titanic

October 31, 2020

SURVIVAL IN TITANIC PREDICTION

[0]:
```
import pandas as pd
from google.colab import drive
import seaborn as sns
%matplotlib inline
from matplotlib import pyplot as plt
from matplotlib import style
drive.mount('/content/gdrive')
#reading datasets from gdrive
train = pd.read_csv('gdrive/My Drive/Colab Notebooks/titanic/train.csv',␣
 ↪index_col = 'PassengerId')
x_test = pd.read_csv('gdrive/My Drive/Colab Notebooks/titanic/test.csv',␣
 ↪index_col = 'PassengerId')
y_test = pd.read_csv('gdrive/My Drive/Colab Notebooks/titanic/gender_submission.
 ↪csv', index_col = 'PassengerId')
test = x_test.merge(y_test, left_index=True,right_index=True,how='inner')
X =␣
 ↪train[['Pclass','Name','Sex','Age','SibSp','Parch','Ticket','Fare','Cabin','Embarked',]]
Y = train[['Survived']]

train.head()
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call
drive.mount("/content/gdrive", force_remount=True).
```

[0]:
```
             Survived  Pclass  … Cabin Embarked
PassengerId                    …
1                   0       3  …   NaN        S
2                   1       1  …   C85        C
3                   1       3  …   NaN        S
4                   1       1  …  C123        S
5                   0       3  …   NaN        S

[5 rows x 11 columns]
```

[0]:
```
import numpy as np
import matplotlib.pyplot as plt
```

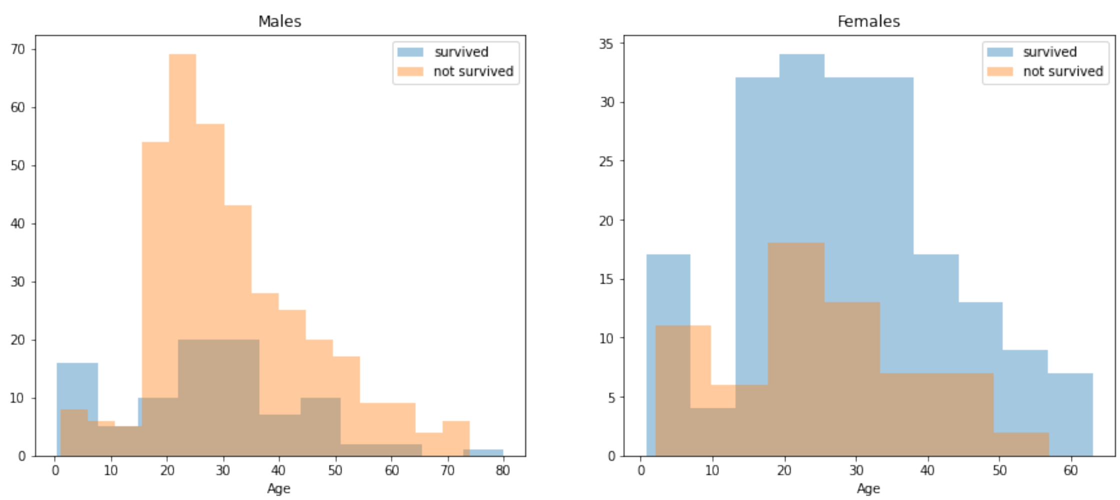
```
from astropy.visualization import hist
#checking the dataset's features for inconsistency
#getting column names
columns = train.columns.values
print(columns)
#here's table with each feature's missing values
total = train.isnull().sum().sort_values(ascending=False)
percent_1 = train.isnull().sum()/train.isnull().count()*100
percent_2 = (round(percent_1, 1)).sort_values(ascending=False)
missing_data = pd.concat([total, percent_2], axis=1, keys=['Total', '%'])
print(missing_data)

#looking at age and sex
male_survived = train[['Survived','Sex','Age']]
female_survived = train[['Survived','Sex','Age']]
#filtering for respective sex
male_survived = male_survived[male_survived.Sex == 'male']
female_survived = female_survived[female_survived.Sex == 'female']

male_alive = len(male_survived[male_survived.Survived == 1])/len(male_survived.
 ↪Survived)
female_alive = len(female_survived[female_survived.Survived == 1])/
 ↪len(female_survived.Survived)

print('males alive : ',male_alive*100,"%")
print('female alive : ',female_alive*100,"%")
#dropping rows with empty age field
male_survived = male_survived[pd.isnull(male_survived.Age)== False]
female_survived = female_survived[pd.isnull(female_survived.Age)== False]
#plotting the graph
survived = 'survived'
not_survived = 'not survived'
fig, axes = plt.subplots(nrows=1, ncols=2,figsize=(15, 6))
ax = sns.distplot(male_survived[male_survived['Survived']==1].Age, bins='auto',␣
 ↪label = survived, ax = axes[0], kde =False)
ax = sns.distplot(male_survived[male_survived['Survived']==0].Age, bins='auto',␣
 ↪label = not_survived, ax = axes[0], kde =False)
ax.legend()
ax.set_title('Males')
ax = sns.distplot(female_survived[female_survived['Survived']==1].Age,␣
 ↪bins='auto', label = survived, ax = axes[1], kde = False)
ax = sns.distplot(female_survived[female_survived['Survived']==0].Age,␣
 ↪bins='auto', label = not_survived, ax = axes[1], kde = False)
ax.legend()
ax.set_title('Females')
```

```
['Survived' 'Pclass' 'Name' 'Sex' 'Age' 'SibSp' 'Parch' 'Ticket' 'Fare'
```

```
 'Cabin' 'Embarked']
          Total     %
Cabin       687  77.1
Age         177  19.9
Embarked      2   0.2
Fare          0   0.0
Ticket        0   0.0
Parch         0   0.0
SibSp         0   0.0
Sex           0   0.0
Name          0   0.0
Pclass        0   0.0
Survived      0   0.0
males alive :  18.890814558058924 %
female alive :  74.20382165605095 %
```

