PROJECT

CS6375: Machine Learning.

Learning from disaster: Predicting survival on the Titanic using Excel, Python, R & Random Forests

FALL 2016
Instructor: Dr. ANURAG NAGAR

Submitted By,

Devendra Lad(DVL160030)

Ian Laurain(ICL140030)

Raghunandan Ramesha(RXN150230)

Saptharsh R Heggade (SRH140130)

<u>Table of Contents</u>

1. Introduction

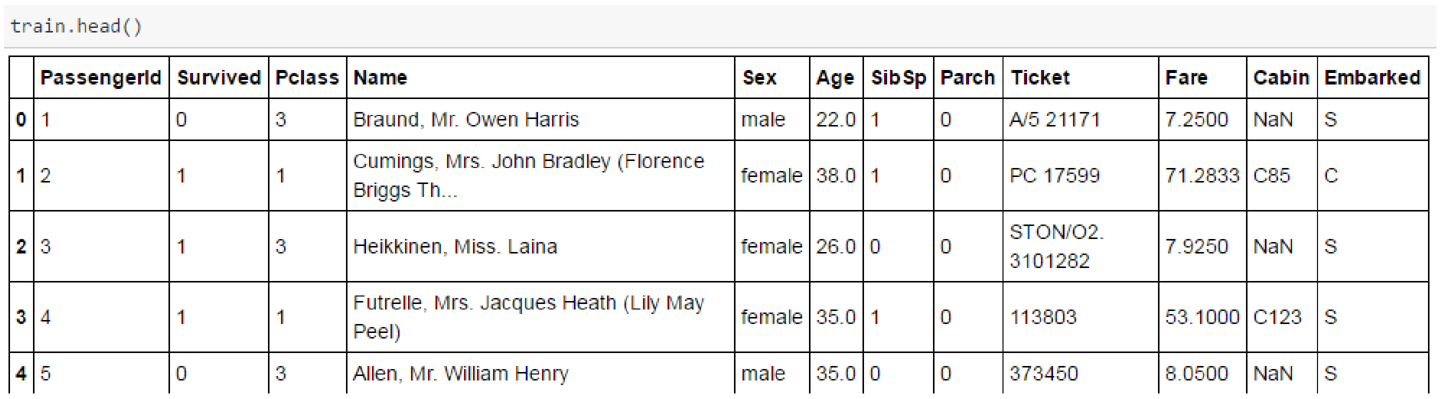
The tragedy involving the RMS Titanic is one of the most outrageous events in the history of shipping. This tragedy witnessed killing of 1502 out of 2224 including passengers and crew members. Out of the total number of people survived, the majority were women, children, and the upper-class. The lack in the number of lifeboats led to the killing of 1522 people. Although there was a strong possibility of luck playing a role in surviving some people had a better chance of surviving than the others. For instance, women, children and people who bought higher class tickets could have had a better chance of surviving and getting on the life boats than the others.

2. Problem Statement

The project uses the machine learning techniques to predict the classes of the people who are more likely to survive and also the passengers who survived the tragedy based on given information. **The accuracy of the prediction is recorded in the scale 0 to 1.** The close the accuracy to 1, the better is the prediction from the classifier. Our aim was to use a random forest classifier for this purpose. However, along with the Random Forest we have implemented 7 other classifiers in this project.

3. Dataset Distribution

Sample Training Dataset:
The training dataset comprises of 891 instances with 12 attributes.
Below is the snapshot of training dataset:

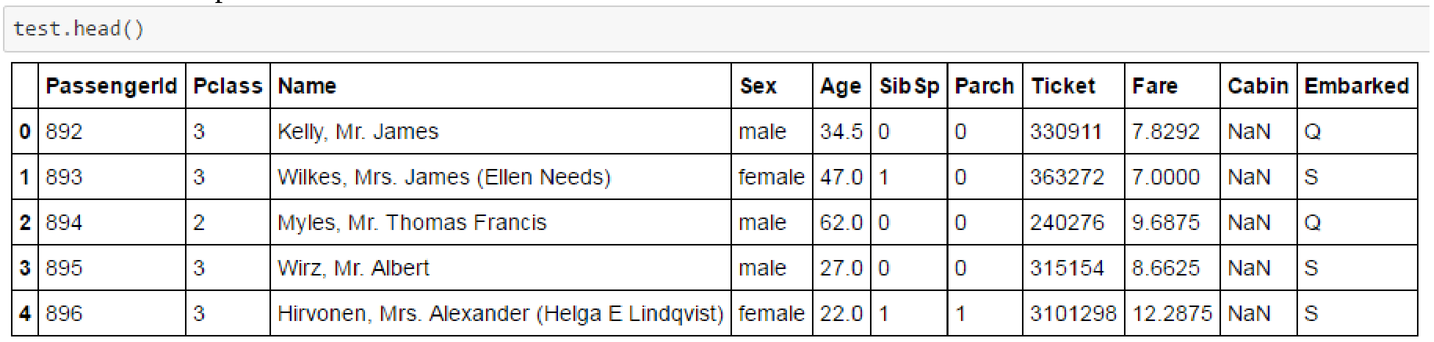


```
train.head()
```

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

Sample Testing Dataset:

Test dataset comprises of 418 instances with 11 attributes.



```
test.head()
```

	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

ATTRIBUTE DESCRIPTIONS:

survival	(0 = No; 1 = Yes)
pclass	Passenger Class (1 = 1st; 2 = 2nd; 3 = 3rd)
	1st ~ Upper; 2nd ~ Middle; 3rd ~ Lower
sibsp	Siblings or Spouses Aboard
parch	Parents or Children Aboard
	(parch=0, for the children who travelled only with a nanny)
embarked	Embarkation Port (C = Cherbourg; Q = Queenstown; S = Southampton)

Definitions Used:

Sibling:	Stepsister of Passenger Aboard Titanic or Brother, Sister, Stepbrother
Spouse:	Wife of Passenger Aboard Titanic or Husband
Parent:	Father of Passenger Aboard Titanic or Mother
Child:	Stepdaughter of Passenger Aboard Titanic or Son, Daughter, Stepson

(Some people travelled with close friends or neighbors, the definitions do not support such relations.)

4. Pre-Processing Techniques

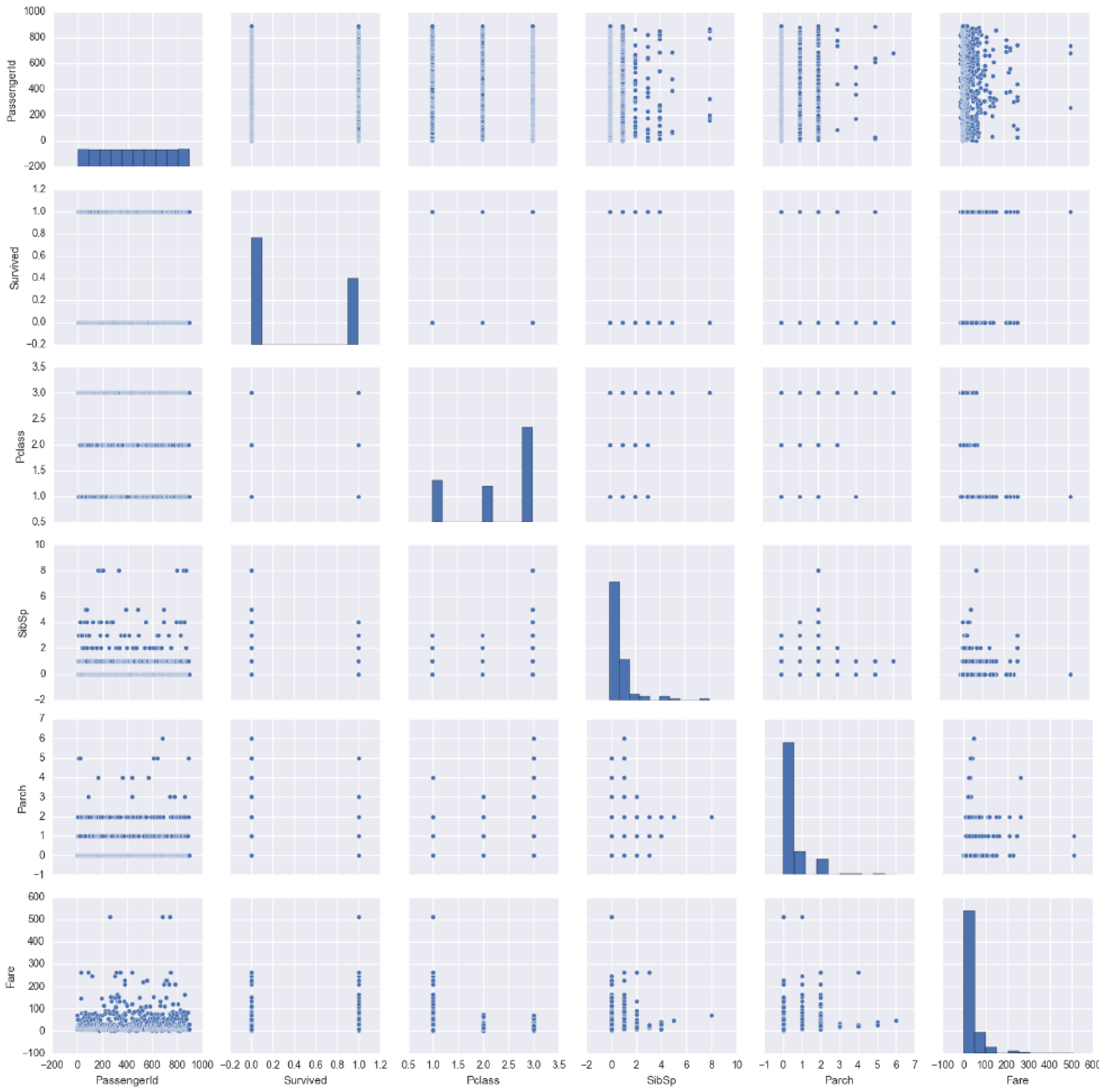
The NaN values in the training dataset was found as below. The attributes such as “Age”, “Cabin” and “Embarked” has impurities.

```
# see which columns have NaN values
print(train.isnull().sum())
```

```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age            177
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

Below, pair-wise plot is shown which is run on the training dataset. This helps in visualizing each data in the dataset by drawing a grid of small subplots.
We have dropped the attributes with NaN values for now in order to see the data distribution.

```
sns.pairplot(train.drop(['Age', 'Cabin', 'Embarked'], axis=1))
sns.plt.show()
```



<u>Handling the missing/NaN values in the dataset:</u>

<u>Age Column:</u>
We have replaced the missing values in the "age" attribute with the most occurred age value in the dataset.

