CIS053 Final - ML titanic

May 21, 2024

1 CIS053 Final Project - ML Titanic

Anne-Elise Chung
Hermanda Mak

1.1 Executive Summary

The Titanic shipwreck is a widely known as a major historic event. In this report, we will analyze some training data to try and predict which features have the highest impact on survival. In order to guide our analyses, we survey some existing other work on kaggle. From there, we start looking at the shape of the data in our training dataset. We found that a significant portion of the training data has missing values, and greatly reduces the viable amount of data that we can work with. So we take some steps to handle the missing data. After which, we also observed that some of the data is provided as strings, such as 'S' for Southampton for our 'Embarked' feature. To better prepare the data for use in the classification models, we converted these string values into numbers.

We look closely at the distribution of our quantitative features and use this to make decisions on how scaling the data may potentially affect the classification step. We also analyze the categorical data with various count plots to try and find correlations between the survival rate and different features such as gender, cabin location, and age. To prepare the data for use in the classification models, we create a train/test split of our training dataset. We then employed various techniques, such as but not limited to logistic regression, decision trees, bagging, random forest, boost, and neural networks. Each of these techniques have various results, all of which we compare in using cross validation. Our best performing model was the bagging classifier, but we make note of reasons why it performed better than some of the other models. We also note how the way we handled the missing data may have amplified the noise or obfuscated important characteristics in the training data.

Looking beyond the scope of this report, it is important to note that there is more data needed to get a better picture of how well we can predict survival aboard the Titanic.

1.2 Background

The purpose of this project is to build a predictive model that can classify the sort of passenger on Titatic that will most likely survive after the shipwreck. Moreover, we aim to discover the most influential features that affect the possibility of survival among the Titanic passengers.

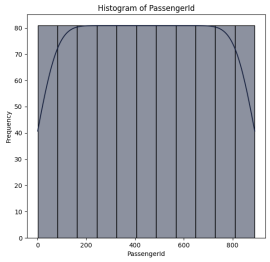
The sinking of the Titanic happened on April 15, 1912, where the resulting death of 1502 out of 2224 passengers led to global attention regarding safety of sea traveling and way to avoid another shipwreck of such sort. The major reason behind the disastrous outcome is the lack of lifeboats.

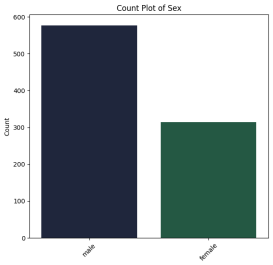
Based on the collected data of survival history on the Titanic, we can understand and build a predictive model that would identify which type of passenger could survive on such shipwreck.

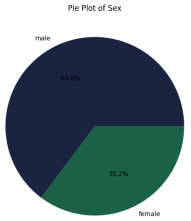
1.3 Survey of Existing Work

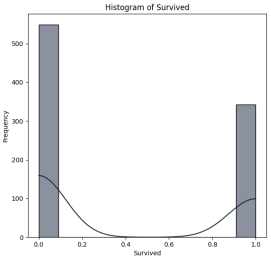
https://www.kaggle.com/code/abdmental01/exploring-data-visual-insights-unveiled#About-Function

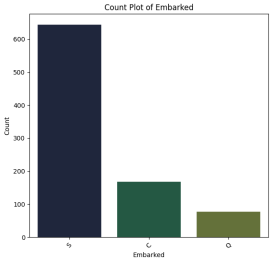
- Author: MUHAMMAD ABDULLAH
- Techniques Used: this submission was purely showing some data visualization techniques, but it serves as a helpful comparison for the upcoming EDA work in this report.
- Results:

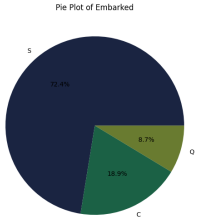

Histogram of PassengerId
Frequency
PassengerId

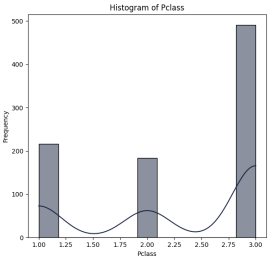

Count Plot of Sex
Count
male
female
Sex

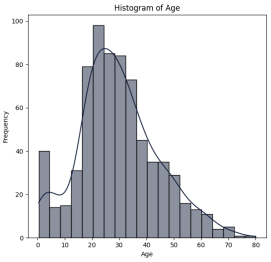

Pie Plot of Sex
male
64.8%
35.2%
female

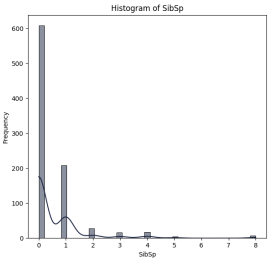

Histogram of Survived
Frequency
Survived

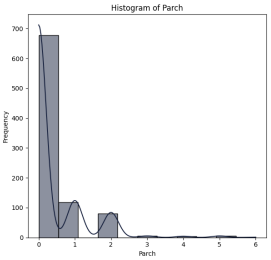

Count Plot of Embarked
Count
S
C
Q
Embarked

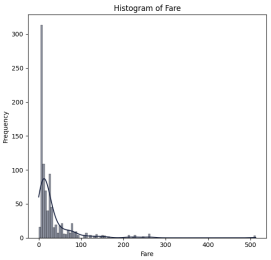

Pie Plot of Embarked
S
72.4%
8.7%
Q
18.9%
C


Histogram of Pclass
Frequency
Pclass


Histogram of Age
Frequency
Age


Histogram of SibSp
Frequency
SibSp


Histogram of Parch
Frequency
Parch


Histogram of Fare
Frequency
Fare

-

https://www.kaggle.com/code/gusthema/titanic-competition-w-tensorflow-decision-forests - Author: GUSTHEMA - Techniques Used: TensorFlow Decision Trees - Results: In the submission, they use a gradient boosted tree model from Keras. The trained model assigned the sex, age, fare, and name as the most important variables. - Notes: This Kaggle submission received 675 upvotes, compared to many of the other submissions that only received 2 or 3 upvotes. The popularity of this post might indicate that the results and techniques are probably sound and would be good for comparison.

[2]:
```
import pandas as pd
decisiontree_submission = pd.
 ↪read_csv('tensorflow_decisiontrees_kaggle_submission.csv')
decisiontree_submission.info
```

[2]:
```
<bound method DataFrame.info of      PassengerId  Survived
0            892         0
1            893         0
2            894         0
3            895         0
4            896         1
..           …       …
413         1305         0
414         1306         1
415         1307         0
416         1308         0
417         1309         0

[418 rows x 2 columns]>
```

https://www.kaggle.com/code/eneskosar19/titanic-sample-submission-random-forest

- Author: ENES KOŞAR
- Techniques Used: Random Forest Classifier
- Results: The kaggle submission does not include any accuracy predictions, and there is no way to get their output for comparison.
- From this submission, we were able to see some example of how to handle missing data and also an example of grid search cross validation for different random forest models of depth 2.

https://www.kaggle.com/code/whitedevil4648/neural-networks-titanic

- Author: WHITEDEVIL4648
- Techniques Used: neural networks
- Results: this submission did not include any meaningful prediction scores, but it is helpful to see how they use label encoding to transform categorical data into numerical data.

https://www.kaggle.com/code/gunesevitan/titanic-advanced-feature-engineering-tutorial

- Author: GUNES EVITAN
- Techniques Used: random forest

- Results: this submission used extensive feature engineering and does some in-depth research to handle missing features. They have a public score 0.83, which is the top 2 percent on the public leader board. Their work is really helpful in understanding important considerations in the exploratory data analysis step.

1.4 Our Approach

As the goal is to analyze the kind of people that could survive in the shipwreck, classification model will be used to predict two outcomes: survived or not survived. Moreover, the training dataset provides with labels for each data entity. We will focus on using supervised classification models, including logistic regression, tree models, support vector machine and use cross validation to select the best performing model for use.

Before fitting any models, EDA analysis will be performed to see if there are collinearity among the features and if any pre-processing is required, including standardization or normalization, dropping of rows and encoding of categorical features, will provide with better resulting models.

1.5 Code and Results

[3]:
```
from pandas import read_csv
import matplotlib.pyplot as plt
from pandas import set_option
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
import seaborn as sns
import numpy as np
```

[4]:
```
test_data = read_csv('./test.csv')
train_data_raw = read_csv('./train.csv')
print(train_data_raw.shape)

train_data = train_data_raw.copy()
train_data_dropped = train_data_raw.copy()

train_data_dropped.dropna(inplace=True)
train_data_dropped.reset_index(drop=True, inplace=True)
print(train_data_dropped.shape)
```

```
(891, 12)
(183, 12)
```

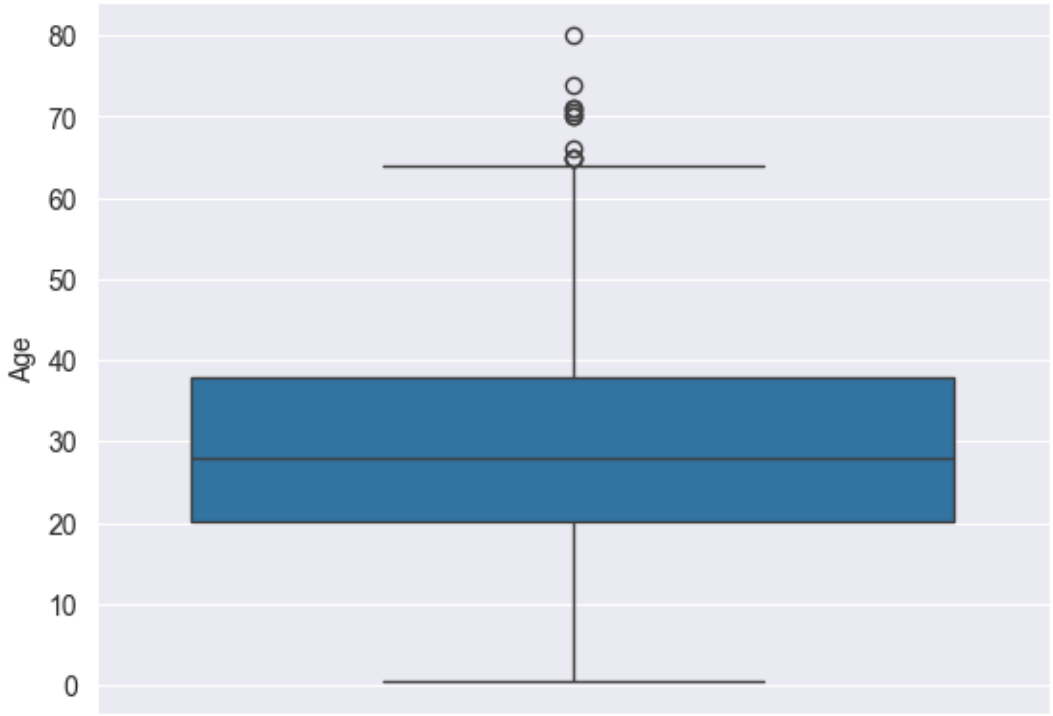
Unfortunately, when we drop all the rows with null values, we go from 891 rows to 183 rows, which significantly reduces the amount of data we have to work with by more than 75%.

In order to make use of more of the input data, let's try to handle the missing values. We will

be using this example: https://www.kaggle.com/code/rushikeshlavate/handling-missing-data-in-titanic-train-dataset

[5]:
```
sns.boxplot(train_data_raw['Age'])
```

[5]:
```
<Axes: ylabel='Age'>
```



We can see that there are a few outliers in the raw input, this is something we can try to remedy along with missing information. The technique we will try using is mean, mode, and median imputation Here's an interesting quote from a Kaggle submission: https://www.kaggle.com/code/pagenotfound/mean-and-median-imputation

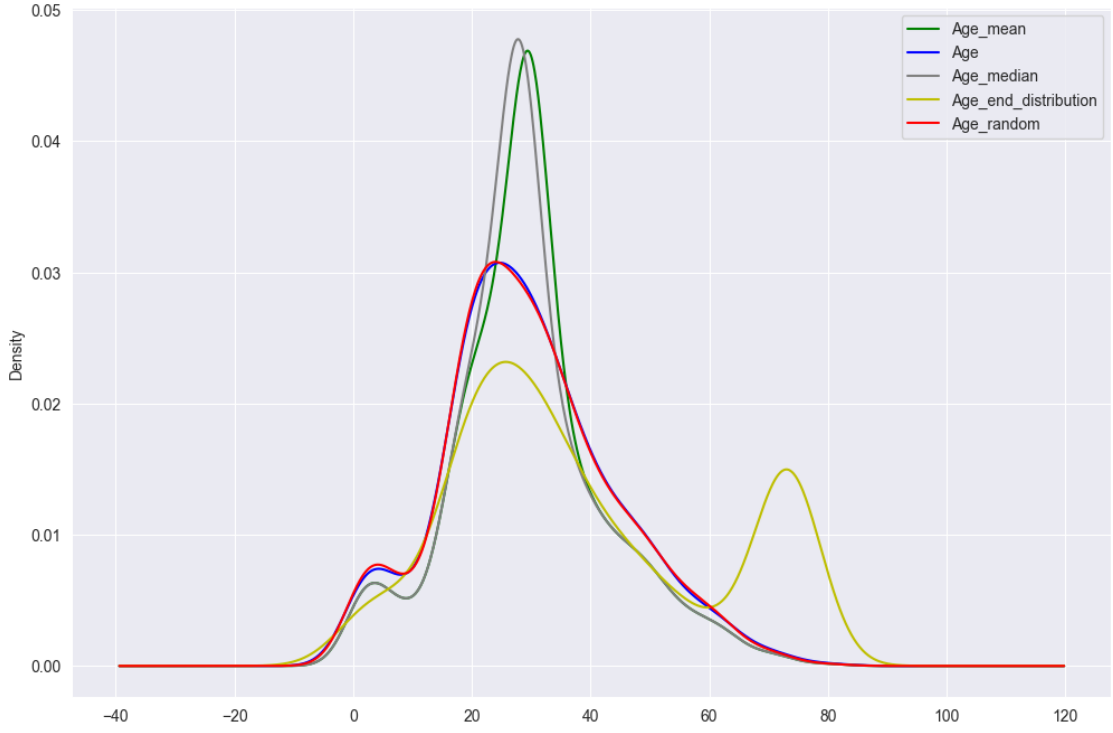
> Imputation should be done over the training set, and then propagated to the test set. This means that the mean/median to be used to fill missing values both in train and test set, should be extracted from the train set only. And this is to avoid overfitting.

This means that we will need to modify our training data set and also fill missing values in the test set. As to how it avoids overfitting, if we fill in the missing values on the training data, but don't fill it out on the test data, when we measure the performance of the model, it will be fitted to the data that we filled but, it won't handle the missing information in the test data adequately.

However, instead of blindly following this with the mean or median imputation, let's test some other imputation techniques to see if we can get a version that matches the data that isn't missing.

[6]:
```
# we fill the missing age data

def mean_median(df,variable):
    df[variable+'_mean']  = df[variable].fillna(df[variable].mean())
    df[variable+'_median']  = df[variable].fillna(df[variable].median())

mean_median(train_data_raw,'Age')

train_data_raw['Age_nan'] = np.where(train_data_raw['Age'].isnull,0,1)
def Age_fill(df,variable):
    df[variable+'_random'] = df[variable]
    random_values = df[variable].dropna().sample(df[variable].isnull().
 ↪sum(),random_state=3)
    random_values.index = df[df[variable].isnull()].index
    df.loc[df[variable].isnull(),variable+'_random'] = random_values

Age_fill(train_data_raw, 'Age')

extreme = (train_data_raw['Age'].mean()+(3*train_data_raw['Age'].std()))

def End_distribution(df,variable,extreme):
    df[variable+'_end_distribution'] = df[variable].fillna(extreme)

End_distribution(train_data_raw,'Age',extreme)

plt.figure(figsize=(12,8))
train_data_raw.Age_mean.plot(kind='kde',color='g')
train_data_raw.Age.plot(kind='kde',color='b')
train_data_raw.Age_median.plot(kind='kde',color='grey')
train_data_raw.Age_end_distribution.plot(kind='kde',color='y')
train_data_raw.Age_random.plot(kind='kde',color='r')
plt.legend()
```

[6]:
```
<matplotlib.legend.Legend at 0x29faf5c37d0>
```



From the graph above, we can see that the Age curve is very similar to the random imputation age curve. Thus, filling the missing values by using random imputation may be the best strategy in this scenario.

[7]:
```
# drop the Age column with missing values and replace with Age_random
train_data = train_data.drop(['Age'], axis = 1)
train_data['Age'] = train_data_raw['Age_random']
```

[8]:
```
# trying to fill the missing values in embarked
train_data_raw['Embarked'].isnull().sum()
```

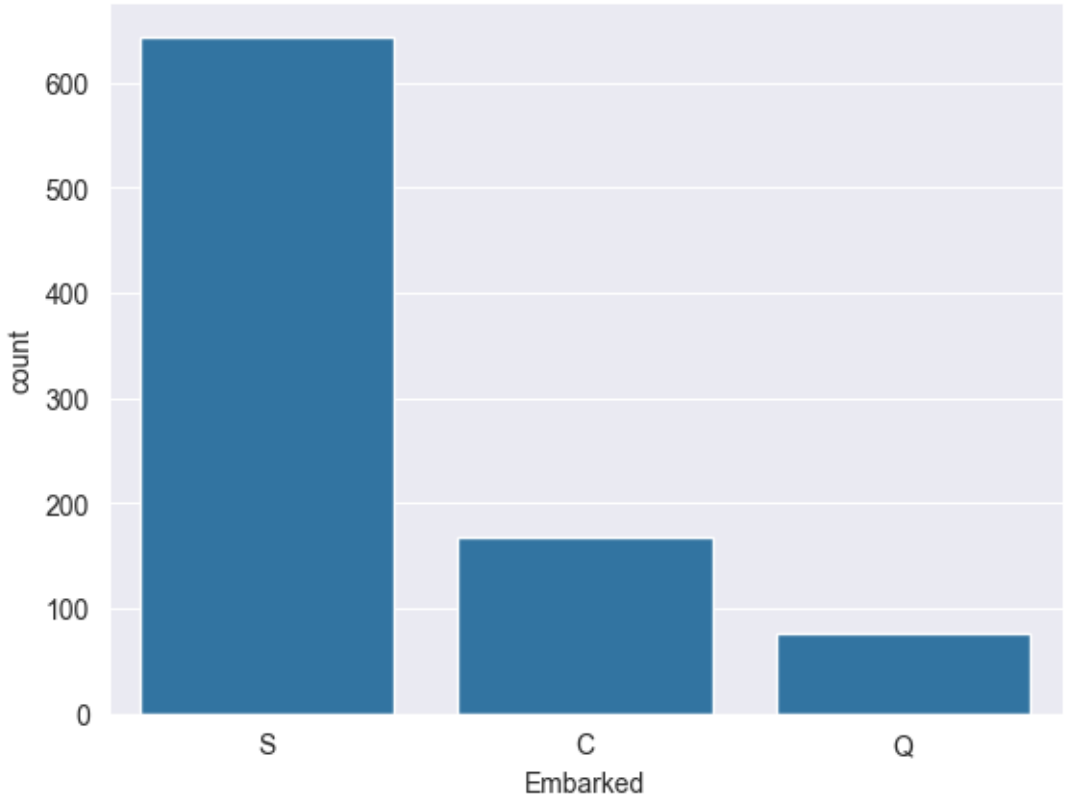
[8]:
```
2
```

[9]:
```
train_data_raw['Embarked'].unique()
```

[9]:
```
array(['S', 'C', 'Q', nan], dtype=object)
```

[10]:
```
sns.countplot(train_data_raw, x='Embarked')
```

[10]:
```
<Axes: xlabel='Embarked', ylabel='count'>
```



Since the amount of missing data is so small, let's fill it with mode imputation.