[0]: Text(0.5, 1.0, 'Females')

[0]:



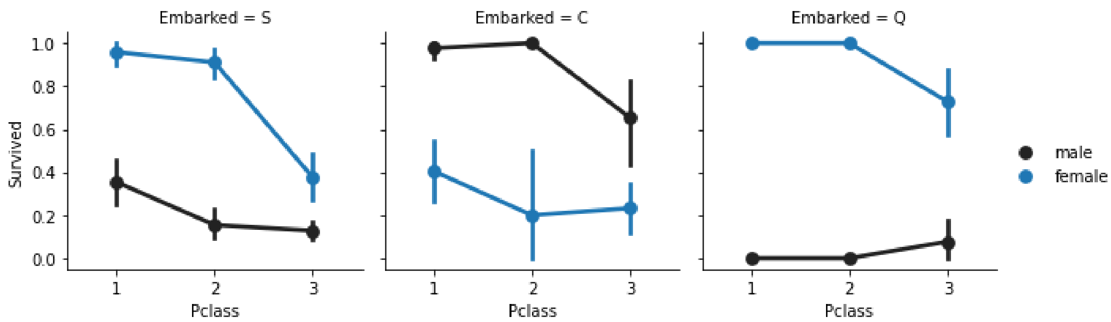
Age and sex are pretty relevant features since the data is distributed. Above data shows that males only 19% males survived. Most fatalities were among young (16-55 yrs) were the people who died mostly. In Females there wasn't much fatalities as 74% of them survived. Thus we can use these features in the training dataset.

[0]:
```
#Embarking represents which passenger entered from which gate, which could␣
↪differ based on passenger class
#checking embarking's correlation with survival rate and Passenger class
print(train.Embarked.unique())
g = sns.FacetGrid(train, col="Embarked")
g.map(sns.pointplot, 'Pclass', 'Survived', 'Sex', palette=None,  order=None,␣
↪hue_order=None)
g.add_legend()
```

```
['S' 'C' 'Q' nan]
```

[0]: <seaborn.axisgrid.FacetGrid at 0x7fc9157a84a8>

[0]:



Here we compare passenger class and survival rate based on their port of entry which is defined in the embarked class.

This data suggests that people who entered the ship from port S, among them first and second class females had a fairly high rate of survival rate. Men apart from first class passgengers barely survived.

In the second graph, people who entered the ship from port C, among them male passengers had a fairly high rate of survival rate. Woman apart from first class passgengers barely survived.

In the thrid graph, people who entered the ship from port Q, among them female passengers had a fairly high rate of survival rate. Man apart from third class passgengers barely survived.

