Statistical Analysis of Titanic Dataset along with Machine Learning Model

There are three primary work in this notebook:

- **Do a statistical analysis** of how some group of people was survived more than others.
- **Do an exploratory data analysis(EDA)** of titanic with visualizations and storytelling.
- **Predict**: Use machine learning classification models to predict the chances of passengers survival.

Import Libraries

In [1]:
```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns

%matplotlib inline
# %config InlineBackend.figure_format = 'retina' ## This is preferable for retina display.

import warnings ## importing warnings library.
warnings.filterwarnings('ignore') ## Ignore warning

import os ## imporing os
print(os.listdir("../input/"))
```

```
['titanic']
```

Import the data

In [2]:
```
train = pd.read_csv("../input/titanic/train.csv")
test = pd.read_csv("../input/titanic/test.csv")
```

In [3]:
```
train.head()
```

Out[3]:

	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

In [4]:
```
temp = train.groupby("Sex")['Age'].min().to_frame().reset_index()
```

In [5]:
```
temp
```

Out[5]:

	Sex	Age
0	female	0.75
1	male	0.42

In [6]:
```
temp = temp.rename(columns={"Age": "min_age"})
```

In [7]:
```
temp
```

Out[7]:

	Sex	min_age
0	female	0.75
1	male	0.42

In [8]:
```
train.head()
```

Out[8]:

	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

In [9]:
```
train.dtypes
```

Out[9]:
```
PassengerId      int64
Survived         int64
Pclass           int64
Name            object
Sex             object
Age            float64
SibSp            int64
Parch            int64
Ticket          object
Fare           float64
Cabin           object
Embarked        object
dtype: object
```

Train Set

In [10]:
```
%%time
train.sample(5)
```

```
CPU times: user 254 µs, sys: 1.04 ms, total: 1.3 ms
Wall time: 4.06 ms
```

Out[10]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
279	280	1	3	Abbott, Mrs. Stanton (Rosa Hunt)	female	35.0	1	1	C.A. 2673	20.25	NaN	S
853	854	1	1	Lines, Miss. Mary Conover	female	16.0	0	1	PC 17592	39.40	D28	S
770	771	0	3	Lievens, Mr. Rene Aime	male	24.0	0	0	345781	9.50	NaN	S
815	816	0	1	Fry, Mr. Richard	male	NaN	0	0	112058	0.00	B102	S
15	16	1	2	Hewlett, Mrs. (Mary D Kingcome)	female	55.0	0	0	248706	16.00	NaN	S

Test Set

In [11]:
```
## Take a look at the overview of the dataset.
%timeit test.sample(5)
```

```
552 µs ± 19.7 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

This is a sample of train and test dataset. Lets find out a bit more about the train and test dataset.

In [12]:
```
print ("The shape of the train data is (row, column):"+ str(train.shape))
print (train.info())
print ("The shape of the test data is (row, column):"+ str(test.shape))
print (test.info())
```

```
The shape of the train data is (row, column):(891, 12)
<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: 83.7+ KB
None
The shape of the test data is (row, column):(418, 11)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 418 entries, 0 to 417
Data columns (total 11 columns):
PassengerId    418 non-null int64
Pclass         418 non-null int64
Name           418 non-null object
Sex            418 non-null object
Age            332 non-null float64
SibSp          418 non-null int64
Parch          418 non-null int64
Ticket         418 non-null object
Fare           417 non-null float64
Cabin          91 non-null object
Embarked       418 non-null object
dtypes: float64(2), int64(4), object(5)
memory usage: 36.0+ KB
None
```

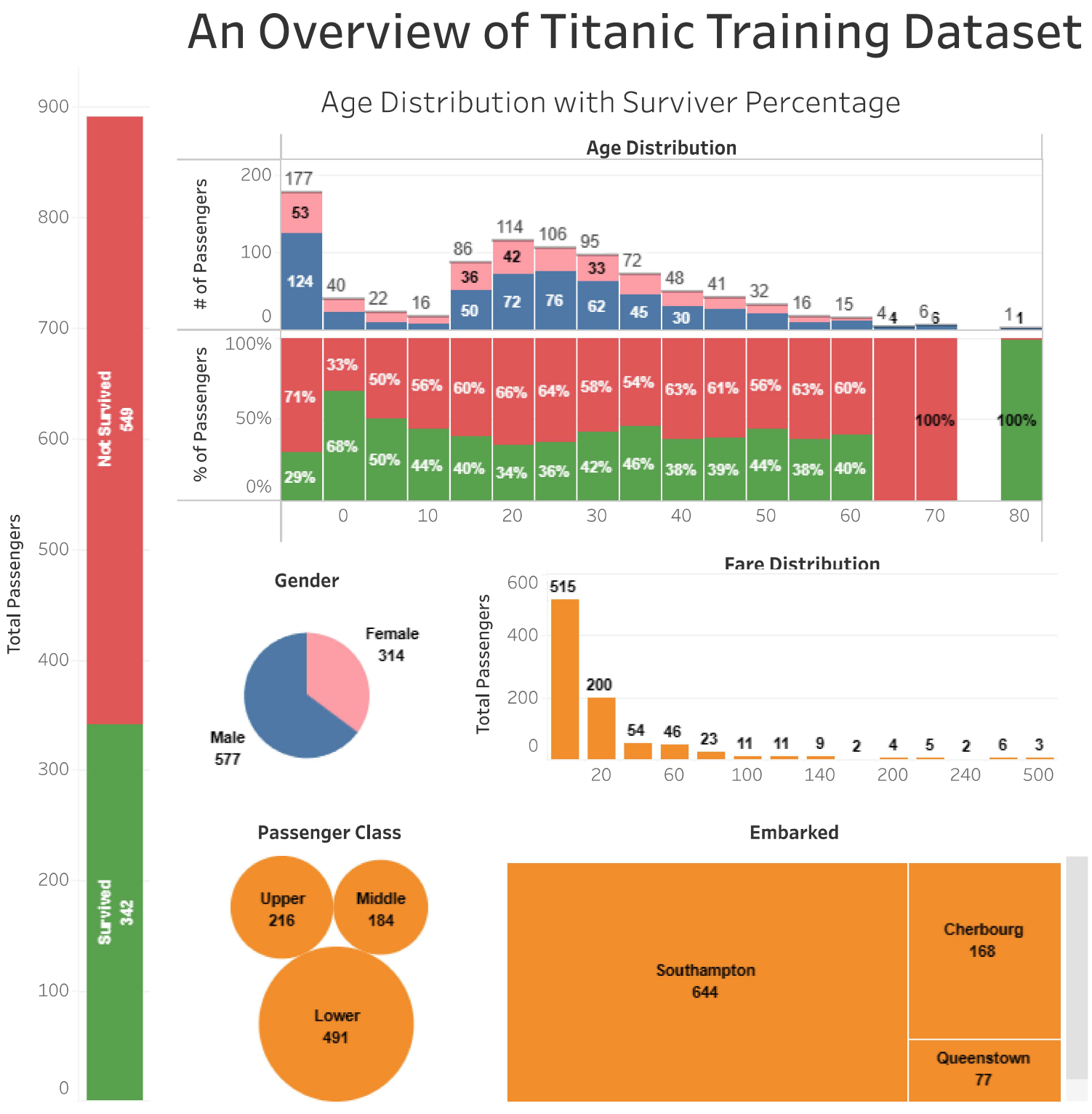
Tableau Visualization of the Data

I have incorporated a tableau visualization below of the training data. This visualization...

- is for us to have an overview and play around with the dataset.
- is done without making any changes(including Null values) to any features of the dataset.

Let's get a better perspective of the dataset through this visualization.

In [13]:
```
%%HTML
<div class='tableauPlaceholder' id='viz1516349898238' style='position: relative'><noscript><a href='#'><img alt='An Ove
```



We want to observe how the left vertical bar changes when filtering out unique values of certain features. Multiple filters can be applied to check for correlations among them. For instance, by selecting the upper and Female tabs, we would notice that the green color dominates the bar with a ratio of 91:3 for survived and non-survived female passengers, indicating a 97% survival rate for females. Resetting the filters is as simple as clicking anywhere in the white space. The age distribution chart at the top offers additional information. For example, it reveals the age range of those three unfortunate females, indicated by the red color that denotes the non-survivors.

Overview the Data

Datasets in the real world are often messy, However, this dataset is almost clean. Lets analyze and see what we have here.

In [14]:
```
## saving passenger id in advance in order to submit later.
passengerid = test.PassengerId
## We will drop PassengerID and Ticket since it will be useless for our data.
#train.drop(['PassengerId'], axis=1, inplace=True)
#test.drop(['PassengerId'], axis=1, inplace=True)

print (train.info())
print ("*"*40)
print (test.info())
```

```
<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: 83.7+ KB
None
****************************************
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 418 entries, 0 to 417
Data columns (total 11 columns):
PassengerId    418 non-null int64
Pclass         418 non-null int64
Name           418 non-null object
Sex            418 non-null object
Age            332 non-null float64
SibSp          418 non-null int64
Parch          418 non-null int64
Ticket         418 non-null object
Fare           417 non-null float64
Cabin          91 non-null object
Embarked       418 non-null object
dtypes: float64(2), int64(4), object(5)
memory usage: 36.0+ KB
None
```

It looks like, the features have unequal amount of data entries for every column and they have many different types of variables. This can happen for the following reasons...

- We may have missing values in our features.
- We may have categorical features.
- We may have alphanumerical or/and text features.

Dealing with Missing values

Missing values in *train* dataset.

In [15]:
```
# Let's write a functin to print the total percentage of the missing values.
def missing_percentage(df):
    """This function takes a DataFrame(df) as input and returns two columns, total missing values and total missing valu
    total = df.isnull().sum().sort_values(ascending = False)
    percent = round(df.isnull().sum().sort_values(ascending = False)/len(df)*100,2)
    return pd.concat([total, percent], axis=1, keys=['Total','Percent'])
```

In [16]:
```
%timeit -r2 -n10 missing_percentage(train) # setting the number of runs(-r) and/or loops (-n)
```

```
5.7 ms ± 749 µs per loop (mean ± std. dev. of 2 runs, 10 loops each)
```

In [17]:
```
missing_percentage(train)
```

Out[17]:

	Total	Percent
Cabin	687	77.10
Age	177	19.87
Embarked	2	0.22
Fare	0	0.00
Ticket	0	0.00
Parch	0	0.00
SibSp	0	0.00
Sex	0	0.00
Name	0	0.00
Pclass	0	0.00
Survived	0	0.00
PassengerId	0	0.00

Missing values in *test* set.

In [18]:
```
%%timeit -r2 -n10
missing_percentage(test)
```

```
4.92 ms ± 409 µs per loop (mean ± std. dev. of 2 runs, 10 loops each)
```

In [19]:
```
missing_percentage(test)
```

Out[19]:

	Total	Percent
Cabin	327	78.23
Age	86	20.57
Fare	1	0.24
Embarked	0	0.00
Ticket	0	0.00
Parch	0	0.00
SibSp	0	0.00
Sex	0	0.00
Name	0	0.00
Pclass	0	0.00
PassengerId	0	0.00

We see that in both **train**, and **test** dataset have missing values. Let's make an effort to fill these missing values starting with "Embarked" feature.

Embarked feature

In [20]:
```
def percent_value_counts(df, feature):
    """This function takes in a dataframe and a column and finds the percentage of the value_counts"""
    percent = pd.DataFrame(round(df.loc[:,feature].value_counts(dropna=False, normalize=True)*100,2))
    ## creating a df with th
    total = pd.DataFrame(df.loc[:,feature].value_counts(dropna=False))
    ## concating percent and total dataframe

    total.columns = ["Total"]
    percent.columns = ['Percent']
    return pd.concat([total, percent], axis = 1)
```

In [21]:
```
percent_value_counts(train, 'Embarked')
```

Out[21]:

	Total	Percent
S	644	72.28
C	168	18.86
Q	77	8.64
NaN	2	0.22

In [22]:
```
percent_value_counts(train, 'Embarked')
```

Out[22]:

	Total	Percent
S	644	72.28
C	168	18.86
Q	77	8.64
NaN	2	0.22

It looks like there are only two null values(~ 0.22 %) in the Embarked feature, we can replace these with the mode value "S".

Let's see what are those two null values

In [23]:
```
train[train.Embarked.isnull()]
```

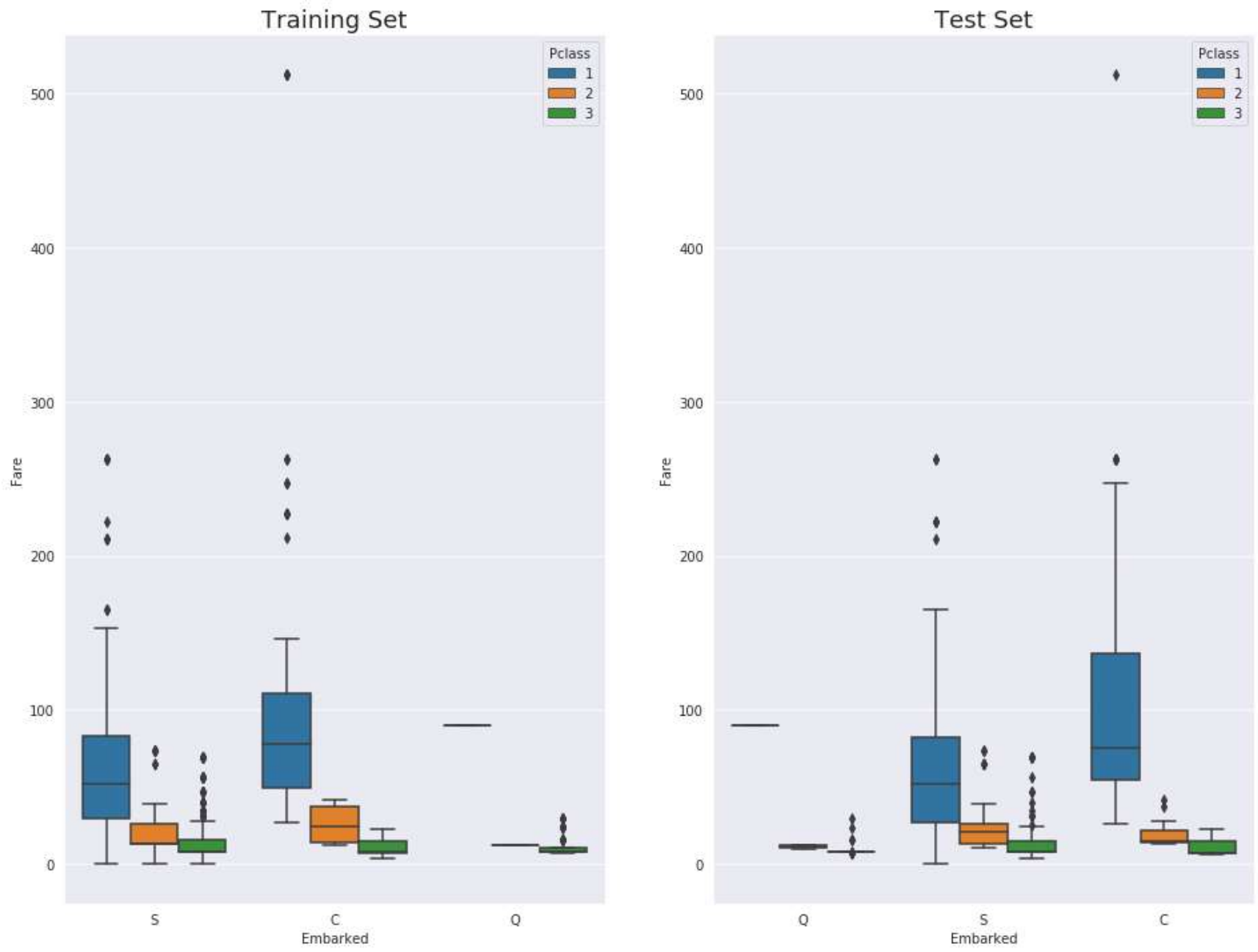
Out[23]:

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

We may be able to solve these two missing values by looking at other independent variables of the two raws. Both passengers paid a fare of $80, are of Pclass 1 and female Sex. Let's see how the **Fare** is distributed among all **Pclass** and **Embarked** feature values

In [24]:
```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_style('darkgrid')
fig, ax = plt.subplots(figsize=(16,12),ncols=2)
ax1 = sns.boxplot(x="Embarked", y="Fare", hue="Pclass", data=train, ax = ax[0]);
ax2 = sns.boxplot(x="Embarked", y="Fare", hue="Pclass", data=test, ax = ax[1]);
ax1.set_title("Training Set", fontsize = 18)
ax2.set_title('Test Set',  fontsize = 18)
```

```
# ## Fixing legends
# leg_1 = ax1.get_legend()
# leg_1.set_title("PClass")
# legs = leg_1.texts
# legs[0].set_text('Upper')
# legs[1].set_text('Middle')
# legs[2].set_text('Lower')

fig.show()
```



Here, in both training set and test set, the average fare closest to $80 are in the **C** Embarked values where pclass is 1. So, let's fill in the missing values as "C"

In [25]:
```
## Replacing the null values in the Embarked column with the mode.
train.Embarked.fillna("C", inplace=True)
```

Cabin Feature

In [26]:
```
print("Train Cabin missing: " + str(train.Cabin.isnull().sum()/len(train.Cabin)))
print("Test Cabin missing: " + str(test.Cabin.isnull().sum()/len(test.Cabin)))
```