[11]:
```
# this will fill the missing values with the most common category
train_data['Embarked'] = train_data_raw['Embarked'].
 ↪fillna(train_data_raw['Embarked'].mode()[0])
train_data['Embarked'].isnull().sum()
```

[11]: 0

[12]:
```
train_data['Embarked'].unique()
```

[12]: array(['S', 'C', 'Q'], dtype=object)

[13]:
```
# moving on to the cabin feature
train_data['Cabin'].isnull().sum()
```

[13]: 687

Wow, there is a lot of missing information in the cabin feature. We can try to fix this with frequent category imputation, which is similar to the mode imputation for the Embarked feature. However, that might severely affect the survival rate if the most frequent category survival rate is very different compared to the unknown cabin survival rate.

[14]:
```
unknown_cabins = train_data_raw.loc[train_data_raw['Cabin'].
  ↪isnull()]['Survived']
rate_survived = sum(unknown_cabins)/len(unknown_cabins)
print(rate_survived)
```

```
0.29985443959243085
```

Since the rate of survival seems very low for all the missing cabins, it seems more valid to replace it with its own separate category. We will replace the null values with the 'Unknown' label.

[15]:
```
train_data['Cabin'] = train_data['Cabin'].apply(lambda s: s[0] if pd.notnull(s)␣
  ↪else 'Unknown')
train_data['Cabin'].isnull().sum()
```

[15]:
```
0
```

[16]:
```
print(train_data.isnull().sum())
train_data.shape
```

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

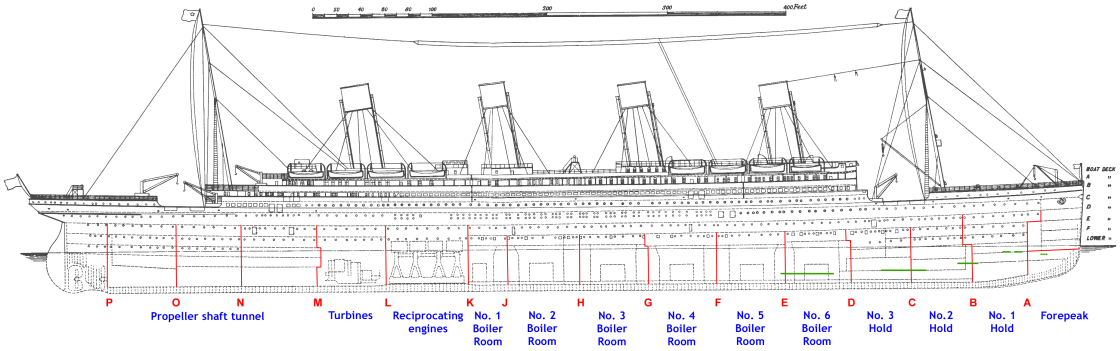
[16]:
```
(891, 12)
```

Now that we have handled all the missing values we can continue to separate our data into X,Y, and testing vs training sets. We will also drop some of the features that have little or no effect on the survival rate:

- PassengerId: this is a unique id per row. We do not expect it to really affect the survival rate. This can be dropped.
- Survived: this is what we are trying to predict, where 1 indicates survival and 0 indicates not survived. We separate this from our X into Y
- Pclass: class might have an impact on who survived. Maybe the upper class had better access to lifeboats? Or maybe the lower class was better at navigating and acting in such a desperate situation. This is a categorical ordinal feature with 3 values: 1 is the upper class, and 3 is the lower class.
- Name: we think it is not likely that the name would attribute to the survival rate. Some of the titles, such as Dr. might help indicate different roles or professions that might have been

prioritized when boarding lifeboats. If we want to make use of this possibility, we may need to engineer a new feature.

- Sex & Age: both of these may have had a large impact on affecting survival. We are assuming that the lifeboat boarding protocol would prioritize women and children before adult men. Therefore, it is more likely that those that are younger and women are more likely to survive. We will explore this as we look at the data below
- SibSp & Parch: the total number of siblings and spouse, and the total number of parents and children respectively. These feature might have an affect on survival. Maybe if you have family that is boarding lifeboat you would be more likely to be there with them and vice versa? If you can't find your family in the chaos, you might spend more time looking for them on the ship, and decrease the chances of your own survival.
- Ticket: the ticket number. Like the passenger id, we don't expect this to have a significant impact on survival
- Fare: this is the amount paid to board the ship. We are expecting this to be highly correlated with Pclass. If there is a high collinearity between the two, we may consider dropping one of these features to reduce the noise or overfitting to redundant features.
- Cabin: the cabin number. This is a string that starts with a character (A,B,C, D, E, F, G, T) followed by a number. This feature has a very high amount of missing data. The leading character indicates which section of the ship the cabin was located.



(Image from https://en.m.wikipedia.org/wiki/File:Titanic_side_plan_annotated_English.png)

- Embarked: this is the port from where passengers embarked. The 3 ports, C for Cherbourg, Q for Queenstown, and S for Southampton. We are not sure what effect this could have on the survival rate.

[17]:
```
# x = data without output and unique identifiers
x = train_data.drop(['PassengerId', 'Name', 'Ticket', 'Survived'], axis=1)
y = train_data['Survived']
x_names = x.columns

# x_clean = input that are not categorical
x_clean = x.drop(['Cabin', 'Sex', 'Embarked', 'Pclass'], axis=1)
x_clean_names = x_clean.columns

x_num = x

# convert categorical value to numerical value using LabelEncoder
```

```
le_cabin = LabelEncoder()
le_sex = LabelEncoder()
le_embarked = LabelEncoder()
x_num['Cabin'] = le_cabin.fit_transform(x_num['Cabin'])
x_num['Sex'] = le_sex.fit_transform(x_num['Sex'].fillna('Unknown'))
x_num['Embarked'] = le_embarked.fit_transform(x_num['Embarked'].
 ↪fillna('Unknown'))

# x_category = dataframe for categorical features
x_category = pd.concat([x_num['Cabin'], x_num['Sex'], x_num['Embarked'],␣
 ↪x_num['Pclass']], axis=1)
x_category
```

[17]:

	Cabin	Sex	Embarked	Pclass
0	8	1	2	3
1	2	0	0	1
2	8	0	2	3
3	2	0	2	1
4	8	1	2	3
..	…	…	…	…
886	8	1	2	2
887	1	0	2	1
888	8	0	2	3
889	2	1	0	1
890	8	1	1	3

[891 rows x 4 columns]

As most of the models being used in this project require numerical inputs for evaluation. LabelEncoder is used to encode the categorical features. The following map displays the corresponding category to the integral value after conversion.

[18]:
```
# Mapping of category value with numerical value

sex_mapping = le_sex.classes_
embarked_mapping = le_embarked.classes_
cabin_mapping = le_cabin.classes_

mappings_df = pd.DataFrame({
    'Sex': pd.Series(sex_mapping),
    'Embarked': pd.Series(embarked_mapping),
    'Cabin': pd.Series(cabin_mapping)
})

mappings_df = mappings_df.fillna("")

print("Mapping of Categorical Value with Numerical Value")
```

```
mappings_df
```

Mapping of Categorical Value with Numerical Value

[18]:

	Sex	Embarked	Cabin
0	female	C	A
1	male	Q	B
2		S	C
3			D
4			E
5			F
6			G
7			T
8			Unknown

[19]:
```
X_arr = x_num.values
Y_arr = y.values

# create train_data with numerical value of categories
train_data_num = pd.concat([y, x_num], axis=1)

# train_data_eda = dataset without categorical data
train_data_eda = train_data.drop(['PassengerId', 'Name', 'Sex', 'Ticket',␣
 ↪'Cabin', 'Embarked', 'Pclass'], axis =1)
```

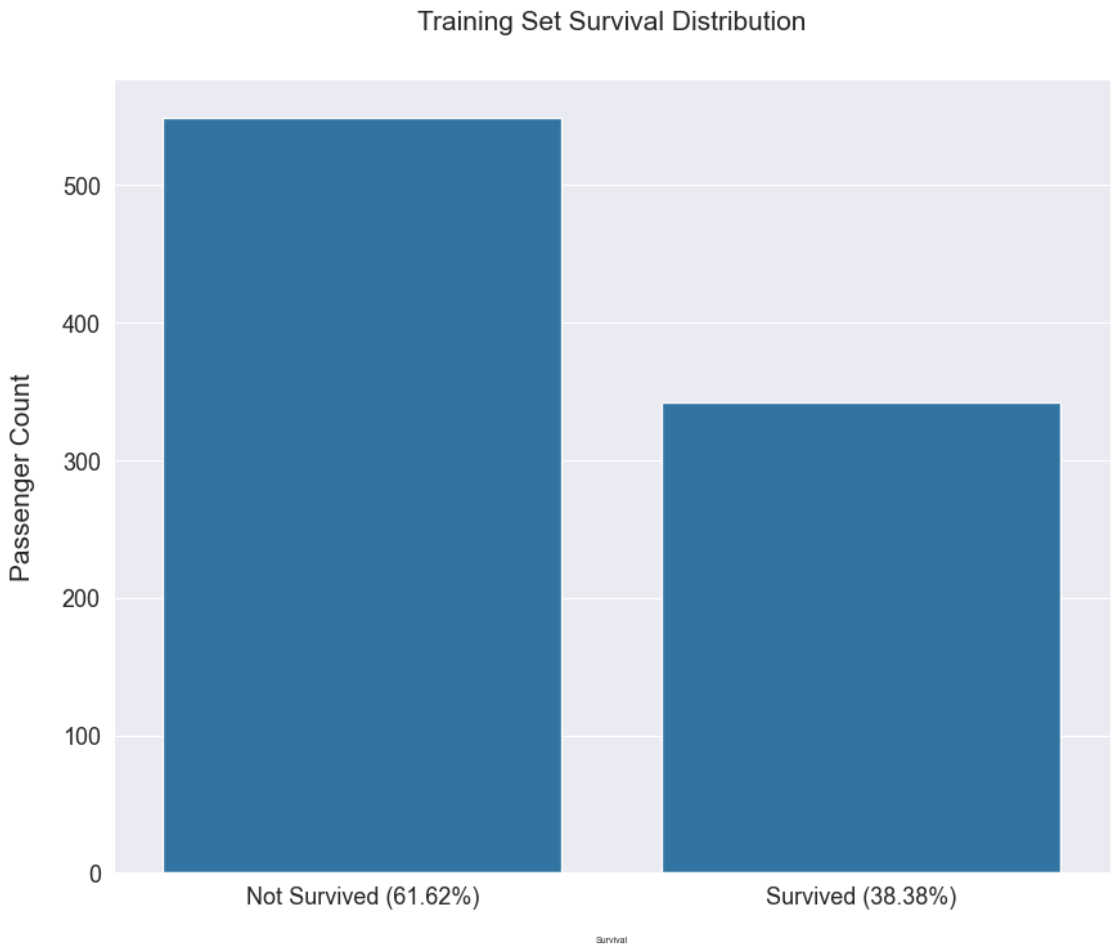
1.6 EDA with No Preprocessing

Categorical data analysis Target Distribution

[20]:
```
# using code from kaggle as an example: https://www.kaggle.com/code/gunesevitan/
 ↪titanic-advanced-feature-engineering-tutorial

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

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

plt.figure(figsize=(10, 8))
sns.countplot(train_data, x='Survived')

plt.xlabel('Survival', size=5, 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()
```

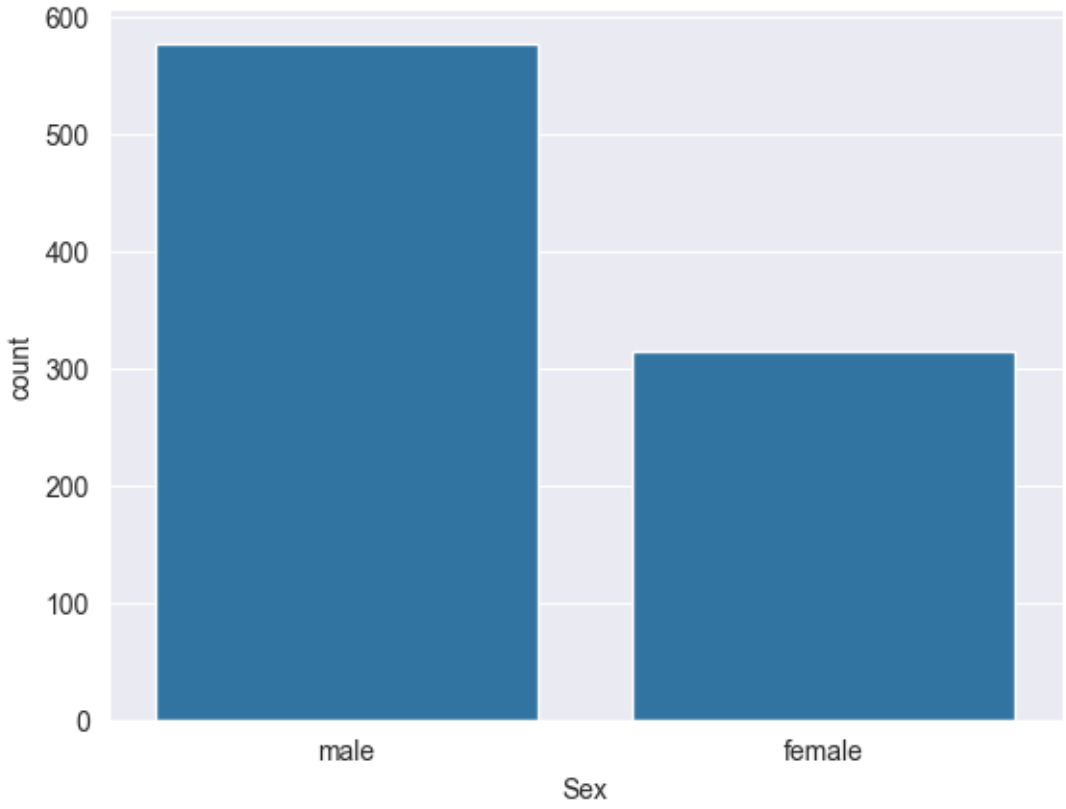
```
342 of 891 passengers survived and it is the 38.38% of the training set.
549 of 891 passengers didnt survive and it is the 61.62% of the training set.
```



We can see that more than 60 percent of the passengers in our training set did not survive. This is not an overwhelming majority by any means, but we can see that the general trend is not to survive.

[83]:
```
sns.countplot(train_data, x='Sex')
plt.figure(figsize=(10, 8))
```

[83]: <Figure size 1000x800 with 0 Axes>

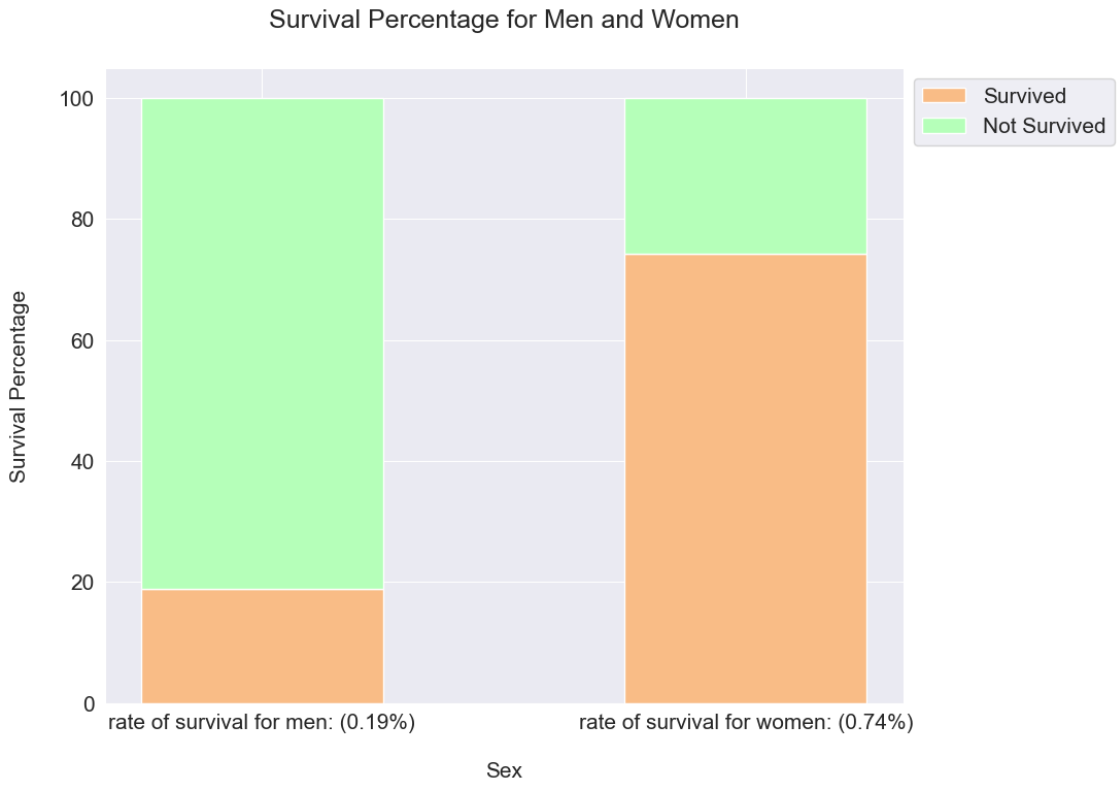


<Figure size 1000x800 with 0 Axes>

By plotting the distribution of both genders, we can see that most of the passengers in the training set are male. We can try to see the survival rates of either gender to see if there is a strong correlation between gender and survival.