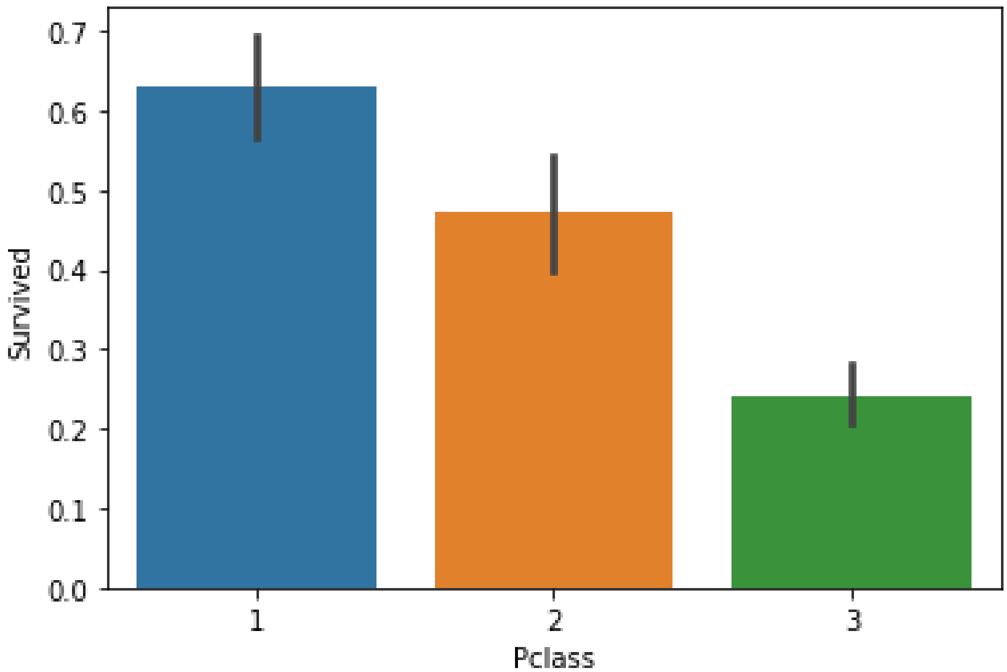
[0]:

```
print(train.Pclass.unique())

sns.barplot(x='Pclass', y='Survived', data=train)
```

```
[3 1 2]
```

[0]: <matplotlib.axes._subplots.AxesSubplot at 0x7fc9157a02e8>

[0]:

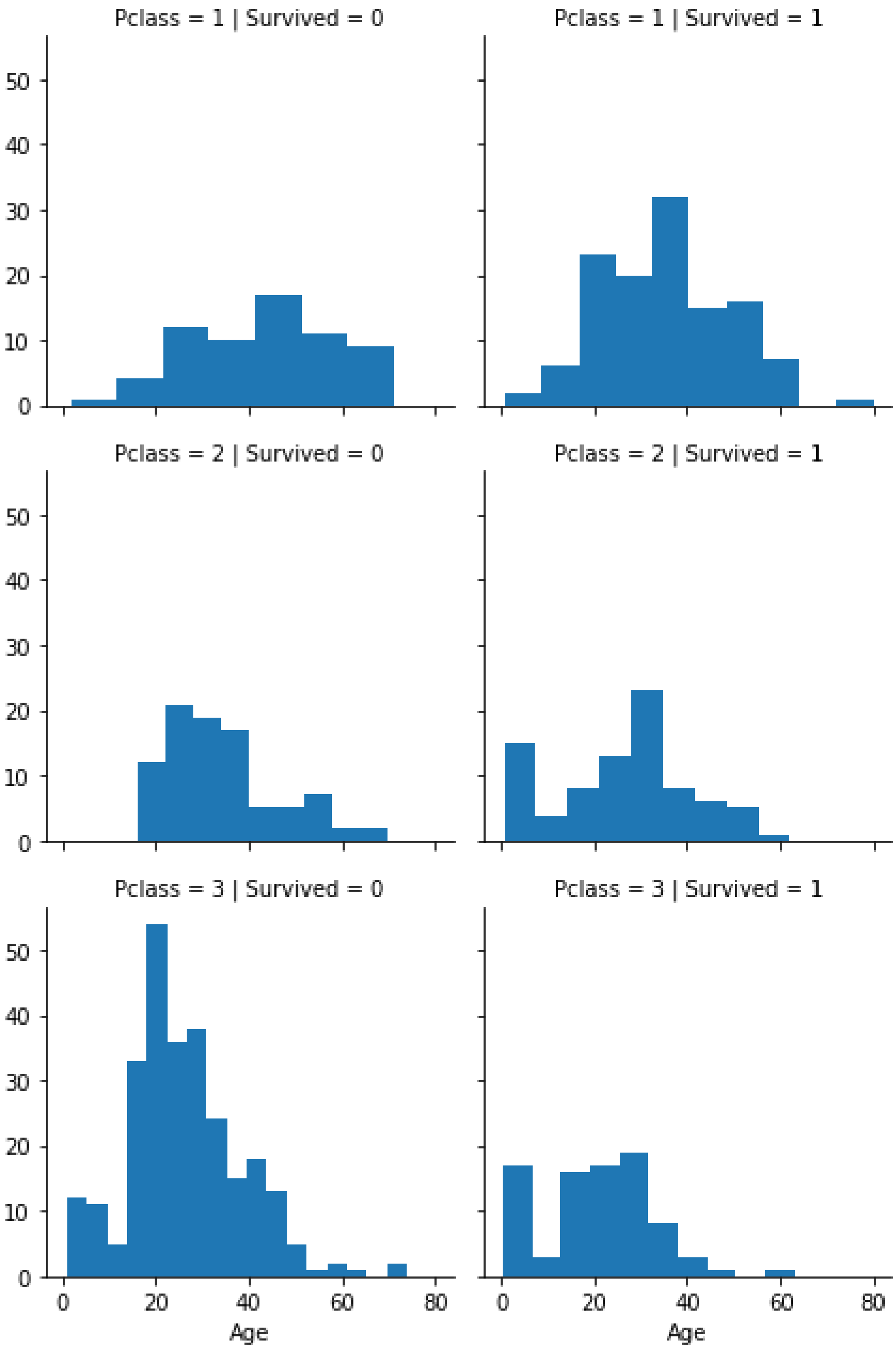


In the embarked feature test, we can see that the passenger class plays an important role in the determining the survival rate. Thus, we plot a graph to see the correlation of passenger class with survival rate.

As we can see, people with higher passenger class had higher survival rate. Thus, its safe to say that passenger class plays an important role in survival.

[0]:
```
g = sns.FacetGrid(train, col='Survived', row='Pclass')
g.map(plt.hist, 'Age', bins='auto')
g.add_legend();
```

[0]:


Pclass = 1 | Survived = 0
Pclass = 1 | Survived = 1
Pclass = 2 | Survived = 0
Pclass = 2 | Survived = 1
Pclass = 3 | Survived = 0
Pclass = 3 | Survived = 1
0
10
20
30
40
50
0
20
40
60
80
Age
Age

To further elaborate on passenger class, we relate it to age class and see who survived. Turns out maximum, fatalities were among passengers from third class and most survivers were first class passengers.

[0]:
```
print(train.Parch.unique())
print(train.SibSp.unique())
data = [train, test]
for dataset in data:
    #dataset['relatives'] = dataset['SibSp'] + dataset['Parch']
    dataset.loc[(dataset['SibSp'] + dataset['Parch']) > 0, 'not_alone'] = 1
    dataset.loc[(dataset['SibSp'] + dataset['Parch']) == 0, 'not_alone'] = 0
    dataset['not_alone'] = dataset['not_alone'].astype(int)
print(train['not_alone'].value_counts())
g = sns.factorplot((train.Parch+train.SibSp),'Survived', data=train , aspect =␣
 ↪3.5)
```

```
[0 1 2 5 3 4 6]
[1 0 3 4 2 5 8]
0    537
1    354
Name: not_alone, dtype: int64
```