```
# look at age values to get an idea for value imputation
train.Age.value_counts()
```

```
24.00    30
22.00    27
18.00    26
19.00    25
30.00    25
28.00    25
21.00    24
25.00    23
36.00    22
29.00    20
32.00    18
27.00    18
35.00    18
26.00    18
16.00    17
31.00    17
20.00    15
33.00    15
```

From the above result, looks like Age: 24 is the most frequently occurred value in the dataset.

<u>Imputation Strategy:</u>
The below process replaces the missing age values of the passengers with the most frequently occurred value in the age column

```
ages = train.Age
```

```
# initialize imputer to replace missing age values using the most frequent strategy
imputer = Imputer(missing_values='NaN', strategy='most_frequent', axis=0)
```

```
ages = imputer.fit_transform(ages.values.reshape(-1, 1))
```

```
ages
```

```
array([[ 22.  ],
       [ 38.  ],
       [ 26.  ],
       [ 35.  ],
       [ 35.  ],
       [ 24.  ],
       [ 54.  ],
       [  2.  ],
       [ 27.  ],
       [ 14.  ],
       [  4.  ],
       [ 58.  ],
       [ 20.  ],
       [ 39.  ],
       [ 14.  ],
       [ 55.  ],
       [  2.  ],
       [ 24.  ],
       [ 31.  ],
       [ 24.  ],
       [ 35.  ],
       [ 34.  ],
       [ 15.  ],
       [ 28.  ],
       [  8.  ],
       [ 38.  ],
       [ 24.  ],
       [ 19.  ],
       [ 24.  ],
       [ 24.  ],
```

As a result of imputation, now you can see there are no NaN values under the "Age" column of the training dataset.

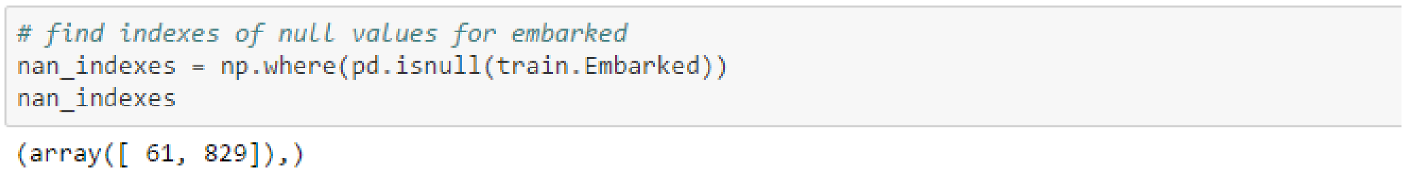
```
train.Age = ages
```

```
# see which columns now have NaN values
print(train.isnull().sum())
```

```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age              0
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         2
dtype: int64
```

<u>Embarked Column:</u>
The Strategy for handling the missing 2 values under the "Embarked" column is as explained below:

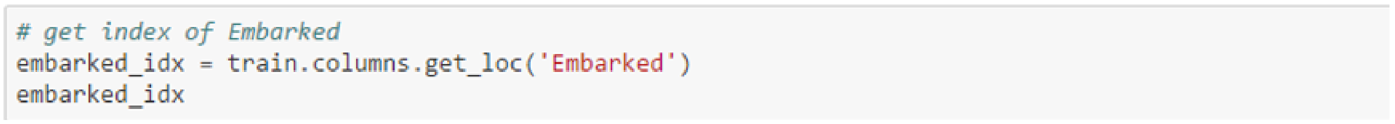
Get the Indexes of the null values,



```
# find indexes of null values for embarked
nan_indexes = np.where(pd.isnull(train.Embarked))
nan_indexes
```

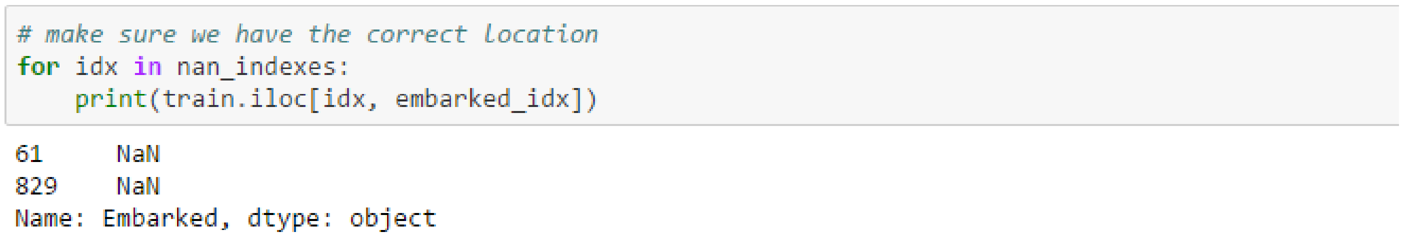
```
(array([ 61, 829]),)
```

Get the Index of the "Embarked" column,



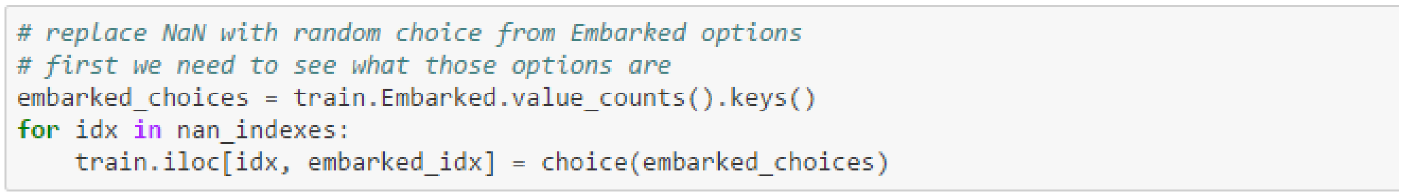
```
# get index of Embarked
embarked_idx = train.columns.get_loc('Embarked')
embarked_idx
```

Double check and replace the missing values by randomly picking a value out of the 3 form the "Embarked" column:



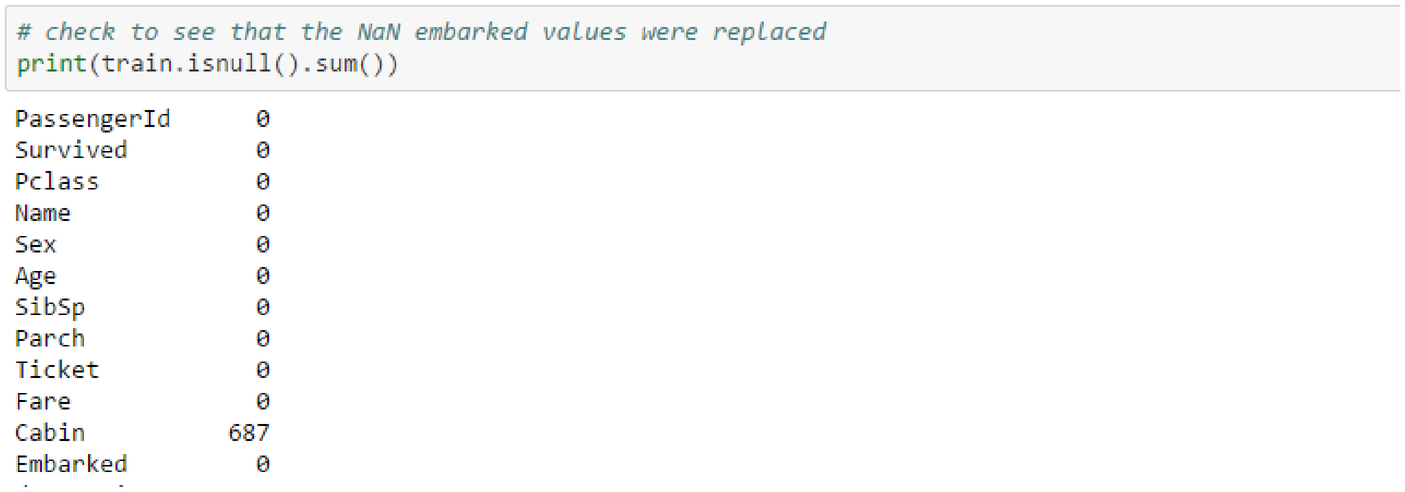
```
# make sure we have the correct location
for idx in nan_indexes:
    print(train.iloc[idx, embarked_idx])
```

```
61     NaN
829    NaN
Name: Embarked, dtype: object
```



```
# replace NaN with random choice from Embarked options
# first we need to see what those options are
embarked_choices = train.Embarked.value_counts().keys()
for idx in nan_indexes:
    train.iloc[idx, embarked_idx] = choice(embarked_choices)
```

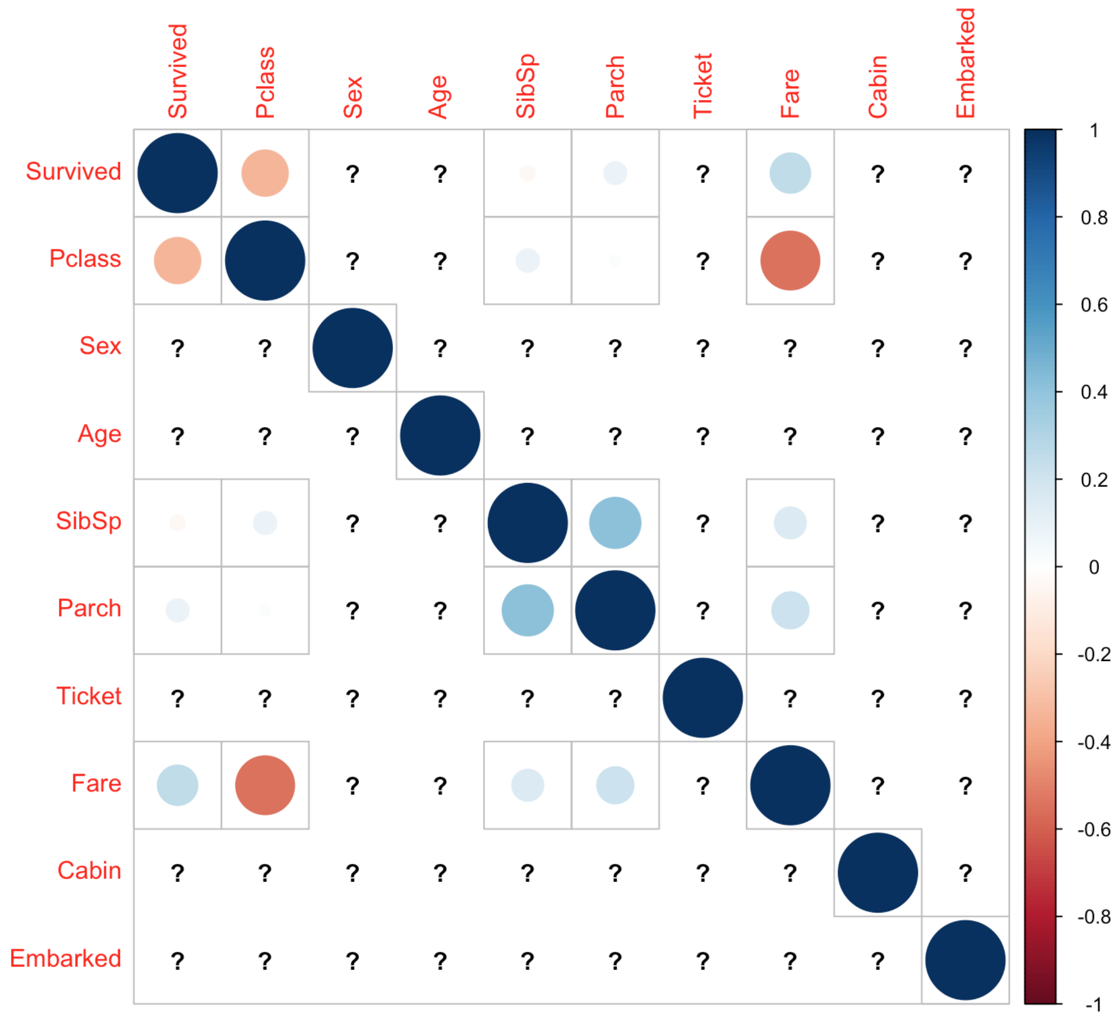
As a result, we have successfully replaced the missing values:



```
# check to see that the NaN embarked values were replaced
print(train.isnull().sum())
```

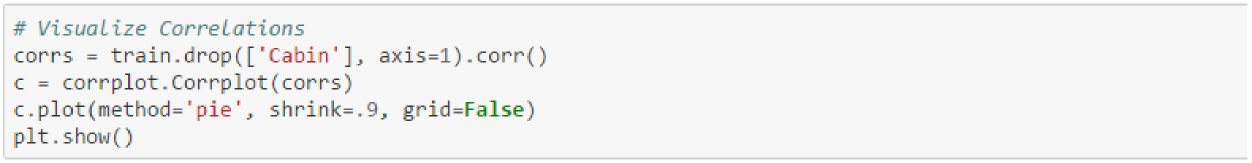
```
PassengerId      0
Survived         0
Pclass           0
Name             0
Sex              0
Age              0
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          687
Embarked         0
```

Co-Relation plot of some attributes and Survived class label



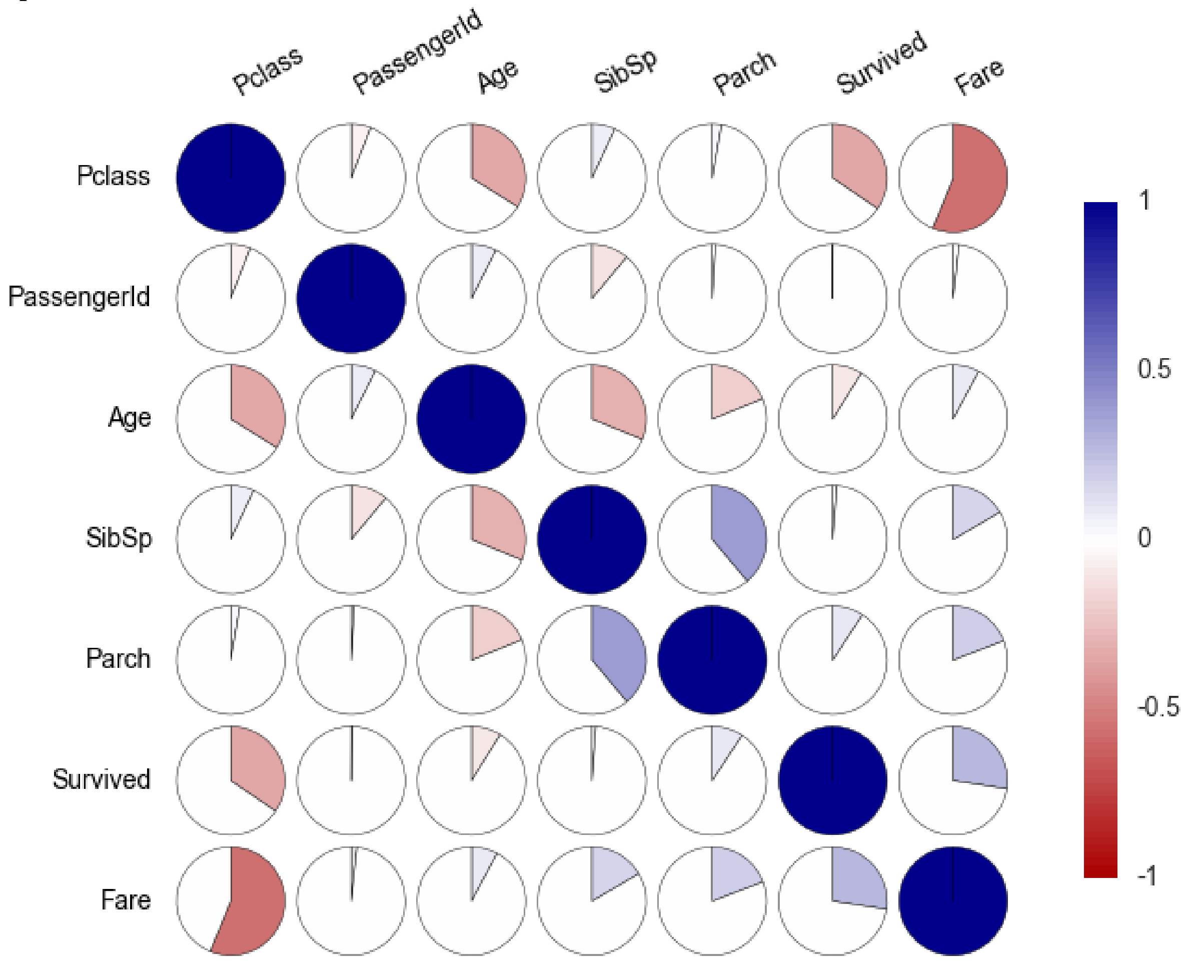
The above plot shows the correlation of the attributes and the survived class after the preprocessing step.

Visualizing the Correlations plot after dropping the “cabin”, “embarked” and “age” attributes:

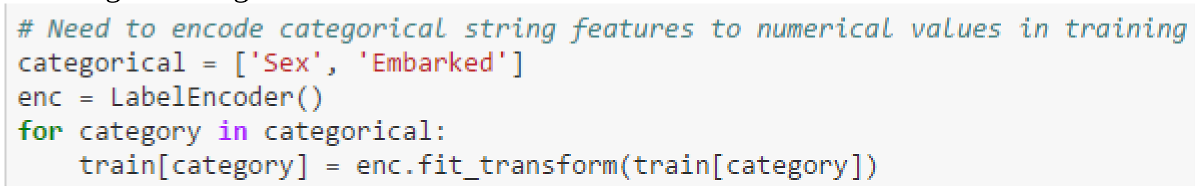


```
# Visualize Correlations
corrs = train.drop(['Cabin'], axis=1).corr()
c = corrplot.Corrplot(corrs)
c.plot(method='pie', shrink=.9, grid=False)
plt.show()
```

The plot is as below:

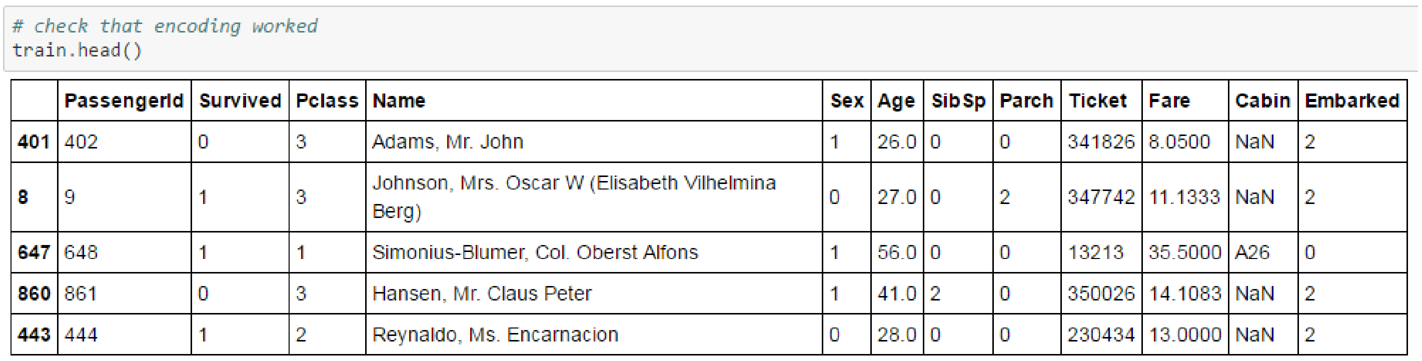


Encoding the string features into numerical values:



```
# Need to encode categorical string features to numerical values in training
categorical = ['Sex', 'Embarked']
enc = LabelEncoder()
for category in categorical:
    train[category] = enc.fit_transform(train[category])
```

Cross verifying the encoding:



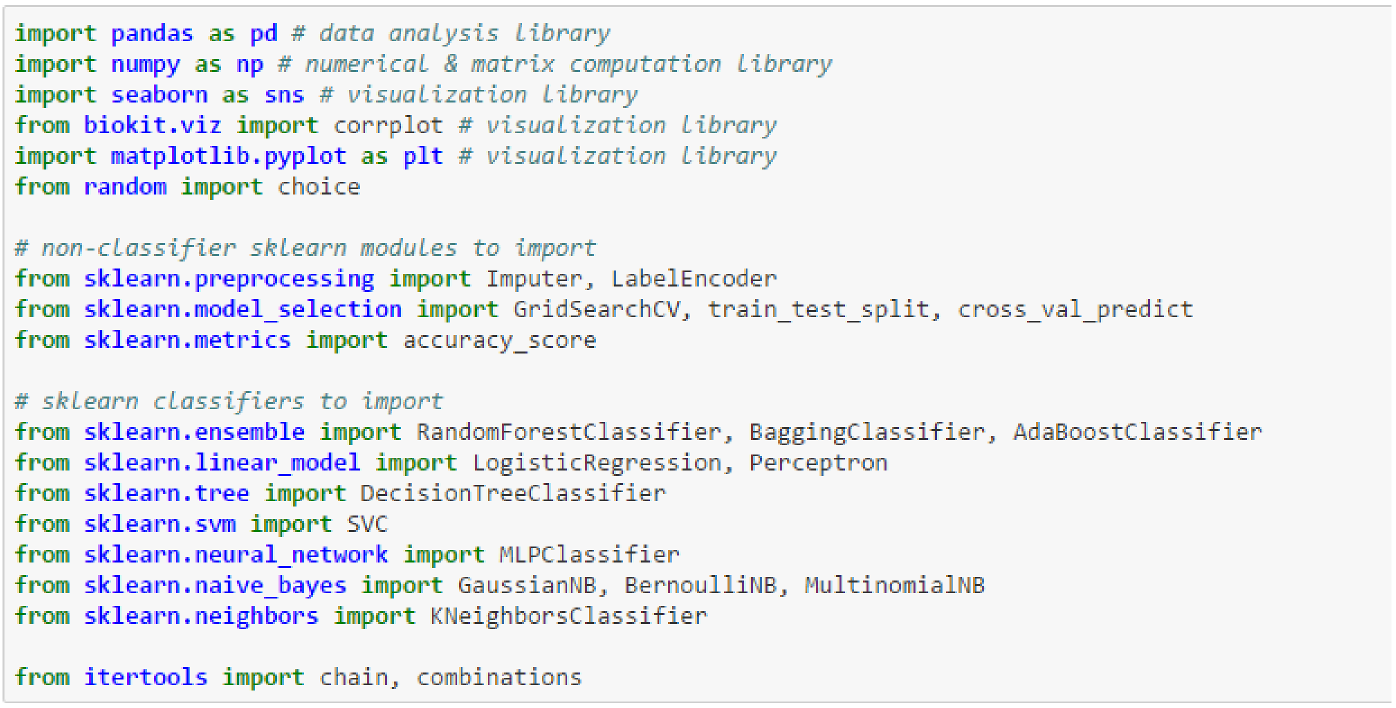
```
# check that encoding worked
train.head()
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
401	402	0	3	Adams, Mr. John	1	26.0	0	0	341826	8.0500	NaN	2
8	9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	0	27.0	0	2	347742	11.1333	NaN	2
647	648	1	1	Simonius-Blumer, Col. Oberst Alfons	1	56.0	0	0	13213	35.5000	A26	0
860	861	0	3	Hansen, Mr. Claus Peter	1	41.0	2	0	350026	14.1083	NaN	2
443	444	1	2	Reynaldo, Ms. Encarnacion	0	28.0	0	0	230434	13.0000	NaN	2

5. <u>Coding Strategy for classifiers:</u>

After preprocessing of data we are building models for every classifier. Though our main strategy is random forest, we decided to go ahead and build few more classifiers in order to see how they are performing on our data set and why they perform how they performed that well or that poor.

<u>Tools and Libraries:</u>



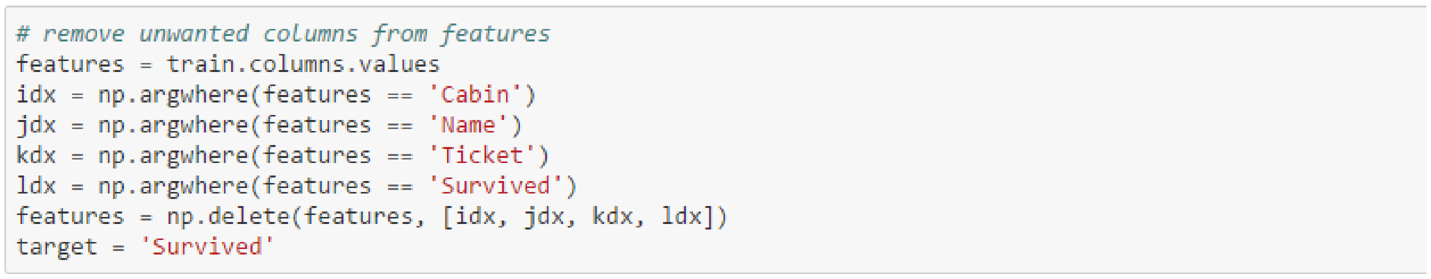
```
import pandas as pd # data analysis library
import numpy as np # numerical & matrix computation library
import seaborn as sns # visualization library
from biokit.viz import corrplot # visualization library
import matplotlib.pyplot as plt # visualization library
from random import choice

# non-classifier sklearn modules to import
from sklearn.preprocessing import Imputer, LabelEncoder
from sklearn.model_selection import GridSearchCV, train_test_split, cross_val_predict
from sklearn.metrics import accuracy_score

# sklearn classifiers to import
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier, AdaBoostClassifier
from sklearn.linear_model import LogisticRegression, Perceptron
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.naive_bayes import GaussianNB, BernoulliNB, MultinomialNB
from sklearn.neighbors import KNeighborsClassifier

from itertools import chain, combinations
```

We are using Scikit-Learn libraries in Python. The reason that we chose Sklearn APIs over CRAN libraries in R is that these API are robust, well documented, consistent over all classifiers.

Though Sklearn API is mostly written in python which is interpreted, some of its code is written in Cython which compiles to C which in turn increases performance. This is a reason that Sklearn is fast.

For writing Python code we used a non traditional editor which is Jupyter Notebook. It is a web application which contain live code on a web page along with the visualization and proper explanation. The advantage we got using this web application is better visualization at every step in code, simple data cleaning.

<u>Removing unnecessary attributes form the Dataset:</u>



```
# remove unwanted columns from features
features = train.columns.values
idx = np.argwhere(features == 'Cabin')
jdx = np.argwhere(features == 'Name')
kdx = np.argwhere(features == 'Ticket')
ldx = np.argwhere(features == 'Survived')
features = np.delete(features, [idx, jdx, kdx, ldx])
target = 'Survived'
```

<u>Feature Selection:</u>

As feature selection is most important when we build a classifier. So first we selected all the feature set and then generated one power set which has all the combinations of features. We tried to get Accuracy on each of such combination and selected the feature set which was most productive.

The following code snippets shows the feature selection:

```
def generate_powerset(items):
    '''
    Generates the powerset of a group of items,
    in this case the features available to a model
    '''
    p = list(items)
    return chain.from_iterable(combinations(p, r) for r in
range(len(p)+1))


def get_best_feature_group_and_acc(classifier, features, train, test):
    '''
    Using the powerset of a group of features, finds
    the best combination of features for a model
    that produces the best accuracy
    '''
    accs = []
    name, clf = classifier
    print('Performing feature find on {}-classifier'.format(name))
    for feature_group in generate_powerset(features):
        if len(feature_group) > 0:
            f = list(feature_group)
            #clf = classifier()
            clf = clf.fit(train[f], train[target])
            preds = clf.predict(test[f])
            acc = accuracy_score(test[target], preds)
            accs.append((name, acc, f))
    return max(accs)
```

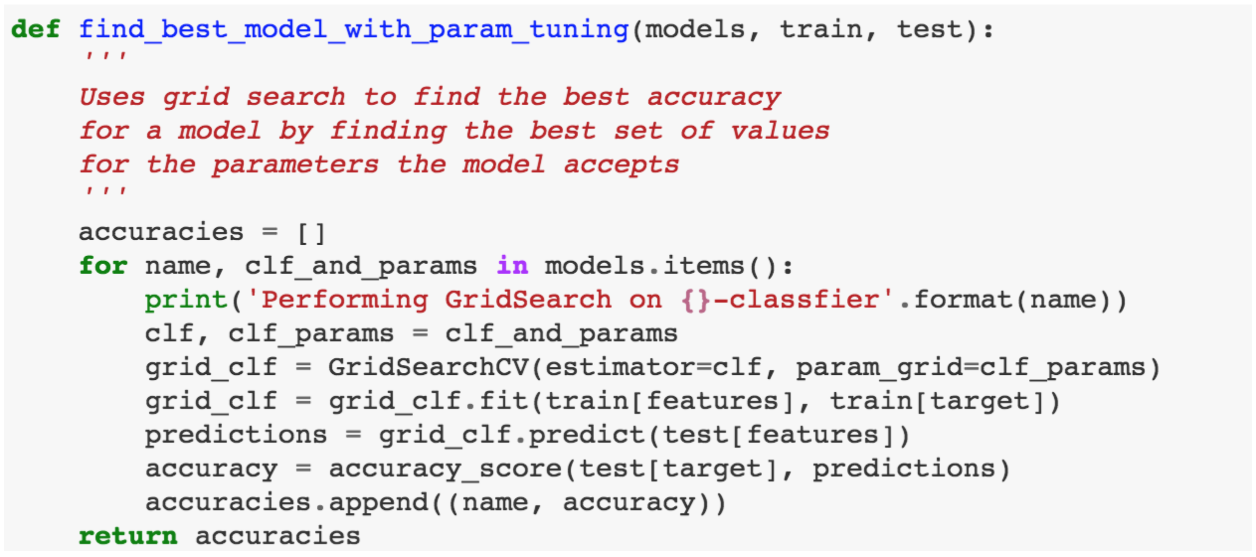
Feature Extraction:

Feature extraction is nothing but to get features which are more meaningful and related to the label out of the existing dataset. As the titanic dataset has very less number of attributes and most of them are straightforward & well defined when we relate them to the problem statement domain. So in order to avoid redundancy in information we avoided feature extraction in our dataset.

Parameter Tuning:

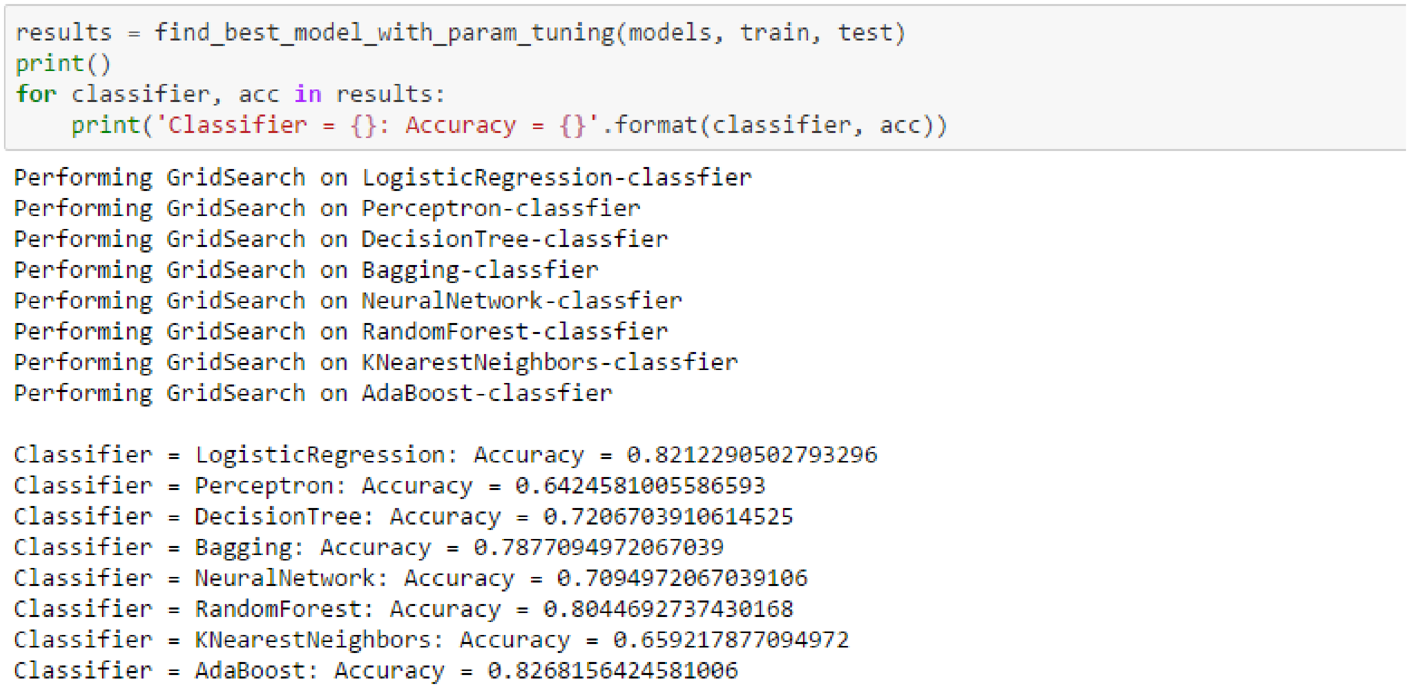
Parameter tuning is a process in which we try out various attributes that are needed to build a model In order to get the best accuracy I.e. we try to create best fit on training data so that it doesn't overfit and learns enough to get higher accuracies.
In all the classifiers below we can see the screens for parameters and their different values that we are using for parameter tuning. We initialize all those values and we try every possible combination of those parameters in order to get the best accuracy. This way we automated the process for parameter tuning.

Here's the code for the same.



```python
def find_best_model_with_param_tuning(models, train, test):
    '''
    Uses grid search to find the best accuracy
    for a model by finding the best set of values
    for the parameters the model accepts
    '''
    accuracies = []
    for name, clf_and_params in models.items():
        print('Performing GridSearch on {}-classfier'.format(name))
        clf, clf_params = clf_and_params
        grid_clf = GridSearchCV(estimator=clf, param_grid=clf_params)
        grid_clf = grid_clf.fit(train[features], train[target])
        predictions = grid_clf.predict(test[features])
        accuracy = accuracy_score(test[target], predictions)
        accuracies.append((name, accuracy))
    return accuracies
```

Usage of the above function:



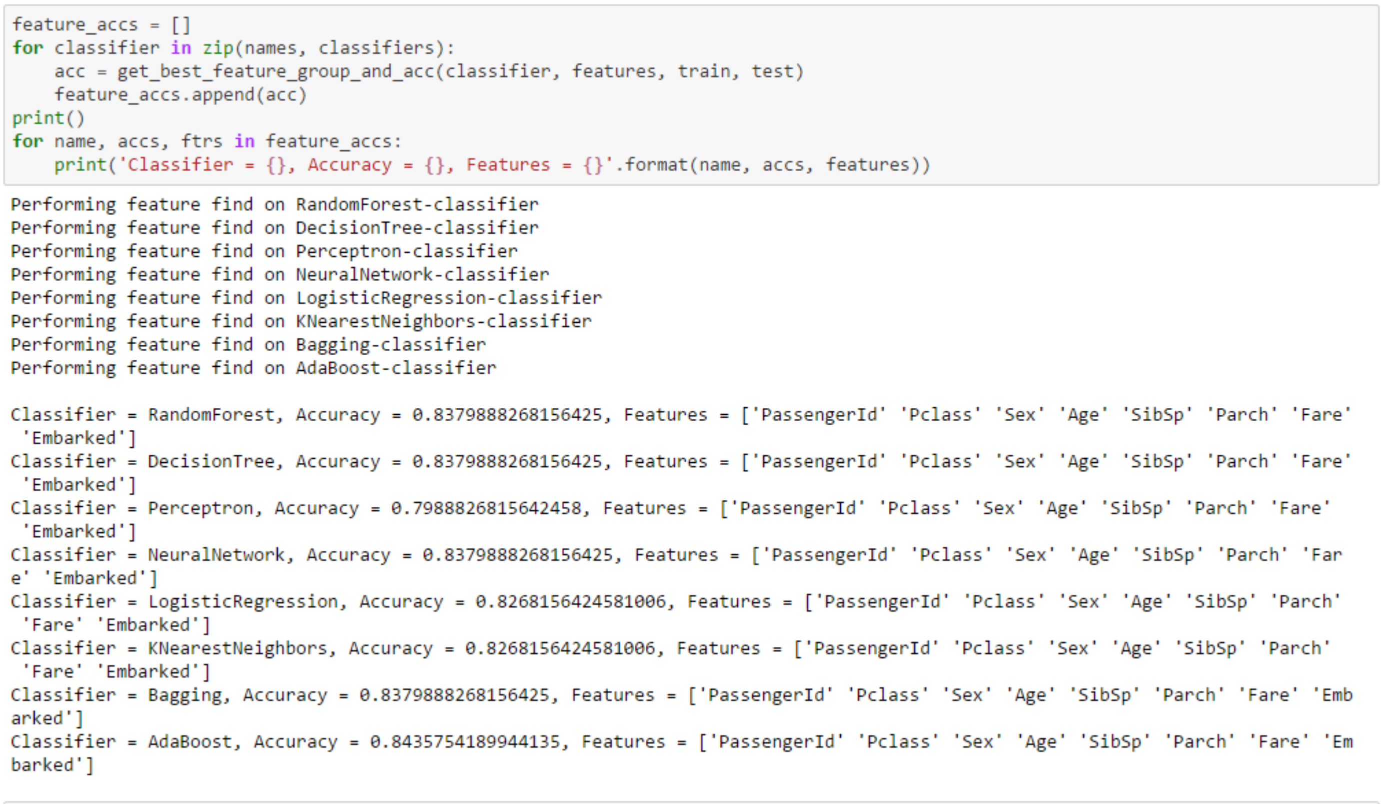
```python
results = find_best_model_with_param_tuning(models, train, test)
print()
for classifier, acc in results:
    print('Classifier = {}: Accuracy = {}'.format(classifier, acc))
```

```
Performing GridSearch on LogisticRegression-classfier
Performing GridSearch on Perceptron-classfier
Performing GridSearch on DecisionTree-classfier
Performing GridSearch on Bagging-classfier
Performing GridSearch on NeuralNetwork-classfier
Performing GridSearch on RandomForest-classfier
Performing GridSearch on KNearestNeighbors-classfier
Performing GridSearch on AdaBoost-classfier

Classifier = LogisticRegression: Accuracy = 0.8212290502793296
Classifier = Perceptron: Accuracy = 0.6424581005586593
Classifier = DecisionTree: Accuracy = 0.7206703910614525
Classifier = Bagging: Accuracy = 0.7877094972067039
Classifier = NeuralNetwork: Accuracy = 0.7094972067039106
Classifier = RandomForest: Accuracy = 0.8044692737430168
Classifier = KNearestNeighbors: Accuracy = 0.659217877094972
Classifier = AdaBoost: Accuracy = 0.8268156424581006
```

Cross Fold Validation:

We are using 10 Fold validation on test dataset on the selected feature group. We used K-fold validation to get lower Variance than single set estimator.

The group of features can be found as below:



```
feature_accs = []
for classifier in zip(names, classifiers):
    acc = get_best_feature_group_and_acc(classifier, features, train, test)
    feature_accs.append(acc)
print()
for name, accs, ftrs in feature_accs:
    print('Classifier = {}, Accuracy = {}, Features = {}'.format(name, accs, features))
```

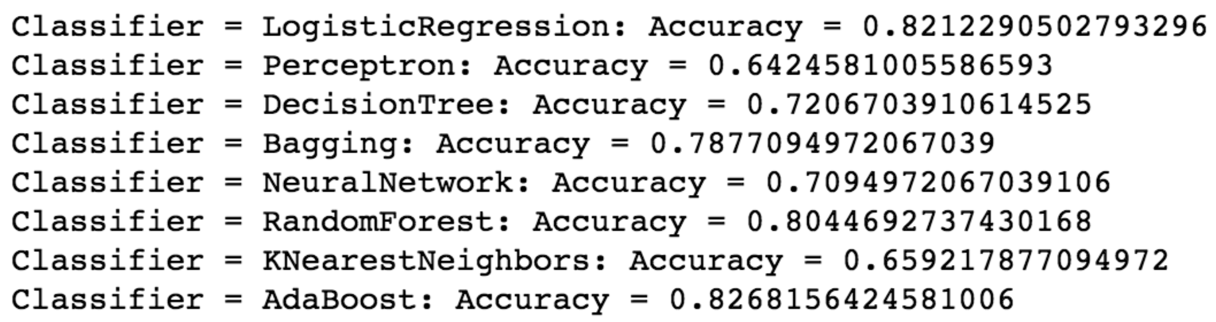
```
Performing feature find on RandomForest-classifier
Performing feature find on DecisionTree-classifier
Performing feature find on Perceptron-classifier
Performing feature find on NeuralNetwork-classifier
Performing feature find on LogisticRegression-classifier
Performing feature find on KNearestNeighbors-classifier
Performing feature find on Bagging-classifier
Performing feature find on AdaBoost-classifier

Classifier = RandomForest, Accuracy = 0.8379888268156425, Features = ['PassengerId' 'Pclass' 'Sex' 'Age' 'SibSp' 'Parch' 'Fare'
 'Embarked']
Classifier = DecisionTree, Accuracy = 0.8379888268156425, Features = ['PassengerId' 'Pclass' 'Sex' 'Age' 'SibSp' 'Parch' 'Fare'
 'Embarked']
Classifier = Perceptron, Accuracy = 0.7988826815642458, Features = ['PassengerId' 'Pclass' 'Sex' 'Age' 'SibSp' 'Parch' 'Fare'
 'Embarked']
Classifier = NeuralNetwork, Accuracy = 0.8379888268156425, Features = ['PassengerId' 'Pclass' 'Sex' 'Age' 'SibSp' 'Parch' 'Far
e' 'Embarked']
Classifier = LogisticRegression, Accuracy = 0.8268156424581006, Features = ['PassengerId' 'Pclass' 'Sex' 'Age' 'SibSp' 'Parch'
 'Fare' 'Embarked']
Classifier = KNearestNeighbors, Accuracy = 0.8268156424581006, Features = ['PassengerId' 'Pclass' 'Sex' 'Age' 'SibSp' 'Parch'
 'Fare' 'Embarked']
Classifier = Bagging, Accuracy = 0.8379888268156425, Features = ['PassengerId' 'Pclass' 'Sex' 'Age' 'SibSp' 'Parch' 'Fare' 'Emb
arked']
Classifier = AdaBoost, Accuracy = 0.8435754189944135, Features = ['PassengerId' 'Pclass' 'Sex' 'Age' 'SibSp' 'Parch' 'Fare' 'Em
barked']
```

The following code is for 10-Fold validation on each classifier.

```
# Perform cross validation with feature groups discovered above

scores = []
feature_classifier_map = dict(zip(names, classifiers))
for name, _, ftrs in feature_accs:
    print('Performing Cross-Validation on {}-classifier'.format(name))
    clf = feature_classifier_map[name]
    predictions = cross_val_predict(clf, test[ftrs], test[target], cv=10)
    score = accuracy_score(test[target], predictions)
    scores.append((name, score))
print()

for name, scr in scores:
    print('Classifier-{}: Accuracy = {}'.format(name, scr))
```

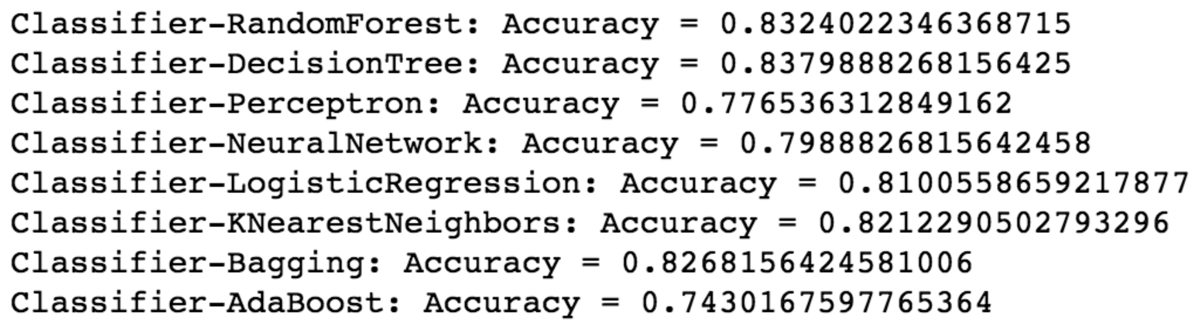
We could see increase in accuracy after using 10-Fold Validation

<u>Before 10-Fold Validation:</u>



```
Classifier = LogisticRegression: Accuracy = 0.8212290502793296
Classifier = Perceptron: Accuracy = 0.6424581005586593
Classifier = DecisionTree: Accuracy = 0.7206703910614525
Classifier = Bagging: Accuracy = 0.7877094972067039
Classifier = NeuralNetwork: Accuracy = 0.7094972067039106
Classifier = RandomForest: Accuracy = 0.8044692737430168
Classifier = KNearestNeighbors: Accuracy = 0.659217877094972
Classifier = AdaBoost: Accuracy = 0.8268156424581006
```

After 10-Fold Validation:



```
Classifier-RandomForest: Accuracy = 0.8324022346368715
Classifier-DecisionTree: Accuracy = 0.8379888268156425
Classifier-Perceptron: Accuracy = 0.776536312849162
Classifier-NeuralNetwork: Accuracy = 0.7988826815642458
Classifier-LogisticRegression: Accuracy = 0.8100558659217877
Classifier-KNearestNeighbors: Accuracy = 0.8212290502793296
Classifier-Bagging: Accuracy = 0.8268156424581006
Classifier-AdaBoost: Accuracy = 0.7430167597765364
```

6. Classifier Selection

Before explaining the work done on each of the classifier, we are here giving the entire look of the code used to pass the parameters for training the classifiers and then printing out the results on the whole set of classifiers that we used in this project.

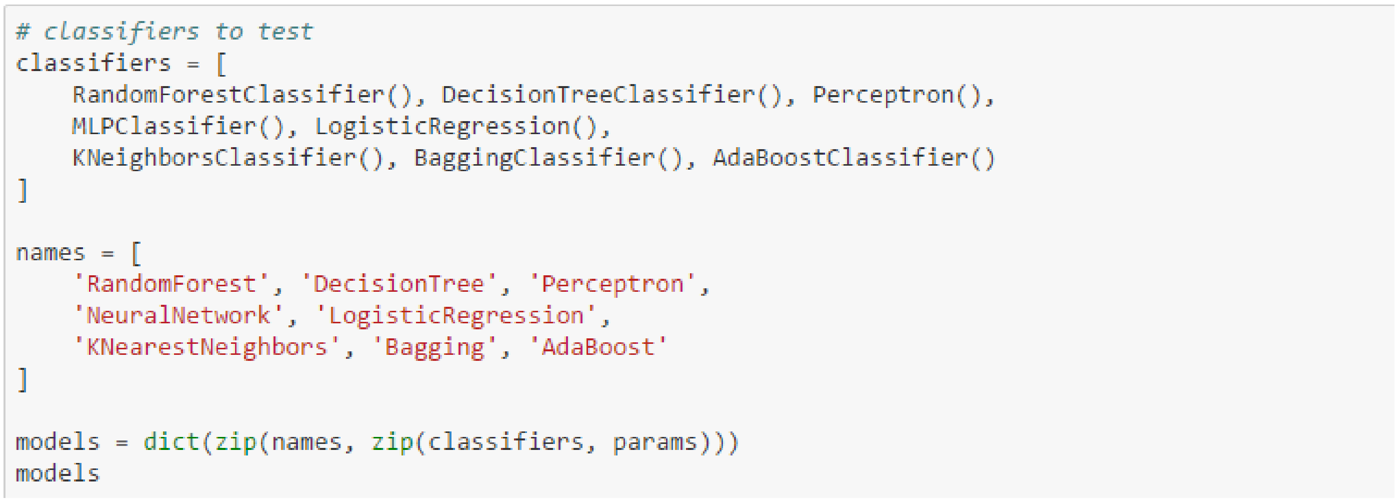
The aim of our project was to build the Random Forest Classifier for predicting the survival of the people travelling. **The classifier we built is approximately 85 percent accurate in predicting the survival chances of an individual in the Titanic.**

Along with the Random Forest Classifier, we have implemented Decision Tree, Perceptron, Neural Net, Logistic Regression, KNN, Bagging, Ada – Boosting.

The parameters set for the respective classifiers as given below:

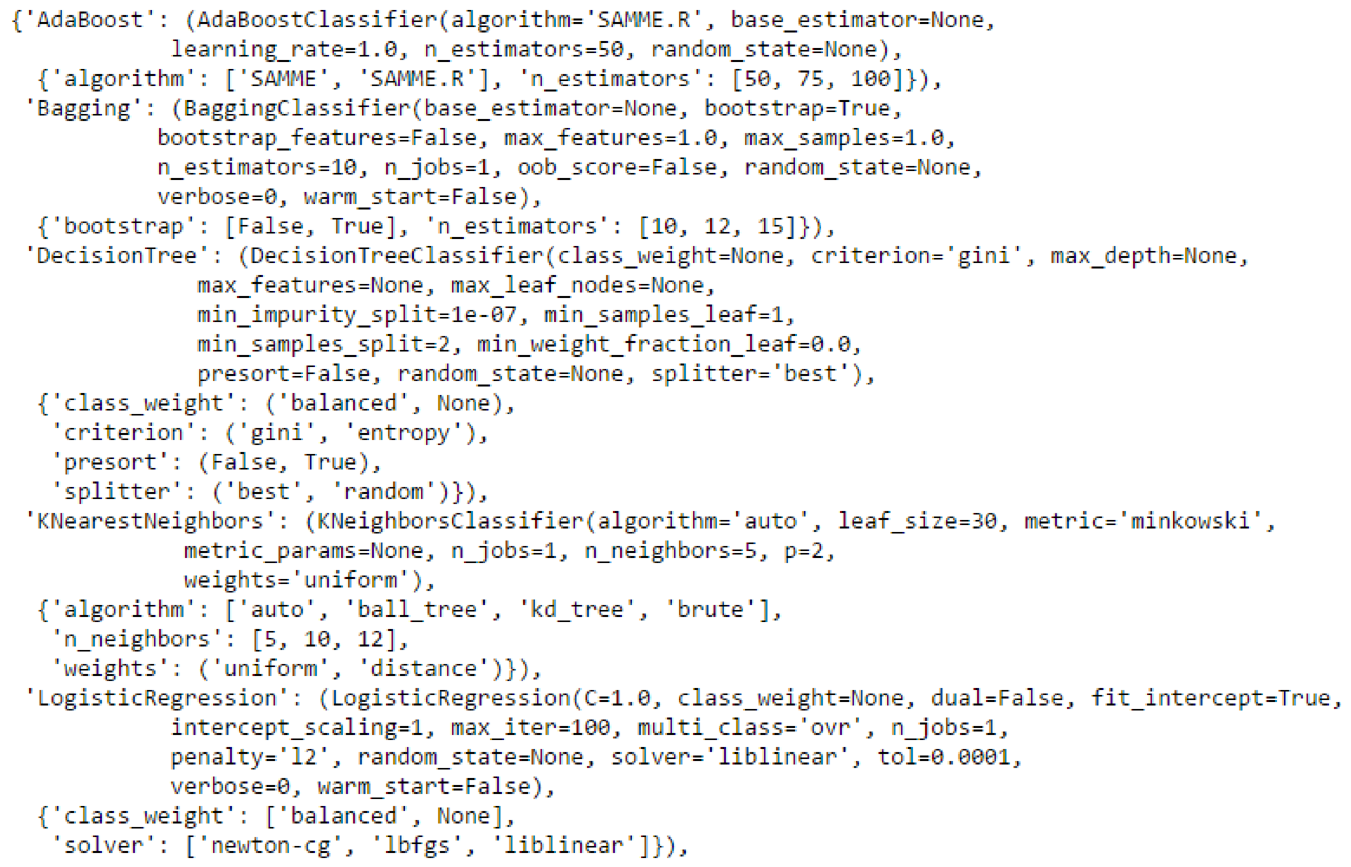
```python
# set up parameter grid to use with exhaustive grid search
# in order to find the best combination of parameters for the
# classifier
random_forest_params = {
    'n_estimators': [10, 12, 15], 'criterion': ('gini', 'entropy'),
    'bootstrap': (True, False), 'class_weight': ('balanced', None)
}

decision_tree_params = {
    'criterion': ('gini', 'entropy'), 'splitter': ('best', 'random'),
    'class_weight': ('balanced', None), 'presort': (False, True)
}

perceptron_params = {
    'penalty': [None, 'l2', 'l1', 'elasticnet'], 'shuffle': [False, True],
    'class_weight': ['balanced', None]
}

'''
svm_params = {
    'kernel': ['linear', 'rbf'],
    'shrinking': [False, True], 'class_weight': ['balanced', None]
}
'''

neural_net_params = {
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
    'solver': ['adam'],
    'learning_rate': ['constant', 'invscaling', 'adaptive']
}

log_reg_params = {
    'class_weight': ['balanced', None],
    'solver': ['newton-cg', 'lbfgs', 'liblinear']
}
knn_params = {
    'n_neighbors': [5, 10, 12], 'weights': ('uniform', 'distance'),
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute']
}

bagging_params = {
    'n_estimators': [10, 12, 15], 'bootstrap': [False, True]
}

ada_boost_params = {
    'n_estimators': [50, 75, 100], 'algorithm': ['SAMME', 'SAMME.R']
}

params = [
    random_forest_params, decision_tree_params, perceptron_params,
    neural_net_params, log_reg_params, knn_params,
    bagging_params, ada_boost_params
]
```

Using the parameters and the arrays in python, we are creating the classifier models using the “dict” and “zip” packages available in python 3. The method is as shown below,



```
# classifiers to test
classifiers = [
    RandomForestClassifier(), DecisionTreeClassifier(), Perceptron(),
    MLPClassifier(), LogisticRegression(),
    KNeighborsClassifier(), BaggingClassifier(), AdaBoostClassifier()
]

names = [
    'RandomForest', 'DecisionTree', 'Perceptron',
    'NeuralNetwork', 'LogisticRegression',
    'KNearestNeighbors', 'Bagging', 'AdaBoost'
]

models = dict(zip(names, zip(classifiers, params)))
models
```

<u>The models created with the parameters passed is as seen below:</u>



```
{'AdaBoost': (AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None,
            learning_rate=1.0, n_estimators=50, random_state=None),
  {'algorithm': ['SAMME', 'SAMME.R'], 'n_estimators': [50, 75, 100]}),
 'Bagging': (BaggingClassifier(base_estimator=None, bootstrap=True,
           bootstrap_features=False, max_features=1.0, max_samples=1.0,
           n_estimators=10, n_jobs=1, oob_score=False, random_state=None,
           verbose=0, warm_start=False),
  {'bootstrap': [False, True], 'n_estimators': [10, 12, 15]}),
 'DecisionTree': (DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
              max_features=None, max_leaf_nodes=None,
              min_impurity_split=1e-07, min_samples_leaf=1,
              min_samples_split=2, min_weight_fraction_leaf=0.0,
              presort=False, random_state=None, splitter='best'),
  {'class_weight': ('balanced', None),
   'criterion': ('gini', 'entropy'),
   'presort': (False, True),
   'splitter': ('best', 'random')}),
 'KNearestNeighbors': (KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
             metric_params=None, n_jobs=1, n_neighbors=5, p=2,
             weights='uniform'),
  {'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
   'n_neighbors': [5, 10, 12],
   'weights': ('uniform', 'distance')}),
 'LogisticRegression': (LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
            intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
            penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
            verbose=0, warm_start=False),
  {'class_weight': ['balanced', None],
   'solver': ['newton-cg', 'lbfgs', 'liblinear']}),
'NeuralNetwork': (MLPClassifier(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9,
        beta_2=0.999, early_stopping=False, epsilon=1e-08,
        hidden_layer_sizes=(100,), learning_rate='constant',
        learning_rate_init=0.001, max_iter=200, momentum=0.9,
        nesterovs_momentum=True, power_t=0.5, random_state=None,
        shuffle=True, solver='adam', tol=0.0001, validation_fraction=0.1,
        verbose=False, warm_start=False),
 {'activation': ['identity', 'logistic', 'tanh', 'relu'],
  'learning_rate': ['constant', 'invscaling', 'adaptive'],
  'solver': ['adam']}),
'Perceptron': (Perceptron(alpha=0.0001, class_weight=None, eta0=1.0, fit_intercept=True,
       n_iter=5, n_jobs=1, penalty=None, random_state=0, shuffle=True,
       verbose=0, warm_start=False),
 {'class_weight': ['balanced', None],
  'penalty': [None, 'l2', 'l1', 'elasticnet'],
  'shuffle': [False, True]}),
'RandomForest': (RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
             max_depth=None, max_features='auto', max_leaf_nodes=None,
             min_impurity_split=1e-07, min_samples_leaf=1,
             min_samples_split=2, min_weight_fraction_leaf=0.0,
             n_estimators=10, n_jobs=1, oob_score=False, random_state=None,
             verbose=0, warm_start=False),
 {'bootstrap': (True, False),
  'class_weight': ('balanced', None),
  'criterion': ('gini', 'entropy'),
  'n_estimators': [10, 12, 15]})}
```

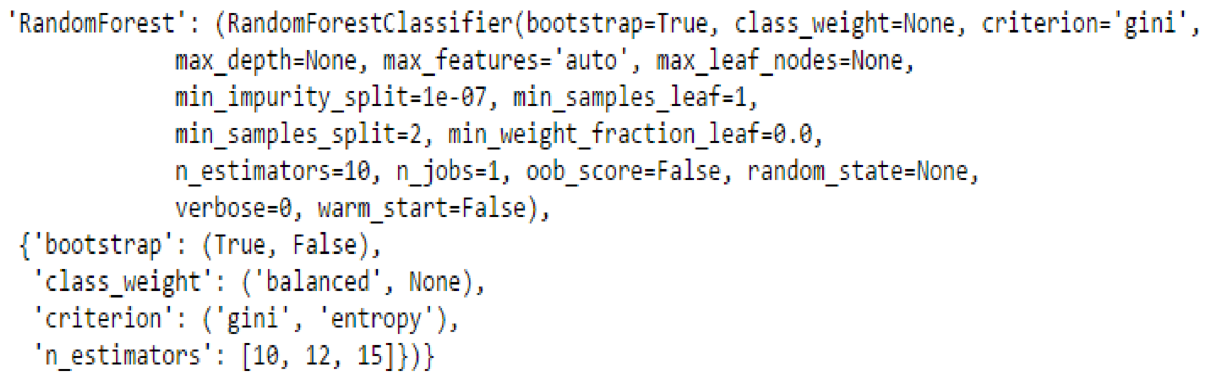
Explanation by considering the classifiers individually:

- **Random Forest**

In Random Forest method we grow many classification trees. To classify a new object from an input vector, we put the input vector down each of the trees in the forest. Each tree gives a classification, and we say the tree "votes" for that class. The forest chooses the classification having the most votes.
As part of the first step we set up parameter grid to use with exhaustive grid search in order to find the best combination of parameters for this classifier.

```
random_forest_params = {
    'n_estimators': [10, 12, 15], 'criterion': ('gini', 'entropy'),
    'bootstrap': (True, False), 'class_weight': ('balanced', None)
}
```

Once we decide on the parameters we include the Random Forest classifier in the classifiers to test and create a model by passing the information of the parameters and the classifier as follows:



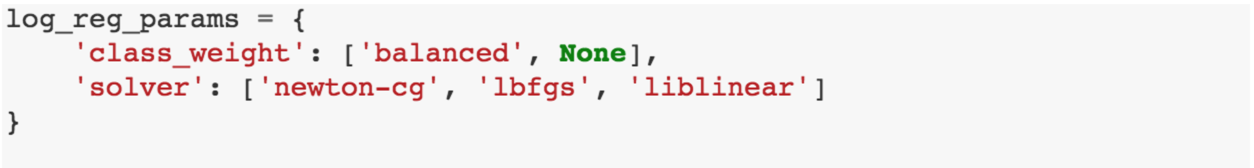
```
'RandomForest': (RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
             max_depth=None, max_features='auto', max_leaf_nodes=None,
             min_impurity_split=1e-07, min_samples_leaf=1,
             min_samples_split=2, min_weight_fraction_leaf=0.0,
             n_estimators=10, n_jobs=1, oob_score=False, random_state=None,
             verbose=0, warm_start=False),
  {'bootstrap': (True, False),
   'class_weight': ('balanced', None),
   'criterion': ('gini', 'entropy'),
   'n_estimators': [10, 12, 15]})}
```

As part of predicting accuracy of the created Random Forest model we build a helper function that uses grid search for finding the best set of values for the parameters the model accepts

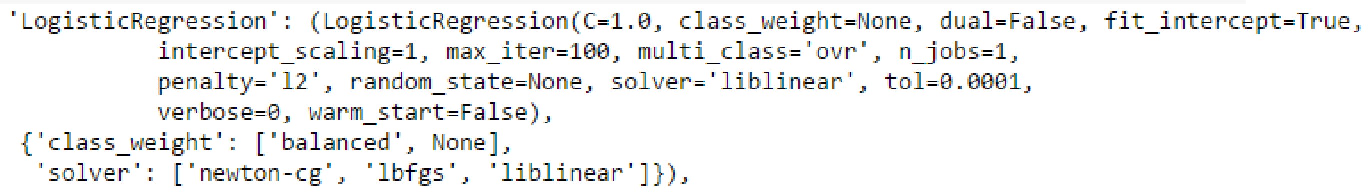
Classifier- Random Forest: **Accuracy in Prediction** = 0.8324022346368715

- ## Logistic Regression

Logistic Regression is a classification technique that helps us come up with a probability function that can give the chance for an input to belong to any of the various classes we have in our classification.
Once we decide on the parameters we include the Logistic Regression classifier in the classifiers to test and create a model by passing the information of the parameters and the classifier as follows –



```
log_reg_params = {
    'class_weight': ['balanced', None],
    'solver': ['newton-cg', 'lbfgs', 'liblinear']
}
```



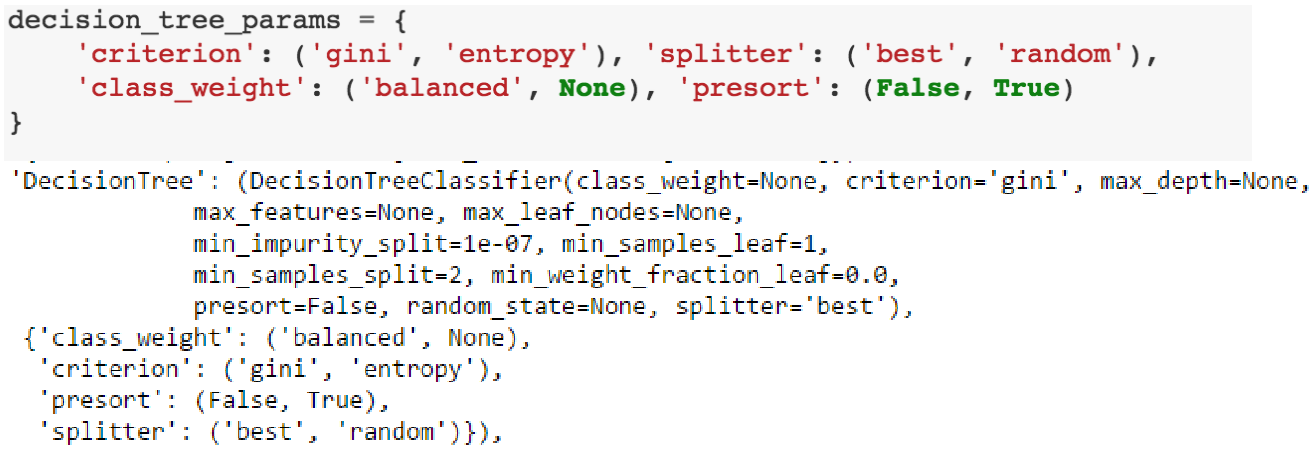
```
'LogisticRegression': (LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False),
  {'class_weight': ['balanced', None],
   'solver': ['newton-cg', 'lbfgs', 'liblinear']}),
```

Classifier-Logistic Regression: **Accuracy in Prediction** = 0.8100558659217877

- ## Decision Tree

Decision trees can be used as Classification trees or regression trees. Here we will use decision trees for classification. Every non-leaf node of decision tree splits dataset on some attribute of the given dataset. If we traverse from root toward any of the leaf, the path we follow will basically represent our choices for the values of various attributes. If we map the data, the Decision divides the data using parallel lines making rectangles. Due to its flexibility, It can classify non linear data as well.

We are building tree model in the following code:



```
decision_tree_params = {
    'criterion': ('gini', 'entropy'), 'splitter': ('best', 'random'),
    'class_weight': ('balanced', None), 'presort': (False, True)
}
```

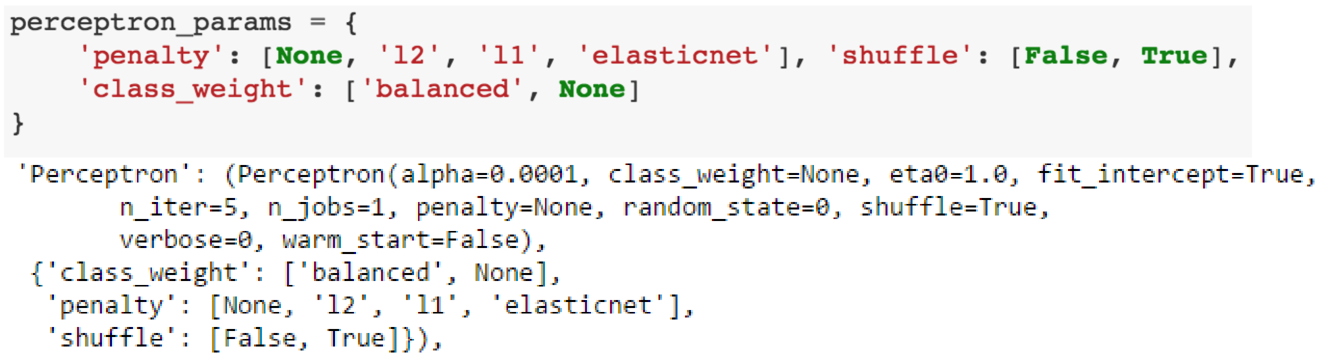
```
'DecisionTree': (DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
             max_features=None, max_leaf_nodes=None,
             min_impurity_split=1e-07, min_samples_leaf=1,
             min_samples_split=2, min_weight_fraction_leaf=0.0,
             presort=False, random_state=None, splitter='best'),
 {'class_weight': ('balanced', None),
  'criterion': ('gini', 'entropy'),
  'presort': (False, True),
  'splitter': ('best', 'random')}),
```

Classifier-Decision Tree: **Accuracy in Prediction** = 0.8379888268156425

- **Perceptron**

Perceptron resembles the concept of Neurons that occur in our brains. Perceptron is in use since 1940. Perceptron is not a non linear classifier but it is used as a building block for other classifier. Input to Perceptron are feature values, each feature has a weight. Perceptron has activation function, which is generally a sigmoid function. We generally compare the output with some threshold. If the output is below the threshold the set of values contribute to the negative class else it contributes to the positive class.

We are building perceptron model in the following code:



```
perceptron_params = {
    'penalty': [None, 'l2', 'l1', 'elasticnet'], 'shuffle': [False, True],
    'class_weight': ['balanced', None]
}
```

```
 'Perceptron': (Perceptron(alpha=0.0001, class_weight=None, eta0=1.0, fit_intercept=True,
       n_iter=5, n_jobs=1, penalty=None, random_state=0, shuffle=True,
       verbose=0, warm_start=False),
  {'class_weight': ['balanced', None],
   'penalty': [None, 'l2', 'l1', 'elasticnet'],
   'shuffle': [False, True]}),
```

Classifier-Perceptron: **Accuracy in Prediction** = 0.776536312849162

- **Neural Network**

Let us suppose we chain bunch of neutrons together to make a Network, this structure is nothing but a Neural Net. Neural net takes input feature vectors along with some weights and the outputs are basically weights which are given to the next set of Perceptrons to fine tune over the weights of newly learned features. So basically the Neural Nets has layers which learns the new features out of the input features and feed them to next layer with better set of Weights and finally the last perceptron predicts the output with better accuracy. As it has multiple number of Perceptron layers it can handle non linear data.

We are building Neural Net model in the following code:

```
neural_net_params = {
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
    'solver': ['adam'],
    'learning_rate': ['constant', 'invscaling', 'adaptive']
}
```

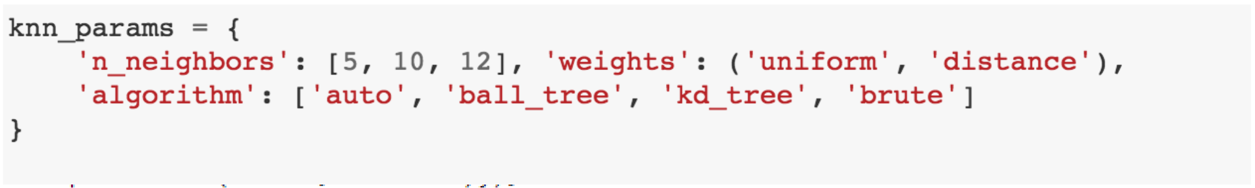
```
'NeuralNetwork': (MLPClassifier(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9,
        beta_2=0.999, early_stopping=False, epsilon=1e-08,
        hidden_layer_sizes=(100,), learning_rate='constant',
        learning_rate_init=0.001, max_iter=200, momentum=0.9,
        nesterovs_momentum=True, power_t=0.5, random_state=None,
        shuffle=True, solver='adam', tol=0.0001, validation_fraction=0.1,
        verbose=False, warm_start=False),
 {'activation': ['identity', 'logistic', 'tanh', 'relu'],
  'learning_rate': ['constant', 'invscaling', 'adaptive'],
  'solver': ['adam']}),
```

Classifier-Neural Network: **Accuracy in Prediction** = 0.7988826815642458

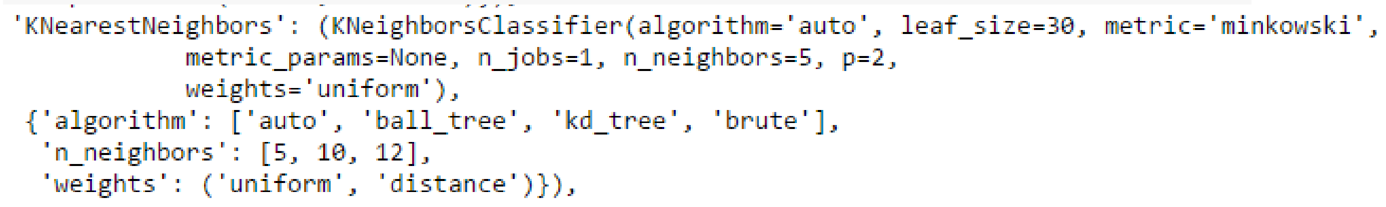
- **K-Nearest Neighbor's**

KNN algorithm is one of the simplest algorithm in Inductive learning which uses lazy learning approach. As the name say, It considers the class label for the k nearest neighbors and assigns the label of the majority to the point.(KNN can also be used for regression) The nearer points are given higher weightage in oppose to the farther points. There are various techniques to find the distance metric like Euclidean distance, Hamming distance which really depends the kind of Feature sets we are working on.

We are building KNN model in the following code:



```
knn_params = {
    'n_neighbors': [5, 10, 12], 'weights': ('uniform', 'distance'),
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute']
}
```



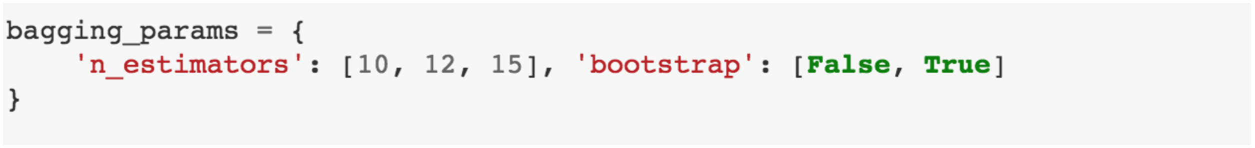
```
'KNearestNeighbors': (KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
           metric_params=None, n_jobs=1, n_neighbors=5, p=2,
           weights='uniform'),
  {'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
   'n_neighbors': [5, 10, 12],
   'weights': ('uniform', 'distance')}),
```

Classifier-K Nearest Neighbors: **Accuracy in Prediction** = 0.8212290502793296

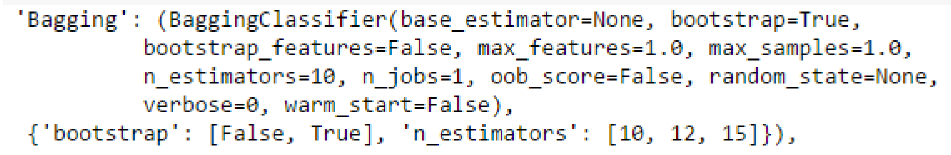
- ## Bagging Classifier

Bagging is called **B**ootstrap **Agg**regation. In this we sample input data set and generate multiple data sets. On each newly generated datasets we train some classifier and build a models.
Now to get the testing accuracy, test instances are run through each of these models and the majority output class is selected as a label for that particular testing instance. (The output of these models is averaged in case of LR). Bagging helps us reduce variance keeping the bias constant.

We are building bagging models in the following code:



```
bagging_params = {
    'n_estimators': [10, 12, 15], 'bootstrap': [False, True]
}
```



```
'Bagging': (BaggingClassifier(base_estimator=None, bootstrap=True,
         bootstrap_features=False, max_features=1.0, max_samples=1.0,
         n_estimators=10, n_jobs=1, oob_score=False, random_state=None,
         verbose=0, warm_start=False),
  {'bootstrap': [False, True], 'n_estimators': [10, 12, 15]}),
```

Classifier-Bagging: **Accuracy in Prediction** = 0.8268156424581006

- ## AdaBoost Classifier

The curiosity that can a group of weak classifiers perform better than a strong classifier. A weak classifier is a classifier whose accuracy is slightly better than 50% i.e. it is slightly better than random guessing. The instances are weighted. Whenever a weak classifier is added, the instances are reweighted. So the instances misclassified by previous classifiers now have more weight and the ones classified properly have less weight.

We are building bagging models in the following code:

```
ada_boost_params = {
    'n_estimators': [50, 75, 100], 'algorithm': ['SAMME', 'SAMME.R']
}
```

```
'AdaBoost': (AdaBoostClassifier(algorithm='SAMME.R', base_estimator=None,
           learning_rate=1.0, n_estimators=50, random_state=None),
 {'algorithm': ['SAMME', 'SAMME.R'], 'n_estimators': [50, 75, 100]}),
'Bagging': (BaggingClassifier(base_estimator=None, bootstrap=True,
          bootstrap_features=False, max_features=1.0, max_samples=1.0,
          n_estimators=10, n_jobs=1, oob_score=False, random_state=None,
          verbose=0, warm_start=False),
 {'bootstrap': [False, True], 'n_estimators': [10, 12, 15]}),
```

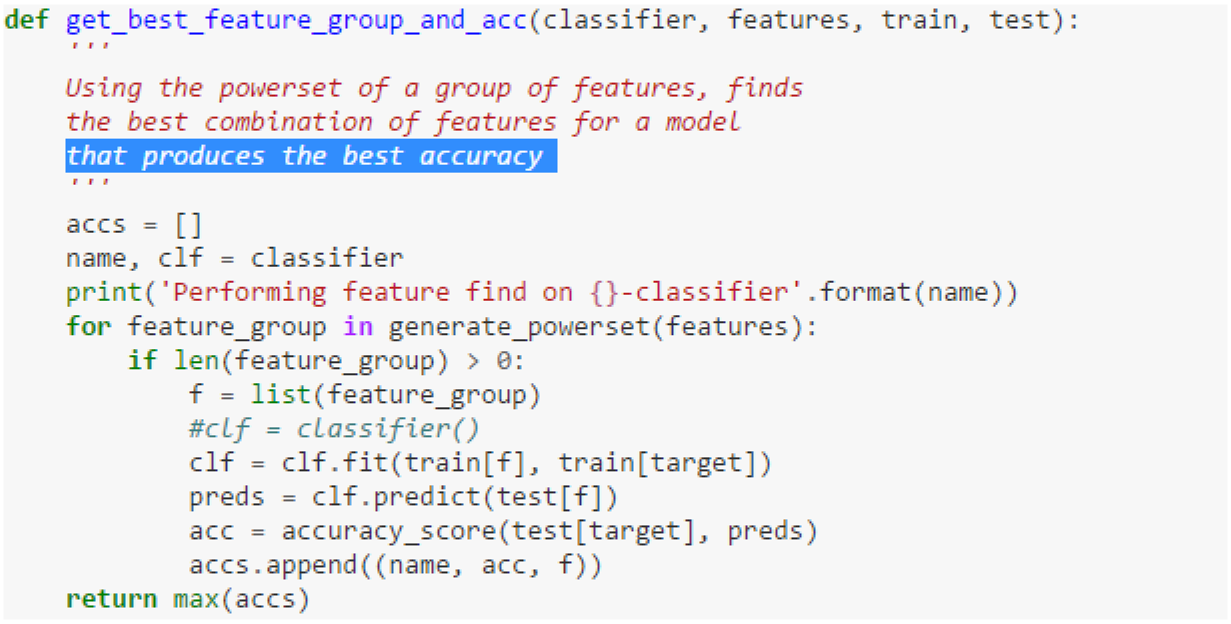
Classifier-Ada Boost: **Accuracy in Prediction** = 0.7430167597765364

7. Validation/Predicting Accuracy of results

To find the best model with parameter tuning we use grid search technique which helps to get the best set of values for the parameters the model accepts.

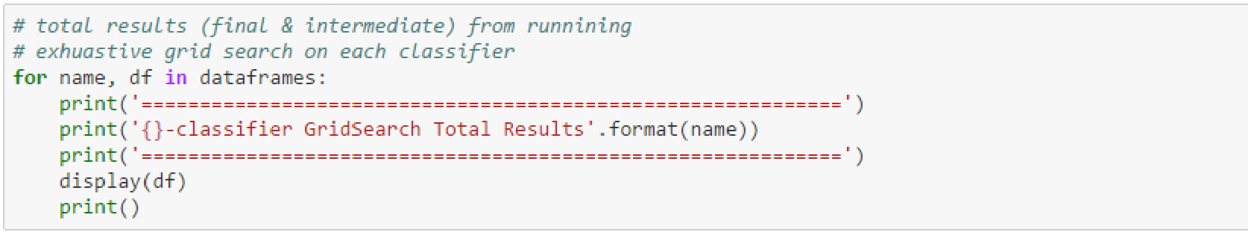
```
def find_best_model_with_param_tuning(models, train, test):
    '''
    Uses grid search to find the best accuracy
    for a model by finding the best set of values
    for the parameters the model accepts
    '''
    accuracies = []
    for name, clf_and_params in models.items():
        print('Performing GridSearch on {}-classfier'.format(name))
        clf, clf_params = clf_and_params
        grid_clf = GridSearchCV(estimator=clf, param_grid=clf_params)
        grid_clf = grid_clf.fit(train[features], train[target])
        predictions = grid_clf.predict(test[features])
        accuracy = accuracy_score(test[target], predictions)
        accuracies.append((name, accuracy))
    return accuracies


def generate_powerset(items):
    '''
    Generates the powerset of a group of items,
    in this case the features available to a model
    '''
    p = list(items)
    return chain.from_iterable(combinations(p, r) for r in range(len(p)+1))
```

Using the power set of a group of features, finds the best combination of features for a model that produces the best accuracy.



```python
def get_best_feature_group_and_acc(classifier, features, train, test):
    '''
    Using the powerset of a group of features, finds
    the best combination of features for a model
    that produces the best accuracy
    '''
    accs = []
    name, clf = classifier
    print('Performing feature find on {}-classifier'.format(name))
    for feature_group in generate_powerset(features):
        if len(feature_group) > 0:
            f = list(feature_group)
            #clf = classifier()
            clf = clf.fit(train[f], train[target])
            preds = clf.predict(test[f])
            acc = accuracy_score(test[target], preds)
            accs.append((name, acc, f))
    return max(accs)
```

The Intermediate results on running a Grid Search (**Refer the HTML file submitted to view the complete result**) in the case of Parameter tuning to achieve the best accuracy on the classifiers. This can be obtained using the below code:



```python
# total results (final & intermediate) from runnining
# exhuastive grid search on each classifier
for name, df in dataframes:
    print('============================================================')
    print('{}-classifier GridSearch Total Results'.format(name))
    print('============================================================')
    display(df)
    print()
```

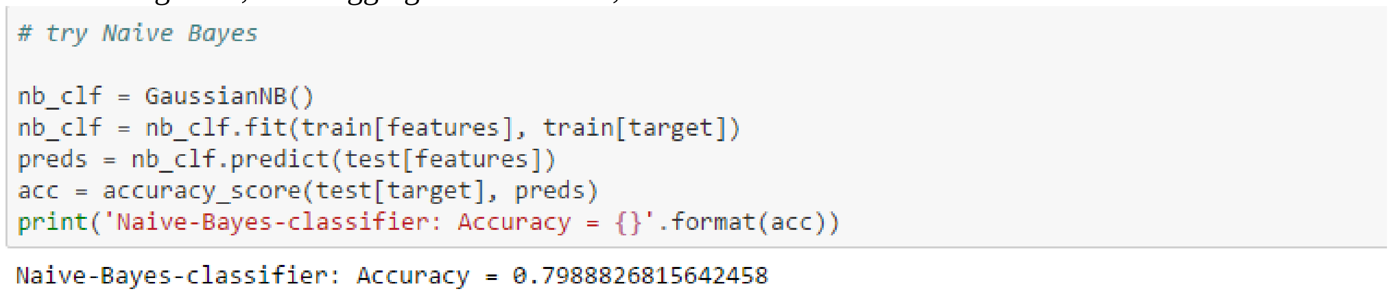
8. <u>Summary</u>

This project had the students attempting to find the best machine learning classification model for predicting the survivor label for passengers on the Titanic, where the survivor label is a binary label with 1 representing survival and 0 representing death. Several different classifiers were tried and compared, among which were: Random Forest, Decision Tree, Neural Network, Logistic Regression, K-Nearest Neighbors, Bagging, and Ada Boost. To make sure the parameters each classifier uses were as optimal as possible, exhaustive grid search was performed for each classification model and the intermediate and final results for this procedure was returned, which included model the accuracy for the best combination of parameter values. The best performing classifiers from the grid search results were Bagging, Logistic Regression, Ada Boost, Random Forest, and Decision Tree.

To also ensure that the best set of features was used for each classification model, the powerset of features was generated and for each set in the powerset, the model was fit on the training set, with the intermediate accuracy recorded. The set of features that led to the best performance was saved, along with the accuracy achieved by using this set of features.

Comparing the most successful sets of features, with the initial correlation chart done in preprocessing, gives the students some intuitive sense of what features mattered most in terms of survival. Ticket Fare, Sex, Passenger Ticket Class, and Age all seem to play a big part, no matter which classifier was used. As explained, this makes intuitive sense if you know anything about the history of the Titanic disaster. Women and children were put into life rafts first, and some have speculated that lower class passengers were delayed in escaping the sinking ship. For feature power set, the classifiers that performed best were Ada Boost, Random Forest, Decision Tree, and Bagging.

Cross-validation was also performed on each classifier. The models that achieved the best results from cross validation were Random Forest, Decision Tree, Logistic Regression, K-Nearest Neighbors, and Bagging. As a final test,



```python
# try Naive Bayes

nb_clf = GaussianNB()
nb_clf = nb_clf.fit(train[features], train[target])
preds = nb_clf.predict(test[features])
acc = accuracy_score(test[target], preds)
print('Naive-Bayes-classifier: Accuracy = {}'.format(acc))
```

```
Naive-Bayes-classifier: Accuracy = 0.7988826815642458
```

a simple Gaussian Naive Bayes classifier was run on the dataset to see how well it performed. It did not work well with Grid Search or Feature Power Set, since it does not accept parameters in the Sklearn implementations that were used in this project. It achieved decent, but not great, accuracy at around 79%.

This project gave the students experience using multiple machine learning classification models. Not only did the students get experience with utilizing machine learning library and choosing models, but also tuning model parameters and choosing the best set of features. It also helped the students get a clear understanding of how machine learning can be applied to real world situations in order to solve a problem and gain insight. The topic of the project was also interesting, and would not normally seem like a scenario where machine learning could be applied. In a way, this made the project all the more interesting.

9. Reference Links

https://www.kaggle.com/c/titanic
http://courses.washington.edu/css490/2012.Winter/lecture_slides/05b_logistic_regression.pdf
http://scikit-learn.org/stable/developers/