[84]:
```
women = train_data.loc[train_data.Sex == 'female']['Survived']
rate_women = sum(women)/len(women)

men = train_data.loc[train_data.Sex == 'male']['Survived']
rate_men = sum(men)/len(men)

bar_count = np.arange(2)
bar_width = 0.5

not_survived = np.array([(1-rate_men)*100, (1-rate_women)*100])
survived = np.array([rate_men*100, rate_women*100])

plt.figure(figsize=(10, 8))
```

```
plt.bar(bar_count, survived, color='#f9bc86', edgecolor='white',␣
 ↪width=bar_width, label="Survived")
plt.bar(bar_count, not_survived, bottom=survived, color='#b5ffb9',␣
 ↪edgecolor='white', width=bar_width, label="Not Survived")

plt.xlabel('Sex', size=15, labelpad=20)
plt.ylabel('Survival Percentage', size=15, labelpad=20)
plt.xticks(bar_count, ['rate of survival for men: ({0:.2f}%)'.format(rate_men),␣
 ↪'rate of survival for women: ({0:.2f}%)'.format(rate_women)])
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 for Men and Women', size=18, y=1.05)

plt.show()
```

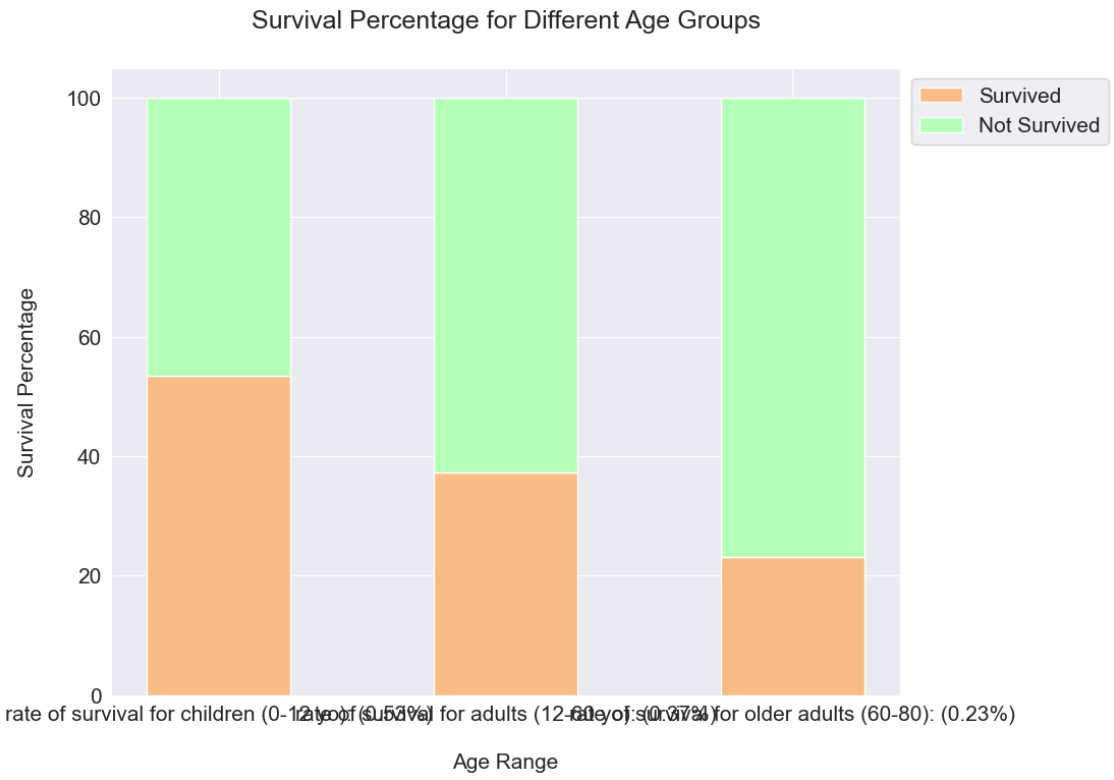


We can see that the survival rate for women is much higher than that for men. This aligns with our prediction of gender having a high impact on the survival rate.

[23]:
```
train_data['Age'].describe()
```

[23]:
```
count    891.000000
mean      29.618788
std       14.541098
min        0.420000
25%       20.000000
50%       28.000000
75%       38.000000
max       80.000000
Name: Age, dtype: float64
```

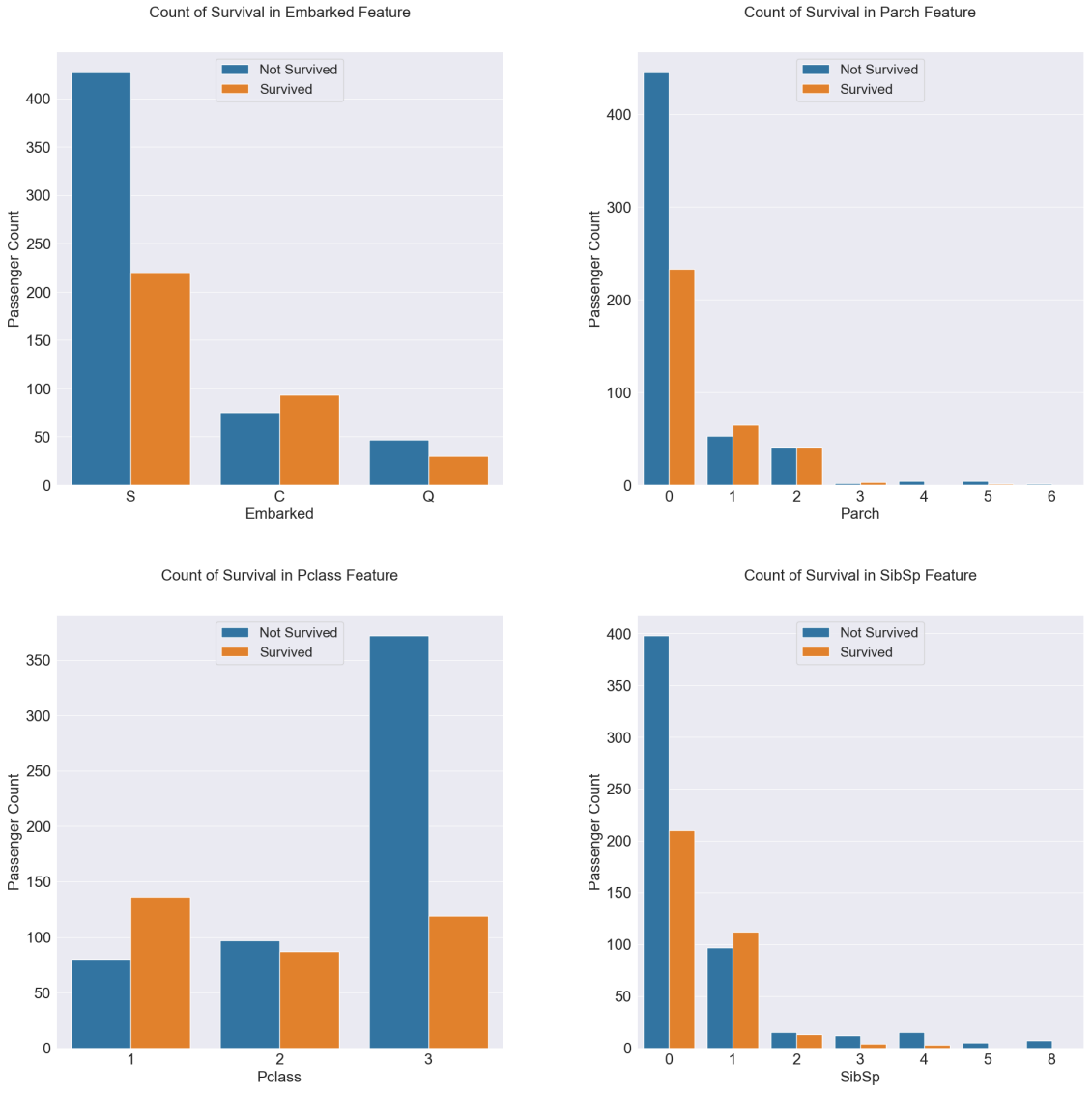
[85]:
```
# using some arbitrary age values based on the min, max and IQR
youth = train_data.loc[train_data.Age <= 12]['Survived']
rate_youth = sum(youth)/len(youth)

mid = train_data.loc[(train_data.Age > 12) & (train_data.Age <= 60)]['Survived']
rate_mid = sum(mid)/len(mid)

older = train_data.loc[train_data.Age > 60]['Survived']
rate_older = sum(older)/len(older)

bar_count = np.arange(3)
bar_width = 0.5

not_survived = np.array([(1-rate_youth)*100, (1-rate_mid)*100,␣
 ↪(1-rate_older)*100])
survived = np.array([rate_youth*100, rate_mid*100, rate_older*100])

plt.figure(figsize=(10, 8))
plt.bar(bar_count, survived, color='#f9bc86', edgecolor='white',␣
 ↪width=bar_width, label="Survived")
plt.bar(bar_count, not_survived, bottom=survived, color='#b5ffb9',␣
 ↪edgecolor='white', width=bar_width, label="Not Survived")

plt.xlabel('Age Range', size=15, labelpad=20)
plt.ylabel('Survival Percentage', size=15, labelpad=20)
plt.xticks(bar_count, ['rate of survival for children (0-12 yo): ({0:.2f}%)'.
 ↪format(rate_youth), 'rate of survival for adults (12-60 yo): ({0:.2f}%)'.
 ↪format(rate_mid), 'rate of survival for older adults (60-80): ({0:.2f}%)'.
 ↪format(rate_older)])
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 for Different Age Groups', size=18, y=1.05)

plt.show()
```



Sadly, it seems like the odds of survival for children was only 50 percent, and the odds only decrease as you get older. We can use this to infer that there is a weak negative correlation between age and survival rate.

[87]:
```
# code is adopted from https://www.kaggle.com/code/gunesevitan/
 ↪titanic-advanced-feature-engineering-tutorial?
 ↪scriptVersionId=27280410&cellId=42
cat_features = ['Embarked', 'Parch', 'Pclass', 'SibSp']

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

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

    plt.xlabel('{}'.format(feature), size=20)
    plt.ylabel('Passenger Count', size=20)
    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.
↪07)

plt.tight_layout()
plt.subplots_adjust(hspace=0.3, wspace=0.3)
plt.show()
```



Some interesting notes on each categorical feature: - The upper class has the best survival to non-survival ratio. The lower class survival ratio is really low. Passengers in the lower class seem to have a very high percentage of not surviving - The passengers that embarked in Southampton have a much lower chance of survival compared to Cherbourg and Queenstown. Perhaps many of the passengers that boarded in Southampton were lower class? There may be some collinearity between class and the port of embarkment. - Parch and Sibsp have very similar distributions. On

the left side, having 0 parents, children, siblings, or spouses seems to indicate a lower survival rate. The survival rate is slightly better if you have one parent/child or one sibling/spouse.

[26]:
```
## descriptive stats
set_option('display.width', 100)
set_option('display.precision', 2)

description = train_data_eda.describe()
description
```

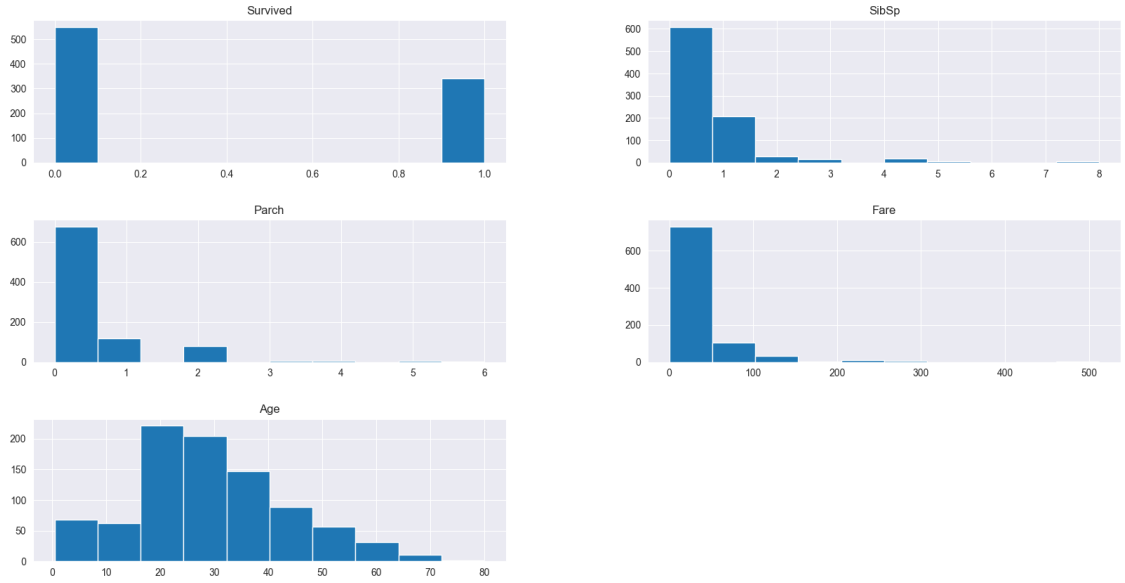
[26]:

	Survived	SibSp	Parch	Fare	Age
count	891.00	891.00	891.00	891.00	891.00
mean	0.38	0.52	0.38	32.20	29.62
std	0.49	1.10	0.81	49.69	14.54
min	0.00	0.00	0.00	0.00	0.42
25%	0.00	0.00	0.00	7.91	20.00
50%	0.00	0.00	0.00	14.45	28.00
75%	1.00	1.00	0.00	31.00	38.00
max	1.00	8.00	6.00	512.33	80.00

1.7 Notes on Descriptive Statistics

For the number of siblings and spouses, it seems like most passengers do not have siblings or spouses, because both the 25% quartile and 50% quartile are 0. It only just reaches one at the 75 percent quartile, but there is a max of 8 (that's a big family!). It seems most of the passengers on board were both single and were not traveling with a sibling. For the number of parents and children, we also see that the 25% quartile, 50% quartile, and 75% quartile passengers were not on board with any parents or children. The max number of parents and children for a passenger is six. Overall, we have very few passengers traveling on the Titanic with family. For the fare, first class tickets started at 30 pounds and went up to 105,000 pounds. (https://www.nbclosangeles.com/news/national-international/history-of-the-titanic-10-questions-about-the-ill-fated-ship/3173692/#:~:text=Second%2Dclass%20tickets%20were%2012,class%20tickets%20would%
Second class fare was 12 pounds, and third class fare was 7 pounds. Interesting enough, the IQR for fare reflects the price points for the fare pretty closely. Our earlier count plot of the class indicates that there were many more third class passengers than first or second class. For age, because the IQR ranges from 20-28-38, it seems that most passengers were in their 20s and 30s. This makes sense, given that there are so many passengers traveling without family. It's starting to look like the most common passenger on the Titanic was single, male, and third class.

[27]:
```
## histograms
train_data_eda.hist(figsize = (20, 10))
plt.margins(2,2)
plt.subplots_adjust(hspace = 0.4)
plt.show()
```



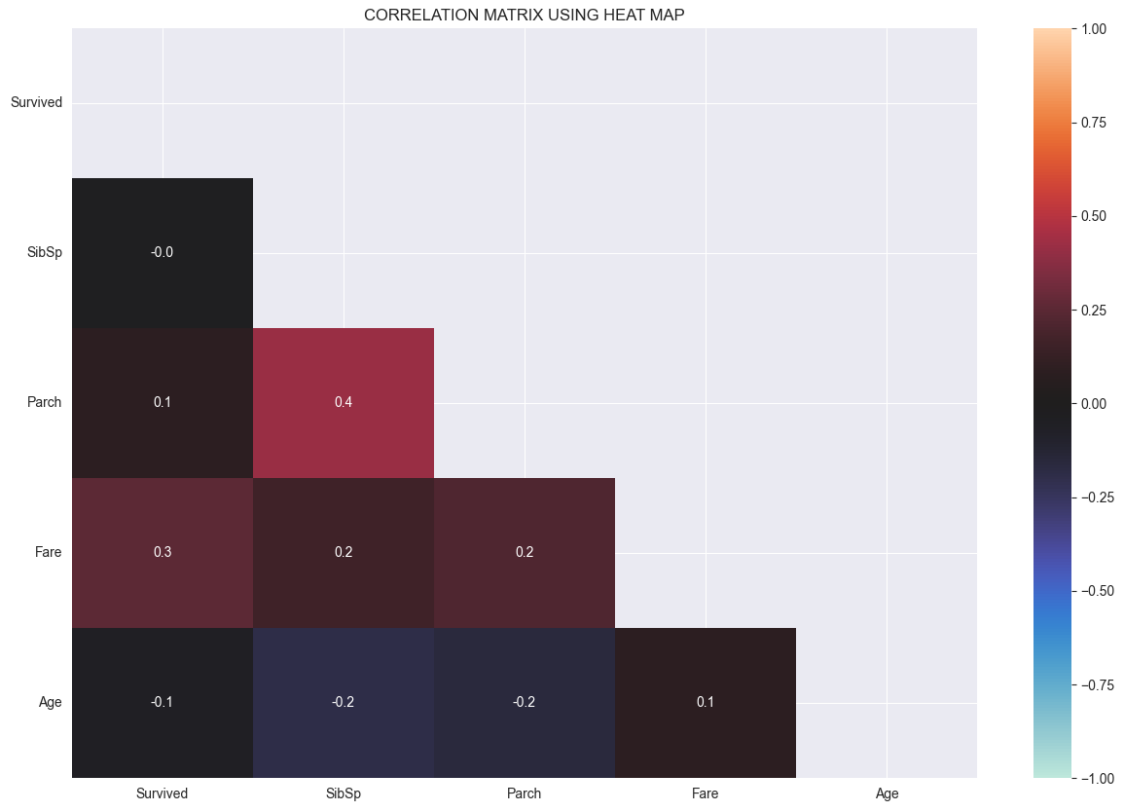
2 Notes on Histograms

Similar to the observations from our descriptive statistics, these histograms indicate that our most common passenger was single and third-class. The age graph is the only one that even slightly resembles a bell curve. Our models may have better performance if we standardize the age data to reduce the impact of outliers, like those of passengers over the age of 60.

The reason why this feature stands out compared to the others might be because age is actually ordinal compared to the other features. The distribution of age on a ship is best described with numbers. However, features such as fare, parch, and sibSp are kind of categorical ordinal features. Even though they are represented by numbers, those numbers indicate different subcategories of information. For instance, you either have one sibling on the ship with you, or you have zero. There's no situation where one would get a smooth gradient of numeric information between either of those cases. As such, it might not be very helpful to try to process these specific features with standardization or normalization.