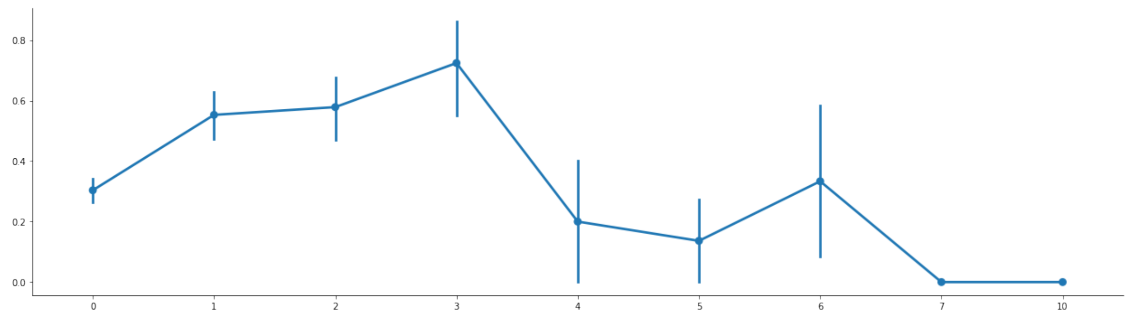
```
/usr/local/lib/python3.6/dist-packages/seaborn/categorical.py:3669: UserWarning:
The `factorplot` function has been renamed to `catplot`. The original name will
be removed in a future release. Please update your code. Note that the default
`kind` in `factorplot` (`'point'`) has changed `'strip'` in `catplot`.
  warnings.warn(msg)
/usr/local/lib/python3.6/dist-packages/pandas/core/ops/array_ops.py:253:
FutureWarning: elementwise comparison failed; returning scalar instead, but in
the future will perform elementwise comparison
  res_values = method(rvalues)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-23-1d35ca25c525> in <module>()
      8     dataset['not_alone'] = dataset['not_alone'].astype(int)
      9 print(train['not_alone'].value_counts())
---> 10 g = sns.factorplot((train.Parch+train.SibSp),'Survived', data=train ,␣
 ↪aspect = 3.5)

/usr/local/lib/python3.6/dist-packages/seaborn/categorical.py in␣
 ↪factorplot(*args, **kwargs)
   3677     kwargs.setdefault("kind", "point")
   3678
-> 3679     return catplot(*args, **kwargs)
   3680
   3681
```

```
/usr/local/lib/python3.6/dist-packages/seaborn/categorical.py in catplot(x, y, hue, data, row, col, col_wrap, estimator, ci, n_boot, units, seed, order, hue_order, row_order, col_order, kind, height, aspect, orient, color, palette, legend, legend_out, sharex, sharey, margin_titles, facet_kws, **kwargs)
   3763
   3764     # Draw the plot onto the facets
-> 3765     g.map_dataframe(plot_func, x, y, hue, **plot_kws)
   3766
   3767     # Special case axis labels for a count type plot

/usr/local/lib/python3.6/dist-packages/seaborn/axisgrid.py in map_dataframe(self, func, *args, **kwargs)
    834
    835         # Finalize the annotations and layout
--> 836         self._finalize_grid(args[:2])
    837
    838         return self

/usr/local/lib/python3.6/dist-packages/seaborn/axisgrid.py in _finalize_grid(self, axlabels)
    857     def _finalize_grid(self, axlabels):
    858         """Finalize the annotations and layout."""
--> 859         self.set_axis_labels(*axlabels)
    860         self.set_titles()
    861         self.fig.tight_layout()

/usr/local/lib/python3.6/dist-packages/seaborn/axisgrid.py in set_axis_labels(self, x_var, y_var)
    883         if x_var is not None:
    884             self._x_var = x_var
--> 885             self.set_xlabels(x_var)
    886         if y_var is not None:
    887             self._y_var = y_var

/usr/local/lib/python3.6/dist-packages/seaborn/axisgrid.py in set_xlabels(self, label, **kwargs)
    894             label = self._x_var
    895         for ax in self._bottom_axes:
--> 896             ax.set_xlabel(label, **kwargs)
    897         return self
    898

/usr/local/lib/python3.6/dist-packages/matplotlib/axes/_axes.py in set_xlabel(self, xlabel, fontdict, labelpad, **kwargs)
    246         if labelpad is not None:
    247             self.xaxis.labelpad = labelpad
--> 248         return self.xaxis.set_label_text(xlabel, fontdict, **kwargs)
```

```
    249
    250     def get_ylabel(self):

/usr/local/lib/python3.6/dist-packages/matplotlib/axis.py in␣
↪set_label_text(self, label, fontdict, **kwargs)
   1611         """
   1612         self.isDefault_label = False
-> 1613         self.label.set_text(label)
   1614         if fontdict is not None:
   1615             self.label.update(fontdict)

/usr/local/lib/python3.6/dist-packages/matplotlib/text.py in set_text(self, s)
   1163         if s is None:
   1164             s = ''
-> 1165         if s != self._text:
   1166             self._text = str(s)
   1167             self.stale = True

/usr/local/lib/python3.6/dist-packages/pandas/core/generic.py in␣
↪__nonzero__(self)
   1477     def __nonzero__(self):
   1478         raise ValueError(
-> 1479             f"The truth value of a {type(self).__name__} is ambiguous.
   1480             "Use a.empty, a.bool(), a.item(), a.any() or a.all()."
   1481         )

ValueError: The truth value of a Series is ambiguous. Use a.empty, a.bool(), a.
↪item(), a.any() or a.all().
```

[0]:



Lastly we look at Sibsp(# of siblings present on the ship) and parch(# of parent present on the ship) classes. The above graph shows that people with relatives had a higher chances. The chances seems higher when someone has 1-3 relatives on board.