```
Train Cabin missing: 0.7710437710437711
Test Cabin missing: 0.7822966507177034
```

Approximately 77% of Cabin feature is missing in the training data and 78% missing on the test data. We have two choices,

- we can either get rid of the whole feature, or
- we can brainstorm a little and find an appropriate way to put them in use. For example, We may say passengers with cabin record had a higher socio-economic-status then others. We may also say passengers with cabin record were more likely to be taken into consideration when loading into the boat.

Let's combine train and test data first and for now, will assign all the null values as **"N"**

In [27]:
```
## Concat train and test into a variable "all_data"
survivers = train.Survived

train.drop(["Survived"],axis=1, inplace=True)

all_data = pd.concat([train,test], ignore_index=False)

## Assign all the null values to N
all_data.Cabin.fillna("N", inplace=True)
```

All the cabin names start with an English alphabet following by multiple digits. It seems like there are some passengers that had booked multiple cabin rooms in their name. This is because many of them travelled with family. However, they all seem to book under the same letter followed by different numbers. It seems like there is a significance with the letters rather than the numbers. Therefore, we can group these cabins according to the letter of the cabin name.

In [28]:
```
all_data.Cabin = [i[0] for i in all_data.Cabin]
```

Now let's look at the value counts of the cabin features and see how it looks.

In [29]:
```
percent_value_counts(all_data, "Cabin")
```

Out[29]:

	Total	Percent
N	1014	77.46
C	94	7.18
B	65	4.97
D	46	3.51
E	41	3.13
A	22	1.68
F	21	1.60
G	5	0.38
T	1	0.08

So, We still haven't done any effective work to replace the null values. Let's stop for a second here and think through how we can take advantage of some of the other features here.

- We can use the average of the fare column We can use pythons ***groupby*** function to get the mean fare of each cabin letter.

In [30]:
```
all_data.groupby("Cabin")['Fare'].mean().sort_values()
```

Out[30]:
```
Cabin
G     14.205000
F     18.079367
N     19.132707
T     35.500000
A     41.244314
D     53.007339
E     54.564634
C    107.926598
B    122.383078
Name: Fare, dtype: float64
```

Now, these means can help us determine the unknown cabins, if we compare each unknown cabin rows with the given mean's above. Let's write a simple function so that we can give cabin names based on the means.

In [31]:
```
def cabin_estimator(i):
    """Grouping cabin feature by the first letter"""
    a = 0
    if i<16:
        a = "G"
    elif i>=16 and i<27:
        a = "F"
    elif i>=27 and i<38:
        a = "T"
    elif i>=38 and i<47:
        a = "A"
    elif i>= 47 and i<53:
        a = "E"
    elif i>= 53 and i<54:
        a = "D"
    elif i>=54 and i<116:
        a = 'C'
    else:
        a = "B"
    return a
```

Let's apply **cabin_estimator** function in each unknown cabins(cabin with **null** values). Once that is done we will separate our train and test to continue towards machine learning modeling.

In [32]:
```
with_N = all_data[all_data.Cabin == "N"]

without_N = all_data[all_data.Cabin != "N"]
```

In [33]:
```
##applying cabin estimator function.
with_N['Cabin'] = with_N.Fare.apply(lambda x: cabin_estimator(x))

## getting back train.
all_data = pd.concat([with_N, without_N], axis=0)

## PassengerId helps us separate train and test.
all_data.sort_values(by = 'PassengerId', inplace=True)

## Separating train and test from all_data.
train = all_data[:891]

test = all_data[891:]
```

```
# adding saved target variable with train.
train['Survived'] = survivers
```

Fare Feature

If you have paid attention so far, you know that there is only one missing value in the fare column. Let's have it.

In [34]:
```
test[test.Fare.isnull()]
```

Out[34]:

	PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
152	1044	3	Storey, Mr. Thomas	male	60.5	0	0	3701	NaN	B	S

Here, We can take the average of the **Fare** column to fill in the NaN value. However, for the sake of learning and practicing, we will try something else. We can take the average of the values where **Pclass** is ***3***, **Sex** is ***male*** and **Embarked** is ***S***

In [35]:
```
missing_value = test[(test.Pclass == 3) &
                     (test.Embarked == "S") &
                     (test.Sex == "male")].Fare.mean()
## replace the test.fare null values with test.fare mean
test.Fare.fillna(missing_value, inplace=True)
```

Age Feature

We know that the feature "Age" is the one with most missing values, let's see it in terms of percentage.

In [36]:
```
print ("Train age missing value: " + str((train.Age.isnull().sum()/len(train))*100)+str("%"))
print ("Test age missing value: " + str((test.Age.isnull().sum()/len(test))*100)+str("%"))
```

```
Train age missing value: 19.865319865319865%
Test age missing value: 20.574162679425836%
```

We will take a different approach since **~20% data in the Age column is missing** in both train and test dataset. The age variable seems to be promising for determining survival rate. Therefore, It would be unwise to replace the missing values with median, mean or mode. We will use machine learning model Random Forest Regressor to impute missing value instead of Null value. We will keep the age column unchanged for now and work on that in the feature engineering section.

Visualization and Feature Relations

Before we dive into finding relations between independent variables and our dependent variable(survivor), let us create some assumptions about how the relations may turn-out among features.

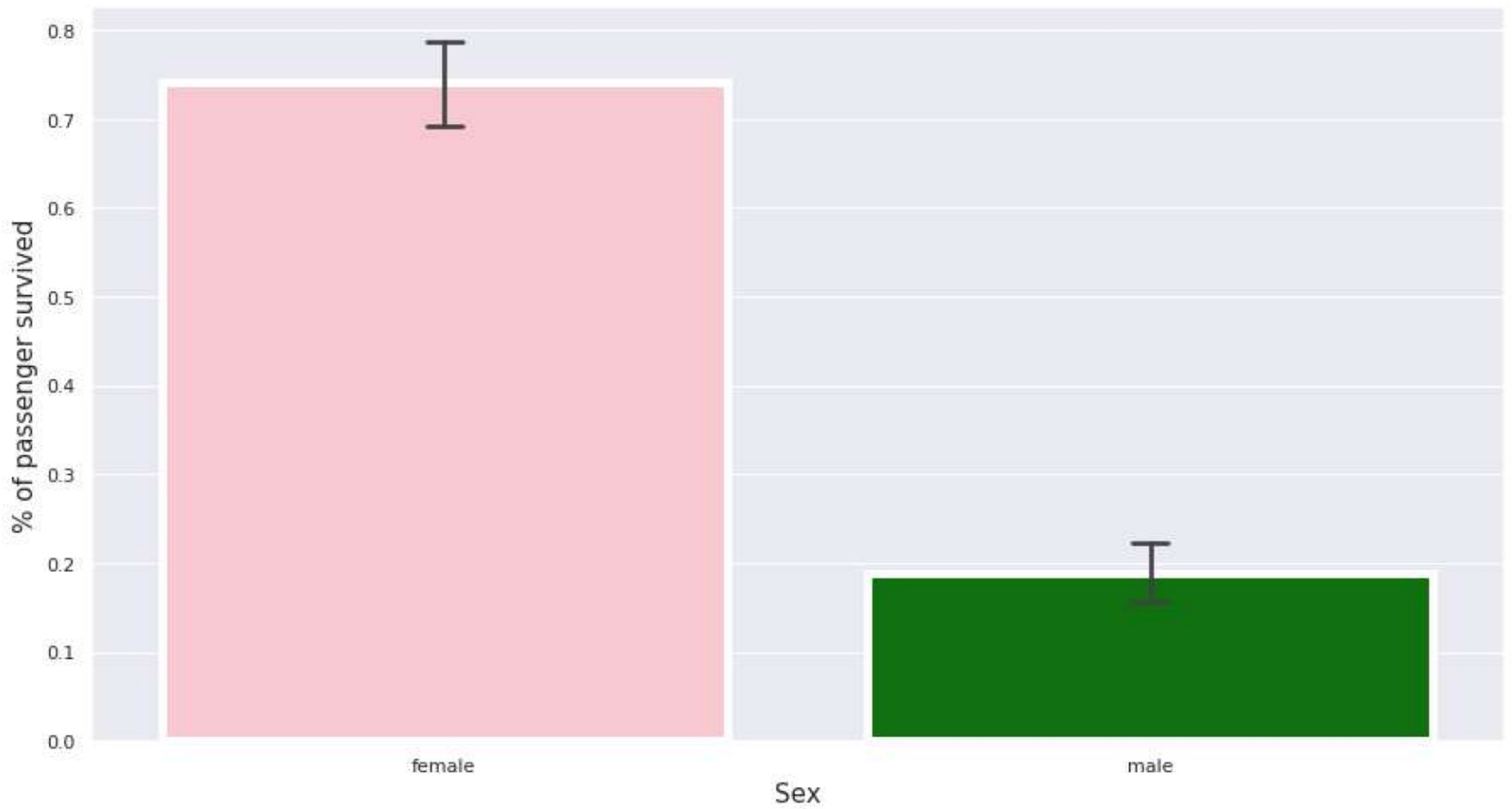
Assumptions:

- Gender: More female survived than male
- Pclass: Higher socio-economic status passenger survived more than others.
- Age: Younger passenger survived more than other passengers.
- Fare: Passenger with higher fare survived more that other passengers. This can be quite correlated with Pclass.

Now, let's see how the features are related to each other by creating some visualizations.

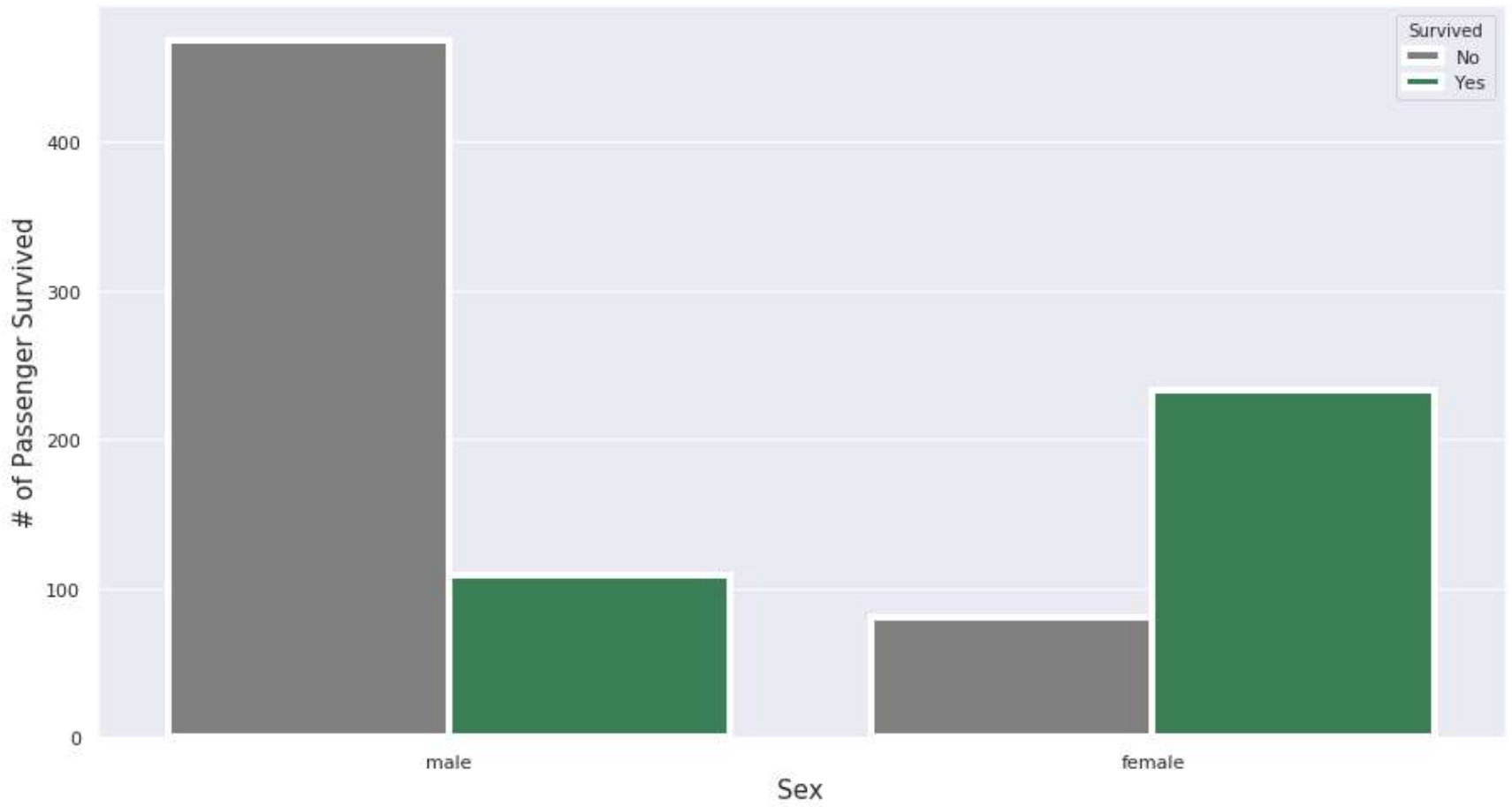
Gender and Survived

In [37]:
```
import seaborn as sns
pal = {'male':"green", 'female':"Pink"}
sns.set(style="darkgrid")
plt.subplots(figsize = (15,8))
ax = sns.barplot(x = "Sex",
                 y = "Survived",
                 data=train,
                 palette = pal,
                 linewidth=5,
                 order = ['female','male'],
                 capsize = .05,

                )

plt.title("Survived/Non-Survived Passenger Gender Distribution", fontsize = 25,loc = 'center', pad = 40)
plt.ylabel("% of passenger survived", fontsize = 15, )
plt.xlabel("Sex",fontsize = 15);
```



This bar plot above shows the distribution of female and male survived. The ***x_label*** represents **Sex** feature while the ***y_label*** represents the % of **passenger survived**. This bar plot shows that ~74% female passenger survived while only ~19% male passenger survived.

In [38]:
```
pal = {1:"seagreen", 0:"gray"}
sns.set(style="darkgrid")
plt.subplots(figsize = (15,8))
ax = sns.countplot(x = "Sex",
                   hue="Survived",
                   data = train,
                   linewidth=4,
                   palette = pal
)

## Fixing title, xlabel and ylabel
plt.title("Passenger Gender Distribution - Survived vs Not-survived", fontsize = 25, pad=40)
plt.xlabel("Sex", fontsize = 15);
plt.ylabel("# of Passenger Survived", fontsize = 15)