[82]:
```
## correlation heatmap
import seaborn as sns
import numpy as np

plt.figure(figsize=(15,10))
corMat = train_data_eda.corr(method="pearson")
mask_upper = np.triu(corMat)
sns.heatmap(corMat, annot=True, vmin = -1, vmax = 1, center = 0, mask =␣
 ↪mask_upper, fmt=".1f")
plt.yticks(rotation=0)
plt.xticks(rotation=0)
plt.title("CORRELATION MATRIX USING HEAT MAP")
```

```
plt.show()
```



2.1 Notes on Heatmap

None of our features have a strong positive or negative correlation with the survival rate. The number of parents & children feature seems somewhat positively correlated with the number of siblings & spouse(s). It is not significantly high as to infer collinearity.

[29]:
```
## scatterplots
from pandas.plotting import scatter_matrix

scatter_matrix(train_data_eda, figsize = (15,10))
plt.show()
```

2.2 Notes on Scatter plots

None of the scatter plots show any strong linear relationship with each other. Like we mentioned above, the only continuous quantitative types of data included in this dataset are age and fare. All the quantitative data used in this analysis are discrete quantitative datatypes, which explains the distinct lines in the scatter plots for Parch and SibSp.

2.3 EDA with Standardized Data

[30]:
```
data_stan = pd.DataFrame(StandardScaler().fit_transform(x_clean),␣
  ↪columns=x_clean_names)
x_stan = pd.concat([data_stan, x_category], axis=1)
x_stan
```

[30]:

	SibSp	Parch	Fare	Age	Cabin	Sex	Embarked	Pclass
0	0.43	-0.47	-0.50	-0.52	8	1	2	3
1	0.43	-0.47	0.79	0.58	2	0	0	1
2	-0.47	-0.47	-0.49	-0.25	8	0	2	3
3	0.43	-0.47	0.42	0.37	2	0	2	1
4	-0.47	-0.47	-0.49	0.37	8	1	2	3
..	…	…	…	…	…	…	…	…
886	-0.47	-0.47	-0.39	-0.18	8	1	2	2
887	-0.47	-0.47	-0.04	-0.73	1	0	2	1

```
888   0.43   2.01 -0.18  0.37      8    0         2       3
889  -0.47  -0.47 -0.04 -0.25      2    1         0       1
890  -0.47  -0.47 -0.49  0.16      8    1         1       3

[891 rows x 8 columns]
```

[31]:
```
data_stan = pd.concat([y, data_stan], axis=1)
data_stan
```

[31]:
```
     Survived  SibSp  Parch  Fare   Age
0           0   0.43  -0.47 -0.50 -0.52
1           1   0.43  -0.47  0.79  0.58
2           1  -0.47  -0.47 -0.49 -0.25
3           1   0.43  -0.47  0.42  0.37
4           0  -0.47  -0.47 -0.49  0.37
..        …    …     …     …     …
886         0  -0.47  -0.47 -0.39 -0.18
887         1  -0.47  -0.47 -0.04 -0.73
888         0   0.43   2.01 -0.18  0.37
889         1  -0.47  -0.47 -0.04 -0.25
890         0  -0.47  -0.47 -0.49  0.16

[891 rows x 5 columns]
```

[32]:
```
X_arr_stan = x_stan.values
Y_arr_stan = y.values

data_stan.describe()
```

[32]:
```
       Survived     SibSp     Parch      Fare       Age
count    891.00  8.91e+02  8.91e+02  8.91e+02  8.91e+02
mean       0.38  4.39e-17  5.38e-17  3.99e-18  9.57e-17
std        0.49  1.00e+00  1.00e+00  1.00e+00  1.00e+00
min        0.00 -4.75e-01 -4.74e-01 -6.48e-01 -2.01e+00
25%        0.00 -4.75e-01 -4.74e-01 -4.89e-01 -6.62e-01
50%        0.00 -4.75e-01 -4.74e-01 -3.57e-01 -1.11e-01
75%        1.00  4.33e-01 -4.74e-01 -2.42e-02  5.77e-01
max        1.00  6.78e+00  6.97e+00  9.67e+00  3.47e+00
```

[33]:
```
train_data_eda.describe()
```

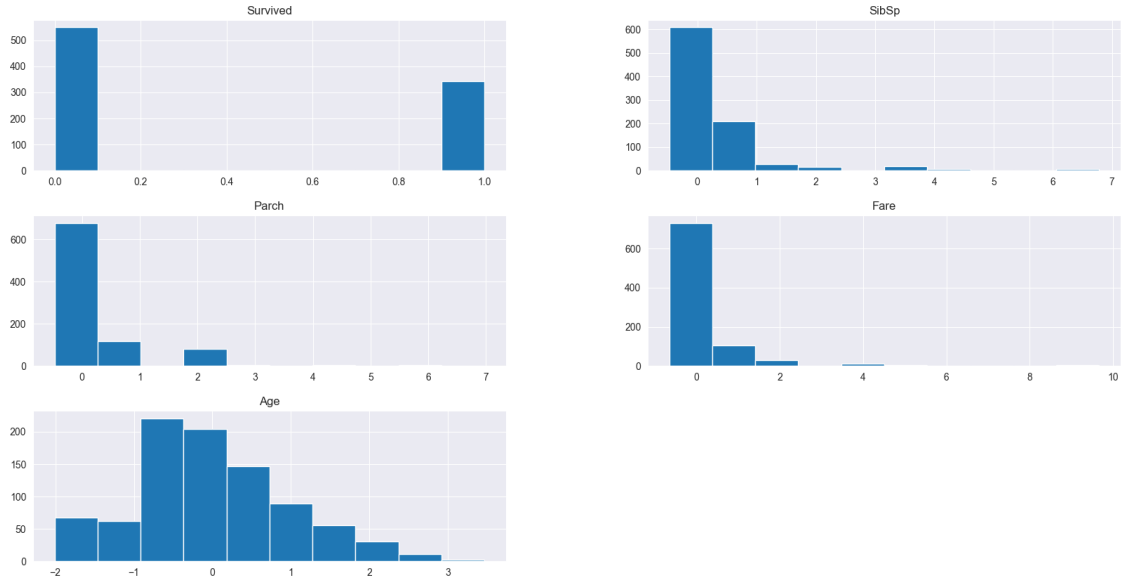
[33]:
```
       Survived   SibSp   Parch    Fare     Age
count    891.00  891.00  891.00  891.00  891.00
mean       0.38    0.52    0.38   32.20   29.62
std        0.49    1.10    0.81   49.69   14.54
min        0.00    0.00    0.00    0.00    0.42
25%        0.00    0.00    0.00    7.91   20.00
```

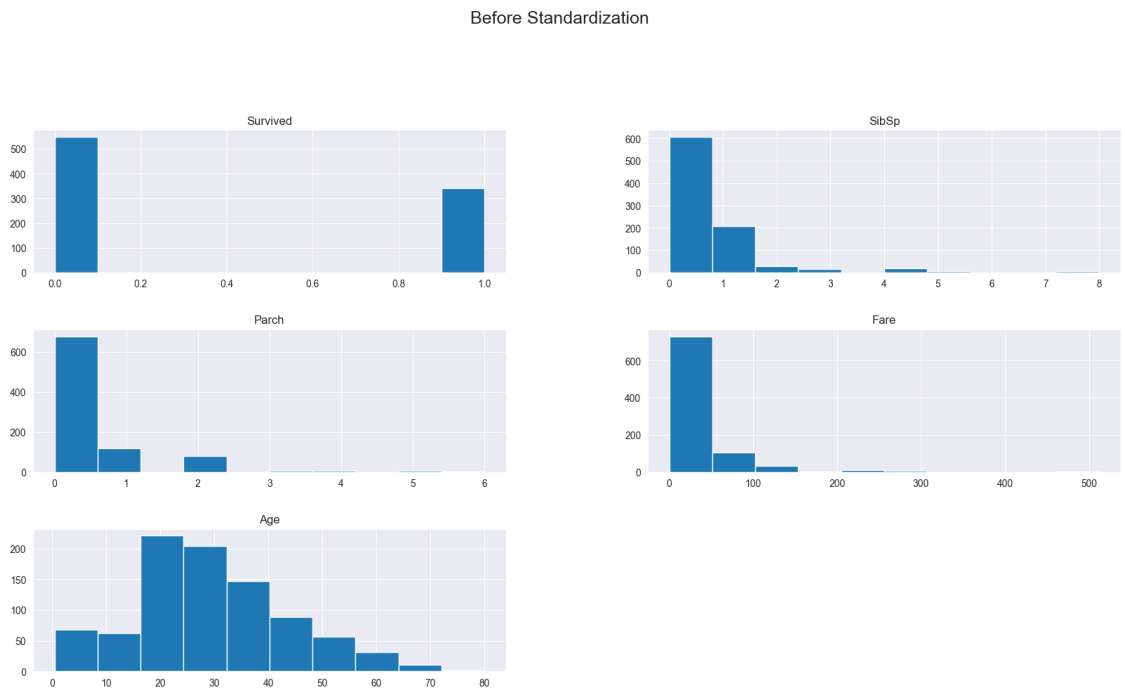
50%	0.00	0.00	0.00	14.45	28.00
75%	1.00	1.00	0.00	31.00	38.00
max	1.00	8.00	6.00	512.33	80.00

2.4 Notes on Descriptive Statistics for Standardized Data

By standardizing the data we can see that the standard deviation for the Age and Fare is reduced greatly. The standard deviation for Age moved from 14.54 to 1, so we can expect to see fewer outliers when plotting the age histogram. Hopefull, this will also reduce the impact of outliers for model predictions as well. The standard deviation for Fare moved from 49.69 yo 1, this will probably reduce the outliers for fare from the upper-class range (30-150,000 pounds). Like we predicted, neither Parch and SibSp are much affected by standardizing the data.

[34]:
```
## histograms
data_stan.hist( figsize = (20, 10))
train_data_eda.hist(figsize = (20, 10))
plt.margins(2,2)
plt.subplots_adjust(hspace = 0.4)
plt.suptitle("Before Standardization", size=18, y=1.05)
plt.show()
```

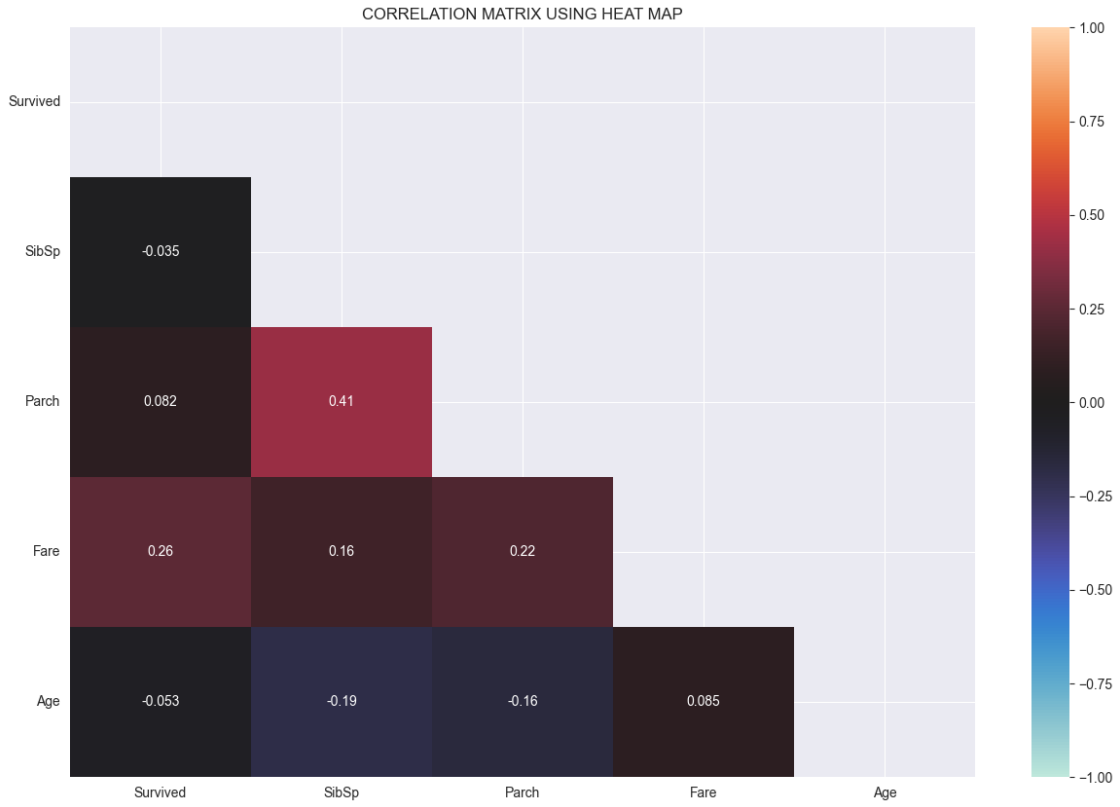




3 Notes on Histograms

Surprisingly, the distribution curve before and after standardization do not seem very different for any of the features.

[91]:
```
## correlation heatmap
plt.figure(figsize=(15,10))
corMat = data_stan.corr(method="pearson")
mask_upper = np.triu(corMat)
sns.heatmap(corMat, annot=True, vmin = -1, vmax = 1, center = 0, mask =␣
 ↪mask_upper)
plt.yticks(rotation=0)
plt.xticks(rotation=0)
plt.title("CORRELATION MATRIX USING HEAT MAP")
plt.show()
```



3.1 Notes on Heatmap

Our correlations also do not seem much affected by standardizing the values.

[90]:
```
## scatter plots
scatter_matrix(data_stan, figsize = (15,10))
plt.show()
```

3.2 Notes on Scatter Plots With Standardized Data

Same as the heatmap, there are no big visual differences between the scatter plots that were not standardized and those that were standardized.

Let's see if there are any distinctive changes once we normalize the data.

3.3 EDA with Normalized Data

Typically, data normalization should help with data redundancy and keep features with higher magnitudes from having a larger impact than those with smaller values. Because the age and fare categories have larger numeric values (0-80 years of age and 7 to 512 pounds) compared to sibsp, and parch (0-8) we might see some slight performance changes when it comes to model predictions.

[37]:
```
data_norm = pd.DataFrame(MinMaxScaler().fit_transform(x_clean),␣
↪columns=x_clean_names)
x_norm = pd.concat([data_norm, x_category], axis=1)
x_norm
```

[37]:

	SibSp	Parch	Fare	Age	Cabin	Sex	Embarked	Pclass
0	0.12	0.00	0.01	0.27	8	1	2	3
1	0.12	0.00	0.14	0.47	2	0	0	1
2	0.00	0.00	0.02	0.32	8	0	2	3
3	0.12	0.00	0.10	0.43	2	0	2	1

```
4     0.00   0.00  0.02  0.43      8    1         2       3
..     …     …     …     …     …  …       …      …
886   0.00   0.00  0.03  0.33      8    1         2       2
887   0.00   0.00  0.06  0.23      1    0         2       1
888   0.12   0.33  0.05  0.43      8    0         2       3
889   0.00   0.00  0.06  0.32      2    1         0       1
890   0.00   0.00  0.02  0.40      8    1         1       3

[891 rows x 8 columns]
```

[38]:
```
X_arr_norm = x_norm.values
Y_arr_norm = y.values

data_norm = pd.concat([y, data_norm], axis=1)
data_norm
```

[38]:
```
     Survived  SibSp  Parch  Fare   Age
0           0   0.12   0.00  0.01  0.27
1           1   0.12   0.00  0.14  0.47
2           1   0.00   0.00  0.02  0.32
3           1   0.12   0.00  0.10  0.43
4           0   0.00   0.00  0.02  0.43
..        …     …     …     …     …
886         0   0.00   0.00  0.03  0.33
887         1   0.00   0.00  0.06  0.23
888         0   0.12   0.33  0.05  0.43
889         1   0.00   0.00  0.06  0.32
890         0   0.00   0.00  0.02  0.40

[891 rows x 5 columns]
```

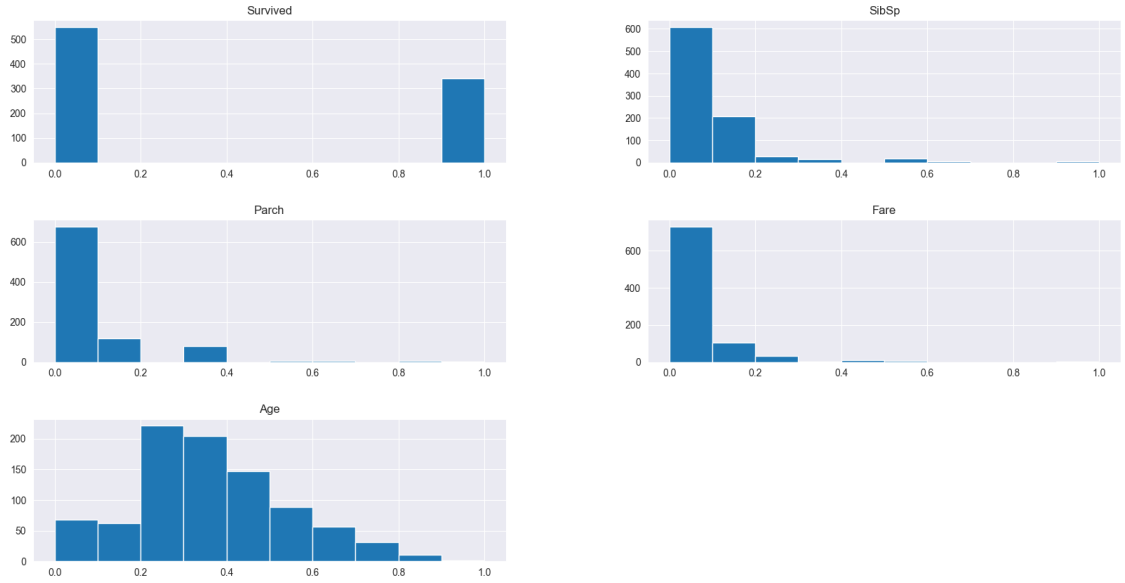
[39]:
```
## descriptive stats
set_option('display.width', 100)
set_option('display.precision', 2)
description = data_norm.describe()
description
```

[39]:
```
       Survived   SibSp   Parch    Fare     Age
count    891.00  891.00  891.00  891.00  891.00
mean       0.38    0.07    0.06    0.06    0.37
std        0.49    0.14    0.13    0.10    0.18
min        0.00    0.00    0.00    0.00    0.00
25%        0.00    0.00    0.00    0.02    0.25
50%        0.00    0.00    0.00    0.03    0.35
75%        1.00    0.12    0.00    0.06    0.47
max        1.00    1.00    1.00    1.00    1.00
```

3.4 Notes on Descriptive Statistics on Normalized Data

The standard deviation for Parch, SibSp, Fare, Age are now all pretty similar. Normalization did a good job of making it easier to compare features with different ranges of ordinal values. We can still see that SibSp, Parch and Fare are heavily skewed left, which we should expect to see in the histogram visualization below.