[0]:
```
#Pre Processing data
#Filling the missing values
```

```
import re
#make class deck out of cabin and then drop it
deck = {"A": 1, "B": 2, "C": 3, "D": 4, "E": 5, "F": 6, "G": 7, "N": 8}
data = [train, test]

for dataset in data:
    dataset['Cabin'] = dataset['Cabin'].fillna("NO")
    dataset['Deck'] = dataset['Cabin'].map(lambda x: re.compile("([a-zA-Z]+)").
 ↪search(x).group())
    dataset['Deck'] = dataset['Deck'].map(deck)
    dataset['Deck'] = dataset['Deck'].fillna(0)
    dataset['Deck'] = dataset['Deck'].astype(int)# we can now drop the cabin␣
 ↪feature
train = train.drop(columns='Cabin')
test = test.drop(columns='Cabin')

#filling the values of Embarked with the most common ones and then mapping them␣
 ↪to their numeric values
print(train['Embarked'].describe())

train['Embarked'] = train['Embarked'].fillna('S')
test['Embarked'] = test['Embarked'].fillna('S')

ports = {"S": 0, "C": 1, "Q": 2}

train['Embarked'] = train['Embarked'].map(ports)
test['Embarked'] = test['Embarked'].map(ports)
```

```
count     889
unique      3
top         S
freq      644
Name: Embarked, dtype: object
```

[0]:

```
#filling the missing values in age
mean = train["Age"].mean()
std = test["Age"].std()
is_null = train["Age"].isnull().sum()
print(mean,std,is_null)
# compute random numbers between the mean, std and is_null
rand_age = np.random.randint(mean - std, mean + std, size = is_null)
print(rand_age)
# fill NaN values in Age column with random values generated
new_age = train["Age"].copy()
new_age[np.isnan(new_age)] = rand_age
train["Age"] = new_age
train["Age"] = train["Age"].astype(int)
```

```python
mean = train["Age"].mean()
std = test["Age"].std()
is_null = test["Age"].isnull().sum()
print(mean,std,is_null)
# compute random numbers between the mean, std and is_null
rand_age = np.random.randint(mean - std, mean + std, size = is_null)
print(rand_age)
# fill NaN values in Age column with random values generated
new_age = test["Age"].copy()
new_age[np.isnan(new_age)] = rand_age
test["Age"] = new_age
test["Age"] = test["Age"].astype(int)

print("Number of NaN's in the training dataset : ",train["Age"].isnull().sum())
print("Number of NaN's in the testing dataset : ",test["Age"].isnull().sum())

# Changing the age to categories of age

train['Age'] = train['Age'].astype(int)
train.loc[ train['Age'] <= 10, 'Age'] = 0
train.loc[(train['Age'] > 10) & (train['Age'] <= 20), 'Age'] = 1
train.loc[(train['Age'] > 20) & (train['Age'] <= 30), 'Age'] = 2
train.loc[(train['Age'] > 30) & (train['Age'] <= 40), 'Age'] = 3
train.loc[(train['Age'] > 40) & (train['Age'] <= 50), 'Age'] = 4
train.loc[(train['Age'] > 50) & (train['Age'] <= 60), 'Age'] = 5
train.loc[(train['Age'] > 60) & (train['Age'] <= 70), 'Age'] = 6
train.loc[ train['Age'] > 70, 'Age'] = 7

test['Age'] = test['Age'].astype(int)
test.loc[ test['Age'] <= 10, 'Age'] = 0
test.loc[(test['Age'] > 10) & (test['Age'] <= 20), 'Age'] = 1
test.loc[(test['Age'] > 20) & (test['Age'] <= 30), 'Age'] = 2
test.loc[(test['Age'] > 30) & (test['Age'] <= 40), 'Age'] = 3
test.loc[(test['Age'] > 40) & (test['Age'] <= 50), 'Age'] = 4
test.loc[(test['Age'] > 50) & (test['Age'] <= 60), 'Age'] = 5
test.loc[(test['Age'] > 60) & (test['Age'] <= 70), 'Age'] = 6
test.loc[ test['Age'] > 70, 'Age'] = 7
# let's see how it's distributed
print('Values distribution in training set : ',train['Age'].value_counts())
print('Values distribution in testing set : ',test['Age'].value_counts())
```

```
29.69911764705882 14.18120923562442 177
[16 27 20 24 20 35 23 22 37 17 31 34 19 40 41 28 29 40 33 21 24 28 42 35
 30 38 26 39 36 39 22 26 21 22 32 41 16 29 25 41 42 22 31 24 22 19 40 40
 40 42 27 37 27 42 29 21 31 41 33 28 27 19 28 19 38 23 15 36 34 37 33 26
 24 15 41 41 36 40 20 42 38 17 33 19 29 23 21 22 33 42 37 32 36 35 28 35
```

```
 24 28 21 35 28 28 32 28 41 17 37 15 42 26 39 40 34 16 28 19 40 22 28 24
 37 36 36 30 22 17 16 18 37 23 15 19 27 40 39 37 30 23 40 34 22 36 26 42
 27 32 29 20 31 35 18 28 38 19 30 24 19 22 26 39 37 20 21 41 21 26 22 17
 32 26 25 34 17 34 40 23 42]
29.60830527497194 14.18120923562442 86
[25 38 34 37 18 28 22 29 42 19 15 24 38 20 35 30 16 24 17 26 28 25 39 31
 22 37 20 42 21 15 27 25 32 21 36 25 26 42 18 16 40 21 27 33 28 15 42 27
 30 28 23 35 29 33 28 40 39 42 39 40 20 26 21 41 27 23 24 40 33 38 36 37
 22 39 28 42 35 16 18 27 22 15 29 30 16 18]
Number of NaN's in the training dataset :  0
Number of NaN's in the testing dataset :  0
Values distribution in training set :  2    299
3    217
1    146
4    101
0     64
5     42
6     18
7      4
Name: Age, dtype: int64
Values distribution in testing set :  2    168
3     81
1     64
4     52
0     22
5     21
6      9
7      1
Name: Age, dtype: int64
```

