Titanic Machine Learning Project

The sinking of the Titanic in 1912 was a tragedy that resulted in the loss of many lives. The collision with an iceberg sank the "unsinkable" ship and resulted in the death of 1502 out of 2224 passengers and crew. In this project I am tasked with analyzing available data on the passengers, such as their name, age, gender, socio-economic class, and fare to potentially provide insights into the factors that influenced survival. For this project I will be utilizing the power of machine learning, Random Forest Classifier and XGBoost algorithms, to predict what sort of people were more likely to survive.

Loading The Data

In this project, I am given 2 datasets with passenger information. The dataset **train.csv** comprises of data on 891 passengers and includes the target variable "Survived" which indicates whether or not a passenger survived. The dataset **test.csv** on the otherhand contains data on a seperate 418 passengers, but does not contain a variable that indicates the fate of the passenger. I will analyze and use Machine Learning techniques on the **train.csv** data in order to create a model. This model will then be used to predict the outcome of those passengers in the **test.csv** dataset. This project is a "competition project" on the Kaggle site, so I will never know what passengers in the **test.csv** dataset survived, instead I will submit my guess on which 418 passengers survived using my model and will recieve a score based on how accurate I was.

I will begin by importing packages that I will use throughout this project. I will also import the datasets from the Kaggle site.

In [1]:
```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.metrics import mean_absolute_error

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

```
/kaggle/input/titanic/train.csv
/kaggle/input/titanic/test.csv
/kaggle/input/titanic/gender_submission.csv
```

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

Out[2]:

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

In [3]:
```
test_data = pd.read_csv("/kaggle/input/titanic/test.csv")
test_data.head()
```

Out[3]:

	PassengerId	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	892	3	Kelly, Mr. James	male	34.5	0	0	330911	7.8292	NaN	Q
1	893	3	Wilkes, Mrs. James (Ellen Needs)	female	47.0	1	0	363272	7.0000	NaN	S
2	894	2	Myles, Mr. Thomas Francis	male	62.0	0	0	240276	9.6875	NaN	Q
3	895	3	Wirz, Mr. Albert	male	27.0	0	0	315154	8.6625	NaN	S
4	896	3	Hirvonen, Mrs. Alexander (Helga E Lindqvist)	female	22.0	1	1	3101298	12.2875	NaN	S

Here I can see that both datasets contain explanitory variables such as, PassengerId, Pclass, Name, Sex, Age, SibSp, Parch, Ticket, Fare, Cabin, and Embarked. The **train_data** also contains the "Survived" variable, which will obviously be used as the target variable. The variable appears to be categorical, that is a 1 if the passenger survived and a 0 if they did not survive.

I will now take a closer look at the variables using the descibe method.

In [4]:
```
train_data.describe()
```

Out[4]:

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

In [5]:
```
test_data.describe()
```

Out[5]:

	PassengerId	Pclass	Age	SibSp	Parch	Fare
count	418.000000	418.000000	332.000000	418.000000	418.000000	417.000000
mean	1100.500000	2.265550	30.272590	0.447368	0.392344	35.627188
std	120.810458	0.841838	14.181209	0.896760	0.981429	55.907576
min	892.000000	1.000000	0.170000	0.000000	0.000000	0.000000
25%	996.250000	1.000000	21.000000	0.000000	0.000000	7.895800
50%	1100.500000	3.000000	27.000000	0.000000	0.000000	14.454200
75%	1204.750000	3.000000	39.000000	1.000000	0.000000	31.500000
max	1309.000000	3.000000	76.000000	8.000000	9.000000	512.329200

There are a couple of things this shows:

- The count for the Age is different for both datasets indicating that some passengers are missing an Age in their data. (Also the case for 1 passenger in the **test_data** with the Fare variable.)
- Name, Sex, Ticket, Cabin, and Embarked are not included in the result, since they are categorical variables.
- In general all the variables tend to have similar numbers across both datasets hopefully indicating that the data was properly/randomly split when separating the **train** and **test**

datasets.

Split the Data

Next I will split the **train_data** into 4 groups using the train_test_split method. (Using 80% train, 20% Validation)

- X_train: 80% of the data with all variables except the target variable. Will be used to build the model(s).
- X_valid: 20% of the data with all variables except the target variable. Will be used to test the accuracy of model(s), and check results of adjustments against the model.
- y_train: The same 80% of the data as the X_train set, but only comprised of the target "Survived" variable.
- y_test: The same 20% of the data as the X_test set, but only comprised of the target "Survived" variable.

I need to split the data since like I mentioned before I will never know the Survived variable of the **test_data**. Thus I cannot use that data to assess the accuracy of my models and determine what proper adjustments to make against it.

In [6]:
```
# Create a copy of the train_data that I will make adjustments against.
X = train_data

# Separate the target or prediction variable from the explanatory variables.
y = X.Survived
X.drop(['Survived'], axis=1, inplace=True)

# Split the taining data set and validation set from training data
X_train, X_valid, y_train, y_valid = train_test_split(X, y, train_size=0.8, test_size=
                                                      random_state=0)
```

Assess and Clean Variables in the Data

Missing Data

Next I have to handle the missing data points. There are a couple of approaches to handle missing values in the data. In this case I will be using simple imputation to fill in the missing values. For this project I will be calculating the missing values using the "constant" strategy of the SimpleImputer method.

In [7]:
```
# Select numerical columns to impute against
numerical_cols = [cname for cname in X_train.columns if
                  X_train[cname].dtype in ['int64', 'float64']]

# Preprocessing for numerical data
numerical_transformer = SimpleImputer(strategy = "constant")
```

Categorical Data

As I mentioned earlier, many of the explanatory variables are categorical. There are also a couple of ways to handle categorical variables, but here I will be using One-Hot Encoding which creates new columns in the data indicating the presence (or absence) of each possible value of the categorical data. Since this typically does not perform well with categorical variables with a large number of differing values, I will find the cardinality of the categorical variables. I will then use One-Hot Encoding on those with less than 10 cardinality, and drop the variables with 10 or more cardinality from the data.

In [8]:
```
# "Cardinality" means the number of unique values in a column
# Select categorical columns with relatively low cardinality.
categorical_cols = [cname for cname in X_train.columns if
                    X_train[cname].nunique() < 10 and
                    X_train[cname].dtype == "object"]

# Using Pipeline to help preprocess the categorical data both Imputing and One-Hot Enc
categorical_transformer = Pipeline(steps=[
    ("imputer",SimpleImputer(strategy = "constant")),
    ("one hot",OneHotEncoder(handle_unknown = "ignore"))])
```

Selecting Varaibles For Model

Now based on how I handled the data in the previous section, I will select which columns to use for the model to predict. Keeping only the selected variables for each of the 3 X datasets. (X_train, X_valid, X_test)

In [9]:
```
# Keep selected columns only
my_cols = categorical_cols + numerical_cols

X_train_mycol = X_train[my_cols].copy()
X_valid_mycol = X_valid[my_cols].copy()
X_test_mycol = test_data[my_cols].copy()
```

In [10]:
```
# Display the first 5 rows of the valid data to show which columns I am keeping.
X_valid_mycol.head()
```

Out[10]:

	Sex	Embarked	PassengerId	Pclass	Age	SibSp	Parch	Fare
495	male	C	496	3	NaN	0	0	14.4583
648	male	S	649	3	NaN	0	0	7.5500
278	male	Q	279	3	7.0	4	1	29.1250
31	female	C	32	1	NaN	1	0	146.5208
255	female	C	256	3	29.0	0	2	15.2458

In [11]:
```
# Bundle preprocessing for numerical and categorical data
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numerical_transformer, numerical_cols),
        ('cat', categorical_transformer, categorical_cols)
    ])
```

Creating The Models

For this project I will using both Random Forest Classification and XGBoost (gradient boosting) to create models.

The Random Forest Classifier works by constructing a collection of decision trees. Data is inputted into each decision tree in the forest, and each tree independently makes a prediction. The final prediction is then determined by combining the predictions of all the trees, then deciding the outcome by majority voting.

Gradient boosting works by going through cycles to iteratively add models into an ensemble. It begins by initializing the ensemble with a single model, whose predictions can be pretty naive. It then iteratively trains new models to correct the errors made by previous models in the ensemble. The errors or residuals from the previous models are used as targets for the next model to improve upon. This process continues iteratively, with each new model trying to minimize the residual errors of the ensemble.

I will create 5 models for using each technique with differing parameters.

In [12]:
```
# Define the Random Forsest models for different n_estimators.
model_RF1 = RandomForestClassifier(n_estimators = 100, random_state = 0)
model_RF2 = RandomForestClassifier(n_estimators = 200, random_state = 0)
model_RF3 = RandomForestClassifier(n_estimators = 300, random_state = 0)
model_RF4 = RandomForestClassifier(n_estimators = 400,  random_state = 0)
model_RF5 = RandomForestClassifier(n_estimators = 500, random_state = 0)

# Define the XGBoost models for different n_estimators.
model_XGB1 = XGBClassifier(n_estimators = 100, learning_rate = 0.05,
                                                    random_state = 0)
model_XGB2 = XGBClassifier(n_estimators = 200, learning_rate = 0.05,
                                                    random_state = 0)
model_XGB3 = XGBClassifier(n_estimators = 300, learning_rate = 0.05,
                                                    random_state = 0)
model_XGB4 = XGBClassifier(n_estimators = 400, learning_rate = 0.05,
                                                    random_state = 0)
model_XGB5 = XGBClassifier(n_estimators = 500, learning_rate = 0.05,
                                                    random_state = 0)
```

Model Testing

Next I will create a score_model function that uses mean absolute error between the predictions based on the X-valid data (calculated using the model trained on the X_train_mycol and y_train

data) and the y_valid data. The model that gives the lowest mean absolute error should be considered the best model.

In [13]:
```
def score_model(model, X_t=X_train_mycol, X_v=X_valid_mycol,
                                        y_t=y_train, y_v=y_valid):
    my_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                                  ('model', model)
                                 ])
    my_pipeline.fit(X_t, y_t)
    preds = my_pipeline.predict(X_v)
    return mean_absolute_error(y_v, preds)

#create set of all the combined models and then iterate
#through them calculating their mean absolute error.
models = [model_RF1, model_RF2, model_RF3, model_RF4, model_RF5,
                model_XGB1, model_XGB2, model_XGB3, model_XGB4, model_XGB5]
for i in range(0, len(models)):
    mae = score_model(models[i])
    print("Model " + str(i+1) + " MAE: " + str(mae))
```

```
Model 1 MAE: 0.13966480446927373
Model 2 MAE: 0.13966480446927373
Model 3 MAE: 0.1340782122905028
Model 4 MAE: 0.12849162011173185
Model 5 MAE: 0.12849162011173185
Model 6 MAE: 0.16201117318435754
Model 7 MAE: 0.1452513966480447
Model 8 MAE: 0.13966480446927373
Model 9 MAE: 0.13966480446927373
Model 10 MAE: 0.1452513966480447
```

It appears **model_RF4** should be considered the best and final model as it is tied for having the lowest mean absolute error with more complex models. When breaking ties between models it is usually best to favor the least complex model.

Creating and Fitting the Final Model

I will now create the final model, once again calculate its mean absolute value under its new final model name, and then use the **X_test** data to predict the survival fate of all the passengers in the **X_test** dataset. An **output** dataset consisting of the PassengerIds and predicted Survial values will be created and submitted to the Kaggle compition.

In [14]:
```
#Creating the Final Model.
model_final = RandomForestClassifier(n_estimators = 400,  random_state = 0)
```

In [15]:
```
# Bundle preprocessing and modeling code in a pipeline
my_pipeline = Pipeline(steps=[('preprocessor', preprocessor),
                              ('model', model_final)])

# Preprocessing of training data, fit model
my_pipeline.fit(X_train_mycol, y_train)

# Preprocessing of validation data, get predictions
preds = my_pipeline.predict(X_valid_mycol)
```

```
# Evaluating the model under the new "final model" name
score = mean_absolute_error(y_valid, preds)
print('MAE:', score)
```

```
MAE: 0.12849162011173185
```

In [16]:
```
# Preprocessing of test data, fit model using the X_test_mycol dataset
preds_test = my_pipeline.predict(X_test_mycol)
```

In [17]:
```
#Create the output dataset and take a quick look at it.
output = pd.DataFrame({'PassengerId': test_data.PassengerId, 'Survived': preds_test})
output.head()
```

Out[17]:

	PassengerId	Survived
0	892	0
1	893	0
2	894	0
3	895	0
4	896	0

In [18]:
```
#Submit the output dataset.
output.to_csv('submission.csv', index=False)
print("Submission was successfully saved!")
```

```
Submission was successfully saved!
```

Result

In conclusion, after submitting the output, the result is that the final model was able to correctly identify the fate of 75.94% of the passengers within the **test** dataset.