## Fixing xticks
#labels = ['Female', 'Male']
#plt.xticks(sorted(train.Sex.unique()), labels)

## Fixing legends
leg = ax.get_legend()
leg.set_title("Survived")
legs = leg.texts
legs[0].set_text("No")
legs[1].set_text("Yes")
plt.show()
```



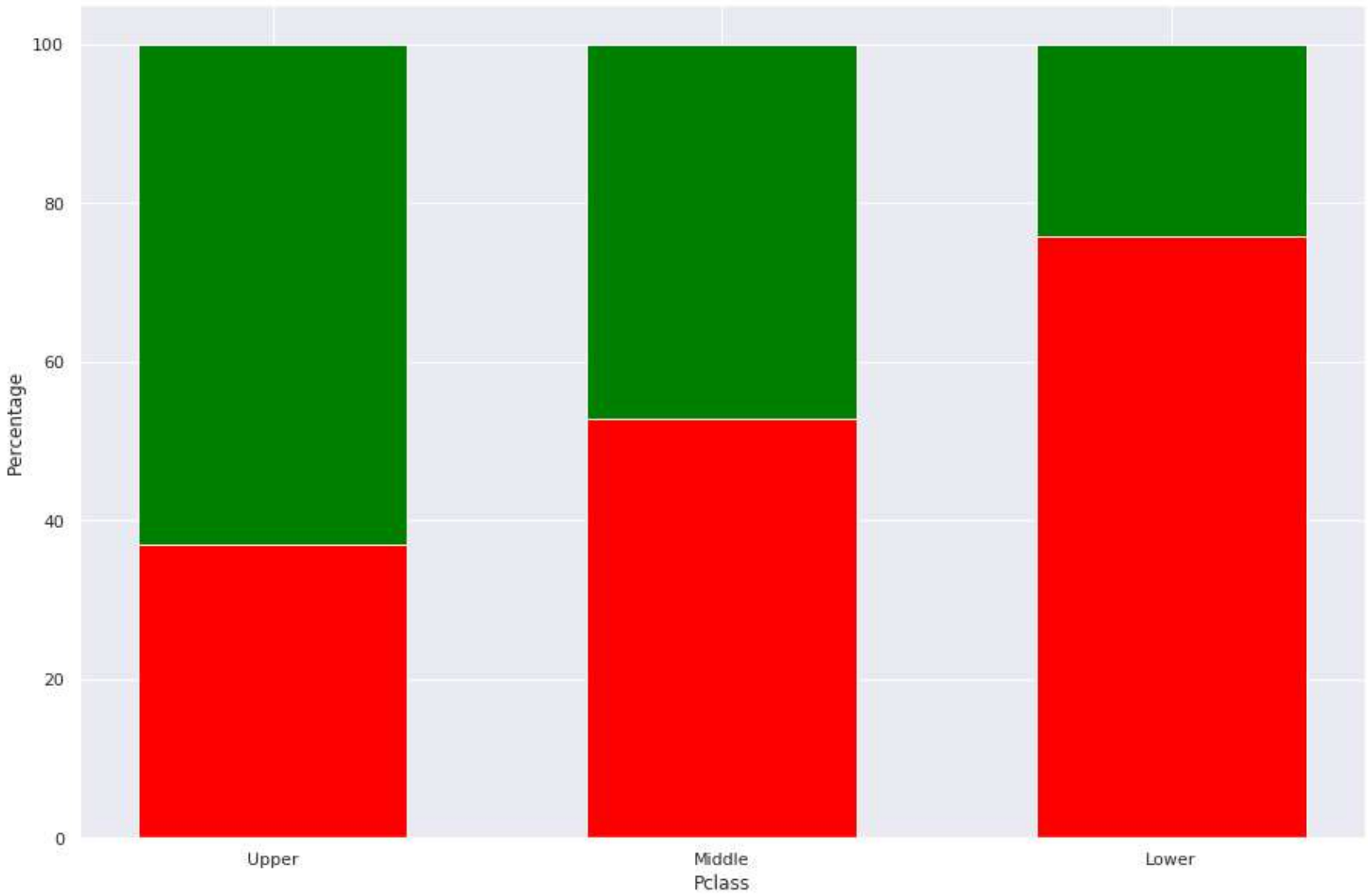
This count plot shows the actual distribution of male and female passengers that survived and did not survive. It shows that among all the females ~ 230 survived and ~ 70 did not survive. While among male passengers ~110 survived and ~480 did not survive.

Summary

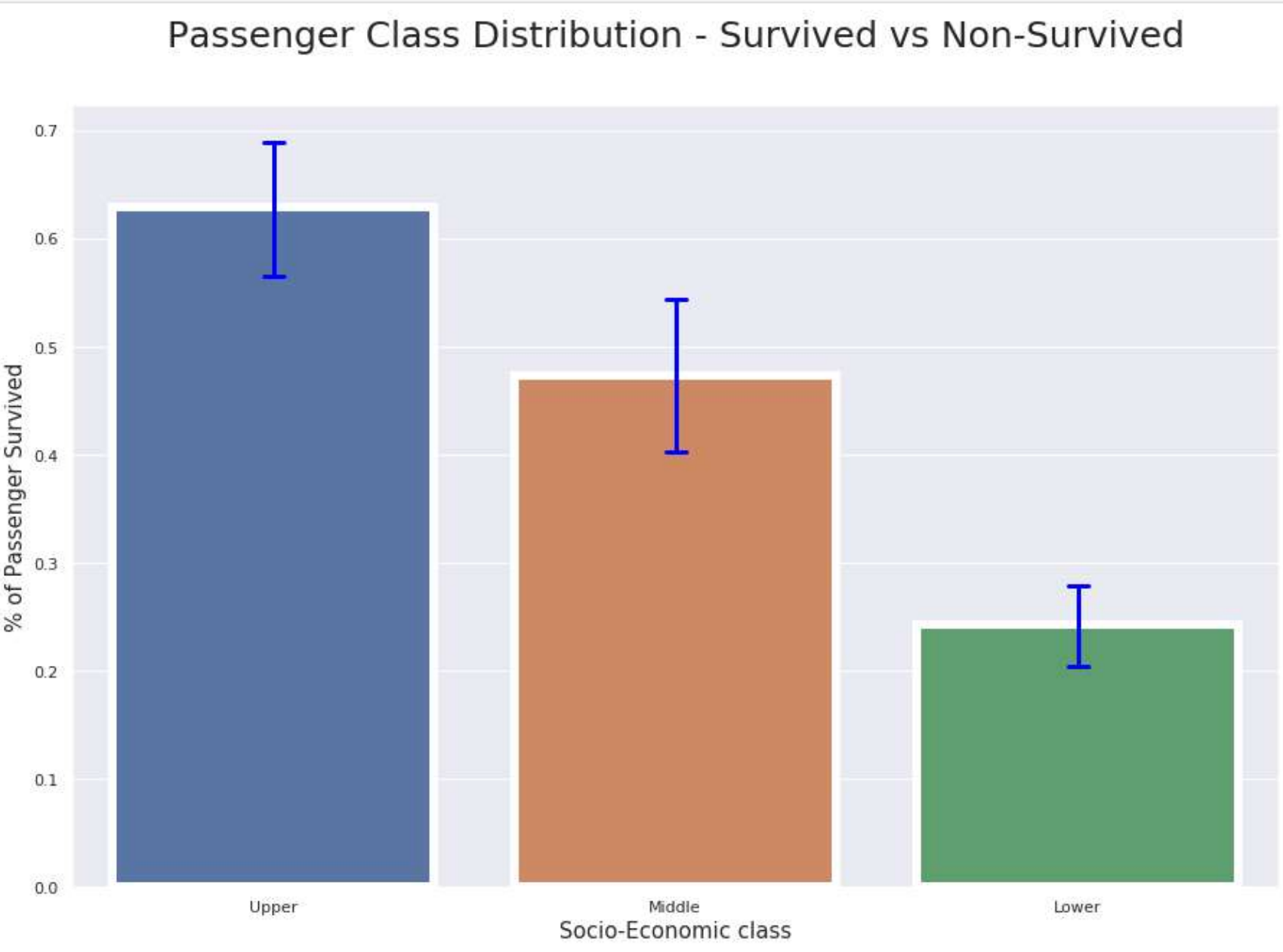
- As we suspected, female passengers have survived at a much better rate than male passengers.
- It seems about right since females and children were the priority.

Pclass and Survived

In [39]:
```
temp = train[['Pclass', 'Survived', 'PassengerId']].groupby(['Pclass', 'Survived']).count().reset_index()
temp_df = pd.pivot_table(temp, values = 'PassengerId', index = 'Pclass',columns = 'Survived')
names = ['No', 'Yes']
temp_df.columns = names
r = [0,1,2]
totals = [i+j for i, j in zip(temp_df['No'], temp_df['Yes'])]
No_s = [i / j * 100 for i,j in zip(temp_df['No'], totals)]
Yes_s = [i / j * 100 for i,j in zip(temp_df['Yes'], totals)]
## Plotting
plt.subplots(figsize = (15,10))
barWidth = 0.60
names = ('Upper', 'Middle', 'Lower')
# Create green Bars
plt.bar(r, No_s, color='Red', edgecolor='white', width=barWidth)
# Create orange Bars
plt.bar(r, Yes_s, bottom=No_s, color='Green', edgecolor='white', width=barWidth)


# Custom x axis
plt.xticks(r, names)
plt.xlabel("Pclass")
plt.ylabel('Percentage')

# Show graphic
plt.show()
```

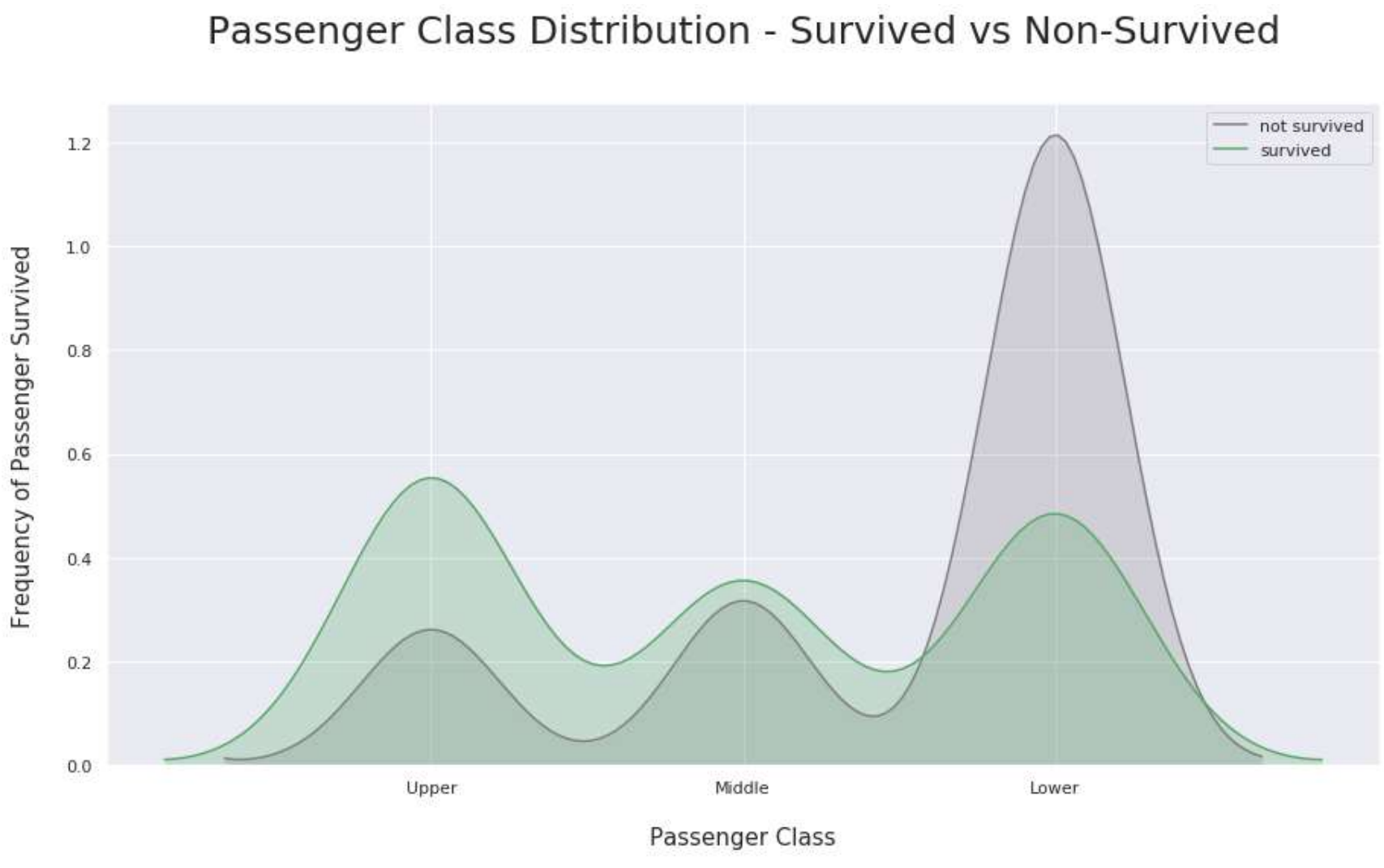


In [40]:
```
plt.subplots(figsize = (15,10))
sns.barplot(x = "Pclass",
            y = "Survived",
            data=train,
            linewidth=6,
            capsize = .05,
            errcolor='blue',
            errwidth = 3


           )
plt.title("Passenger Class Distribution - Survived vs Non-Survived", fontsize = 25, pad=40)
plt.xlabel("Socio-Economic class", fontsize = 15);
plt.ylabel("% of Passenger Survived", fontsize = 15);
names = ['Upper', 'Middle', 'Lower']
#val = sorted(train.Pclass.unique())
val = [0,1,2] ## this is just a temporary trick to get the label right.
plt.xticks(val, names);
```



- It looks like ...
  - ~ 63% first class passenger survived titanic tragedy, while
  - ~ 48% second class and
  - ~ only 24% third class passenger survived.

In [41]:
```
# Density Plot
fig = plt.figure(figsize=(15,8),)
## I have included to different ways to code a plot below, choose the one that suites you.
ax=sns.kdeplot(train.Pclass[train.Survived == 0] ,
               color='gray',
               shade=True,
               label='not survived')
ax=sns.kdeplot(train.loc[(train['Survived'] == 1),'Pclass'] ,
               color='g',
               shade=True,
               label='survived',
              )
plt.title('Passenger Class Distribution - Survived vs Non-Survived', fontsize = 25, pad = 40)
plt.ylabel("Frequency of Passenger Survived", fontsize = 15, labelpad = 20)
plt.xlabel("Passenger Class", fontsize = 15,labelpad =20)
## Converting xticks into words for better understanding
labels = ['Upper', 'Middle', 'Lower']
plt.xticks(sorted(train.Pclass.unique()), labels);
```



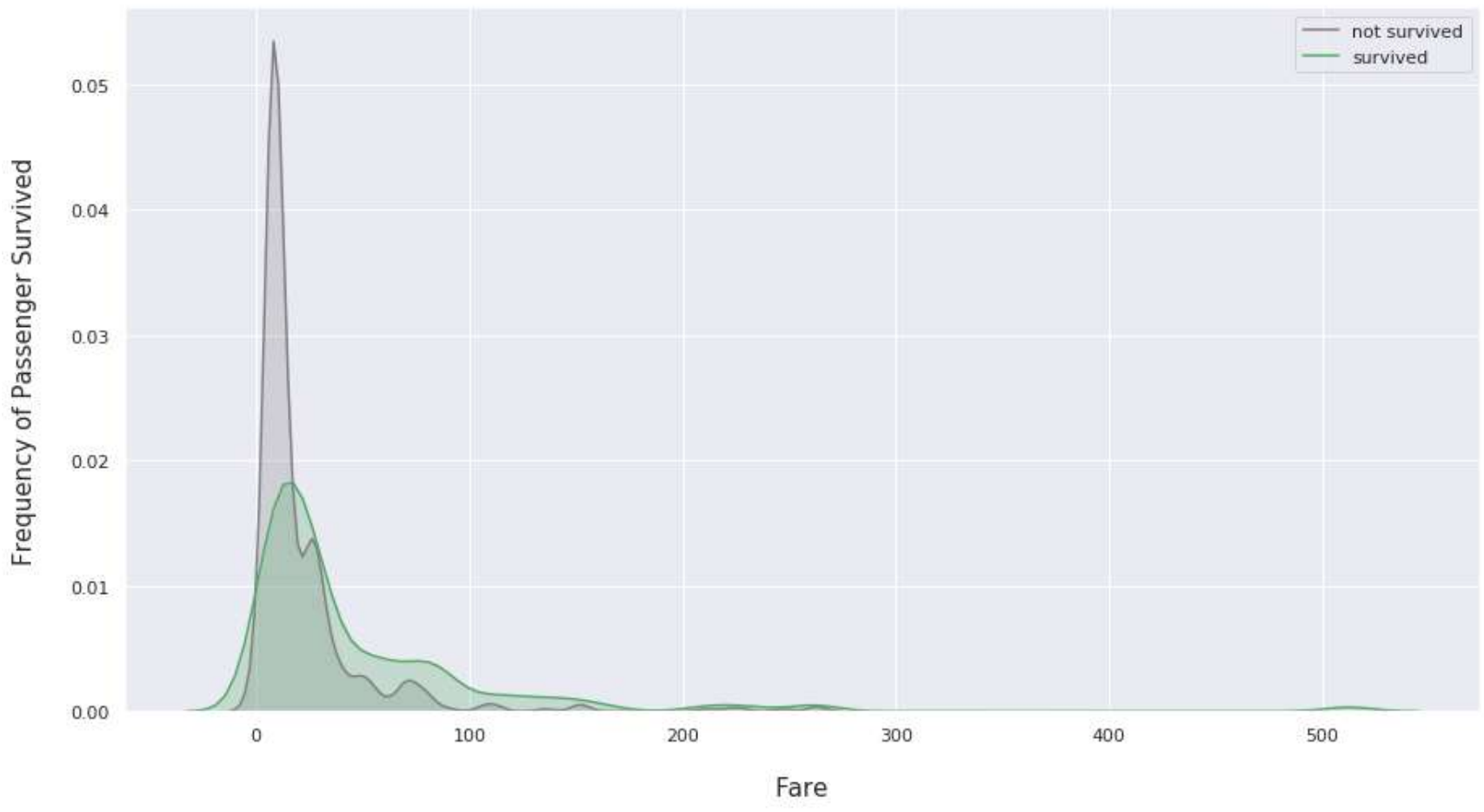
This KDE plot is pretty self-explanatory with all the labels and colors. Something I have noticed that some readers might find questionable is that the lower class passengers have survived more than second-class passengers. It is true since there were a lot more third-class passengers than first and second.

Summary

The first class passengers had the upper hand during the tragedy. You can probably agree with me more on this, in the next section of visualizations where we look at the distribution of ticket fare and survived column.

Fare and Survived

In [42]:
```
# Density Plot
fig = plt.figure(figsize=(15,8),)
ax=sns.kdeplot(train.loc[(train['Survived'] == 0),'Fare'] , color='gray',shade=True,label='not survived')
ax=sns.kdeplot(train.loc[(train['Survived'] == 1),'Fare'] , color='g',shade=True, label='survived')
plt.title('Fare Distribution Survived vs Non Survived', fontsize = 25, pad = 40)
plt.ylabel("Frequency of Passenger Survived", fontsize = 15, labelpad = 20)
plt.xlabel("Fare", fontsize = 15, labelpad = 20);
```



This plot shows something impressive..

- The spike in the plot under 100 dollar represents that a lot of passengers who bought the ticket within that range did not survive.
- When fare is approximately more than 280 dollars, there is no gray shade which means, either everyone passed that fare point survived or maybe there is an outlier that clouds our judgment. Let's check...

In [43]:
```
train[train.Fare > 280]
```

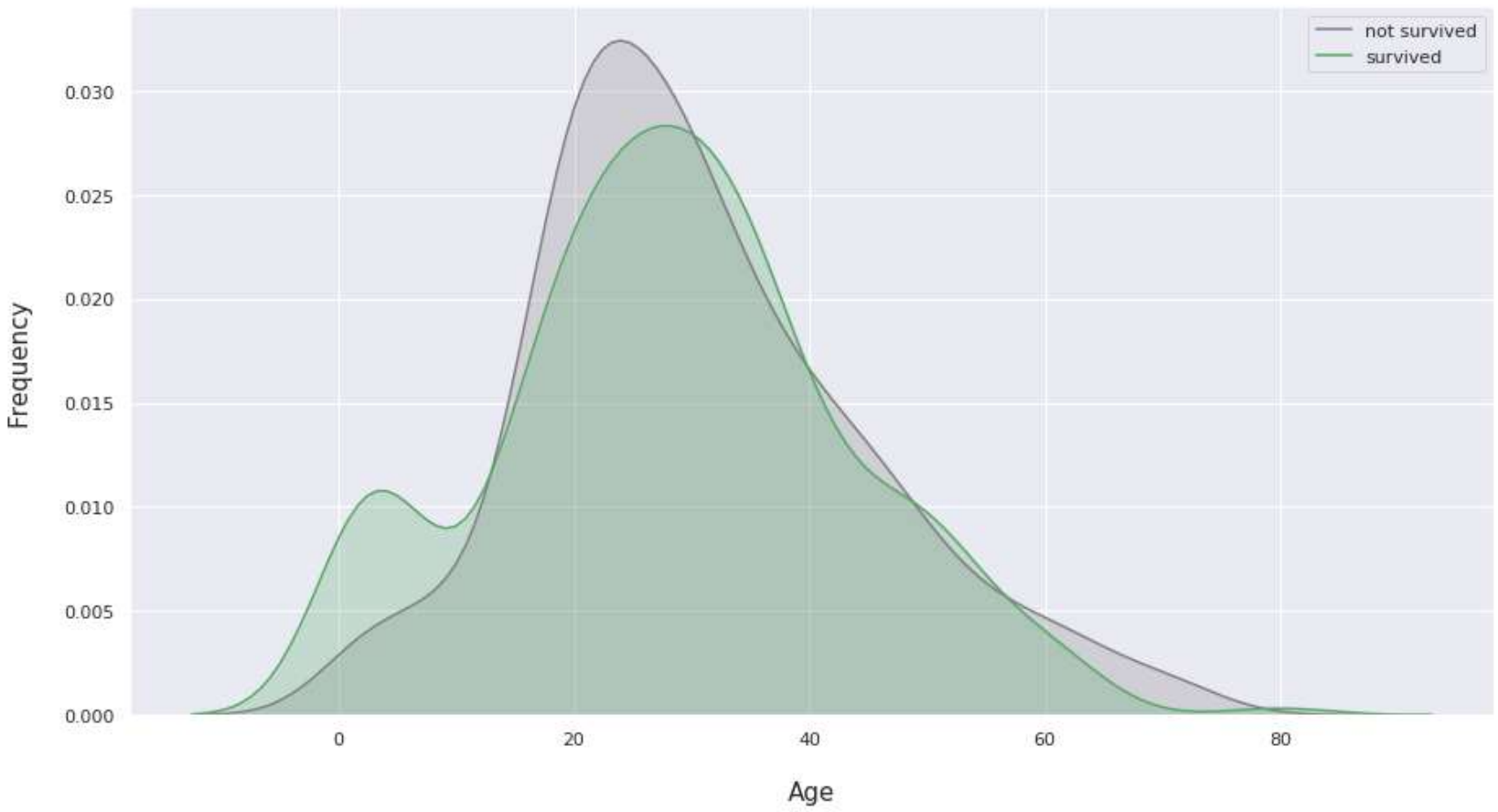
Out[43]:

	PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked	Survived
258	259	1	Ward, Miss. Anna	female	35.0	0	0	PC 17755	512.3292	B	C	1
679	680	1	Cardeza, Mr. Thomas Drake Martinez	male	36.0	0	1	PC 17755	512.3292	B	C	1
737	738	1	Lesurer, Mr. Gustave J	male	35.0	0	0	PC 17755	512.3292	B	C	1

As we assumed, it looks like an outlier with a fare of $512. We sure can delete this point. However, we will keep it for now.

Age and Survived

In [44]:
```
# Density Plot
fig = plt.figure(figsize=(15,8),)
ax=sns.kdeplot(train.loc[(train['Survived'] == 0),'Age'] , color='gray',shade=True,label='not survived')
ax=sns.kdeplot(train.loc[(train['Survived'] == 1),'Age'] , color='g',shade=True, label='survived')
plt.title('Age Distribution - Surviver V.S. Non Survivors', fontsize = 25, pad = 40)
plt.xlabel("Age", fontsize = 15, labelpad = 20)
plt.ylabel('Frequency', fontsize = 15, labelpad= 20);
```

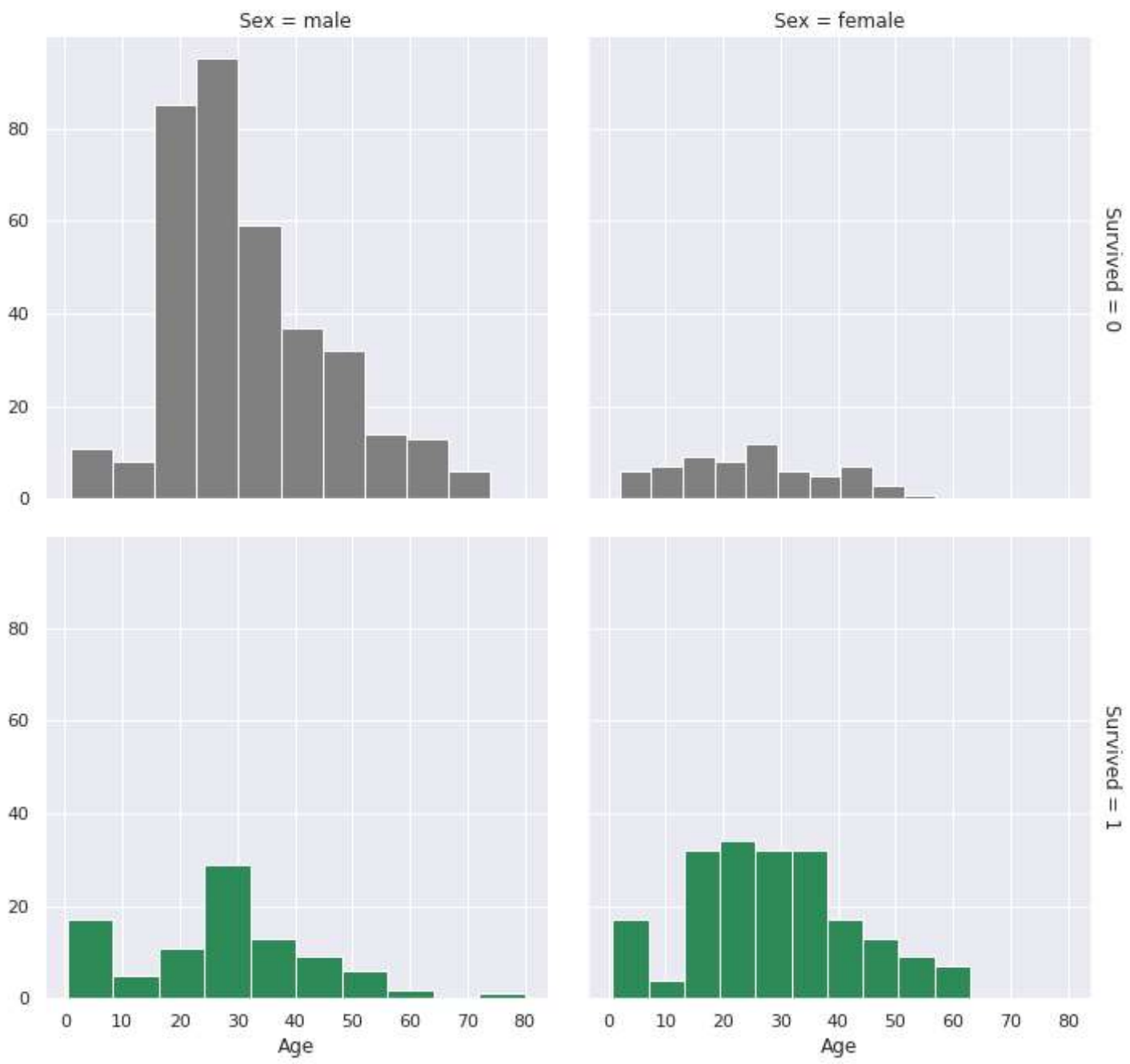


There is nothing out of the ordinary about this plot, except the very left part of the distribution. This may hint on the posibility that children and infants were the priority.

Combined Feature Relations

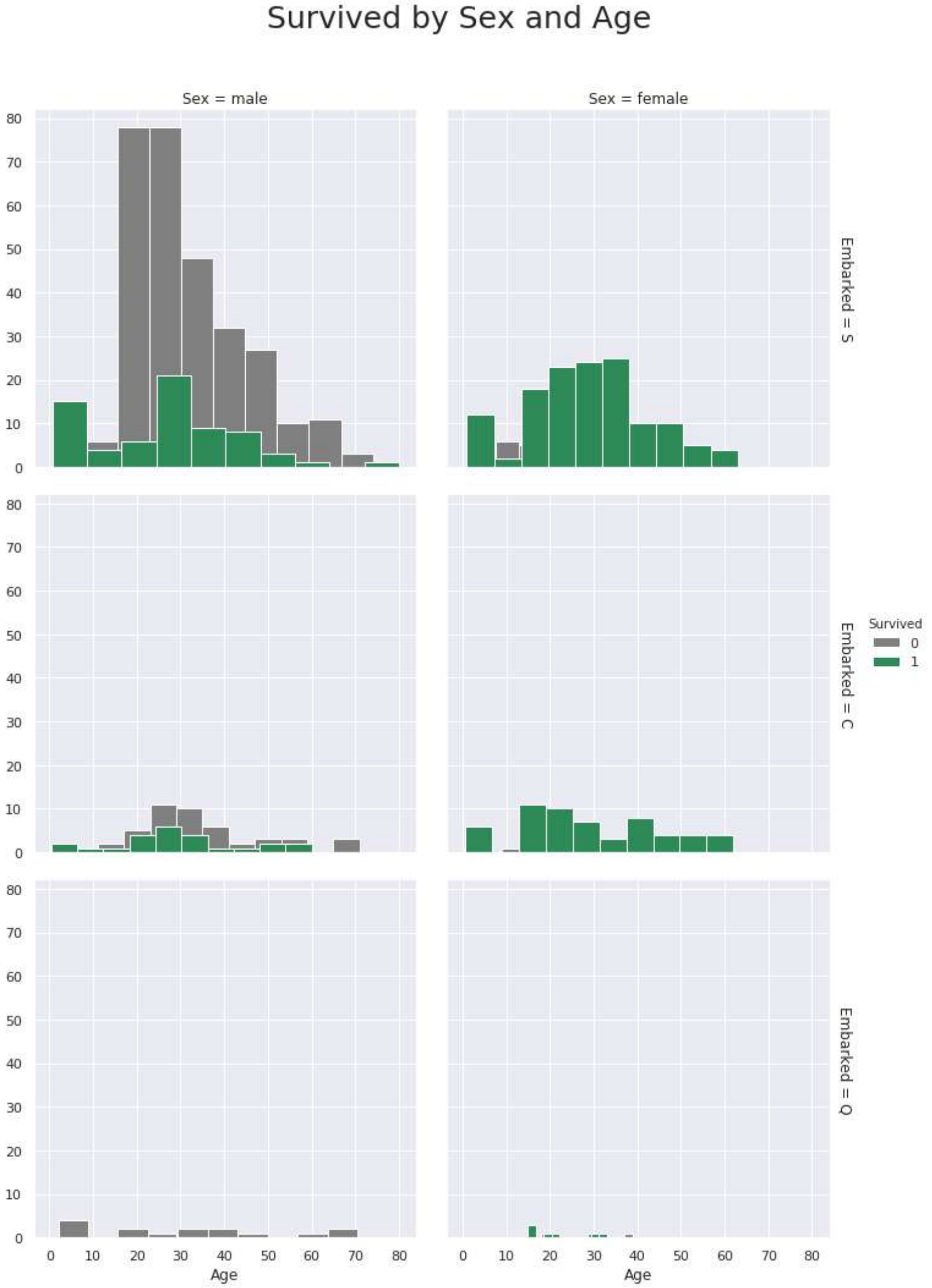
In this section, we are going to discover more than two feature relations in a single graph. I will try to illustrate most of the feature relations. Let's get to it.

In [45]:
```
pal = {1:"seagreen", 0:"gray"}
g = sns.FacetGrid(train,size=5, col="Sex", row="Survived", margin_titles=True, hue = "Survived",
                  palette=pal)
g = g.map(plt.hist, "Age", edgecolor = 'white');
g.fig.suptitle("Survived by Sex and Age", size = 25)
plt.subplots_adjust(top=0.90)
```



From this facet grid, we can also understand which age range groups survived more than others or were not so lucky

In [46]:
```
g = sns.FacetGrid(train,size=5, col="Sex", row="Embarked", margin_titles=True, hue = "Survived",
                  palette = pal
                  )
g = g.map(plt.hist, "Age", edgecolor = 'white').add_legend();
g.fig.suptitle("Survived by Sex and Age", size = 25)
plt.subplots_adjust(top=0.90)
```



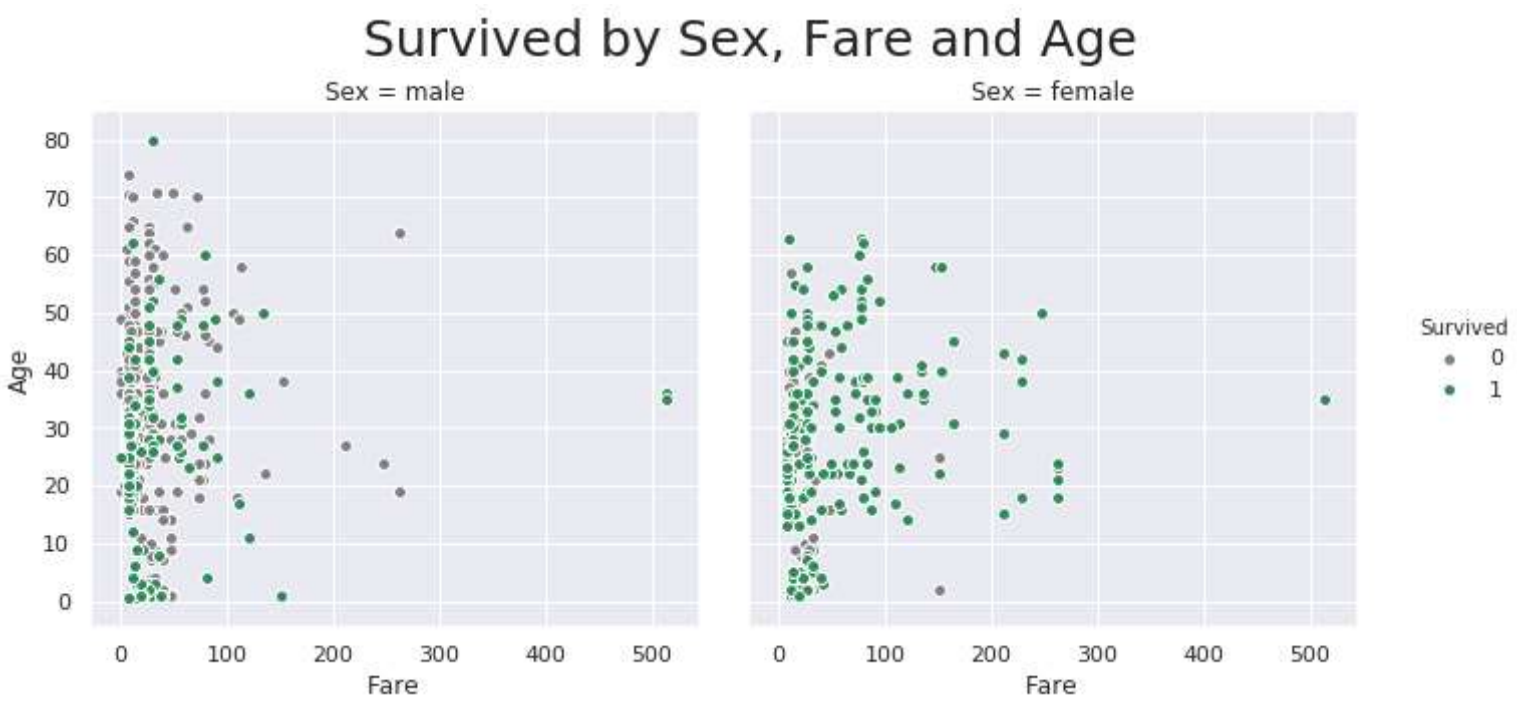
This is another compelling facet grid illustrating four features relationship at once. They are **Embarked, Age, Survived & Sex.**

- The color illustrates passengers survival status(green represents survived, gray represents not survived)
- The column represents Sex(left being male, right stands for female)
- The row represents Embarked(from top to bottom: S, C, Q)

Now that I have steered out the apparent let's see if we can get some insights that are not so obvious as we look at the data.

- Most passengers seem to be boarded on Southampton(S).
- More than 60% of the passengers died boarded on Southampton.
- More than 60% of the passengers lived boarded on Cherbourg(C).
- Pretty much every male that boarded on Queenstown(Q) did not survive.
- There were very few females boarded on Queenstown, however, most of them survived.

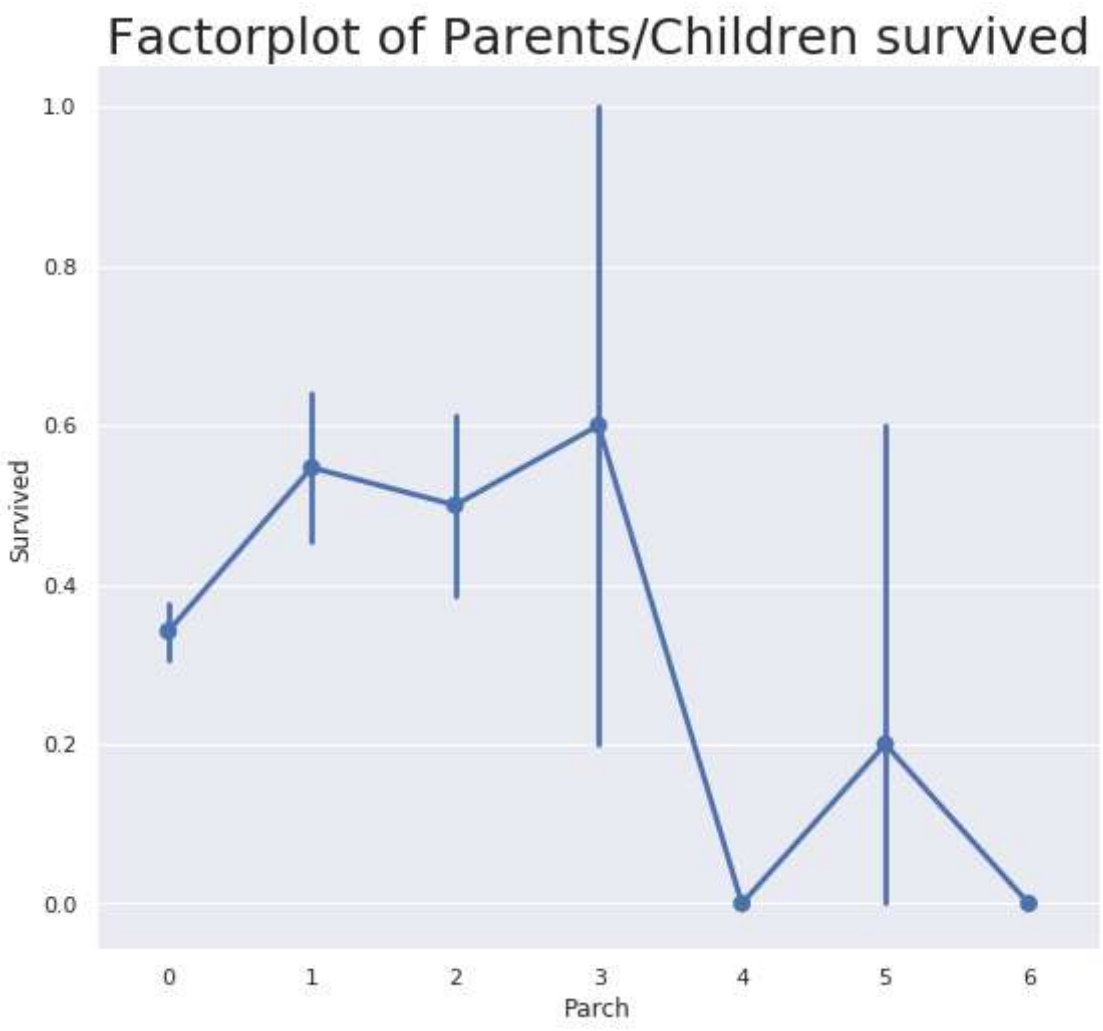
In [47]:
```
g = sns.FacetGrid(train, size=5,hue="Survived", col ="Sex", margin_titles=True,
                  palette=pal,)
g.map(plt.scatter, "Fare", "Age",edgecolor="w").add_legend()
g.fig.suptitle("Survived by Sex, Fare and Age", size = 25)
plt.subplots_adjust(top=0.85)
```



This facet grid unveils a couple of interesting insights. Let's find out.

- The grid above clearly demonstrates the three outliers with Fare of over $500. At this point, I think we are quite confident that these outliers should be deleted.
- Most of the passengers were with in the Fare range of $100.

In [48]:
```
## dropping the three outliers where Fare is over $500
train = train[train.Fare < 500]
## factor plot
sns.factorplot(x = "Parch", y = "Survived", data = train,kind = "point",size = 8)
plt.title("Factorplot of Parents/Children survived", fontsize = 25)
plt.subplots_adjust(top=0.85)
```



Passenger who traveled in big groups with parents/children had less survival rate than other passengers.

In [49]:
```
sns.factorplot(x =  "SibSp", y = "Survived", data = train,kind = "point",size = 8)
plt.title('Factorplot of Sibilings/Spouses survived', fontsize = 25)
plt.subplots_adjust(top=0.85)
```

While, passenger who traveled in small groups with sibilings/spouses had better changes of survival than other passengers.

In [50]:
```
# Placing 0 for female and 1 for male in the "Sex" column.
train['Sex'] = train.Sex.apply(lambda x: 0 if x == "female" else 1)
test['Sex'] = test.Sex.apply(lambda x: 0 if x == "female" else 1)
```

Statistical Overview

Train info

In [51]:
```
train.describe()
```

Out[51]:

	PassengerId	Pclass	Sex	Age	SibSp	Parch	Fare	Survived
count	888.000000	888.000000	888.000000	711.000000	888.000000	888.000000	888.000000	888.000000
mean	445.618243	2.313063	0.647523	29.675345	0.524775	0.381757	30.582164	0.381757
std	257.405474	0.834007	0.478011	14.552495	1.104186	0.806949	41.176366	0.486091
min	1.000000	1.000000	0.000000	0.420000	0.000000	0.000000	0.000000	0.000000
25%	222.750000	2.000000	0.000000	20.000000	0.000000	0.000000	7.895800	0.000000
50%	445.500000	3.000000	1.000000	28.000000	0.000000	0.000000	14.454200	0.000000
75%	667.250000	3.000000	1.000000	38.000000	1.000000	0.000000	30.771850	1.000000
max	891.000000	3.000000	1.000000	80.000000	8.000000	6.000000	263.000000	1.000000

In [52]:
```
train.describe(include =['O'])
```

Out[52]:

	Name	Ticket	Cabin	Embarked
count	888	888	888	888
unique	888	680	8	3
top	Klaber, Mr. Herman	CA. 2343	G	S
freq	1	7	464	644

In [53]:
```
# Overview(Survived vs non survied)
survived_summary = train.groupby("Survived")
survived_summary.mean().reset_index()
```

Out[53]:

Survived	PassengerId	Pclass	Sex	Age	SibSp	Parch	Fare
0	447.016393	2.531876	0.852459	30.626179	0.553734	0.329690	22.117887
1	443.353982	1.958702	0.315634	28.270627	0.477876	0.466077	44.289799

In [54]:
```
survived_summary = train.groupby("Sex")
survived_summary.mean().reset_index()
```

Out[54]:

	Sex	PassengerId	Pclass	Age	SibSp	Parch	Fare	Survived
0	0	431.578275	2.162939	27.888462	0.696486	0.651757	42.985091	0.741214
1	1	453.260870	2.394783	30.705477	0.431304	0.234783	23.830658	0.186087

In [55]:
```
survived_summary = train.groupby("Pclass")
survived_summary.mean().reset_index()
```

Out[55]:

	Pclass	PassengerId	Sex	Age	SibSp	Parch	Fare	Survived
0	1	460.225352	0.563380	38.280984	0.422535	0.356808	78.124061	0.624413
1	2	445.956522	0.586957	29.877630	0.402174	0.380435	20.662183	0.472826
2	3	439.154786	0.706721	25.140620	0.615071	0.393075	13.675550	0.242363

I have gathered a small summary from the statistical overview above. Let's see what they are...

- This train data set has 891 raw and 9 columns.
- only 38% passenger survived during that tragedy.
- ~74% female passenger survived, while only ~19% male passenger survived.
- ~63% first class passengers survived, while only 24% lower class passenger survived.

Correlation Matrix and Heatmap

Correlations

In [56]:
```
pd.DataFrame(abs(train.corr()['Survived']).sort_values(ascending = False))
```

Out[56]:

	Survived
Survived	1.000000
Sex	0.545899
Pclass	0.334068
Fare	0.261742
Parch	0.082157
Age	0.079472
SibSp	0.033395
PassengerId	0.006916

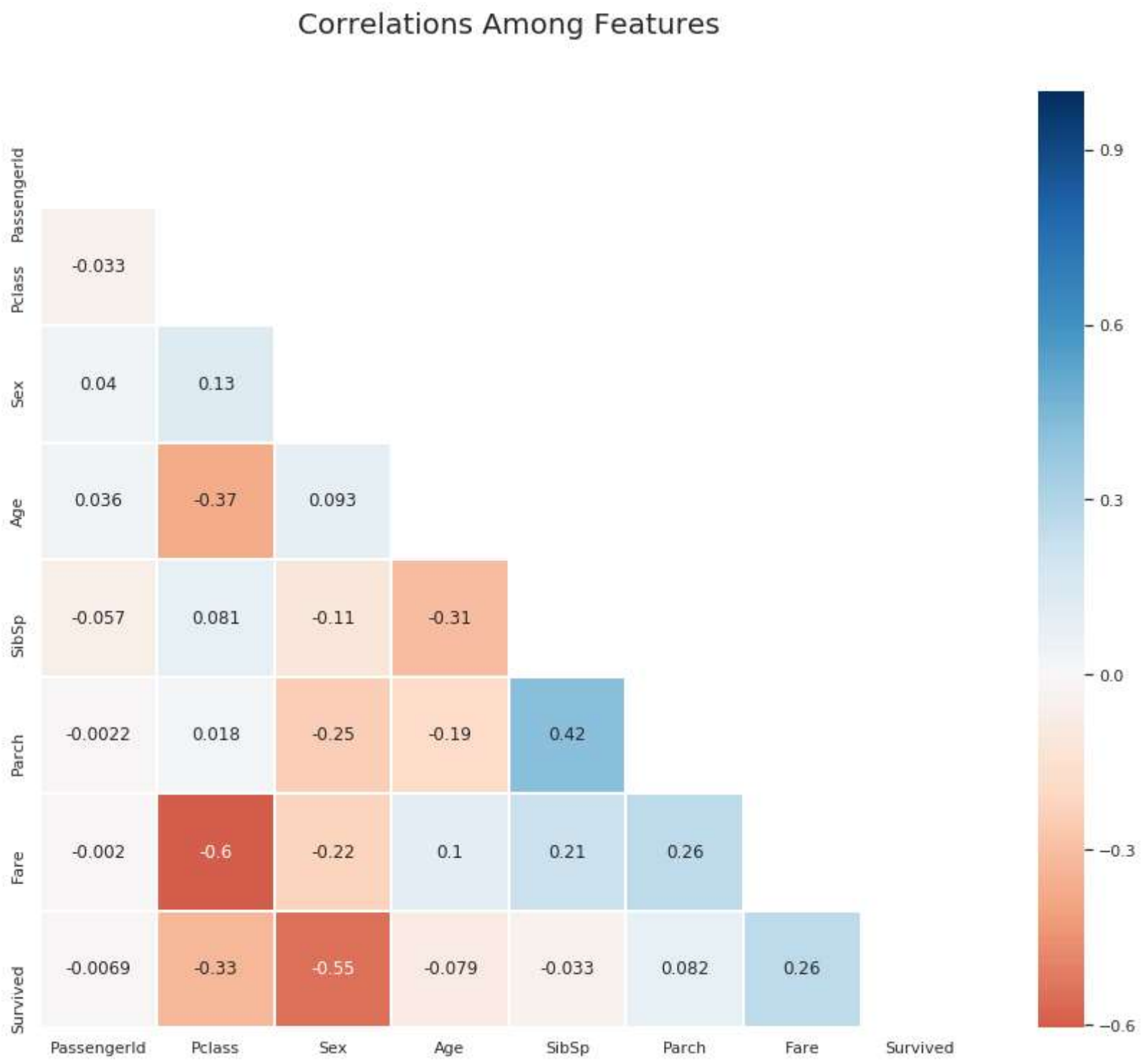
Sex is the most important correlated feature with *Survived(dependent variable)* feature followed by Pclass.

In [57]:
```
## get the most important variables.
corr = train.corr()**2
corr.Survived.sort_values(ascending=False)
```

Out[57]:
```
Survived       1.000000
Sex            0.298006
Pclass         0.111601
Fare           0.068509
Parch          0.006750
Age            0.006316
SibSp          0.001115
PassengerId    0.000048
Name: Survived, dtype: float64
```

Squaring the correlation feature not only gives on positive correlations but also amplifies the relationships.

In [58]:
```
## Heatmeap to see the correlation between features.
# Generate a mask for the upper triangle (taken from seaborn example gallery)
import numpy as np
mask = np.zeros_like(train.corr(), dtype=np.bool)
mask[np.triu_indices_from(mask)] = True
sns.set_style('whitegrid')
plt.subplots(figsize = (15,12))
sns.heatmap(train.corr(),
            annot=True,
            mask = mask,
            cmap = 'RdBu', ## in order to reverse the bar replace "RdBu" with "RdBu_r"
            linewidths=.9,
            linecolor='white',
            fmt='.2g',
            center = 0,
            square=True)
plt.title("Correlations Among Features", y = 1.03,fontsize = 20, pad = 40);
```



Positive Correlation Features:

- Fare and Survived: 0.26

Negative Correlation Features:

- Fare and Pclass: -0.6
- Sex and Survived: -0.55
- Pclass and Survived: -0.33

So, Let's analyze these correlations a bit. We've identified moderately strong relationships between different features. There's a clear positive correlation between Fare and Survival rate. This connection indicates that passengers who paid higher fares were more likely to survive. This theory aligns with another correlation, namely the relationship between Fare and Pclass (-0.6). Here, the correlation suggests that first-class passengers (1) paid more for their fare compared to second-class passengers (2), and similarly, second-class passengers paid more than third-class passengers (3). This theory is further supported by another correlation involving Pclass and our dependent variable, Survived, which stands at -0.33. This correlation implies that first-class passengers had better survival chances than those in second or third class, and so forth.

However, the most significant correlation with our dependent variable is the Sex variable, representing whether the passenger was male or female. This correlation has a substantial negative magnitude of -0.54, indicating some compelling insights. To assess the statistical significance of this correlation, let's conduct some statistical analysis.

Hypothesis Testing for Titanic

Formulating a well developed researched question:

Regarding this dataset, we can formulate the null hypothesis and alternative hypothesis by asking the following questions.

> - **Is there a significant difference in the mean sex between the passenger who survived and passenger who did not survive?.**
> - **Is there a substantial difference in the survival rate between the male and female passengers?**

In [59]:
```
male_mean = train[train['Sex'] == 1].Survived.mean()

female_mean = train[train['Sex'] == 0].Survived.mean()
print ("Male survival mean: " + str(male_mean))
print ("female survival mean: " + str(female_mean))

print ("The mean difference between male and female survival rate: " + str(female_mean - male_mean))
```

```
Male survival mean: 0.18608695652173912
female survival mean: 0.7412140575079872
The mean difference between male and female survival rate: 0.5551271009862481
```

In [60]:
```python
# separating male and female dataframe.
import random
male = train[train['Sex'] == 1]
female = train[train['Sex'] == 0]

## empty list for storing mean sample
m_mean_samples = []
f_mean_samples = []

for i in range(50):
    m_mean_samples.append(np.mean(random.sample(list(male['Survived']),50,)))
    f_mean_samples.append(np.mean(random.sample(list(female['Survived']),50,)))


# Print them out
print (f"Male mean sample mean: {round(np.mean(m_mean_samples),2)}")
print (f"Male mean sample mean: {round(np.mean(f_mean_samples),2)}")
print (f"Difference between male and female mean sample mean: {round(np.mean(f_mean_samples) - np.mean(m_mean_samples),
```

```
Male mean sample mean: 0.2
Male mean sample mean: 0.74
Difference between male and female mean sample mean: 0.54
```

H0: male mean is greater or equal to female mean

H1: male mean is less than female mean.

Feature Engineering

Feature Engineering is exactly what its sounds like. Sometimes we want to create extra features from with in the features that we have, sometimes we want to remove features that are alike. Features engineering is the simple word for doing all those. It is important to remember that we will create new features in such ways that will not cause **multicollinearity(when there is a relationship among independent variables)** to occur.

name_length

Creating a new feature "name_length" that will take the count of letters of each name

In [61]:
```python
# Creating a new colomn with a
train['name_length'] = [len(i) for i in train.Name]
test['name_length'] = [len(i) for i in test.Name]

def name_length_group(size):
    a = ''
    if (size <=20):
        a = 'short'
    elif (size <=35):
        a = 'medium'
    elif (size <=45):
        a = 'good'
    else:
        a = 'long'
    return a


train['nLength_group'] = train['name_length'].map(name_length_group)
test['nLength_group'] = test['name_length'].map(name_length_group)
```

title

Getting the title of each name as a new feature.

In [62]:
```python
## get the title from the name
train["title"] = [i.split('.')[0] for i in train.Name]
train["title"] = [i.split(',')[1] for i in train.title]
## Whenever we split like that, there is a good change that we will end up with while space around our string values. Le
```

In [63]:
```python
print(train.title.unique())
```

```
[' Mr' ' Mrs' ' Miss' ' Master' ' Don' ' Rev' ' Dr' ' Mme' ' Ms' ' Major'
 ' Lady' ' Sir' ' Mlle' ' Col' ' Capt' ' the Countess' ' Jonkheer']
```

In [64]:
```python
## Let's fix that
train.title = train.title.apply(lambda x: x.strip())
```

In [65]:
```python
## We can also combile all three lines above for test set here
test['title'] = [i.split('.')[0].split(',')[1].strip() for i in test.Name]

## However it is important to be able to write readable code, and the line above is not so readable.
```

In [66]:
```python
## Let's replace some of the rare values with the keyword 'rare' and other word choice of our own.
## train Data
```

```
train["title"] = [i.replace('Ms', 'Miss') for i in train.title]
train["title"] = [i.replace('Mlle', 'Miss') for i in train.title]
train["title"] = [i.replace('Mme', 'Mrs') for i in train.title]
train["title"] = [i.replace('Dr', 'rare') for i in train.title]
train["title"] = [i.replace('Col', 'rare') for i in train.title]
train["title"] = [i.replace('Major', 'rare') for i in train.title]
train["title"] = [i.replace('Don', 'rare') for i in train.title]
train["title"] = [i.replace('Jonkheer', 'rare') for i in train.title]
train["title"] = [i.replace('Sir', 'rare') for i in train.title]
train["title"] = [i.replace('Lady', 'rare') for i in train.title]
train["title"] = [i.replace('Capt', 'rare') for i in train.title]
train["title"] = [i.replace('the Countess', 'rare') for i in train.title]
train["title"] = [i.replace('Rev', 'rare') for i in train.title]
```

In [67]:
```
## we are writing a function that can help us modify title column
def name_converted(feature):
    """
    This function helps modifying the title column
    """

    result = ''
    if feature in ['the Countess','Capt','Lady','Sir','Jonkheer','Don','Major','Col', 'Rev', 'Dona', 'Dr']:
        result = 'rare'
    elif feature in ['Ms', 'Mlle']:
        result = 'Miss'
    elif feature == 'Mme':
        result = 'Mrs'
    else:
        result = feature
    return result

test.title = test.title.map(name_converted)
train.title = train.title.map(name_converted)
```

In [68]:
```
print(train.title.unique())
print(test.title.unique())
```

```
['Mr' 'Mrs' 'Miss' 'Master' 'rare']
['Mr' 'Mrs' 'Miss' 'Master' 'rare']
```

family_size

Creating a new feature called "family_size".

In [69]:
```
## Family_size seems like a good feature to create
train['family_size'] = train.SibSp + train.Parch+1
test['family_size'] = test.SibSp + test.Parch+1
```

In [70]:
```
## bin the family size.
def family_group(size):
    """
    This funciton groups(loner, small, large) family based on family size
    """

    a = ''
    if (size <= 1):
        a = 'loner'
    elif (size <= 4):
        a = 'small'
    else:
        a = 'large'
    return a
```

In [71]:
```
## apply the family_group function in family_size
train['family_group'] = train['family_size'].map(family_group)
test['family_group'] = test['family_size'].map(family_group)
```

is_alone

In [72]:
```
train['is_alone'] = [1 if i<2 else 0 for i in train.family_size]
test['is_alone'] = [1 if i<2 else 0 for i in test.family_size]
```

ticket

In [73]:
```
train.Ticket.value_counts().sample(10)
```

Out[73]:
```
2647          1
111240        1
345774        1
345764        2
350407        1
349256        1
2668          2
2649          1
C.A. 24579    1
113792        1
Name: Ticket, dtype: int64
```

In [74]:
```
train.drop(['Ticket'], axis=1, inplace=True)

test.drop(['Ticket'], axis=1, inplace=True)
```

Calculated_fare

In [75]:
```
## Calculating fare based on family size.
train['calculated_fare'] = train.Fare/train.family_size
test['calculated_fare'] = test.Fare/test.family_size
```

Some people have travelled in groups like family or friends. It seems like Fare column kept a record of the total fare rather than the fare of individual passenger, therefore calculated fare will be much handy in this situation.

fare_group

In [76]:
```
def fare_group(fare):
    """
    This function creates a fare group based on the fare provided
    """

    a= ''
    if fare <= 4:
        a = 'Very_low'
    elif fare <= 10:
        a = 'low'
    elif fare <= 20:
        a = 'mid'
    elif fare <= 45:
        a = 'high'
    else:
        a = "very_high"
    return a

train['fare_group'] = train['calculated_fare'].map(fare_group)
test['fare_group'] = test['calculated_fare'].map(fare_group)

#train['fare_group'] = pd.cut(train['calculated_fare'], bins = 4, labels=groups)
```

Fare group was calculated based on *calculated_fare*. This can further help our cause.

PassengerId

It seems like *PassengerId* column only works as an id in this dataset without any significant effect on the dataset. Let's drop it.

In [77]:
```
train.drop(['PassengerId'], axis=1, inplace=True)

test.drop(['PassengerId'], axis=1, inplace=True)
```

Creating dummy variables

In [78]:
```
train = pd.get_dummies(train, columns=['title',"Pclass", 'Cabin','Embarked','nLength_group', 'family_group', 'fare_grou
test = pd.get_dummies(test, columns=['title',"Pclass",'Cabin','Embarked','nLength_group', 'family_group', 'fare_group']
train.drop(['family_size','Name', 'Fare','name_length'], axis=1, inplace=True)
test.drop(['Name','family_size',"Fare",'name_length'], axis=1, inplace=True)
```

age

we are going to use Random forest regressor in this section to predict the missing age values.

In [79]:
```
train.head()
```

Out[79]:

	Sex	Age	SibSp	Parch	Survived	is_alone	calculated_fare	title_Master	title_Miss	title_Mr	...	nLength_group_medium	nLength_group_s
0	1	22.0	1	0	0	0	3.62500	0	0	1	...	1	
1	0	38.0	1	0	1	0	35.64165	0	0	0	...	0	
2	0	26.0	0	0	1	1	7.92500	0	1	0	...	1	
3	0	35.0	1	0	1	0	26.55000	0	0	0	...	0	
4	1	35.0	0	0	0	1	8.05000	0	0	1	...	1	

5 rows × 38 columns

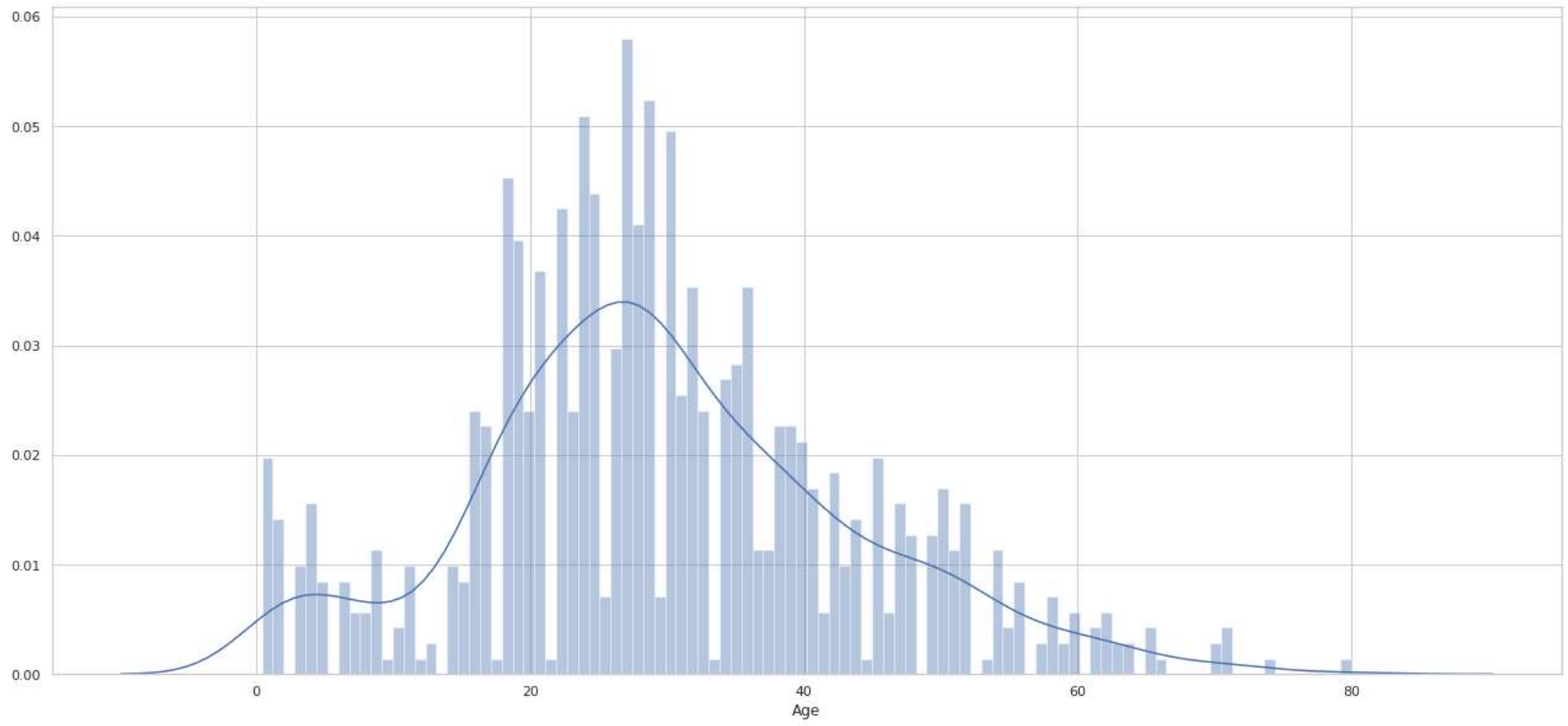
In [80]:
```
## rearranging the columns so that I can easily use the dataframe to predict the missing age values.
train = pd.concat([train[["Survived", "Age", "Sex","SibSp","Parch"]], train.loc[:,"is_alone":]], axis=1)
test = pd.concat([test[["Age", "Sex"]], test.loc[:,"SibSp":]], axis=1)
```

In [81]:
```
## Importing RandomForestRegressor
from sklearn.ensemble import RandomForestRegressor
```

```
## writing a function that takes a dataframe with missing values and outputs it by filling the missing values.
def completing_age(df):
    ## gettting all the features except survived
    age_df = df.loc[:,"Age":]

    temp_train = age_df.loc[age_df.Age.notnull()] ## df with age values
    temp_test = age_df.loc[age_df.Age.isnull()] ## df without age values

    y = temp_train.Age.values ## setting target variables(age) in y
    x = temp_train.loc[:, "Sex":].values

    rfr = RandomForestRegressor(n_estimators=1500, n_jobs=-1)
    rfr.fit(x, y)

    predicted_age = rfr.predict(temp_test.loc[:, "Sex":])

    df.loc[df.Age.isnull(), "Age"] = predicted_age


    return df

## Implementing the completing_age function in both train and test dataset.
completing_age(train)
completing_age(test);
```

Let's take a look at the histgram of the age column.

In [82]:
```
plt.subplots(figsize = (22,10),)
sns.distplot(train.Age, bins = 100, kde = True, rug = False, norm_hist=False);
```



age_group

We can create a new feature by grouping the "Age" column

In [83]:
```
## create bins for age
def age_group_fun(age):
    """
    This function creates a bin for age
    """
    a = ''
    if age <= 1:
        a = 'infant'
    elif age <= 4:
        a = 'toddler'
    elif age <= 13:
        a = 'child'
    elif age <= 18:
        a = 'teenager'
    elif age <= 35:
        a = 'Young_Adult'
    elif age <= 45:
        a = 'adult'
    elif age <= 55:
        a = 'middle_aged'
    elif age <= 65:
        a = 'senior_citizen'
    else:
        a = 'old'
    return a

## Applying "age_group_fun" function to the "Age" column.
train['age_group'] = train['Age'].map(age_group_fun)
test['age_group'] = test['Age'].map(age_group_fun)

## Creating dummies for "age_group" feature.
```

```
train = pd.get_dummies(train,columns=['age_group'], drop_first=True)
test = pd.get_dummies(test,columns=['age_group'], drop_first=True);
```

Pre-Modeling Tasks

In [84]:
```
# separating our independent and dependent variable
X = train.drop(['Survived'], axis = 1)
y = train["Survived"]
```

Splitting the training data

In [85]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,test_size = .33, random_state=0)
```

In [86]:
```
len(X_train)
```

Out[86]:
```
594
```

In [87]:
```
len(X_test)
```

Out[87]:
```
294
```

Feature Scaling

In [88]:
```
train.sample(5)
```

Out[88]:

	Survived	Age	Sex	SibSp	Parch	is_alone	calculated_fare	title_Master	title_Miss	title_Mr	...	fare_group_mid	fare_group_very_h
738	0	27.312834	1	0	0	1	7.8958	0	0	1	...	0	
699	0	42.000000	1	0	0	1	7.6500	0	0	1	...	0	
196	0	51.748639	1	0	0	1	7.7500	0	0	1	...	0	
195	1	58.000000	0	0	0	1	146.5208	0	1	0	...	0	
166	1	41.172778	0	0	1	0	27.5000	0	0	0	...	0	

5 rows × 46 columns

Before Scaling

In [89]:
```
headers = X_train.columns

X_train.head()
```

Out[89]:

	Age	Sex	SibSp	Parch	is_alone	calculated_fare	title_Master	title_Miss	title_Mr	title_Mrs	...	fare_group_mid	fare_group_very_h
170	61.000000	1	0	0	1	33.5000	0	0	1	0	...	0	
187	45.000000	1	0	0	1	26.5500	0	0	1	0	...	0	
849	41.374667	0	1	0	0	44.5521	0	0	0	1	...	0	
433	17.000000	1	0	0	1	7.1250	0	0	1	0	...	0	
651	18.000000	0	0	1	0	11.5000	0	1	0	0	...	1	

5 rows × 45 columns

In [90]:
```
# Feature Scaling
## We will be using standardscaler to transform
from sklearn.preprocessing import StandardScaler
st_scale = StandardScaler()

## transforming "train_x"
X_train = st_scale.fit_transform(X_train)
## transforming "test_x"
X_test = st_scale.transform(X_test)

## transforming "The testset"
#test = st_scale.transform(test)
```

After Scaling

In [91]:
```
pd.DataFrame(X_train, columns=headers).head()
```

Out[91]:

	Age	Sex	SibSp	Parch	is_alone	calculated_fare	title_Master	title_Miss	title_Mr	title_Mrs	...	fare_group_mid	fare
0	2.256512	0.725942	-0.464750	-0.463616	0.794901	0.554725	-0.230633	-0.521487	0.837858	-0.383038	...	-0.455321	
1	1.115436	0.725942	-0.464750	-0.463616	0.794901	0.292298	-0.230633	-0.521487	0.837858	-0.383038	...	-0.455321	
2	0.856887	-1.377520	0.356862	-0.463616	-1.258018	0.972044	-0.230633	-0.521487	-1.193520	2.610707	...	-0.455321	
3	-0.881447	0.725942	-0.464750	-0.463616	0.794901	-0.441176	-0.230633	-0.521487	0.837858	-0.383038	...	-0.455321	
4	-0.810130	-1.377520	-0.464750	0.703282	-1.258018	-0.275979	-0.230633	1.917594	-1.193520	-0.383038	...	2.196253	

5 rows × 45 columns

You can see how the features have transformed above.

Modeling the Data

In [92]:
```
# import LogisticRegression model.
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import mean_absolute_error, accuracy_score

## call on the model object
logreg = LogisticRegression(solver='liblinear',
                            penalty= 'l1',random_state = 42

                            )

## fit the model with "train_x" and "train_y"
logreg.fit(X_train,y_train)

## Once the model is trained we want to find out how well the model is performing, so we test the model.
## we use "X_test" portion of the data(this data was not used to fit the model) to predict model outcome.
y_pred = logreg.predict(X_test)

## Once predicted we save that outcome in "y_pred" variable.
## Then we compare the predicted value( "y_pred") and actual value("test_y") to see how well our model is performing.
```

Evaluating a classification model

In [93]:
```
from sklearn.metrics import classification_report, confusion_matrix
# printing confision matrix
pd.DataFrame(confusion_matrix(y_test,y_pred),\
             columns=["Predicted Not-Survived", "Predicted Survived"],\
             index=["Not-Survived","Survived"] )
```

Out[93]:

	Predicted Not-Survived	Predicted Survived
Not-Survived	149	28
Survived	30	87

In [94]:
```
from sklearn.metrics import accuracy_score
accuracy_score(y_test, y_pred)
```

Out[94]:
```
0.8027210884353742
```

In [95]:
```
from sklearn.metrics import recall_score
recall_score(y_test, y_pred)
```

Out[95]:
```
0.7435897435897436
```

In [96]:
```
from sklearn.metrics import precision_score
precision_score(y_test, y_pred)
```

Out[96]:
```
0.7565217391304347
```

In [97]:
```
from sklearn.metrics import classification_report, balanced_accuracy_score
print(classification_report(y_test, y_pred))
```

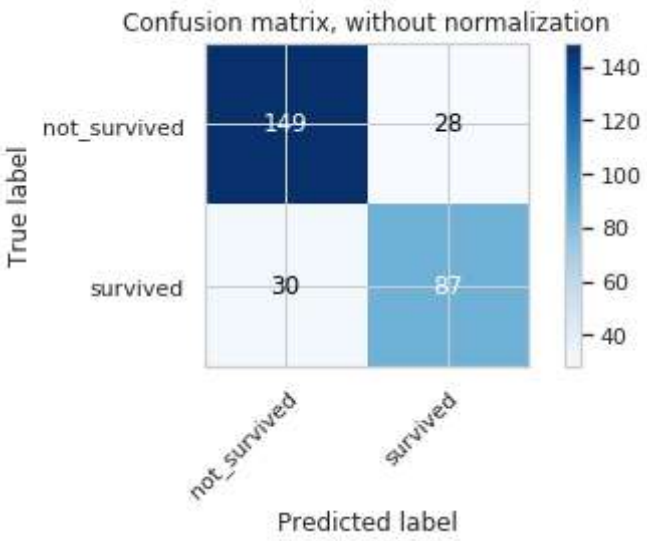
```
              precision    recall  f1-score   support

           0       0.83      0.84      0.84       177
           1       0.76      0.74      0.75       117

    accuracy                           0.80       294
   macro avg       0.79      0.79      0.79       294
weighted avg       0.80      0.80      0.80       294
```

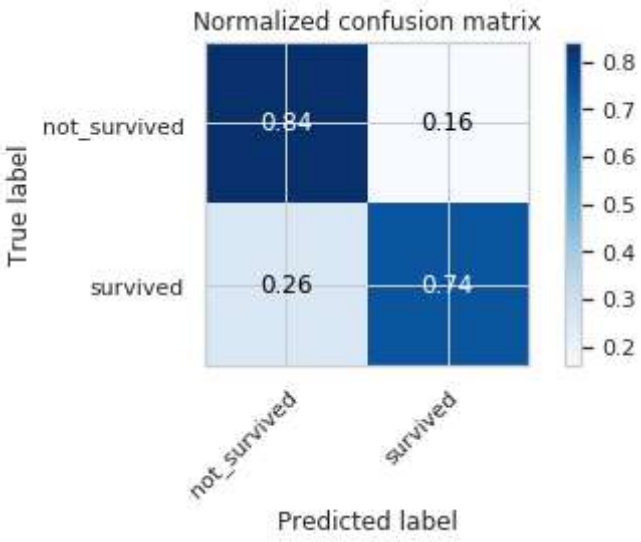
we have our confusion matrix. How about we give it a little more character.

In [98]:
```
from sklearn.utils.multiclass import unique_labels
from sklearn.metrics import confusion_matrix


def plot_confusion_matrix(y_true, y_pred, classes,
                          normalize=False,
```

```
                      title=None,
                      cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if not title:
        if normalize:
            title = 'Normalized confusion matrix'
        else:
            title = 'Confusion matrix, without normalization'

    # Compute confusion matrix
    cm = confusion_matrix(y_true, y_pred)
    # Only use the labels that appear in the data
    classes = classes[unique_labels(y_true, y_pred)]
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    print(cm)

    fig, ax = plt.subplots()
    im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
    ax.figure.colorbar(im, ax=ax)
    # We want to show all ticks...
    ax.set(xticks=np.arange(cm.shape[1]),
           yticks=np.arange(cm.shape[0]),
           # ... and label them with the respective list entries
           xticklabels=classes, yticklabels=classes,
           title=title,
           ylabel='True label',
           xlabel='Predicted label')

    # Rotate the tick labels and set their alignment.
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
             rotation_mode="anchor")

    # Loop over data dimensions and create text annotations.
    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            ax.text(j, i, format(cm[i, j], fmt),
                    ha="center", va="center",
                    color="white" if cm[i, j] > thresh else "black")
    fig.tight_layout()
    return ax


np.set_printoptions(precision=2)

class_names = np.array(['not_survived','survived'])

# Plot non-normalized confusion matrix
plot_confusion_matrix(y_test, y_pred, classes=class_names,
                      title='Confusion matrix, without normalization')

# Plot normalized confusion matrix
plot_confusion_matrix(y_test, y_pred, classes=class_names, normalize=True,
                      title='Normalized confusion matrix')

plt.show()
```

```
Confusion matrix, without normalization
[[149  28]
 [ 30  87]]
Normalized confusion matrix
[[0.84 0.16]
 [0.26 0.74]]
```

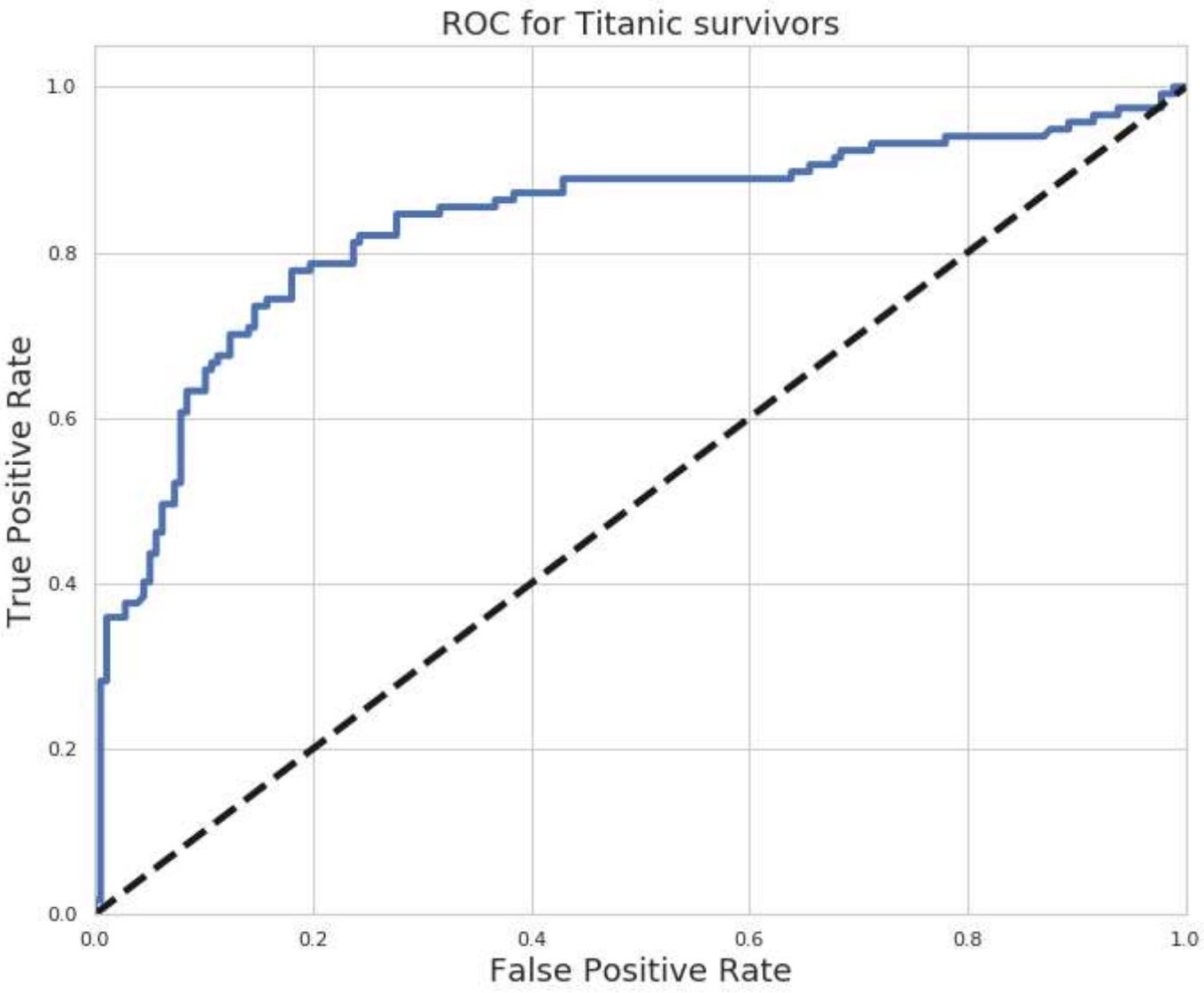




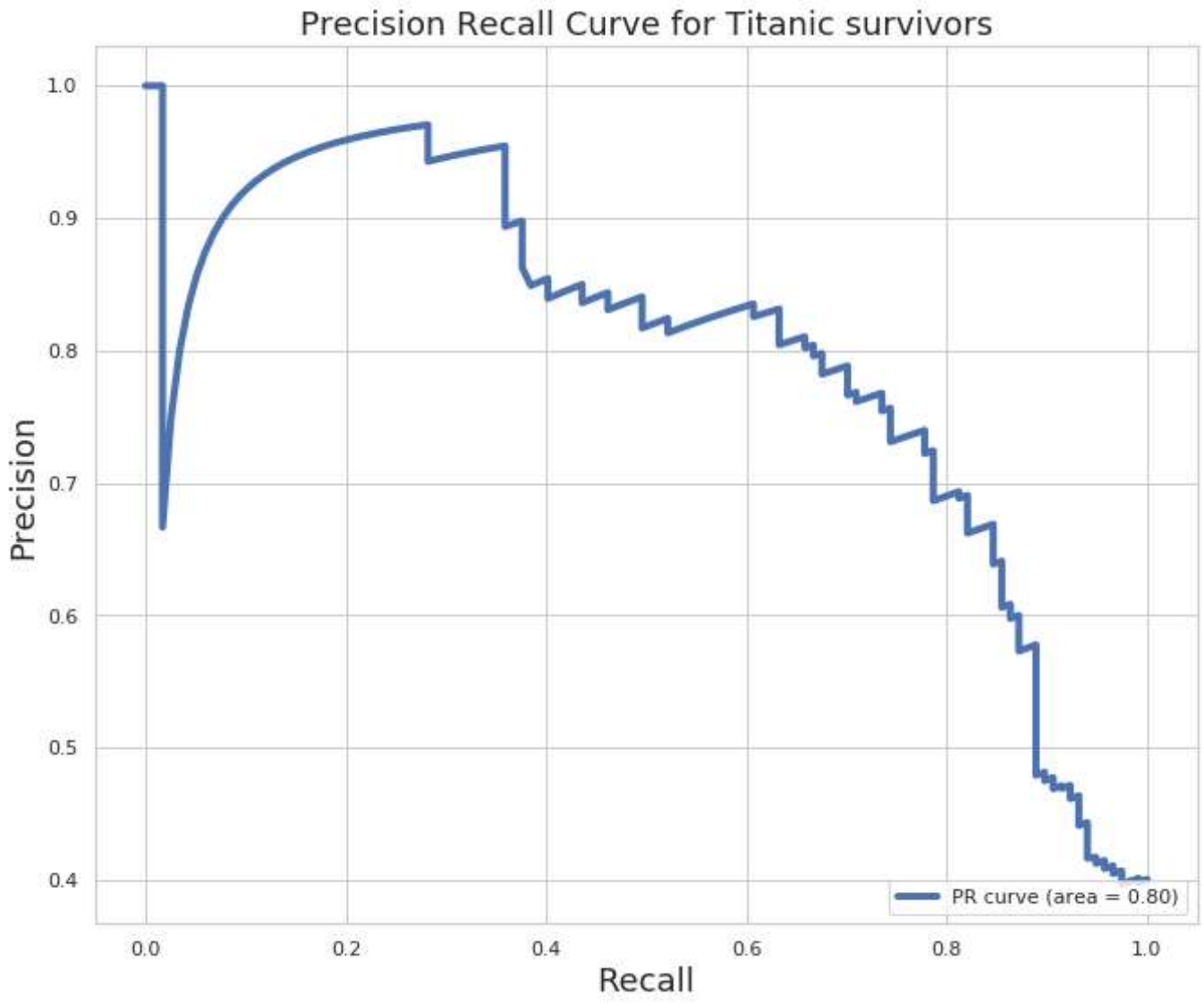
AUC & ROC Curve

In [99]:
```
from sklearn.metrics import roc_curve, auc
#plt.style.use('seaborn-pastel')
y_score = logreg.decision_function(X_test)

FPR, TPR, _ = roc_curve(y_test, y_score)
ROC_AUC = auc(FPR, TPR)
print (ROC_AUC)

plt.figure(figsize =[11,9])
plt.plot(FPR, TPR, label= 'ROC curve(area = %0.2f)'%ROC_AUC, linewidth= 4)
plt.plot([0,1],[0,1], 'k--', linewidth = 4)
plt.xlim([0.0,1.0])
plt.ylim([0.0,1.05])
plt.xlabel('False Positive Rate', fontsize = 18)
plt.ylabel('True Positive Rate', fontsize = 18)
plt.title('ROC for Titanic survivors', fontsize= 18)
plt.show()
```

```
0.8347578347578346
```



In [100…
```
from sklearn.metrics import precision_recall_curve

y_score = logreg.decision_function(X_test)

precision, recall, _ = precision_recall_curve(y_test, y_score)
PR_AUC = auc(recall, precision)

plt.figure(figsize=[11,9])
plt.plot(recall, precision, label='PR curve (area = %0.2f)' % PR_AUC, linewidth=4)
plt.xlabel('Recall', fontsize=18)
plt.ylabel('Precision', fontsize=18)
plt.title('Precision Recall Curve for Titanic survivors', fontsize=18)
plt.legend(loc="lower right")
plt.show()
```



Using Cross-validation:

Pros:

- Helps reduce variance.
- Expends models predictability.

In [101…
```
sc = st_scale
```

In [102…
```
## Using StratifiedShuffleSplit
## We can use KFold, StratifiedShuffleSplit, StratiriedKFold or ShuffleSplit, They are all close cousins. look at sklear
from sklearn.model_selection import StratifiedShuffleSplit, cross_val_score
cv = StratifiedShuffleSplit(n_splits = 10, test_size = .25, random_state = 0 ) # run model 10x with 60/30 split intentio
## Using standard scale for the whole dataset.

## saving the feature names for decision tree display
column_names = X.columns

X = sc.fit_transform(X)
accuracies = cross_val_score(LogisticRegression(solver='liblinear'), X,y, cv  = cv)
print ("Cross-Validation accuracy scores:{}".format(accuracies))
print ("Mean Cross-Validation accuracy score: {}".format(round(accuracies.mean(),5)))
```

```
Cross-Validation accuracy scores:[0.82 0.85 0.82 0.85 0.83 0.82 0.8  0.85 0.82 0.82]
Mean Cross-Validation accuracy score: 0.82793
```

Grid Search on Logistic Regression

In [103…
```
from sklearn.model_selection import GridSearchCV, StratifiedKFold
## C_vals is the alpla value of lasso and ridge regression(as alpha increases the model complexity decreases,)
## remember effective alpha scores are 0<alpha<infinity
C_vals = [0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1,2,3,4,5,6,7,8,9,10,12,13,14,15,16,16.5,17,17.5,18]
## Choosing penalties(Lasso(l1) or Ridge(l2))
penalties = ['l1','l2']
## Choose a cross validation strategy.
cv = StratifiedShuffleSplit(n_splits = 10, test_size = .25)

## setting param for param_grid in GridSearchCV.
param = {'penalty': penalties, 'C': C_vals}

logreg = LogisticRegression(solver='liblinear')
## Calling on GridSearchCV object.
grid = GridSearchCV(estimator=LogisticRegression(),
                           param_grid = param,
                           scoring = 'accuracy',
                            n_jobs =-1,
                           cv = cv
                          )
## Fitting the model
grid.fit(X, y)
```

Out[103]:
```
GridSearchCV(cv=StratifiedShuffleSplit(n_splits=10, random_state=None, test_size=0.25,
            train_size=None),
             error_score='raise-deprecating',
             estimator=LogisticRegression(C=1.0, class_weight=None, dual=False,
                                          fit_intercept=True,
                                          intercept_scaling=1, l1_ratio=None,
                                          max_iter=100, multi_class='warn',
                                          n_jobs=None, penalty='l2',
                                          random_state=None, solver='warn',
                                          tol=0.0001, verbose=0,
                                          warm_start=False),
             iid='warn', n_jobs=-1,
             param_grid={'C': [0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 2, 3,
                               4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 16, 16.5,
                               17, 17.5, 18],
                         'penalty': ['l1', 'l2']},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='accuracy', verbose=0)
```

In [104…
```
## Getting the best of everything.
print (grid.best_score_)
print (grid.best_params_)
print(grid.best_estimator_)
```

```
0.827027027027027
{'C': 0.2, 'penalty': 'l1'}
LogisticRegression(C=0.2, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l1',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

Using the best parameters from the grid-search.

In [105…
```
### Using the best parameters from the grid-search.
logreg_grid = grid.best_estimator_
logreg_grid.score(X,y)
```

Out[105]:
```
0.8412162162162162
```

K-Nearest Neighbor Classifier(KNN)

In [106…
```
## Importing the model.
from sklearn.neighbors import KNeighborsClassifier
## calling on the model oject.
knn = KNeighborsClassifier(metric='minkowski', p=2)
## knn classifier works by doing euclidian distance


## doing 10 fold staratified-shuffle-split cross validation
cv = StratifiedShuffleSplit(n_splits=10, test_size=.25, random_state=2)

accuracies = cross_val_score(knn, X,y, cv = cv, scoring='accuracy')
print ("Cross-Validation accuracy scores:{}".format(accuracies))
print ("Mean Cross-Validation accuracy score: {}".format(round(accuracies.mean(),3)))
```

```
Cross-Validation accuracy scores:[0.82 0.81 0.78 0.8  0.81 0.82 0.79 0.79 0.77 0.82]
Mean Cross-Validation accuracy score: 0.8
```

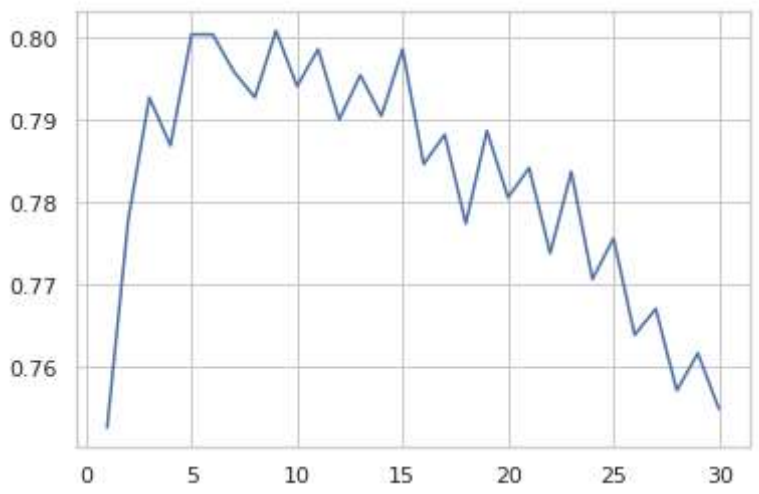
Manually find the best possible k value for KNN

In [107…
```
## Search for an optimal value of k for KNN.
k_range = range(1,31)
k_scores = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(knn, X,y, cv = cv, scoring = 'accuracy')
    k_scores.append(scores.mean())
print("Accuracy scores are: {}\n".format(k_scores))
print ("Mean accuracy score: {}".format(np.mean(k_scores)))
```

```
Accuracy scores are: [0.7527027027027027, 0.777927927927928, 0.7927927927927929, 0.7869369369369369, 0.800450450450450
5, 0.8004504504504505, 0.7959459459459459, 0.7927927927927928, 0.8009009009009009, 0.7941441441441441, 0.79864864864864
86, 0.7900900900900901, 0.7954954954954955, 0.7905405405405406, 0.7986486486486486, 0.7846846846846848, 0.7882882882882
882, 0.7774774774774775, 0.7887387387387388, 0.7806306306306305, 0.7842342342342343, 0.773873873873874, 0.7837837837837
839, 0.7707207207207207, 0.7756756756756756, 0.7639639639639639, 0.767117117117117, 0.7572072072072072, 0.7617117117117
117, 0.754954954954955]

Mean accuracy score: 0.7827177177177177
```

In [108…
```
from matplotlib import pyplot as plt
plt.plot(k_range, k_scores)
```

Out[108]:
```
[<matplotlib.lines.Line2D at 0x7828dc0e0e10>]
```



Grid search on KNN classifier

In [109…
```
from sklearn.model_selection import GridSearchCV
## trying out multiple values for k
k_range = range(1,31)
##
weights_options=['uniform','distance']
#
param = {'n_neighbors':k_range, 'weights':weights_options}
## Using startifiedShufflesplit.
cv = StratifiedShuffleSplit(n_splits=10, test_size=.30, random_state=15)
# estimator = knn, param_grid = param, n_jobs = -1 to instruct scikit learn to use all available processors.
grid = GridSearchCV(KNeighborsClassifier(), param,cv=cv,verbose = False, n_jobs=-1)
## Fitting the model.
grid.fit(X,y)
```

Out[109]:
```
GridSearchCV(cv=StratifiedShuffleSplit(n_splits=10, random_state=15, test_size=0.3,
            train_size=None),
             error_score='raise-deprecating',
             estimator=KNeighborsClassifier(algorithm='auto', leaf_size=30,
                                            metric='minkowski',
                                            metric_params=None, n_jobs=None,
                                            n_neighbors=5, p=2,
                                            weights='uniform'),
             iid='warn', n_jobs=-1,
             param_grid={'n_neighbors': range(1, 31),
                         'weights': ['uniform', 'distance']},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=False)
```

In [110…
```
print(grid.best_score_)
print(grid.best_params_)
print(grid.best_estimator_)
```

```
0.8044943820224719
{'n_neighbors': 5, 'weights': 'uniform'}
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

Using best estimator from grid search using KNN.

In [111…
```
### Using the best parameters from the grid-search.
knn_grid= grid.best_estimator_
knn_grid.score(X,y)
```

Out[111]:
```
0.865990990990991
```

Using RandomizedSearchCV

Randomized search is a close cousin of grid search. It doesn't always provide the best result but its fast.

In [112…
```
from sklearn.model_selection import RandomizedSearchCV
## trying out multiple values for k
k_range = range(1,31)
##
weights_options=['uniform','distance']
#
param = {'n_neighbors':k_range, 'weights':weights_options}
## Using startifiedShufflesplit.
cv = StratifiedShuffleSplit(n_splits=10, test_size=.30)
# estimator = knn, param_grid = param, n_jobs = -1 to instruct scikit learn to use all available processors.
## for RandomizedSearchCV,
grid = RandomizedSearchCV(KNeighborsClassifier(), param,cv=cv,verbose = False, n_jobs=-1, n_iter=40)
## Fitting the model.
grid.fit(X,y)
```

Out[112]:
```
RandomizedSearchCV(cv=StratifiedShuffleSplit(n_splits=10, random_state=None, test_size=0.3,
            train_size=None),
                   error_score='raise-deprecating',
                   estimator=KNeighborsClassifier(algorithm='auto',
                                                  leaf_size=30,
                                                  metric='minkowski',
                                                  metric_params=None,
                                                  n_jobs=None, n_neighbors=5,
                                                  p=2, weights='uniform'),
                   iid='warn', n_iter=40, n_jobs=-1,
                   param_distributions={'n_neighbors': range(1, 31),
                                        'weights': ['uniform', 'distance']},
                   pre_dispatch='2*n_jobs', random_state=None, refit=True,
                   return_train_score=False, scoring=None, verbose=False)
```

In [113…
```
print (grid.best_score_)
print (grid.best_params_)
print(grid.best_estimator_)
```

```
0.804119850187266
{'weights': 'uniform', 'n_neighbors': 5}
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

In [114…
```
### Using the best parameters from the grid-search.
knn_ran_grid = grid.best_estimator_
knn_ran_grid.score(X,y)
```

Out[114]:
```
0.865990990990991
```

Gaussian Naive Bayes

In [115…
```
# Gaussian Naive Bayes
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

gaussian = GaussianNB()
gaussian.fit(X, y)
y_pred = gaussian.predict(X_test)
gaussian_accy = round(accuracy_score(y_pred, y_test), 3)
print(gaussian_accy)
```

```
0.789
```

Support Vector Machines(SVM)

In [116…
```
from sklearn.svm import SVC
Cs = [0.001, 0.01, 0.1, 1,1.5,2,2.5,3,4,5, 10] ## penalty parameter C for the error term.
gammas = [0.0001,0.001, 0.01, 0.1, 1]
param_grid = {'C': Cs, 'gamma' : gammas}
cv = StratifiedShuffleSplit(n_splits=10, test_size=.30, random_state=15)
grid_search = GridSearchCV(SVC(kernel = 'rbf', probability=True), param_grid, cv=cv) ## 'rbf' stands for gaussian kerne
grid_search.fit(X,y)
```

Out[116]:
```
GridSearchCV(cv=StratifiedShuffleSplit(n_splits=10, random_state=15, test_size=0.3,
            train_size=None),
             error_score='raise-deprecating',
             estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='auto_deprecated', kernel='rbf', max_iter=-1,
                           probability=True, random_state=None, shrinking=True,
                           tol=0.001, verbose=False),
             iid='warn', n_jobs=None,
             param_grid={'C': [0.001, 0.01, 0.1, 1, 1.5, 2, 2.5, 3, 4, 5, 10],
                         'gamma': [0.0001, 0.001, 0.01, 0.1, 1]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

In [117…
```
print(grid_search.best_score_)
print(grid_search.best_params_)
print(grid_search.best_estimator_)
```

```
0.8453183520599251
{'C': 2, 'gamma': 0.001}
SVC(C=2, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=0.001, kernel='rbf',
    max_iter=-1, probability=True, random_state=None, shrinking=True, tol=0.001,
    verbose=False)
```

In [118…
```
# using the best found hyper paremeters to get the score.
svm_grid = grid_search.best_estimator_
svm_grid.score(X,y)
```

Out[118]:
```
0.8265765765765766
```

Decision Tree Classifier

Decision tree works by breaking down the dataset into small subsets. This breaking down process is done by asking questions about the features of the datasets. The idea is to unmix the labels by asking fewer questions necessary. As we ask questions, we are breaking down the dataset into more subsets. Once we have a subgroup with only the unique type of labels, we end the tree in that node.

In [119…
```
from sklearn.tree import DecisionTreeClassifier
max_depth = range(1,30)
max_feature = [21,22,23,24,25,26,28,29,30,'auto']
criterion=["entropy", "gini"]

param = {'max_depth':max_depth,
         'max_features':max_feature,
         'criterion': criterion}
grid = GridSearchCV(DecisionTreeClassifier(),
                                param_grid = param,
                                 verbose=False,
                                 cv=StratifiedKFold(n_splits=20, random_state=15, shuffle=True),
                                n_jobs = -1)
grid.fit(X, y)
```

```
/opt/conda/lib/python3.6/site-packages/sklearn/model_selection/_search.py:814: DeprecationWarning: The default of the `
iid` parameter will change from True to False in version 0.22 and will be removed in 0.24. This will change numeric res
ults when test-set sizes are unequal.
  DeprecationWarning)
```

Out[119]:
```
GridSearchCV(cv=StratifiedKFold(n_splits=20, random_state=15, shuffle=True),
             error_score='raise-deprecating',
             estimator=DecisionTreeClassifier(class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features=None,
                                              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,
                                              presort=False, random_state=None,
                                              splitter='best'),
             iid='warn', n_jobs=-1,
             param_grid={'criterion': ['entropy', 'gini'],
                         'max_depth': range(1, 30),
                         'max_features': [21, 22, 23, 24, 25, 26, 28, 29, 30,
                                          'auto']},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=False)
```

In [120…
```
print( grid.best_params_)
print (grid.best_score_)
print (grid.best_estimator_)
```

```
{'criterion': 'entropy', 'max_depth': 4, 'max_features': 29}
0.8322072072072072
DecisionTreeClassifier(class_weight=None, criterion='entropy', max_depth=4,
                       max_features=29, 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, presort=False,
                       random_state=None, splitter='best')
```

In [121…
```
dectree_grid = grid.best_estimator_
## using the best found hyper paremeters to get the score.
dectree_grid.score(X,y)
```

Out[121]:
```
0.8434684684684685
```

Let's look at the feature importance from decision tree grid.

In [122…
```
## feature importance
feature_importances = pd.DataFrame(dectree_grid.feature_importances_,
                                   index = column_names,
                                    columns=['importance'])
feature_importances.sort_values(by='importance', ascending=False).head(10)
```

Out[122]:

	importance
Sex	0.494076
Pclass_3	0.160822
family_group_large	0.120069
calculated_fare	0.115741
title_Master	0.072836
Embarked_S	0.011320
Cabin_D	0.010632
fare_group_very_high	0.009432
Age	0.005071
age_group_child	0.000000

These are the top 10 features determined by **Decision Tree** helped classifing the fates of many passenger on Titanic on that night.

Random Forest Classifier

In [123…
```
from sklearn.model_selection import GridSearchCV, StratifiedKFold, StratifiedShuffleSplit
from sklearn.ensemble import RandomForestClassifier
n_estimators = [140,145,150,155,160];
max_depth = range(1,10);
criterions = ['gini', 'entropy'];
cv = StratifiedShuffleSplit(n_splits=10, test_size=.30, random_state=15)


parameters = {'n_estimators':n_estimators,
              'max_depth':max_depth,
              'criterion': criterions

        }
grid = GridSearchCV(estimator=RandomForestClassifier(max_features='auto'),
                                    param_grid=parameters,
                                    cv=cv,
                                    n_jobs = -1)
grid.fit(X,y)
```

Out[123]:
```
GridSearchCV(cv=StratifiedShuffleSplit(n_splits=10, random_state=15, test_size=0.3,
            train_size=None),
             error_score='raise-deprecating',
             estimator=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='warn', n_jobs=None,
                                              oob_score=False,
                                              random_state=None, verbose=0,
                                              warm_start=False),
             iid='warn', n_jobs=-1,
             param_grid={'criterion': ['gini', 'entropy'],
                         'max_depth': range(1, 10),
                         'n_estimators': [140, 145, 150, 155, 160]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

In [124…
```
print (grid.best_score_)
print (grid.best_params_)
print (grid.best_estimator_)
```

```
0.8445692883895131
{'criterion': 'entropy', 'max_depth': 5, 'n_estimators': 150}
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
                       max_depth=5, 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=150,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

In [125…
```
rf_grid = grid.best_estimator_
rf_grid.score(X,y)
```

Out[125]:
```
0.8457207207207207
```

In [126…
```
from sklearn.metrics import classification_report
# Print classification report for y_test
print(classification_report(y_test, y_pred, labels=rf_grid.classes_))
```

```
              precision    recall  f1-score   support

           0       0.82      0.84      0.83       177
           1       0.74      0.72      0.73       117

    accuracy                           0.79       294
   macro avg       0.78      0.78      0.78       294
weighted avg       0.79      0.79      0.79       294
```

Feature Importance

In [127…
```
## feature importance
feature_importances = pd.DataFrame(rf_grid.feature_importances_,
                                   index = column_names,
                                    columns=['importance'])
feature_importances.sort_values(by='importance', ascending=False).head(10)
```

Out[127]:

	importance
title_Mr	0.203086
Sex	0.158442
calculated_fare	0.085212
title_Miss	0.072980
title_Mrs	0.048577
Pclass_3	0.048434
Age	0.043731
Pclass_1	0.031760
family_group_small	0.031177
family_group_large	0.024231

Bagging Classifier

In [128…

```
from sklearn.ensemble import BaggingClassifier
n_estimators = [10,30,50,70,80,150,160, 170,175,180,185];
cv = StratifiedShuffleSplit(n_splits=10, test_size=.30, random_state=15)

parameters = {'n_estimators':n_estimators,

        }
grid = GridSearchCV(BaggingClassifier(base_estimator= None, ## If None, then the base estimator is a decision tree.
                                      bootstrap_features=False),
                                   param_grid=parameters,
                                   cv=cv,
                                   n_jobs = -1)
grid.fit(X,y)
```

Out[128]:

```
GridSearchCV(cv=StratifiedShuffleSplit(n_splits=10, random_state=15, test_size=0.3,
            train_size=None),
             error_score='raise-deprecating',
             estimator=BaggingClassifier(base_estimator=None, bootstrap=True,
                                         bootstrap_features=False,
                                         max_features=1.0, max_samples=1.0,
                                         n_estimators=10, n_jobs=None,
                                         oob_score=False, random_state=None,
                                         verbose=0, warm_start=False),
             iid='warn', n_jobs=-1,
             param_grid={'n_estimators': [10, 30, 50, 70, 80, 150, 160, 170,
                                          175, 180, 185]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

In [129…

```
print (grid.best_score_)
print (grid.best_params_)
print (grid.best_estimator_)
```

```
0.8198501872659176
{'n_estimators': 175}
BaggingClassifier(base_estimator=None, bootstrap=True, bootstrap_features=False,
                  max_features=1.0, max_samples=1.0, n_estimators=175,
                  n_jobs=None, oob_score=False, random_state=None, verbose=0,
                  warm_start=False)
```

In [130…

```
bagging_grid = grid.best_estimator_
bagging_grid.score(X,y)
```

Out[130]:

```
0.9887387387387387
```

AdaBoost Classifier

In [131…

```
from sklearn.ensemble import AdaBoostClassifier
n_estimators = [100,140,145,150,160, 170,175,180,185];
cv = StratifiedShuffleSplit(n_splits=10, test_size=.30, random_state=15)
learning_r = [0.1,1,0.01,0.5]

parameters = {'n_estimators':n_estimators,
              'learning_rate':learning_r

        }
grid = GridSearchCV(AdaBoostClassifier(base_estimator= None, ## If None, then the base estimator is a decision tree.
                                       ),
                                   param_grid=parameters,
                                   cv=cv,
                                   n_jobs = -1)
grid.fit(X,y)
```

Out[131]:
```
GridSearchCV(cv=StratifiedShuffleSplit(n_splits=10, random_state=15, test_size=0.3,
            train_size=None),
             error_score='raise-deprecating',
             estimator=AdaBoostClassifier(algorithm='SAMME.R',
                                          base_estimator=None,
                                          learning_rate=1.0, n_estimators=50,
                                          random_state=None),
             iid='warn', n_jobs=-1,
             param_grid={'learning_rate': [0.1, 1, 0.01, 0.5],
                         'n_estimators': [100, 140, 145, 150, 160, 170, 175,
                                          180, 185]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

In [132…
```
print (grid.best_score_)
print (grid.best_params_)
print (grid.best_estimator_)
```

```
0.8247191011235955
{'learning_rate': 0.1, 'n_estimators': 100}
AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None, learning_rate=0.1,
                   n_estimators=100, random_state=None)
```

In [133…
```
adaBoost_grid = grid.best_estimator_
adaBoost_grid.score(X,y)
```

Out[133]:
```
0.8355855855855856
```

Gradient Boosting Classifier

In [134…
```
# Gradient Boosting Classifier
from sklearn.ensemble import GradientBoostingClassifier

gradient_boost = GradientBoostingClassifier()
gradient_boost.fit(X, y)
y_pred = gradient_boost.predict(X_test)
gradient_accy = round(accuracy_score(y_pred, y_test), 3)
print(gradient_accy)
```

```
0.864
```

XGBClassifier

In [135…
```
# from xgboost import XGBClassifier
# XGBClassifier = XGBClassifier()
# XGBClassifier.fit(X, y)
# y_pred = XGBClassifier.predict(X_test)
# XGBClassifier_accy = round(accuracy_score(y_pred, y_test), 3)
# print(XGBClassifier_accy)
```

Extra Trees Classifier

In [136…
```
from sklearn.ensemble import ExtraTreesClassifier
ExtraTreesClassifier = ExtraTreesClassifier()
ExtraTreesClassifier.fit(X, y)
y_pred = ExtraTreesClassifier.predict(X_test)
extraTree_accy = round(accuracy_score(y_pred, y_test), 3)
print(extraTree_accy)
```

```
0.949
```

Gaussian Process Classifier

In [137…
```
from sklearn.gaussian_process import GaussianProcessClassifier
GaussianProcessClassifier = GaussianProcessClassifier()
GaussianProcessClassifier.fit(X, y)
y_pred = GaussianProcessClassifier.predict(X_test)
gau_pro_accy = round(accuracy_score(y_pred, y_test), 3)
print(gau_pro_accy)
```

```
0.925
```

Voting Classifier

In [138…
```
from sklearn.ensemble import VotingClassifier

voting_classifier = VotingClassifier(estimators=[
    ('lr_grid', logreg_grid),
    ('svc', svm_grid),
    ('random_forest', rf_grid),
    ('gradient_boosting', gradient_boost),
    ('decision_tree_grid',dectree_grid),
```

```
    ('knn_classifier', knn_grid),
#     ('XGB_Classifier', XGBClassifier),
    ('bagging_classifier', bagging_grid),
    ('adaBoost_classifier',adaBoost_grid),
    ('ExtraTrees_Classifier', ExtraTreesClassifier),
    ('gaussian_classifier',gaussian),
    ('gaussian_process_classifier', GaussianProcessClassifier)
],voting='hard')

#voting_classifier = voting_classifier.fit(train_x,train_y)
voting_classifier = voting_classifier.fit(X,y)
```

In [139…

```
y_pred = voting_classifier.predict(X_test)
voting_accy = round(accuracy_score(y_pred, y_test), 3)
print(voting_accy)
```

```
0.847
```

In [140…

```
#models = pd.DataFrame({
#    'Model': ['Support Vector Machines', 'KNN', 'Logistic Regression',
#              'Random Forest', 'Naive Bayes',
#              'Decision Tree', 'Gradient Boosting Classifier', 'Voting Classifier', 'XGB Classifier','ExtraTrees Class
#    'Score': [svc_accy, knn_accy, logreg_accy,
#              random_accy, gaussian_accy, dectree_accy,
#               gradient_accy, voting_accy, XGBClassifier_accy, extraTree_accy, bagging_accy]})
#models.sort_values(by='Score', ascending=False)
```