[0]:

```
train['Fare'] = train['Fare'].fillna(0)
test['Fare'] = test['Fare'].fillna(0)
train['Fare'] = train['Fare'].astype(int)
test['Fare'] = test['Fare'].astype(int)
print("Number of NaN's in the training dataset : ",train["Fare"].isnull().sum())
print("Number of NaN's in the testing dataset : ",test["Fare"].isnull().sum())

train.loc[ train['Fare'] <= 7.91, 'Fare'] = 0
train.loc[(train['Fare'] > 7.91) & (train['Fare'] <= 14.454), 'Fare'] = 1
train.loc[(train['Fare'] > 14.454) & (train['Fare'] <= 31), 'Fare']   = 2
train.loc[(train['Fare'] > 31) & (train['Fare'] <= 99), 'Fare']   = 3
train.loc[(train['Fare'] > 99) & (train['Fare'] <= 250), 'Fare']   = 4
train.loc[ train['Fare'] > 250, 'Fare'] = 5
train['Fare'] = train['Fare'].astype(int)

test.loc[ test['Fare'] <= 7.91, 'Fare'] = 0
test.loc[(test['Fare'] > 7.91) & (test['Fare'] <= 14.454), 'Fare'] = 1
```

```
test.loc[(test['Fare'] > 14.454) & (test['Fare'] <= 31), 'Fare']   = 2
test.loc[(test['Fare'] > 31) & (test['Fare'] <= 99), 'Fare']   = 3
test.loc[(test['Fare'] > 99) & (test['Fare'] <= 250), 'Fare']   = 4
test.loc[ test['Fare'] > 250, 'Fare'] = 5
test['Fare'] = test['Fare'].astype(int)
```

```
Number of NaN's in the training dataset :  0
Number of NaN's in the testing dataset :  0
```

[0]:

```
#extracting titles out of the names and ranking them
titles = {"Mr": 1, "Miss": 2, "Mrs": 3, "Master": 4, "Rare": 5}
# extract titles
train['Title'] = train.Name.str.extract(' ([A-Za-z]+)\.', expand=False)
print(train.Title.unique())
# replace titles with a more common title or as Rare
train['Title'] = train['Title'].replace(['Lady', 'Countess','Capt',␣
↪'Col','Don', 'Dr','Major', 'Rev', 'Sir', 'Jonkheer', 'Dona'], 'Rare')
train['Title'] = train['Title'].replace('Mlle', 'Miss')
train['Title'] = train['Title'].replace('Ms', 'Miss')
train['Title'] = train['Title'].replace('Mme', 'Mrs')
# convert titles into numbers
train['Title'] = train['Title'].map(titles)
# filling NaN with 0, to get safe
train['Title'] = train['Title'].fillna(0)

# extract titles
test['Title'] = test.Name.str.extract(' ([A-Za-z]+)\.', expand=False)
print(test.Title.unique())
# replace titles with a more common title or as Rare
test['Title'] = test['Title'].replace(['Lady', 'Countess','Capt', 'Col','Don',␣
↪'Dr','Major', 'Rev', 'Sir', 'Jonkheer', 'Dona'], 'Rare')
test['Title'] = test['Title'].replace('Mlle', 'Miss')
test['Title'] = test['Title'].replace('Ms', 'Miss')
test['Title'] = test['Title'].replace('Mme', 'Mrs')
# convert titles into numbers
test['Title'] = test['Title'].map(titles)
# filling NaN with 0, to get safe
test['Title'] = test['Title'].fillna(0)

train = train.drop(columns='Name')
test = test.drop(columns='Name')
```

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

[0]:
```python
#changing gender field
genders = {"male": 0, "female": 1}

train['Sex'] = train['Sex'].map(genders)
test['Sex'] = test['Sex'].map(genders)
```