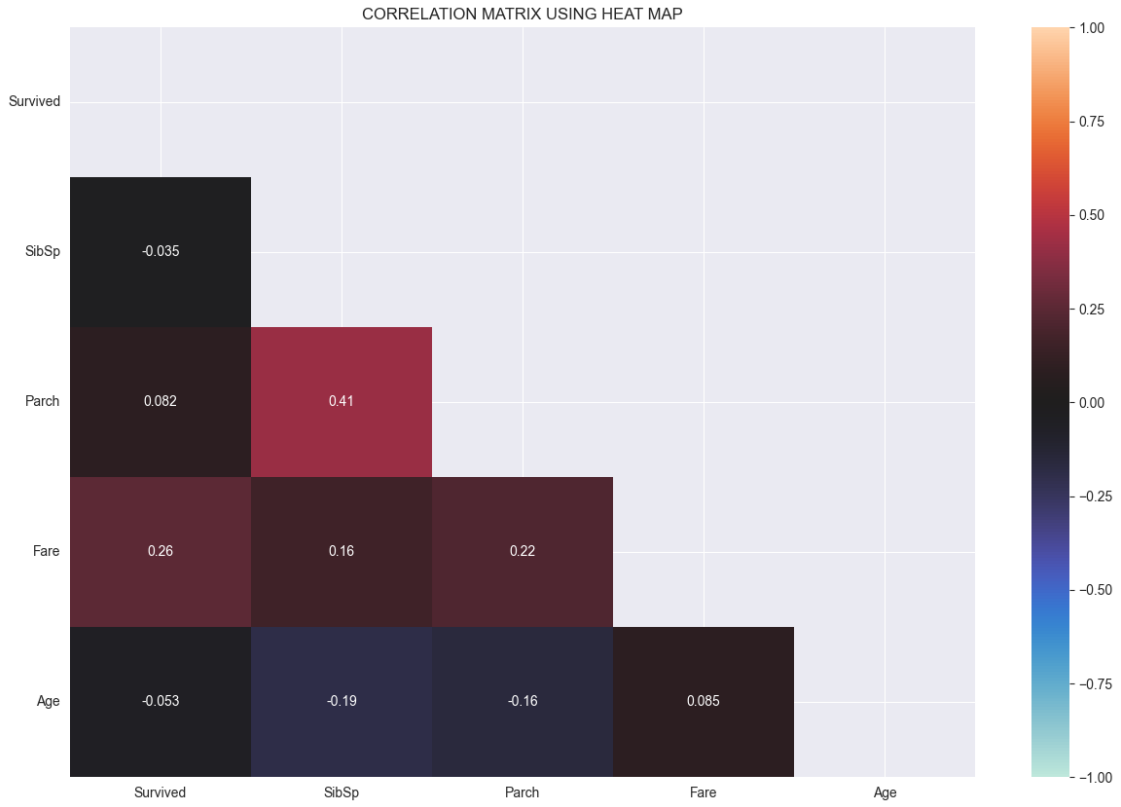
[40]:
```
## histograms
data_norm.hist(figsize = (20, 10))
plt.margins(0,1)
plt.subplots_adjust(hspace = 0.4)
plt.show()
```



4 Notes on Histograms for Normalized Data

There are no visual changes for the different features when looking at the histograms themselves, since normalization just remaps the values from 0-1.

[89]:
```
## correlation heatmap
plt.figure(figsize=(15,10))
data_norm.corr(method="pearson")
mask_upper = np.triu(corMat)
sns.heatmap(corMat, annot=True, vmin = -1, vmax = 1, center = 0, mask =␣
 ↪mask_upper)
plt.yticks(rotation=0)
plt.xticks(rotation=0)
plt.title("CORRELATION MATRIX USING HEAT MAP")
plt.show()
```



4.1 Notes on Heatmap for Normalized Data

Similar to the histograms, normalization seems to have no effect on our heatmap correlation values.

[88]:
```
## scatter plots
scatter_matrix(data_norm, figsize = (15,10))
plt.show()
```

4.2 Notes on Scatter Plots for Normalized Data

Similar to the previous analyses, we are not able to uncover any important changes after normalizing the data, at least not from looking at the scatter plots.

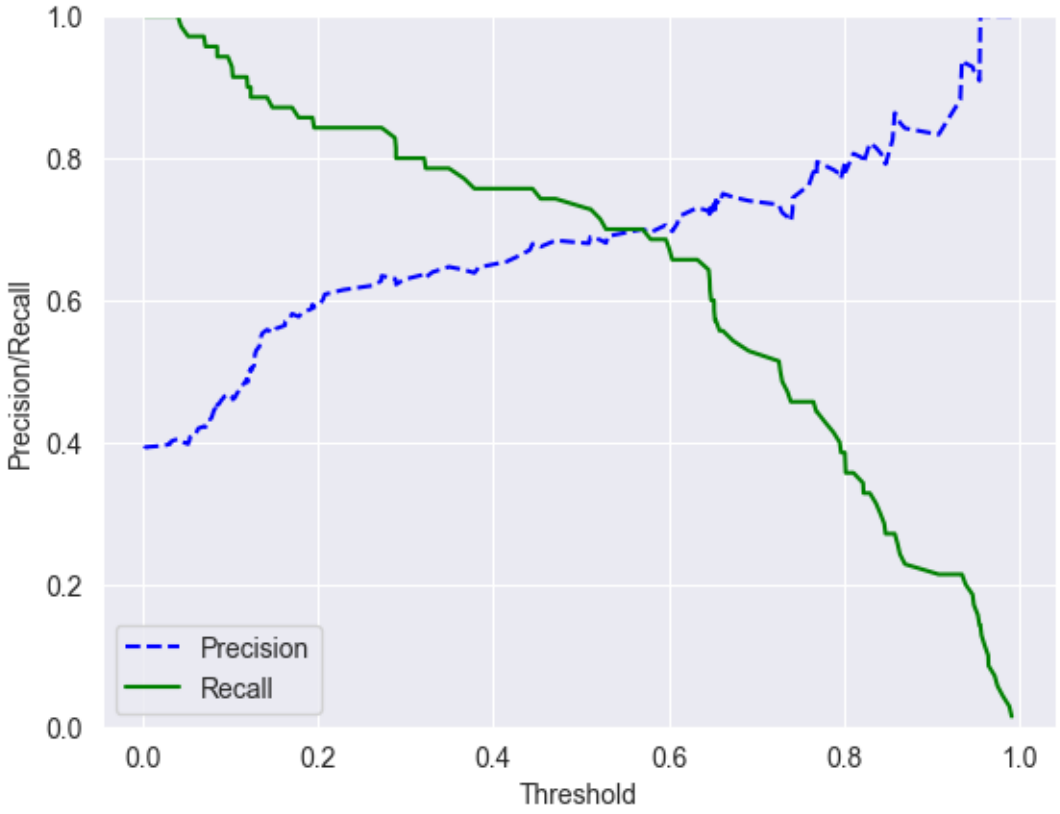
4.3 Model Fitting

[43]:
```
# split train data for calculating accuracy

x_train, x_test, y_train, y_test = train_test_split(x_num, y, test_size=0.20,␣
↪random_state=3)
x_train_stan, x_test_stan, y_train_stan, y_test_stan = train_test_split(x_stan,␣
↪y, test_size=0.20, random_state=3)
x_train_norm, x_test_norm, y_train_norm, y_test_norm = train_test_split(x_norm,␣
↪y, test_size=0.20, random_state=3)
```

4.3.1 Logistic Regression

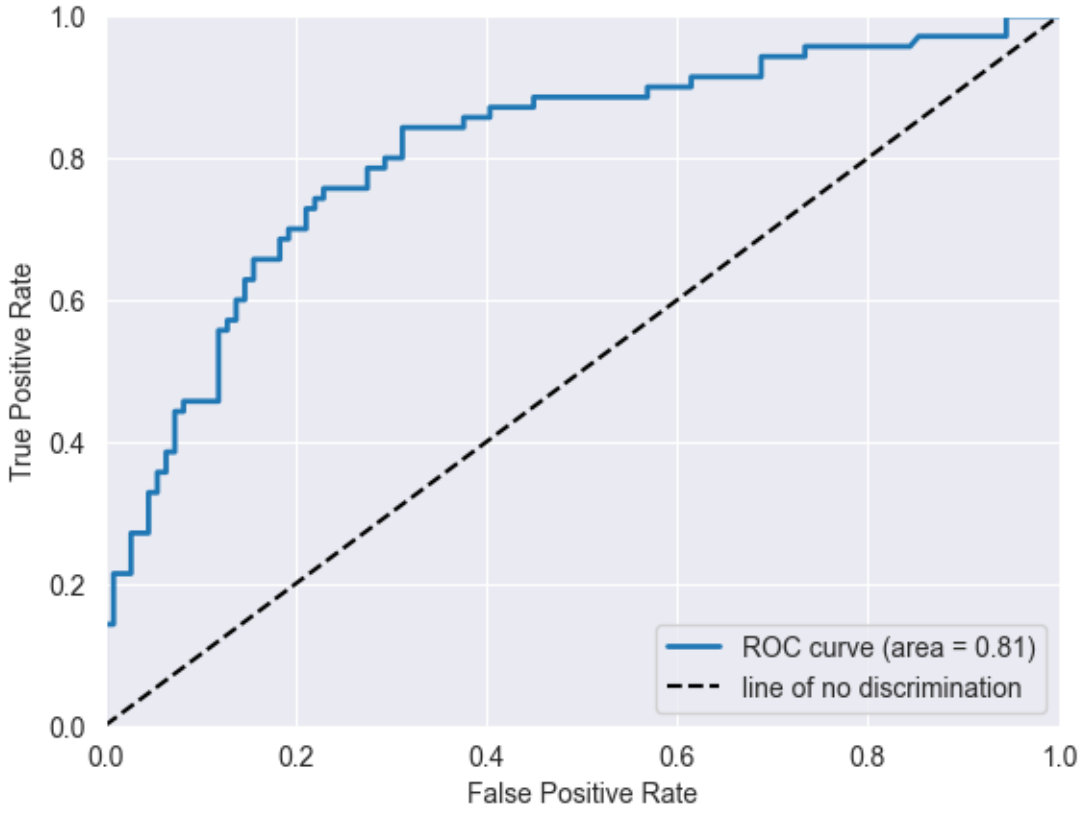
[44]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import precision_recall_curve


logit_model = LogisticRegression(max_iter=10000, random_state=3)
```

```
logit = logit_model.fit(x_train,y_train)

y_scores =logit.predict_proba(x_test)
precisions, recalls, thresholds = precision_recall_curve(y_test, y_scores[:,1])

def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    plt.xlabel("Threshold")
    plt.legend(loc="lower left")
    plt.ylim([0, 1])
    plt.ylabel("Precision/Recall")

plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.show()
```



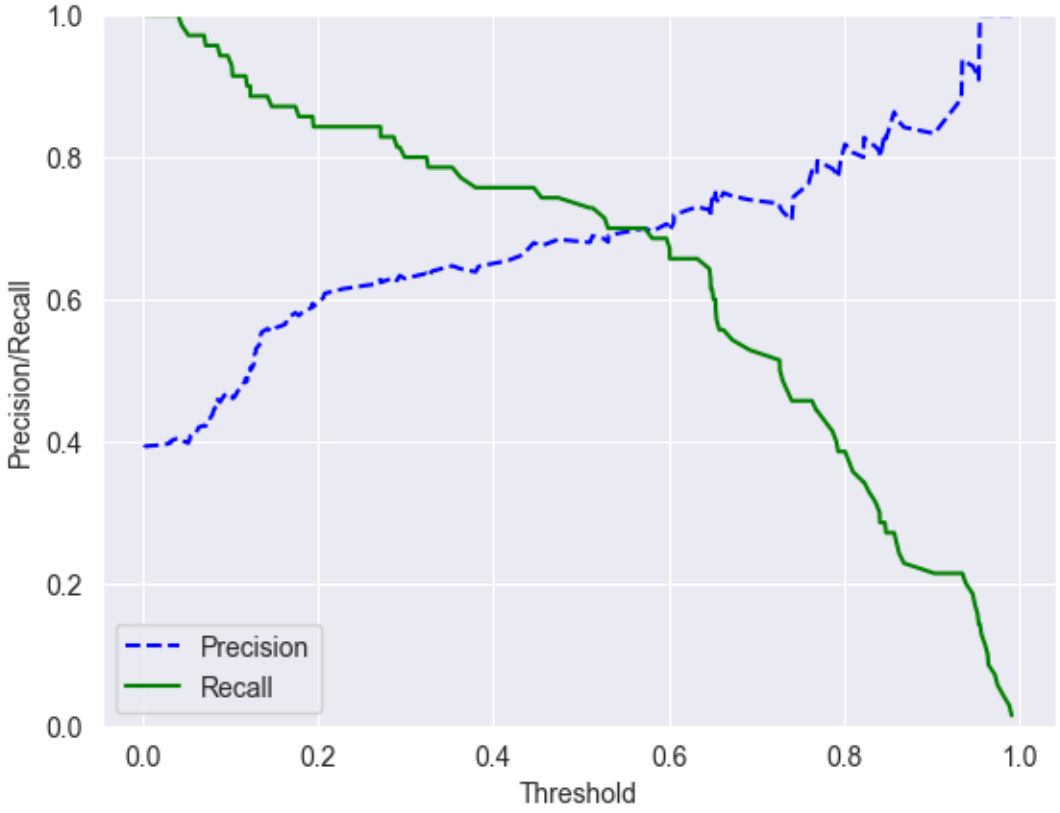
[45]:
```
fpr_lr, tpr_lr, thresholds2 = roc_curve(y_test, y_scores[:,1])
roc_auc = auc(fpr_lr, tpr_lr)

def plot_roc_curve(fpr, tpr, label=None):
```

```
    plt.plot(fpr,tpr, linewidth=2, label=label[0])
    plt.plot([0, 1], [0, 1], 'k--', label=label[1])
    plt.axis((0, 1, 0, 1))
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')

plt.figure()
plot_roc_curve(fpr_lr, tpr_lr, ['ROC curve (area = %0.2f)' % roc_auc, 'line of␣
 ↪no discrimination'])
plt.legend(loc="lower right")
plt.show()
```



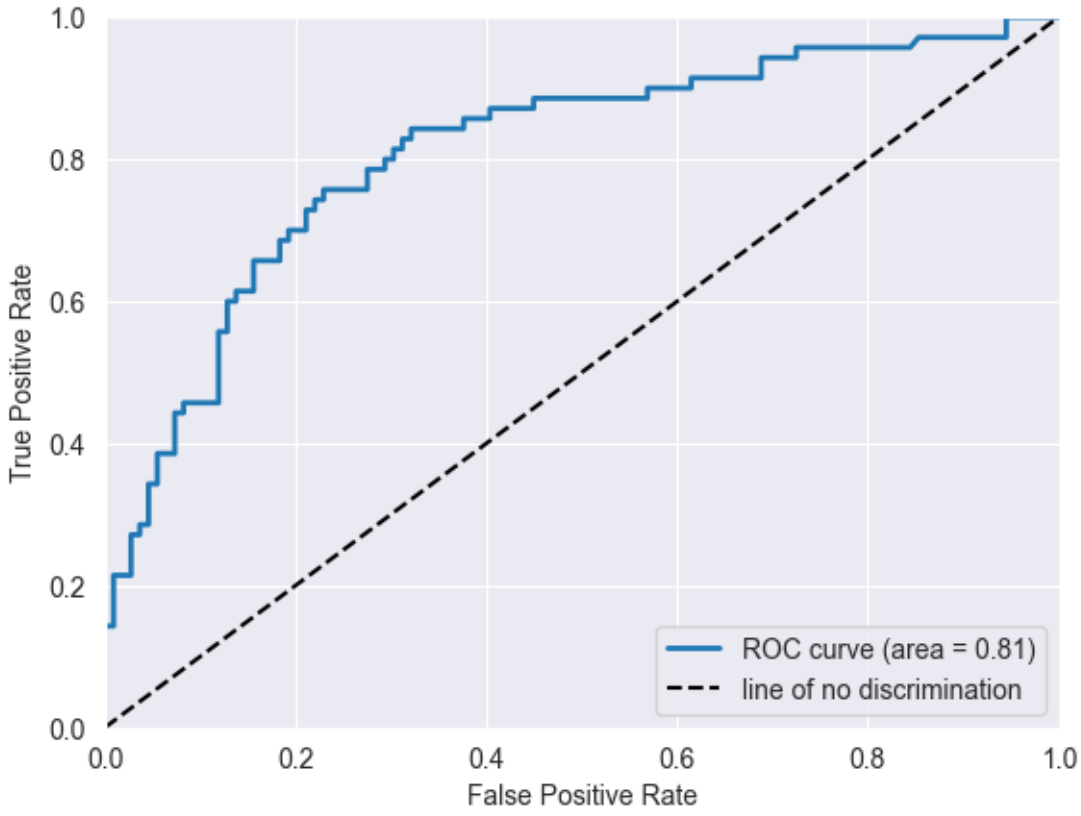
[46]:
```
logit_model_stan = LogisticRegression(max_iter=10000, random_state=3)
logit_stan = logit_model.fit(x_train_stan,y_train_stan)

y_scores =logit_stan.predict_proba(x_test_stan)
precisions, recalls, thresholds = precision_recall_curve(y_test_stan, y_scores[:
 ↪,1])

def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
```

```
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    plt.xlabel("Threshold")
    plt.legend(loc="lower left")
    plt.ylim([0, 1])
    plt.ylabel("Precision/Recall")

plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.show()
```



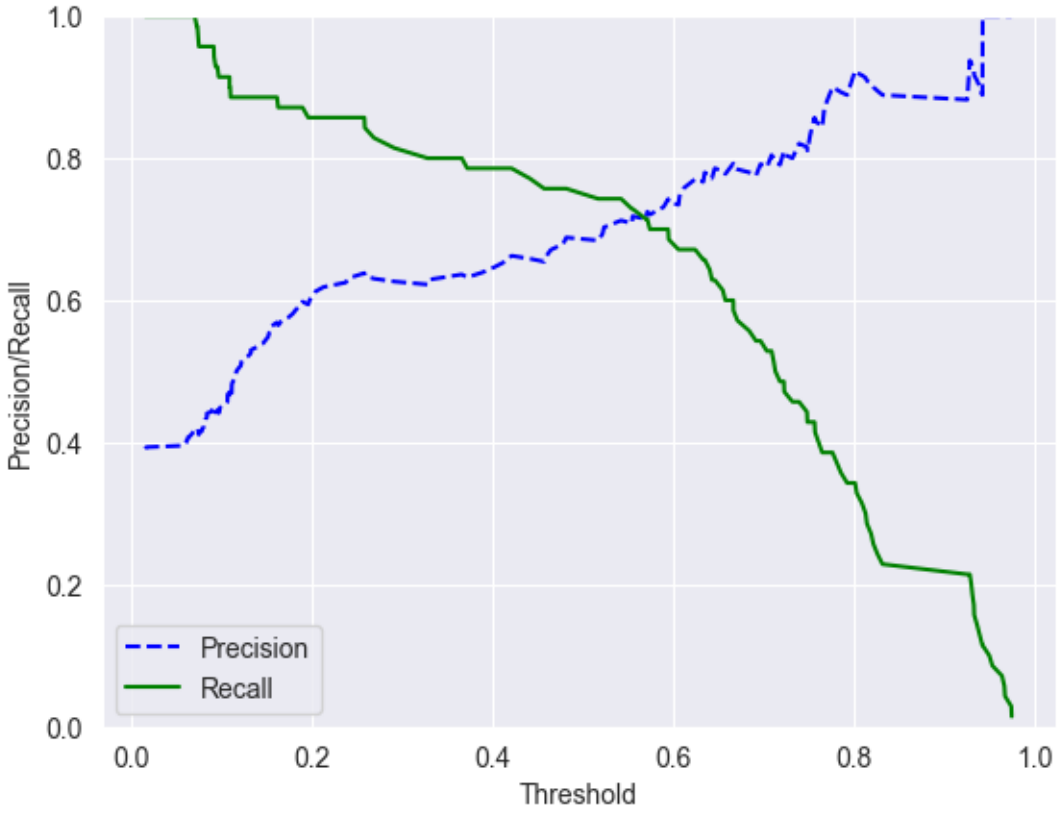
[47]:
```
fpr_lr, tpr_lr, _ = roc_curve(y_test_stan, y_scores[:,1])
roc_auc = auc(fpr_lr, tpr_lr)

def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr,tpr, linewidth=2, label=label[0])
    plt.plot([0, 1], [0, 1], 'k--', label=label[1])
    plt.axis((0, 1, 0, 1))
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
```

```
plt.figure()
plot_roc_curve(fpr_lr, tpr_lr, ['ROC curve (area = %0.2f)' % roc_auc, 'line of␣
 ↪no discrimination'])
plt.legend(loc="lower right")
plt.show()
```

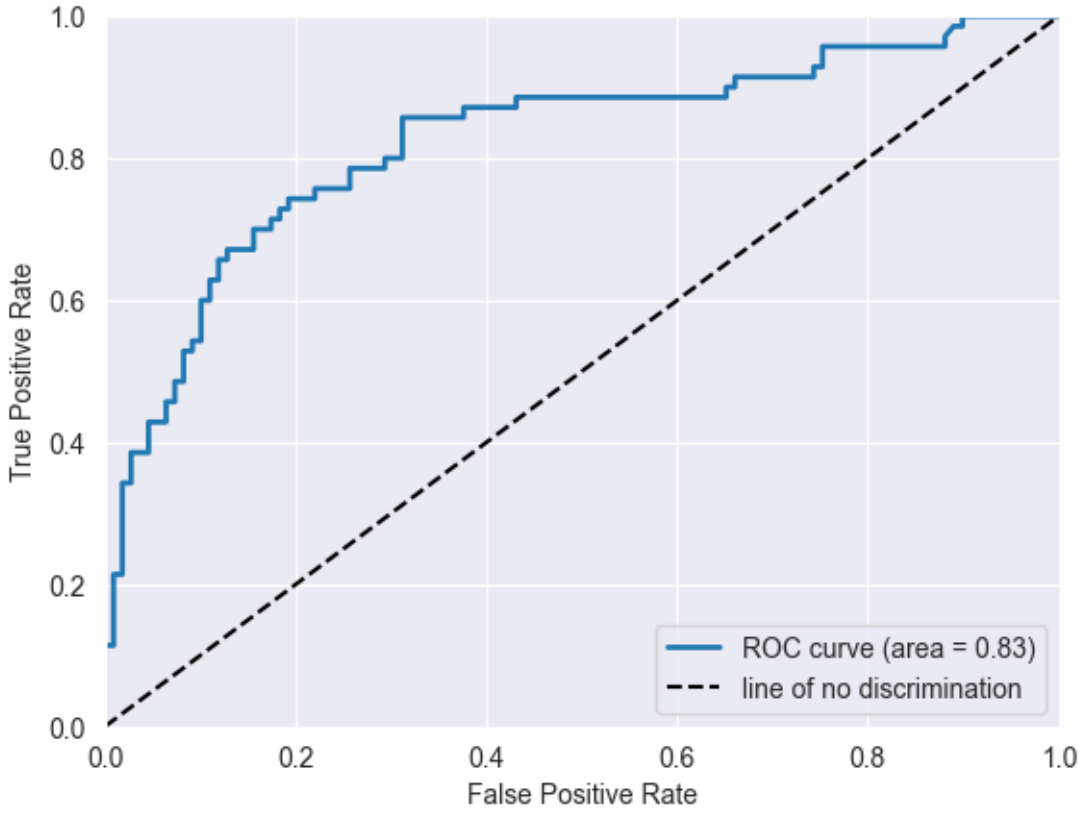


[48]:
```
logit_model_norm = LogisticRegression(max_iter=10000,random_state=3)
logit_norm = logit_model_norm.fit(x_train_norm,y_train_norm)

y_scores =logit_norm.predict_proba(x_test_norm)
precisions, recalls, thresholds = precision_recall_curve(y_test_norm, y_scores[:
 ↪,1])

def plot_precision_recall_vs_threshold(precisions, recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Precision")
    plt.plot(thresholds, recalls[:-1], "g-", label="Recall")
    plt.xlabel("Threshold")
    plt.legend(loc="lower left")
    plt.ylim([0, 1])
    plt.ylabel("Precision/Recall")
```

```
plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.show()
```



[49]:
```
fpr_lr, tpr_lr, _ = roc_curve(y_test_norm, y_scores[:,1])
roc_auc = auc(fpr_lr, tpr_lr)

def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr,tpr, linewidth=2, label=label[0])
    plt.plot([0, 1], [0, 1], 'k--', label=label[1])
    plt.axis((0, 1, 0, 1))
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')

plt.figure()
plot_roc_curve(fpr_lr, tpr_lr, ['ROC curve (area = %0.2f)' % roc_auc, 'line of␣
 ↪no discrimination'])
plt.legend(loc="lower right")
plt.show()
```



From the logistic regression models, all three sets of data yield a similar intercepting threshold of around 0.57, indicating it is a reliable threshold to start with. Yet, the performance of the model built by the normalized set of data is slightly higher by observing the area under the ROC curve, an area of 0.83 compared to 0.81 of the other models. A possible reason behind may be due to the limiting the range of inputs in normalization, which allows all features to weigh similar when building the model, especially with the presence of categorical data involved in this dataset. Therefore, if logistic regression is selected, we can start with the model from the normalized data with a threshold of 0.57, then adjust it according to the need of a stricter or more lenient model.

4.3.2 Decision Tree Classifier

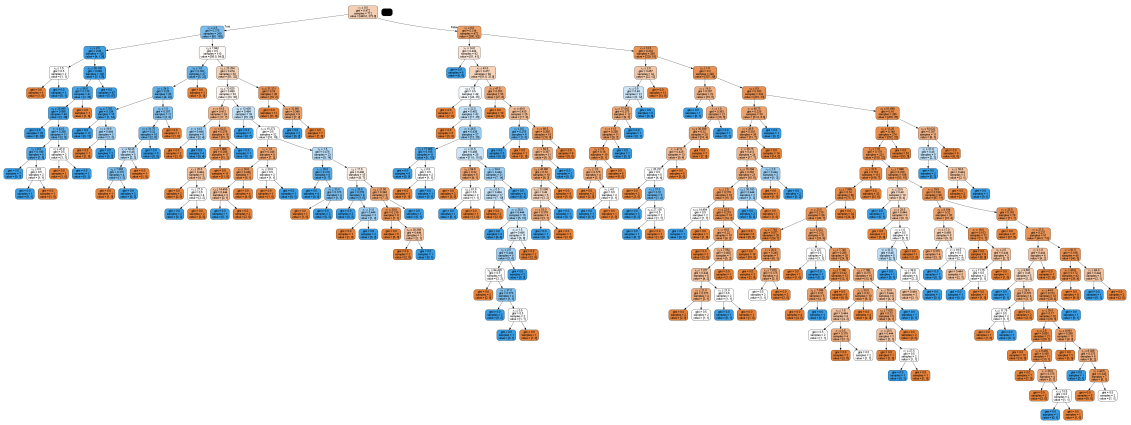
[50]:
```
from sklearn.tree import DecisionTreeClassifier
from io import StringIO
from sklearn.tree import export_graphviz
import pydotplus
from IPython.display import Image


# decision tree with no max depth
dectree = DecisionTreeClassifier(random_state=3)
dectree.fit(x_train, y_train)
```

```
y_pred = dectree.predict(x_test)

df=pd.DataFrame({'Actual':y_test, 'Predicted':y_pred})

print(accuracy_score(y_test, y_pred))
```

0.8212290502793296

[51]:
```
dot_data = StringIO()
export_graphviz(dectree, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

[51]:



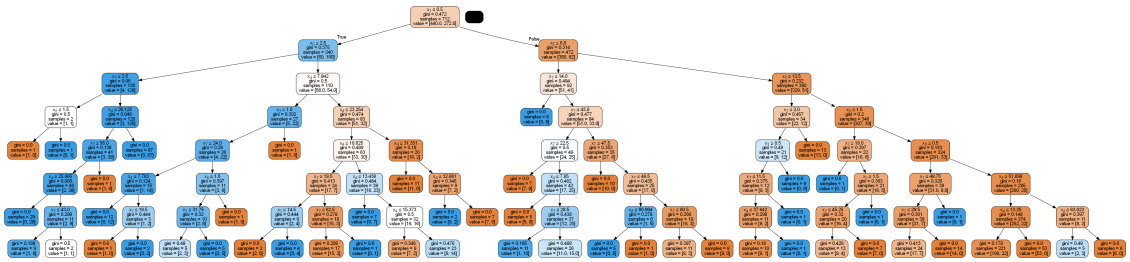
[52]:
```
# decision tree with max depth = 7
dectree_7 = DecisionTreeClassifier(max_depth=7, random_state=3)
dectree_7.fit(x_train, y_train)

y_pred = dectree_7.predict(x_test)

pd.DataFrame({'Actual':y_test, 'Predicted':y_pred})

print(accuracy_score(y_test, y_pred))
```

0.8268156424581006

[53]:
```
dot_data = StringIO()
export_graphviz(dectree_7, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
```

```
Image(graph.create_png())
```

[53]:

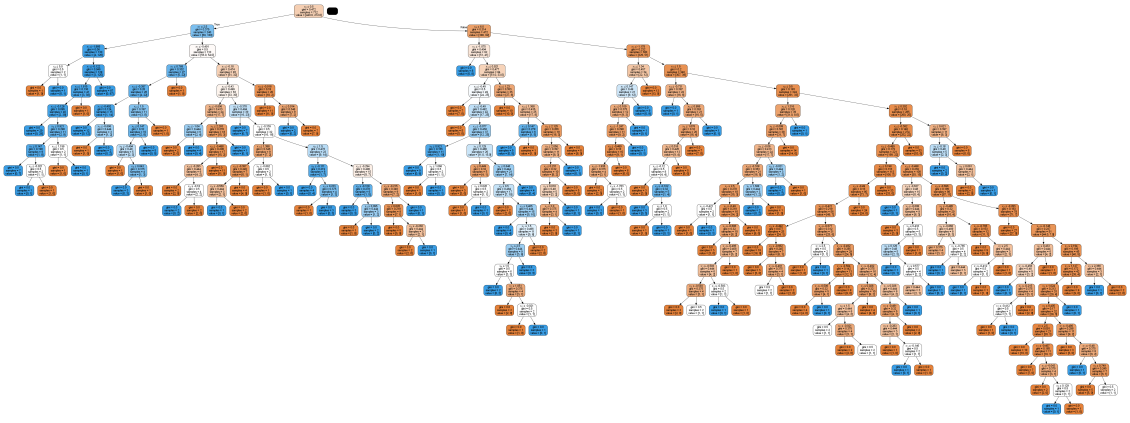


[54]:
```
dectree_stan = DecisionTreeClassifier(random_state=3)
dectree_stan.fit(x_train_stan, y_train_stan)

y_pred_stan = dectree_stan.predict(x_test_stan)

pd.DataFrame({'Actual':y_test_stan, 'Predicted':y_pred_stan})

print(accuracy_score(y_test_stan, y_pred_stan))
```

```
0.7988826815642458
```

[55]:
```
dot_data = StringIO()
export_graphviz(dectree_stan, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

[55]:



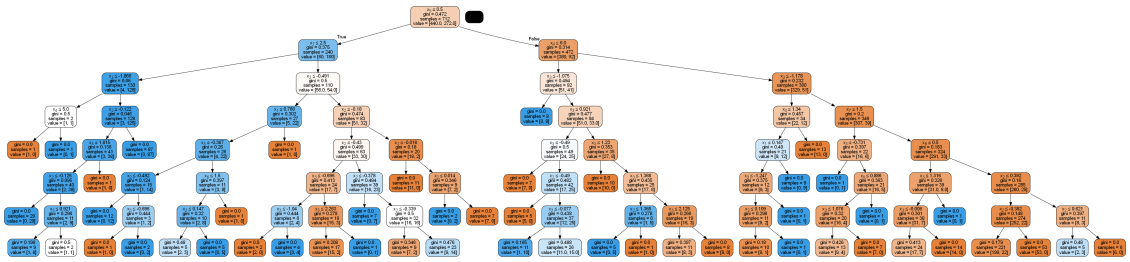
[56]:
```
# decision tree with max depth = 7
dectree_7_stan = DecisionTreeClassifier(max_depth=7, random_state=3)
dectree_7_stan.fit(x_train_stan, y_train_stan)
```

```
y_pred_stan = dectree_7_stan.predict(x_test_stan)

pd.DataFrame({'Actual':y_test_stan, 'Predicted':y_pred_stan})

print(accuracy_score(y_test_stan, y_pred_stan))
```

```
0.8324022346368715
```

[57]:
```
dot_data = StringIO()
export_graphviz(dectree_7_stan, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

[57]:

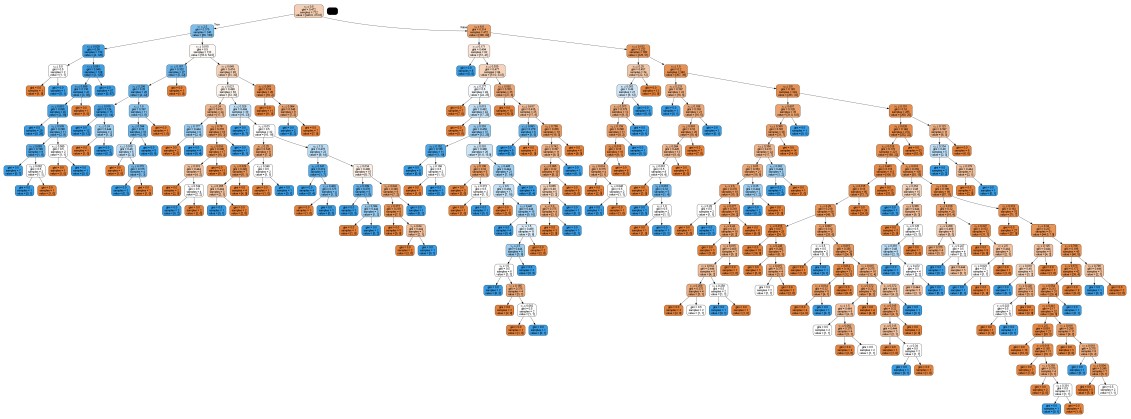


[58]:
```
dectree_norm = DecisionTreeClassifier(random_state=3)
dectree_norm.fit(x_train_norm, y_train_norm)

y_pred = dectree_norm.predict(x_test_norm)

pd.DataFrame({'Actual':y_test_norm, 'Predicted':y_pred})

print(accuracy_score(y_test_norm, y_pred))
```

```
0.7988826815642458
```

[59]:
```
dot_data = StringIO()
export_graphviz(dectree_norm, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

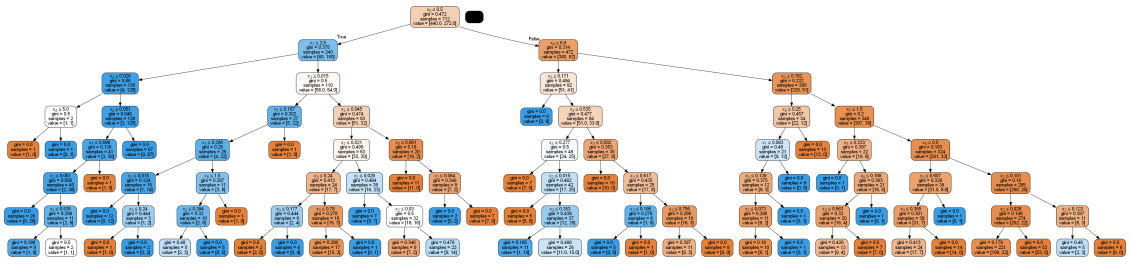
[59]:



[60]:
```
# decision tree with max depth = 7
dectree_7_norm = DecisionTreeClassifier(max_depth=7, random_state=3)
dectree_7_norm.fit(x_train_norm, y_train_norm)

y_pred_norm = dectree_7_norm.predict(x_test_norm)

pd.DataFrame({'Actual':y_test_norm, 'Predicted':y_pred_norm})

print(accuracy_score(y_test_norm, y_pred_norm))
```

```
0.8268156424581006
```

[61]:
```
dot_data = StringIO()
export_graphviz(dectree_7_norm, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

[61]:


Accuracy scores fluctuates in small ranges with decrease in maximum depth allowed until maximum depth reaches 7. A significant drop in score to 0.77 or 0.78 with further reduce in performance is observed with further decrease in maximum depth for all decision tree models. Therefore, a maximum depth of 7 is selected for models created by the three different versions of dataset.

Among the models, the model with maximum depth of 7 with standardized data has the highest performance.

4.3.3 Bagging (Ensemble)

[62]:
```
from sklearn.ensemble import BaggingClassifier

bag_clf = BaggingClassifier(
    dectree, n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1, random_state=3)
bag_clf.fit(x_train, y_train)
y_pred = bag_clf.predict(x_test)

# determine accuracy score for the bagging method
print(accuracy_score(y_test, y_pred))
```

```
0.7932960893854749
```

[63]:
```
bag_clf_stan = BaggingClassifier(
    dectree_stan, n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1, random_state=3)
bag_clf_stan.fit(x_train_stan, y_train_stan)
y_pred = bag_clf_stan.predict(x_test_stan)

# determine accuracy score for the bagging method
print(accuracy_score(y_test_stan, y_pred))
```

```
0.7877094972067039
```

[64]:
```
bag_clf_norm = BaggingClassifier(
    dectree_norm, n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1, random_state=3)
bag_clf_norm.fit(x_train_norm, y_train_norm)
y_pred = bag_clf_norm.predict(x_test_norm)

# determine accuracy score for the bagging method
print(accuracy_score(y_test_norm, y_pred))
```

```
0.7877094972067039
```

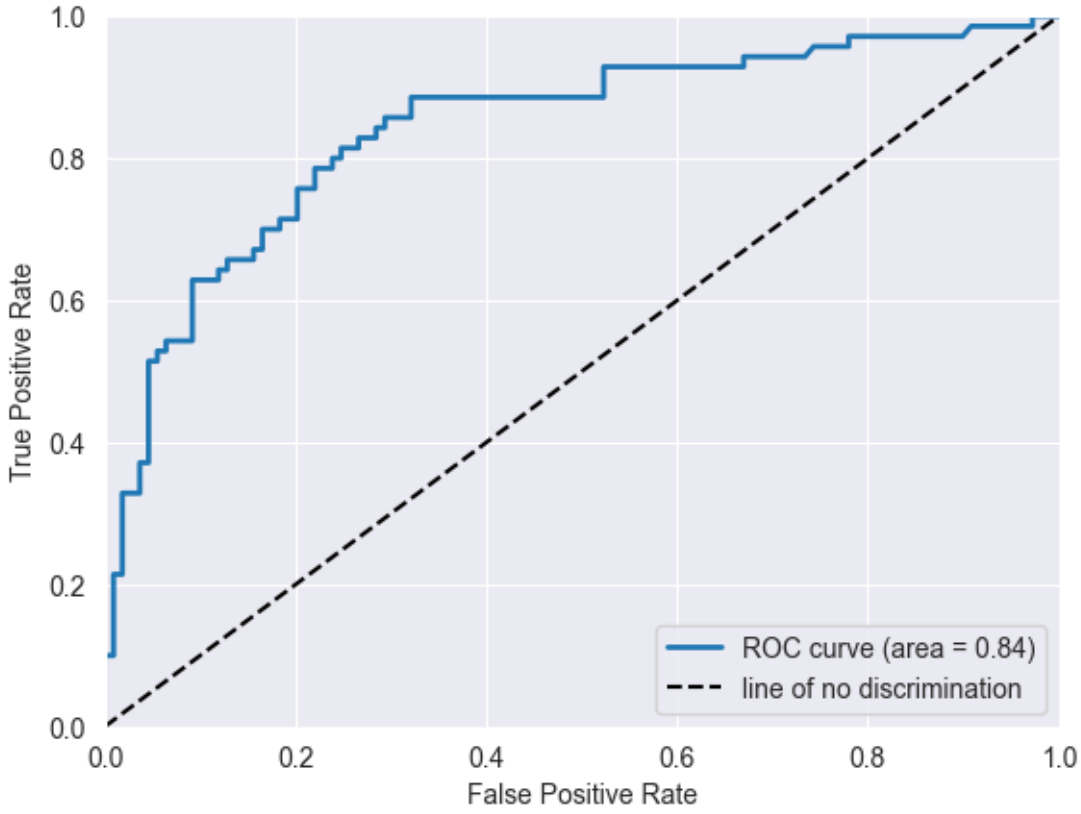
After bagging, the accuracy score dropped from simple decision tree without ensemble techniques. This is possible since ensemble will rearrange the weight of samples after each turn. The random splitting of training and testing data set may coincidently led to an ideal split in which the training data and the test data have similar distribution of sample, leading to the fact that a simple decision tree will perform as well or even better than including bagging. Therefore, we will do a cross validation to further conclude if bagging will increase the performance of the decision tree.

4.3.4 Random Forest Classifier (Ensemble)

[65]:
```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16,␣
 ↪n_jobs=-1, random_state=3)
rnd_clf.fit(x_train, y_train)
y_prob_rf = rnd_clf.predict_proba(x_test)

y_pred_rf = rnd_clf.predict(x_test)

accuracy_score(y_test, y_pred_rf)
```

[65]:
```
0.770949720670391
```

[66]:
```
np.sum(y_pred == y_pred_rf) / len(y_pred)

y_score_rf = y_prob_rf[:,1]
fpr_rf,tpr_rf, threshold_rf = roc_curve(y_test, y_score_rf)
roc_auc = auc(fpr_rf, tpr_rf)

def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr,tpr, linewidth=2, label=label[0])
    plt.plot([0, 1], [0, 1], 'k--', label=label[1])
    plt.axis((0, 1, 0, 1))
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')

plot_roc_curve(fpr_rf,tpr_rf, ['ROC curve (area = %0.2f)' % roc_auc, 'line of␣
 ↪no discrimination'])
plt.legend(loc="lower right")
plt.show()
```

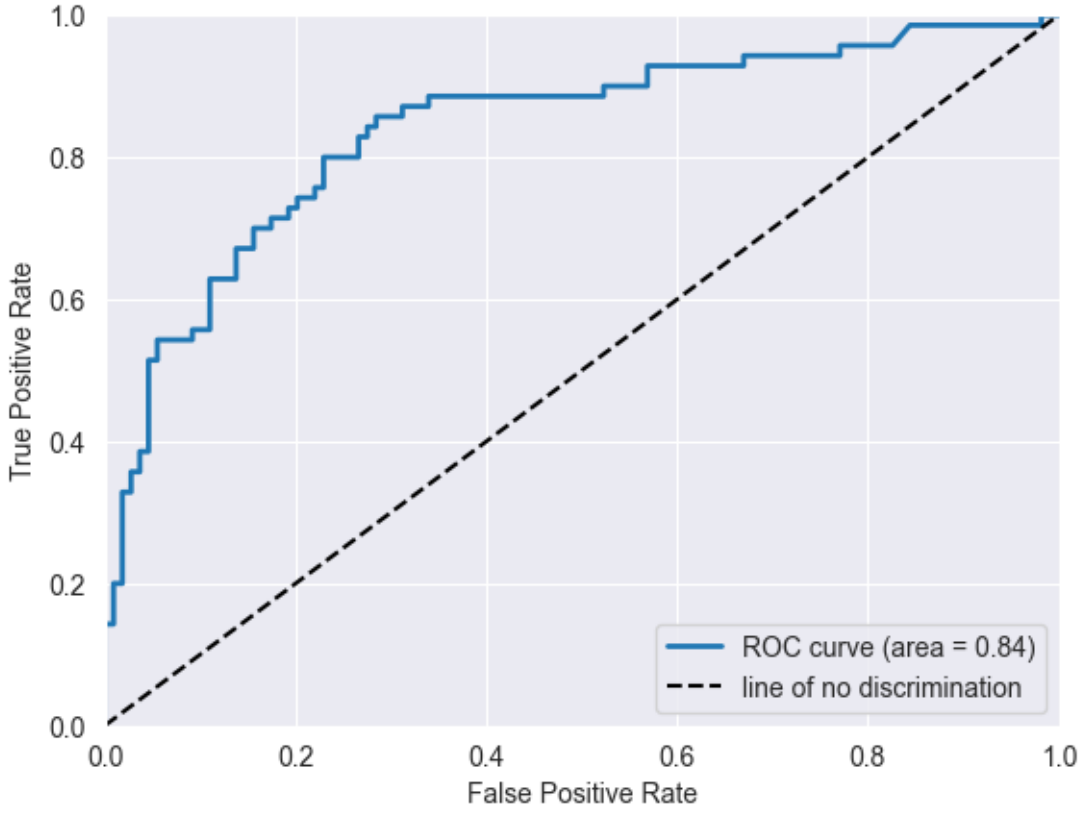


[67]:
```
rnd_clf_stan = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16,␣
 ↪n_jobs=-1, random_state=3)
rnd_clf_stan.fit(x_train_stan, y_train_stan)
y_prob_rf_stan = rnd_clf_stan.predict_proba(x_test_stan)

y_pred_rf_stan = rnd_clf_stan.predict(x_test_stan)

accuracy_score(y_test_stan, y_pred_rf_stan)
```

[67]:
```
0.776536312849162
```

[68]:
```
np.sum(y_pred == y_pred_rf_stan) / len(y_pred)  # almost identical predictions

y_score_rf_stan = y_prob_rf_stan[:,1]
fpr_rf,tpr_rf, threshold_rf = roc_curve(y_test_stan, y_score_rf_stan)
roc_auc = auc(fpr_rf, tpr_rf)

def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr,tpr, linewidth=2, label=label[0])
    plt.plot([0, 1], [0, 1], 'k--', label=label[1])
```

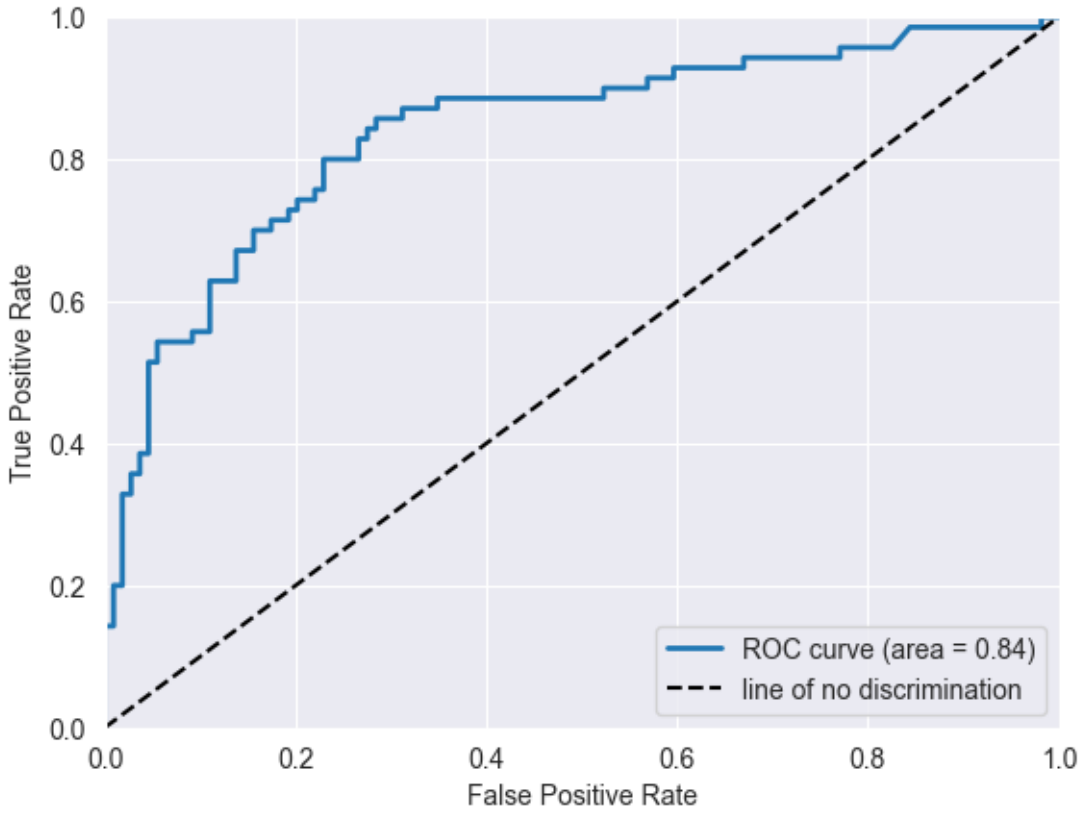
```
    plt.axis((0, 1, 0, 1))
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')

plot_roc_curve(fpr_rf,tpr_rf, ['ROC curve (area = %0.2f)' % roc_auc, 'line of␣
 ↪no discrimination'])
plt.legend(loc="lower right")
plt.show()
```



[69]:
```
rnd_clf_norm = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16,␣
 ↪n_jobs=-1, random_state=3)
rnd_clf_norm.fit(x_train_norm, y_train_norm)
y_prob_rf_norm = rnd_clf_norm.predict_proba(x_test_norm)

y_pred_rf_norm = rnd_clf_norm.predict(x_test_norm)

accuracy_score(y_test_norm, y_pred_rf_norm)
```

[69]: 0.776536312849162

[70]:
```
np.sum(y_pred == y_pred_rf_norm) / len(y_pred)  # almost identical predictions

y_score_rf_norm = y_prob_rf_norm[:,1]
fpr_rf,tpr_rf, _ = roc_curve(y_test_norm, y_score_rf_norm)
roc_auc = auc(fpr_rf, tpr_rf)

def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr,tpr, linewidth=2, label=label[0])
    plt.plot([0, 1], [0, 1], 'k--', label=label[1])
    plt.axis((0, 1, 0, 1))
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')

plot_roc_curve(fpr_rf,tpr_rf, ['ROC curve (area = %0.2f)' % roc_auc, 'line of␣
 ↪no discrimination'])
plt.legend(loc="lower right")
plt.show()
```



Random Forest Classifier is recommended by the original Kaggle Titanic competition. Yet, the accuracy score across the models are lower than that of simple decision tree and decision tree with bagging. Similar to what mentioned before, the random splitting of dataset may lead to a clean

data set that a simple decision tree will be sufficient in creating a reliable model. Yet, this may also be an indicator of overfitting. Therefore, cross validation may help in further analyzing which model provides a reliable prediction.

4.3.5 ADA Boost

[71]:
```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME", learning_rate=0.5, random_state=3)
ada_clf.fit(x_train, y_train)

y_pred_ada = ada_clf.predict(x_test)
accuracy_score(y_test, y_pred_ada)
```

[71]:
```
0.7653631284916201
```

[72]:
```
ada_clf_stan = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME", learning_rate=0.5, random_state=3)
ada_clf_stan.fit(x_train_stan, y_train_stan)

y_pred_ada_stan = ada_clf_stan.predict(x_test_stan)
accuracy_score(y_test_stan, y_pred_ada_stan)
```

[72]:
```
0.7653631284916201
```

[73]:
```
ada_clf_norm = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME", learning_rate=0.5, random_state=3)
ada_clf_norm.fit(x_train_norm, y_train_norm)

y_pred_ada_norm = ada_clf_norm.predict(x_test_norm)
accuracy_score(y_test_norm, y_pred_ada_norm)
```

[73]:
```
0.7653631284916201
```

According to scikit-learn documentation, the AdaBoost classifier fits a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but with different weights. Surprisingly,, our AdaBoost results are about the same with the non-preprocessed data, the standardized data, and the normalized data. This makes sense, given that decision trees are generally not affected by feature scaling (https://www.dataschool.io/comparing-supervised-learning-algorithms/).

4.3.6 Support Vector Machines

[74]:
```
from sklearn.svm import SVC

kernel_types = ['rbf', 'linear', 'poly', 'sigmoid']

for kernel in kernel_types:
    svc = SVC(kernel = kernel, degree = 2, C=1.0, random_state=3,␣
 ↪probability=True)
    svc.fit(x_train, y_train)
    y_prob_svc = svc.predict_proba(x_test)

    y_pred_svc = svc.predict(x_test)



    print(f'accuracy score for {kernel} : ', accuracy_score(y_test, y_pred_svc))

svc_bp = SVC(kernel = 'linear', degree = 2, C=1.0, random_state=3,␣
 ↪probability=True)
```

```
accuracy score for rbf :  0.6089385474860335
accuracy score for linear :  0.7988826815642458
accuracy score for poly :  0.6201117318435754
accuracy score for sigmoid :  0.6424581005586593
```

[75]:
```
for kernel in kernel_types:
    svc = SVC(kernel = kernel, degree = 2, C=1.0, random_state=4, max_iter =␣
 ↪1000000, probability=True)
    svc.fit(x_train_stan, y_train_stan)
    y_prob_svc_stan = svc.predict_proba(x_test_stan)

    y_pred_svc_stan = svc.predict(x_test_stan)



    print(f'accuracy score for {kernel} : ', accuracy_score(y_test_stan,␣
 ↪y_pred_svc_stan))

svc_bp_stan = SVC(kernel = 'linear', degree = 2, C=1.0, random_state=4,␣
 ↪max_iter = 1000000, probability=True)
```

```
accuracy score for rbf :  0.7877094972067039
accuracy score for linear :  0.7988826815642458
accuracy score for poly :  0.776536312849162
accuracy score for sigmoid :  0.6256983240223464
```

[76]:
```
for kernel in kernel_types:
    svc = SVC(kernel = kernel, degree = 2, C=1.0, random_state=4, max_iter =␣
 ↪1000000, probability=True)
    svc.fit(x_train_norm, y_train_norm)
    y_prob_svc_norm = svc.predict_proba(x_test_norm)

    y_pred_svc_norm = svc.predict(x_test_norm)



    print(f'accuracy score for {kernel} : ', accuracy_score(y_test_norm,␣
 ↪y_pred_svc_norm))

svc_bp_norm = SVC(kernel = 'linear', degree = 2, C=1.0, random_state=4,␣
 ↪max_iter = 1000000, probability=True)
```

```
accuracy score for rbf :  0.776536312849162
accuracy score for linear :  0.7877094972067039
accuracy score for poly :  0.7597765363128491
accuracy score for sigmoid :  0.6536312849162011
```

A warning of going over the maximum iteration is seen on creating Support Vector Machines from the raw training data, indicating that standardization and normalization supports in converging the model faster. In this case, we could see that standardization helps the most and create better performed models compared to normalization of the raw data. Moreover, linear kernel creates the best performed models when comparing to other kernals, indicating that the samples in the dataset provides with a clean cut on categorizing survived and not survived. Therefore, Linear SVM is selected for further evaluation by cross validation.

4.3.7 Neural Network

[77]:
```
from sklearn.neural_network import MLPClassifier

def mlp(x_train, y_train, x_test, y_test, label):
    clf_nn = MLPClassifier(random_state=3, max_iter=1000).fit(x_train, y_train)
    clf_nn.predict_proba(x_test)

    clf_nn.predict(x_test)
    score = clf_nn.score(x_test, y_test)
    print("score for " + label + f''': {score}''')
    return clf_nn
```

[78]:
```
mlp_clf = mlp(x_train, y_train, x_test, y_test, "non-preprocessed data")
mlp_clf_stand = mlp(x_train_stan, y_train_stan, x_test_stan, y_test_stan,␣
 ↪"standardized data")
mlp_clf_norm = mlp(x_train_norm, y_train_norm, x_test_norm, y_test_norm,␣
 ↪"normalized data")
```

```
score for non-preprocessed data: 0.7653631284916201
```

```
score for standardized data: 0.7653631284916201
score for normalized data: 0.770949720670391
```

For our neural network classification technique, we use the multi-layer perceptron algorithm from scikit-learn. According to the documentation on scikit-learn trains using backpropagation. It is sensitive to feature scaling, which aligns with our perfomance measure comparison. We can see that the normalised data performed better than the standardized data by 0.01. We can further improve the performance by using grid search cross validation to find which hyperparameters to tune for optimal performance. However, grid search cv is quite intensive on memory, and we are still learning how to best visualize the results from grid search cv, so we will not include it in this report.

4.4 Cross Validation with K-fold

[79]:
```
# prepare models
from sklearn.model_selection import KFold, cross_val_score


models = []
# OVO is selected instead of OVA as there are only 3 classes
models.append(('Logistic Regression', logit))
models.append(('Decision Tree', dectree))
models.append(('Decision Tree with Max Depth', dectree_7))
models.append(('Bagging', bag_clf))
models.append(('Random Forest', rnd_clf))
models.append(('ADA Boost', ada_clf))
models.append(('SVM', svc_bp))
models.append(('MLP', mlp_clf))

modelsStandDf = []

modelsStandDf.append(('Logistic Regression', logit_stan))
modelsStandDf.append(('Decision Tree', dectree_stan))
modelsStandDf.append(('Decision Tree with Max Depth', dectree_7_stan))
modelsStandDf.append(('Bagging', bag_clf_stan))
modelsStandDf.append(('Random Forest', rnd_clf_stan))
modelsStandDf.append(('ADA Boost', ada_clf_stan))
modelsStandDf.append(('SVM', svc_bp_stan))
modelsStandDf.append(('MLP', mlp_clf_stand))

modelsNormDf = []

modelsNormDf.append(('Logistic Regression', logit_norm))
modelsNormDf.append(('Decision Tree', dectree_norm))
modelsNormDf.append(('Decision Tree with Max Depth', dectree_7_norm))
modelsNormDf.append(('Bagging', bag_clf_norm))
modelsNormDf.append(('Random Forest', rnd_clf_norm))
modelsNormDf.append(('ADA Boost', ada_clf_norm))
```

```
modelsNormDf.append(('SVM', svc_bp_norm))
modelsNormDf.append(('MLP', mlp_clf_norm))
```

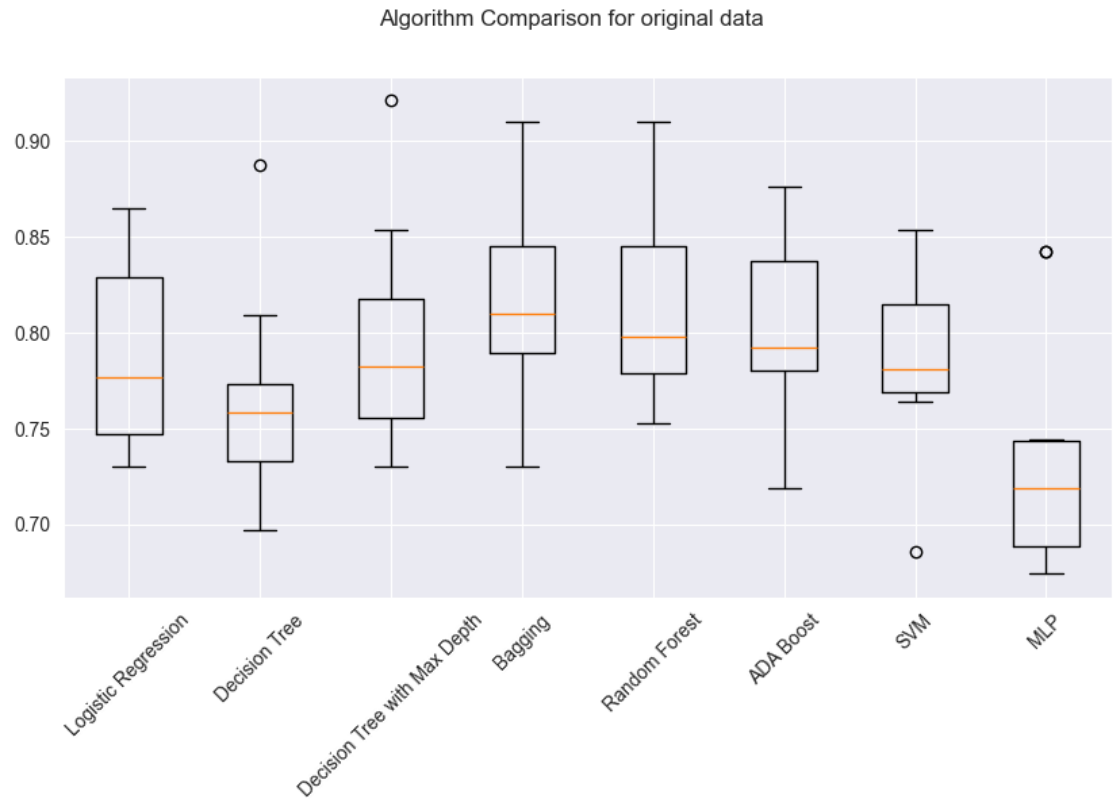
[80]:
```
# evaluate each model in turn
results = []
resultsStandDf = []
resultsNormDf = []
names = []
namesStandDf = []
namesNormDf = []
scoring = 'accuracy'

print("model: mean (std_dev)")
print("======================")
for name, mod in models:
    kfold = KFold(n_splits=10, random_state=7, shuffle = True)
    cv_results = cross_val_score(mod, X_arr, Y_arr, cv=kfold, scoring = scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)


print("\n====================================\n")

print("modelStandDf: mean (std_dev)")
print("======================")
for nameStandDf, modStandDf in modelsStandDf:
    kfold = KFold(n_splits=10, random_state=7, shuffle = True)
    cv_results_StandDf = cross_val_score(modStandDf, X_arr_stan, Y_arr_stan,␣
 ↪cv=kfold, scoring = scoring)
    resultsStandDf.append(cv_results_StandDf)
    namesStandDf.append(nameStandDf)
    msg = "%s: %f (%f)" % (nameStandDf, cv_results_StandDf.mean(),␣
 ↪cv_results_StandDf.std())
    print(msg)

print("\n====================================\n")

print("modelNormDf: mean (std_dev)")
print("======================")
for nameNormDf, modNormDf in modelsNormDf:
    kfold = KFold(n_splits=10, random_state=7, shuffle = True)
    cv_results_NormDf = cross_val_score(modNormDf, X_arr_norm, Y_arr_norm,␣
 ↪cv=kfold, scoring = scoring)
    resultsNormDf.append(cv_results_NormDf)
    namesNormDf.append(nameNormDf)
```

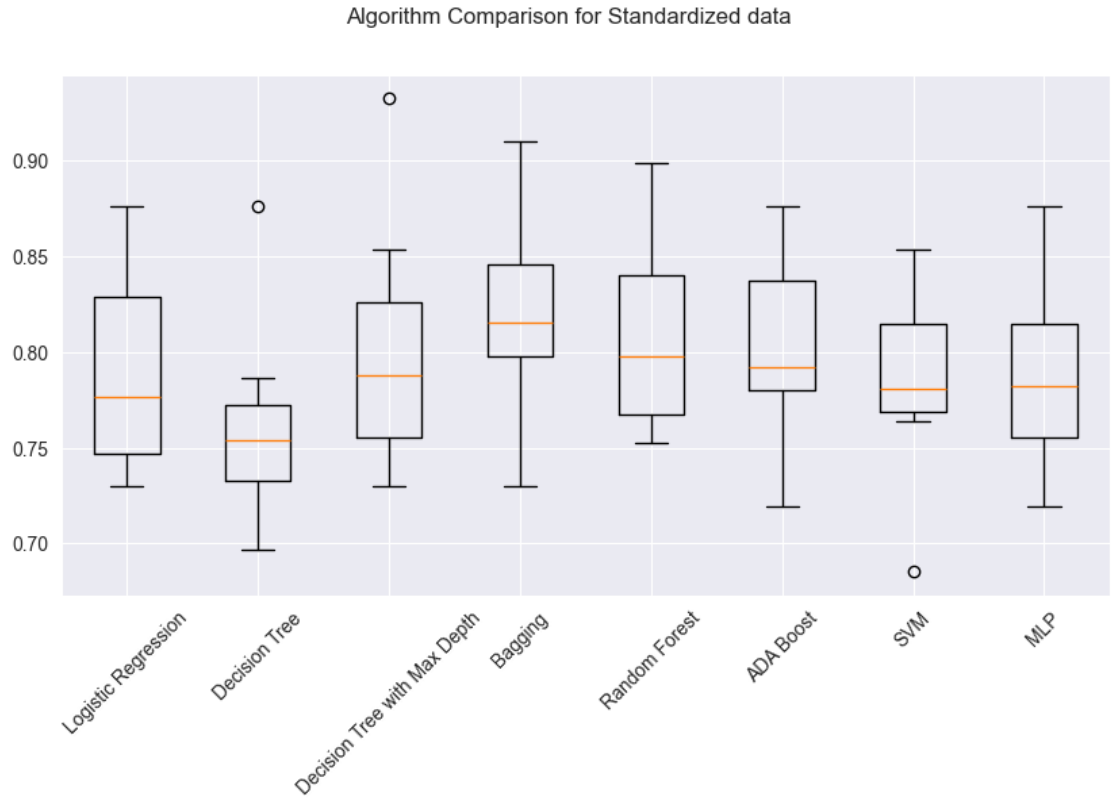
```
    msg = "%s: %f (%f)" % (nameNormDf, cv_results_NormDf.mean(),␣
↪cv_results_NormDf.std())
    print(msg)
```

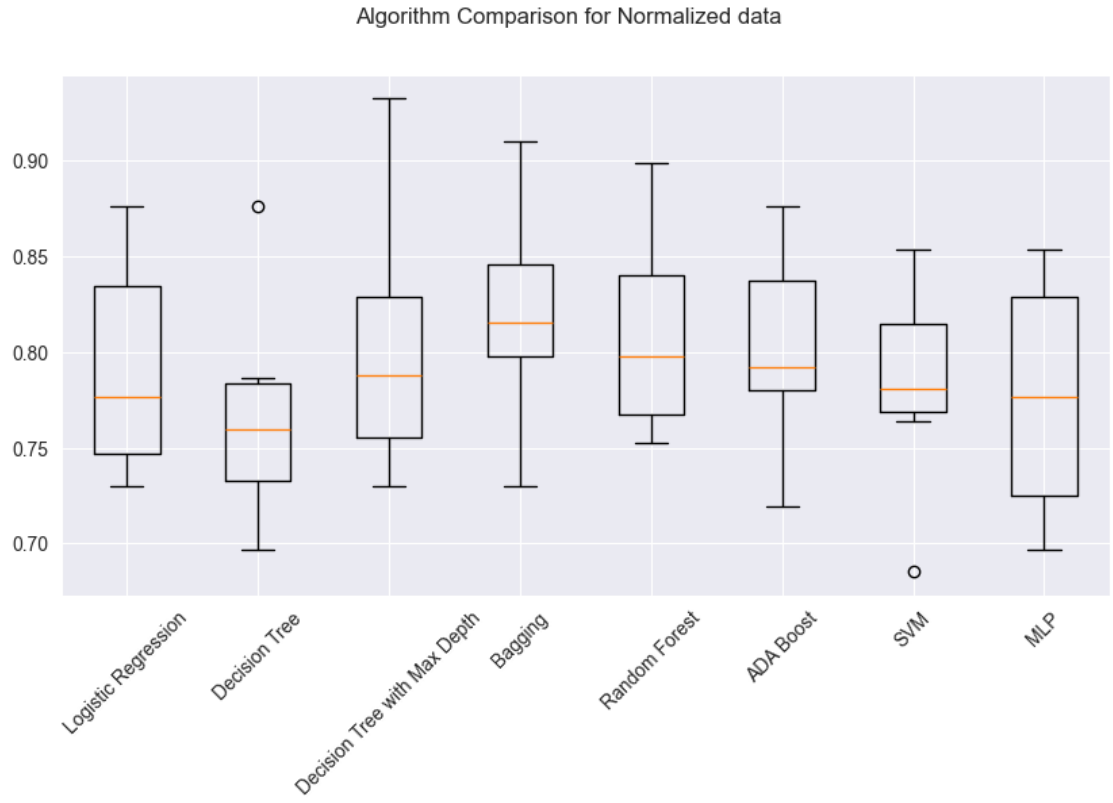
```
model: mean (std_dev)
=======================
Logistic Regression: 0.787890 (0.046660)
Decision Tree: 0.764307 (0.050755)
Decision Tree with Max Depth: 0.796854 (0.054697)
Bagging: 0.817079 (0.051912)
Random Forest: 0.812597 (0.047294)
ADA Boost: 0.802497 (0.044290)
SVM: 0.786779 (0.045100)
MLP: 0.733995 (0.059510)

====================================

modelStandDf: mean (std_dev)
=======================
Logistic Regression: 0.789014 (0.048603)
Decision Tree: 0.759825 (0.046349)
Decision Tree with Max Depth: 0.800225 (0.057243)
Bagging: 0.817066 (0.050980)
Random Forest: 0.809238 (0.046151)
ADA Boost: 0.802497 (0.044290)
SVM: 0.786779 (0.045100)
MLP: 0.790150 (0.046473)

====================================

modelNormDf: mean (std_dev)
=======================
Logistic Regression: 0.790150 (0.047812)
Decision Tree: 0.763184 (0.047003)
Decision Tree with Max Depth: 0.801348 (0.057514)
Bagging: 0.817066 (0.050980)
Random Forest: 0.809238 (0.046151)
ADA Boost: 0.802497 (0.044290)
SVM: 0.786779 (0.045100)
MLP: 0.776629 (0.054084)
```

[81]:
```
# boxplot algorithm comparison
fig = plt.figure(figsize=(10,5))
fig.suptitle('Algorithm Comparison for original data')
ax = fig.add_subplot(111)
plt.boxplot(results)
plt.xticks(rotation=45)
```

```
ax.set_xticklabels(names)

figStandDf = plt.figure(figsize=(10,5))
figStandDf.suptitle('Algorithm Comparison for Standardized data')
ax = figStandDf.add_subplot(111)
plt.boxplot(resultsStandDf)
plt.xticks(rotation=45)
ax.set_xticklabels(namesStandDf)

figNormDf = plt.figure(figsize=(10,5))
figNormDf.suptitle('Algorithm Comparison for Normalized data')
ax = figNormDf.add_subplot(111)
plt.boxplot(resultsNormDf)
plt.xticks(rotation=45)
ax.set_xticklabels(namesNormDf)

plt.show()
```




Algorithm Comparison for Standardized data
0.90
0.85
0.80
0.75
0.70
Logistic Regression
Decision Tree
Decision Tree with Max Depth
Bagging
Random Forest
ADA Boost
SVM
MLP



From the cross validation result of the raw data, the warning of going over the maximum iteration for model building and higher standard deviation overall verify that standardization and normalization helps in removing the effect of outliers in the dataset and converging the model faster. The effect of standardization and normalization is similar, and did not affect the score on the most effective models, such as bagging ensemble tree and ADA boost. This could be caused by the combination of categorical features and quantitative features. Despite standardization and normalization helps in model building, the effects are minimized with the combination of categorical data, especially when the most influential features are mostly categorical data, as described in the EDA.

4.5 Conclusions

While we were running cross validation on non-preprocessed data, we observed some warnings with regards to exceeding the maximum iteration for model building and higher standard deviation. This verifies our prediction that standardization and normalization does help in removing the effect of outliers in the dataset and converging the model faster. Judging by the box plots of the pre-processed data, standardized data and normalized data behave similarly to each other when applying all 8 different techniques, and do not affect the score on the most effective models, such as bagging ensemble tree and ADA boost. This could be caused by the combination of categorical features and quantitative features. Despite the performance gain from standardization and normalization during in model building, the effects are minimized when combined with categorical data, especially when the most influential features are mostly categorical data, as described in the EDA.

Our best performing classifier judging by the mean, median, and standard deviation is the bagging

classifier. Our bagging classifier had the best performance with a mean of 0.817 across all 3 forms of data (non-preprocessed, standardization, and normalization). The standard deviation for the bagging classifier is 0.0519 for the non-preprocessed data, 0.0510 for standardized and normalized data. Scaling the data had a minor improvement on the stratification of performance. Scaling the data seems to have slightly shifted the 25th percentile up.

Our runner-up seems to be the random forest classifier, with a mean of 0.812 for non-scaled data, and 0.809 for standardized and normalized data. When comparing the IQR for the bagging classifier and the random forest classifier, the bagging classifier has a smaller IQR, which means our performance scores are spaced pretty tightly to the mean. This indicates that there is less variance when comparing the bagging classifier model to the random forest.

Looking back, if we compare the results from our individual classifier models (decision tree, bagging, random forest, and ada boost) prior to cross validation, they all score higher than their cross validation correspondents. They drop about an average of 4 percent in accuracy when comparing all the previous models to the cross validation scores. So prior to cross validation, when we train the model on all of the training data (as opposed to cross validation k subsets), the effect of overfitting to the data may have caused our accuracy to be reported higher than what we are now seeing with cross validation.

Overall, this process has demonstrated that noise can have a significant impact when it comes to determining the performance measure of classification models, and that cross validation can be really helpful to provide a better, more complete determination of how one classification technique may perform against another. Another major factor in analyzing the performance is the need to handle missing data with techniques such as random imputation, median imputation, and mode imputation. Because of the way the missing data was handled, noise may have been amplified or decreased, since we had to assume values where there were no values to use.

4.6 References

- https://www.kaggle.com/code/abdmental01/exploring-data-visual-insights-unveiled#About-Function
- https://www.kaggle.com/code/gusthema/titanic-competition-w-tensorflow-decision-forests
- https://www.kaggle.com/code/eneskosar19/titanic-sample-submission-random-forest
- https://www.kaggle.com/code/whitedevil4648/neural-networks-titanic
- https://www.kaggle.com/code/gunesevitan/titanic-advanced-feature-engineering-tutorial
- https://www.kaggle.com/code/rushikeshlavate/handling-missing-data-in-titanic-train-dataset
- https://www.kaggle.com/code/pagenotfound/mean-and-median-imputation
- https://en.m.wikipedia.org/wiki/File:Titanic_side_plan_annotated_English.png
- https://www.nbclosangeles.com/news/national-international/history-of-the-titanic-10-questions-about-the-ill-fated-ship/3173692/#:~:text=Second%2Dclass%20tickets%20were%2012,

4.7 Appendix

https://github.com/aelichung/ml-titanic