabelling_survided_prob + mislabelling_not_survided_prob
          return gini_impurity
```

```
[255] df_train[['Sex', 'Survived']].groupby(['Sex'], as_index=False).agg(['mean', 'count', 'sum'])
      # Since Survived is a binary feature, this metrics grouped by the Sex feature represent:
          # MEAN: survival rate
          # COUNT: total observations
          # SUM: people survived

      # sex_mapping = {{'female': 0, 'male': 1}}
```

| | Survived | | |
|---|---|---|---|
| | mean | count | sum |
| Sex | | | |
| 0 | 0.742038 | 314 | 233.0 |
| 1 | 0.188908 | 577 | 109.0 |

```
[44] # Gini Impurity of starting node
     gini_impurity_starting_node = get_gini_impurity(survived, df_train.shape[0])
     # total: 819, survived: 342
     gini_impurity_starting_node
```

```
0.47301295786144265
```

Next, we calculate the Gini Impurity by splitting it by Sex.

```
[257] # Gini Impurity decrease of node for 'male' observations
      gini_impurity_male = get_gini_impurity(109, 577)
      # male total: 577, male survived: 109
      gini_impurity_male
```

```
0.3064437162277843
```

```
[258] # Gini Impurity decrease if node splited for 'female' observations
      gini_impurity_female = get_gini_impurity(233, 314)
      # female total: 314, female survived: 233
      gini_impurity_female
```

```
0.3828350034484158
```

```
[259] # Gini Impurity decrease if node splited by Sex
      male_weight = 577/891
      female_weight = 314/891
      weighted_gini_impurity_sex_split = (gini_impurity_male * male_weight) + (gini_impurity_female * female_weight)

      sex_gini_decrease = weighted_gini_impurity_sex_split - gini_impurity_starting_node
      sex_gini_decrease
```

```
-0.13964795747285214
```

## ➔ Cross Validation

Then, we perform Cross Validation to find the best depth for the decision tree.

```
[260] # Feature selection: remove variables no longer containing relevant information
      drop_elements = ['PassengerId', 'Name', 'Ticket', 'SibSp',
                       'Family_Size_Grouped', 'Ticket_Frequency', 'Is_Married',
                       'Family', 'Family_Survival_Rate', 'Family_Survival_Rate_NA',
                       'Ticket_Survival_Rate',  'Ticket_Survival_Rate_NA',
                       'Survival_Rate', 'Survival_Rate_NA', 'Pclass_1', 'Pclass_2',
                       'Pclass_3',  'Sex_1', 'Sex_2', 'Deck_1', 'Deck_2', 'Deck_3',
                       'Deck_4',  'Embarked_1', 'Embarked_2', 'Embarked_3', 'Title_1',
                       'Title_2', 'Title_3',  'Title_4',  'Family_Size_Grouped_1',
                       'Family_Size_Grouped_2', 'Family_Size_Grouped_3',
                       'Family_Size_Grouped_4']
      train = df_train.drop(drop_elements, axis = 1)
      test  = df_test.drop(drop_elements, axis = 1)
```

```
[261] from sklearn.model_selection import KFold
      from sklearn.model_selection import cross_val_score

      cv = KFold(n_splits=10)            # Desired number of Cross Validation folds
      accuracies = list()
      max_attributes = len(list(test))
      depth_range = range(1, max_attributes + 1)

      # Testing max_depths from 1 to max attributes
      # Uncomment prints for details about each Cross Validation pass
      for depth in depth_range:
          fold_accuracy = []
          tree_model = tree.DecisionTreeClassifier(max_depth = depth)
          # print("Current max depth: ", depth, "\n")
          for train_fold, valid_fold in cv.split(train):
              f_train = train.loc[train_fold] # Extract train data with cv indices
              f_valid = train.loc[valid_fold] # Extract valid data with cv indices

              model = tree_model.fit(X = f_train.drop(['Survived'], axis=1),
                                     y = f_train["Survived"]) # We fit the model with the fold train data
              valid_acc = model.score(X = f_valid.drop(['Survived'], axis=1),
                                      y = f_valid["Survived"])# We calculate accuracy with the fold validation data
              fold_accuracy.append(valid_acc)

          avg = sum(fold_accuracy)/len(fold_accuracy)
          accuracies.append(avg)
          # print("Accuracy per fold: ", fold_accuracy, "\n")
          # print("Average accuracy: ", avg)
          # print("\n")

      # Just to show results conveniently
      df = pd.DataFrame({"Max Depth": depth_range, "Average Accuracy": accuracies})
      df = df[["Max Depth", "Average Accuracy"]]
      print(df.to_string(index=False))
```

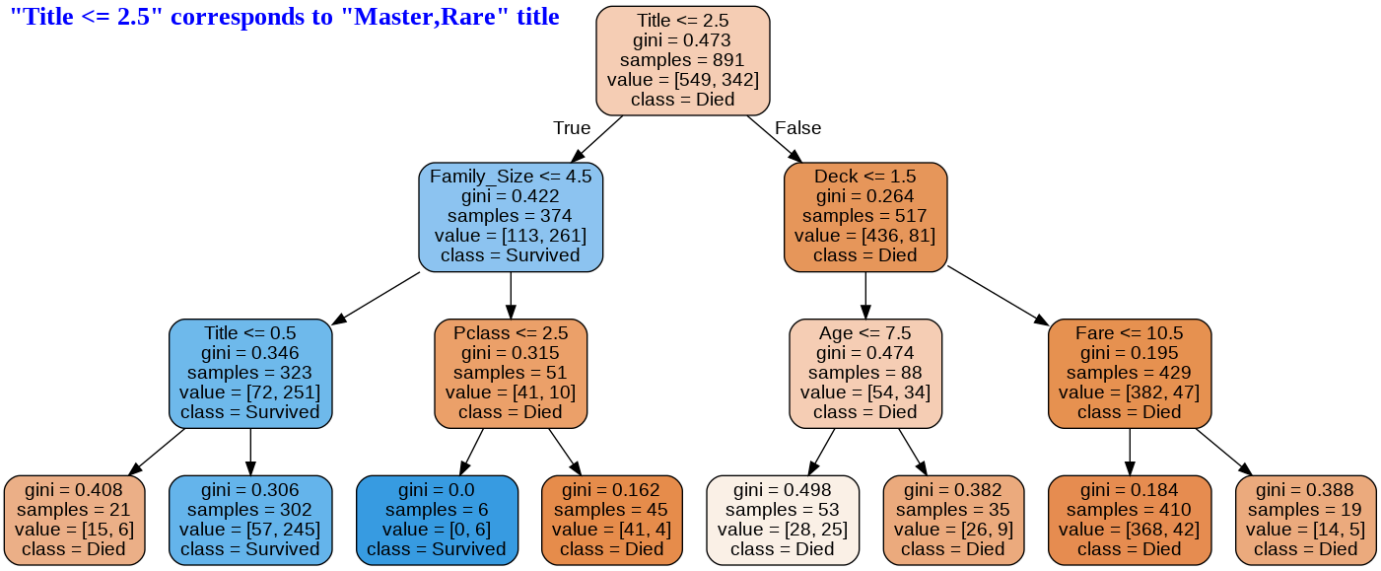
```
 Max Depth  Average Accuracy
         1          0.782285
         2          0.799189
         3          0.823783
         4          0.832772
         5          0.821648
         6          0.820462
         7          0.810362
         8          0.799126
         9          0.796929
        10          0.794669
```

➔ **Final Decision Tree**

Lastly, we import tree function from the *sklearn* library to build a decision tree based on the train test data we built.

```
[265] from sklearn import tree
      from sklearn.tree import DecisionTreeClassifier
      from IPython.display import Image as PImage
      from subprocess import check_call
      from PIL import Image, ImageDraw, ImageFont

      # Create Numpy arrays of train, test and target (Survived) dataframes to feed into our models
      y_train = train['Survived']
      X_train = train.drop(['Survived'], axis=1).values
      X_test = test.drop(['Survived'], axis=1).values

      # Create Decision Tree with max_depth = 3
      decision_tree = tree.DecisionTreeClassifier(max_depth = 3)
      decision_tree.fit(X_train, y_train)

      # Predicting results for test dataset
      y_pred = decision_tree.predict(X_test)
      acc_decision_tree = round(decision_tree.score(X_train, y_train) * 100, 2)
      submission = pd.DataFrame({
              "Survived": y_pred
          })
      submission.to_csv('submission.csv', index=False)

      # Export our trained model as a .dot file
      with open("tree1.dot", 'w') as f:
           f = tree.export_graphviz(decision_tree,
                                    out_file=f,
                                    max_depth = 3,
                                    impurity = True,
                                    feature_names = list(train.drop(['Survived'], axis=1)),
                                    class_names = ['Died', 'Survived'],
                                    rounded = True,
                                    filled= True )

      #Convert .dot to .png to allow display in web notebook
      check_call(['dot','-Tpng','tree1.dot','-o','tree1.png'])

      # Annotating chart with PIL
      img = Image.open("tree1.png")
      draw = ImageDraw.Draw(img)
      font = ImageFont.truetype('/usr/share/fonts/truetype/liberation/LiberationSerif-Bold.ttf', 26)
      draw.text((10, 0), # Drawing offset (position)
                '"Title <= 2.5" corresponds to "Master,Rare" title', # Text to draw
                (0,0,255), # RGB desired color
                font=font) # ImageFont object with desired font
      img.save('sample-out.png')
      PImage("sample-out.png")
```

"Title <= 2.5" corresponds to "Master,Rare" title




From the tree above, the first node starts off with 549 passengers not survived, while the remaining of 342 over 819 survived the crash. But the reason we perform a decision tree is to branch out the survival rate in detail. Although the class of the first node is 'Died', we can still split the branch into two to further see the survival rate of passengers classified by different titles of passenger, passengers' ages, passenger fares, the size of each family, cabin number, and the socio-economic status of passengers.

In the end, we can conclude that passengers with family size less than or equal to 4 and title other than Mr, Miss/Mrs/Ms, and Master have 21 of them who did not survive the crash, while the rest with the title Mr, Miss/Mrs/Ms and Master managed to survive.

For passengers with socio-economic status less than or equal to 2 which are Upper-class and Middle-class, only 6 of them survived and not the rest in Lower-class. And that is the way to interpret the decision tree.

## CONCLUSION

All in all, the Decision Tree algorithm is a supervised machine learning technique that covers both classification and regression. It is mainly used to visualise and represent the decisions in decision making by building a tree. In feature engineering, Decision Tree is used to create or derive new features using a combination of features from the original dataset. In Decision Tree, small groups of features are selected from the columns in the training dataset and are used to train a sub-model, usually what we call a Decision Tree. The sub-model will do predictions which are then assigned back to the main dataset and used as features to help improve the performance of the main model. From this, we can conclude that Feature Engineering is a very important step in machine learning. With Feature Engineering, engineers can analyse the raw data and potential information in order to extract a new or more valuable set of features. These artificial features are then used by the algorithm from Feature Engineering to improve performance or in other words, to reap better results.

Link to the presentation video in Youtube:

https://youtu.be/2RcARGabgSg