[0]:
```python
#ticket id is unique and isnt useful while predicting thus we drop it
train = train.drop(columns='Ticket')
test = test.drop(columns = 'Ticket')
```

[0]:
```python
print('Train sets')
print(train.columns.values)
print(test.columns.values)
Y_TRAIN = train[['Survived']]
#Y_TRAIN.reset_index(drop = True)
X_TRAIN = train
X_TRAIN = X_TRAIN.drop(['Survived'],axis =1)
print(X_TRAIN.columns.values)
print(Y_TRAIN.columns.values)

print('Test sets')
print(test.columns.values)
Y_Test = train[['Survived']]
X_Test = train
X_Test = X_Test.drop(['Survived'],axis =1)
print(X_Test.columns.values)
print(Y_Test.columns.values)
train.head()
```

```
Train sets
['Survived' 'Pclass' 'Sex' 'Age' 'SibSp' 'Parch' 'Fare' 'Embarked' 'Deck'
 'Title']
['Pclass' 'Sex' 'Age' 'SibSp' 'Parch' 'Fare' 'Embarked' 'Survived' 'Deck'
 'Title']
['Pclass' 'Sex' 'Age' 'SibSp' 'Parch' 'Fare' 'Embarked' 'Deck' 'Title']
['Survived']
Test sets
['Pclass' 'Sex' 'Age' 'SibSp' 'Parch' 'Fare' 'Embarked' 'Survived' 'Deck'
 'Title']
['Pclass' 'Sex' 'Age' 'SibSp' 'Parch' 'Fare' 'Embarked' 'Deck' 'Title']
['Survived']
```

[0]:

	Survived	Pclass	Sex	Age	…	Fare	Embarked	Deck	Title
PassengerId					…				
1	0	3	0	2	…	0	0	8	1
2	1	1	1	3	…	3	1	3	3
3	1	3	1	2	…	0	0	8	2

4	1	1	1	3	…	3	0	3	3
5	0	3	0	3	…	1	0	8	1

[5 rows x 10 columns]

[0]:
```
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.ensemble import BaggingClassifier


#training the algorithms before dropping the least important class
clf = RandomForestClassifier(n_estimators=1000, max_depth=None,␣
↪min_samples_split=2)
clf = clf.fit(X_TRAIN, Y_TRAIN.values.ravel())
print('before')
accuracy_clf_before = accuracy_score(Y_Test,clf.predict(X_Test))
print('accuracy random forest before : ',accuracy_clf_before)

clf2 = ExtraTreesClassifier(n_estimators=1000, max_depth=None, random_state=0)
clf2.fit(X_TRAIN,Y_TRAIN.values.ravel())
accuracy_extra_tree = accuracy_score(Y_Test,clf2.predict(X_Test))
print('accuracy extra tree before : ',accuracy_extra_tree)

#plotting a graph to visualize each feature's importance
feature_imp = pd.DataFrame({'feature':X_TRAIN.columns,'importance':np.round(clf.
↪feature_importances_,3)})
feature_imp =  feature_imp.sort_values('importance',ascending=False).
↪set_index('feature')
feature_imp.plot.bar()

#dropping the least important class
X_TRAIN = X_TRAIN.drop(columns='Parch')
X_Test = X_Test.drop(columns='Parch')

#retraining the algorithms to compare
clf = clf.fit(X_TRAIN, Y_TRAIN.values.ravel())
print('after removing parch')
accuracy_clf_after = accuracy_score(Y_Test,clf.predict(X_Test))
print('accuracy random forest after : ',accuracy_clf_after)

scores_random_forest = cross_val_score(clf, X_TRAIN, Y_TRAIN.values.ravel(),␣
↪cv=10)
print(' random forest cross val score: ',scores_random_forest.mean())
```

```
scores_extra_tree = cross_val_score(ExtraTreesClassifier(n_estimators=1000),␣
↪X_TRAIN, Y_TRAIN.values.ravel(), cv=10)
print(' extra tree cross val score : ',scores_extra_tree.mean())

clf2.fit(X_TRAIN,Y_TRAIN.values.ravel())
accuracy_extra_tree = accuracy_score(Y_Test,clf2.predict(X_Test))
print('accuracy extra tree after : ',accuracy_extra_tree)
```

```
before
accuracy random forest before :  0.920314253647587
accuracy extra tree before :  0.920314253647587
after removing parch
accuracy random forest after :  0.9180695847362514
 random forest cross val score:  0.8485018726591761
 extra tree cross val score :  0.8439950062421973
accuracy extra tree after :  0.9180695847362514
```

[0]: