Star Classification - Comparing ML Methods K-Nearest Neighbors, AdaBoost and Logistic Regression

Sanjana Mishra and Gerri Fox

In [1]:
```
import pandas as pd
import numpy as np
from collections import Counter
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import KFold
from sklearn.metrics import r2_score
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA
from sklearn.metrics import roc_curve, roc_auc_score, accuracy_score, f1_score
from sklearn.ensemble import AdaBoostClassifier
from sklearn import tree
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
from sklearn import utils
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.metrics import precision_score, recall_score
from sklearn.neighbors import KNeighborsClassifier
```

Our Data:

The Kaggle Dataset we used.

Our data is provided from NASA via Kaggle and includes the following features collected about stars:

Temperature (Kelvin)

Luminosity (L/Lo*)

Radius (R/Ro**)

Absolute Magnitude (Mv)

General Color of Spectrum (Red, Blue, etc.)

Spectral Class (O,B,A,F,G,K,M)

These features can then be used to predict the type of each provided star, with the types being the following:

Red Dwarf (0)

Brown Dwarf (1)

White Dwarf (2)

Main Sequence (3)

Super Giants (4)

Hyper Giants (5)

*Lo = 3.828 x 10^26 Watts (Avg Luminosity of Sun)

**Ro = 6.9551 x 10^8 m (Avg Radius of Sun)

The types of each star were already converted to integers in our dataset, but the color and spectral class were not. So, to get the data you see below, we first conducted some feature engineering to convert both of these fields to numerical data as follows:

Blue (1)

Blue White (2)

Orange (3)

Orange-Red (4)

Pale yellow orange (5)

Red (6)

White (7)

White-yellow (8)

Whitish (9)

Yellow white (10)

Yellowish (11)

and

O (1)

B (2)

A (3)

F (4)

G (5)

K (6)

M (7)

In [2]:
```
df = pd.read_csv('StarsEdited.csv', header=None) # read in the data into a dataframe
```

```
df = df.dropna() # drop any non existent data
# https://www.codegrepper.com/code-examples/python/how+to+make+1st+row+as+header+in+pan
new_header = df.iloc[0] #grab the first row for the header
df = df[1:] #take the data less the header row
df.columns = new_header #set the header row as the df header
df.head()
```

Out[2]:

	Temperature	L	R	A_M	Color	Spectral_Class	Type
1	33750	220000	26	-6.1	1	2	4
2	19860	0.0011	0.0131	11.34	1	2	2
3	33300	240000	12	-6.5	1	2	4
4	21020	0.0015	0.0112	11.52	1	2	2
5	18290	0.0013	0.00934	12.78	1	2	2

First, we chose to implement **K-Nearest Neighbors** from scratch:

In [3]:

```
class KNN:
    ''' A class that has the necessary methods to train a machine learning
        model following K-Nearest Neighbors.
    '''

    def __init__(self, k):
        ''' Constructor for KNN.

            Args:
              k (int): represents the number of neighbors to consider
        '''
        self.k = k

    def fit(self, X, y):
        ''' Fits this object using the given data.

          Args:
              X (np.array): Represents the X training data
              y (np.array): Represents the y training data
        '''
        self.X_train = X
        self.y_train = y

    #https://www.geeksforgeeks.org/calculate-the-euclidean-distance-using-numpy/
    def distance(self, X1, X2):
        ''' Calculates the euclidean distance between two given points.

            Args:
              X1 (int): the first point to get the distance from
              X2 (int): the second point to get the distance to

            Returns:
              distance (int): the distance between the two given points
        '''
        distance = np.sum(np.square(X1 - X2))
        return np.sqrt(distance)
```

```
    def predict(self, X_test):
        ''' Predicts the Y data from the given X data using the trained model.

            Args:
              X_test (np.array): the X data to classify

            Returns:
              final_output (np.array): the corresponding Y data classifying
                                       the given X data
        '''

        final_output = [] # initialize final output array

        # go through all given x values
        for i in range(len(X_test)):
            d = [] # initialize array to keep track of distances
            votes = [] # initialize array to keep track of votes

            # go through all x training data that the model is fitted with
            for j in range(len(X_train)):
                # calculate the distance between all the training points
                dist = self.distance(X_train[j] , X_test[i])
                # append the distance
                d.append([dist, j])

            d.sort() # sort the distances in ascending order
            d = d[0:self.k] # take the first k nearest neigbors

            # for all distances and indices left in the distances array
            for d, j in d:
                # append a vote for the corresponding y value
                votes.append(float(y_train.iloc[j]))

            # determine the y value with the most votes
            ans = Counter(votes).most_common(1)[0][0]
            # append our final classification for this value to the array
            final_output.append(ans)

        return final_output

    def score(self, X_test, y_test):
        ''' Determines the accuracy of using the trained model to predict the
            classifications of the given X values against the given true y values.

          Args:
              X_test (np.array): the x values to predict
              y_test (np.array): the actual classifications of the given x values

          Returns:
              accuracy (int): the percentage of correct classifications
        '''
        predictions = self.predict(X_test) # get the y predictions

        # calculate the accuracy
        return (predictions == y_test).sum() / len(y_test)

#https://medium.com/analytics-vidhya/implementing-k-nearest-neighbours-knn-without-usin
```

A function made to perform cross validation:

In [4]:
```
def cv_train(x, y_true):
    """ leave one out cross validation regression of x, y using KNN

    Args:
        x (np.array): input features (star classification features)
        y_true (np.array): output features (types of stars)

    Returns:
        y_pred (np.array): cross validated y predictions based on given data
    """
    # initialize kfold
    n_splits = 10
    kfold = KFold(n_splits=n_splits)

    # initialize knn
    clf = KNN(3)

    y_pred = np.empty_like(y_true)
    for train_idx, test_idx in kfold.split(x):
        # split data
        x_train = x[train_idx, :]
        y_train = y_true.iloc[train_idx]
        x_test = x[test_idx, :]

        # fit classifier
        clf.fit(x_train, y_train)

        # predict
        y_pred[test_idx] = clf.predict(x_test)

    return y_pred
```

Evaluating the accuracy, error and R2 score of KNN on classifying our dataset:

*All accuracies/comparison metrics etc. can always be found printed below the code box

In [5]:
```
scaler = MinMaxScaler()

outcome = df[df.columns[-1]] # get the last column which represents the types
features = scaler.fit_transform(df.drop(df.columns[-1], axis=1))

# apply pca on data
pca = PCA(whiten=True)
features_pca = pca.fit_transform(features)

# split data into training and testing data
X_train, X_test, y_train, y_test = train_test_split(features_pca, outcome, test_size =

# construct KNN classifier
clf = KNN(5)
# fit the classifer with training data
clf.fit(X_train, y_train)
# use the training data to predict the test
prediction = clf.predict(X_test)
# calculate the accuracy of this classifier
score = clf.score(X_test, np.array(y_test).astype(float))
print(f'Accuracy Score: {score}')
```

```
print('Error: ', (1-score))

# apply cross validation
y_pred_cross = cv_train(X_train, y_train)
# compute r2 score
r2 = r2_score(y_true=y_train, y_pred=y_pred_cross)
print(f'R2 score: {r2}')
```

```
Accuracy Score: 0.95
Error:  0.050000000000000044
R2 score: 0.9904761904761905
```

Then we implemented the KNN model using sklearn to compare to our own implementation:

In [6]:

```
clf = KNeighborsClassifier(n_neighbors = 5) # initialize sklearn knn model

clf.fit(X_train, y_train) # fit the sklearn knn

predictions = clf.predict(X_test) # predict using sklearn knn model

score = clf.score(X_test, y_test) # get the accuracy of the sklearn knn model

print(f' Accuracy of sklearn KNN: {score}')
```

```
 Accuracy of sklearn KNN: 0.95
```

The next ML model we chose to implement using sckit learn to compare is **AdaBoost**:

In [7]:

```
#dec_tree = tree.DecisionTreeClassifier() # initialize Decision tree classifier

b_classifier = [1, 5, 10] # represents estimators we are testing

# go through each estimator
for classifier in b_classifier:
    dec_tree = tree.DecisionTreeClassifier() # initialize Decision tree classifier
    # initialize AdaBoost classifier using the current estimator and the
    # decision tree and fit it using the x and y training data from above
    adaboost = AdaBoostClassifier(n_estimators = classifier, base_estimator = dec_tree)
    adaboost_pred_train = adaboost.predict(X_train) # predict with training data
    accuracy = accuracy_score(y_train, adaboost_pred_train) # get the accuracy of using
    f1 = f1_score(y_train, adaboost_pred_train, average = 'macro') # get the f1 score o
    #auc = roc_auc_score(y_train, adaboost_pred_train, average = None)
    print("AB Training set values for", classifier,"value: \n",
          "Error: ", (1 - accuracy), "\n",
          "F1 Score: ", f1 , "\n")
          #"AUC: ", auc, "\n")

    adaboost_pred_test = adaboost.predict(X_test) # predict with testing data
    accuracy = accuracy_score(y_test, adaboost_pred_test) # get the accuracy of using t
    f1 = f1_score(y_test, adaboost_pred_test, average = "macro")  # get the f1 score of
    #auc = roc_auc_score(y_test, adaboost_pred_test)
    print("AB Testing Set values for", classifier,"value: \n",
          "Error: ", (1 - accuracy), "\n",
          "F1 Score: ",f1, "\n")
          #"AUC: ", "{:.12f}".format(auc), "\n ______________________\n")
```

```
AB Training set values for 1 value:
 Error:  0.0
 F1 Score:  1.0

AB Testing Set values for 1 value:
 Error:  0.050000000000000044
 F1 Score:  0.9509704212221376

AB Training set values for 5 value:
 Error:  0.0
 F1 Score:  1.0

AB Testing Set values for 5 value:
 Error:  0.050000000000000044
 F1 Score:  0.9509704212221376

AB Training set values for 10 value:
 Error:  0.0
 F1 Score:  1.0

AB Testing Set values for 10 value:
 Error:  0.06666666666666665
 F1 Score:  0.934470013417382
```

Lastly, we chose to implement **Logistic Regression** from scikit learn as our last ML model for our comparison:

In [9]:

```
# fit a Logistic Regression model using the above x and y training data
log = LogisticRegression(max_iter=10000).fit(X_train, y_train)
print ("Logistic Regression: ")
# create a confusion matrix using the train data
CM_train = confusion_matrix(y_train, log.predict(X_train))
# get the accuracy score using the train data
accuracy = accuracy_score(y_train, log.predict(X_train))
# get the error using the train data
error = 1 - accuracy
# get the precision using the train data
precision = precision_score(y_train, log.predict(X_train), average = 'macro')
# get the recall using the train data
recall = recall_score(y_train, log.predict(X_train), average = 'macro')
print("Training Data: \n Accuracy: ", accuracy, "\n", "Error: ", error, "\n Precision:

# get the confusion matrix using the test data
CM_test = confusion_matrix(y_test, log.predict(X_test))
# get the accuracy score using the test data
accuracy = accuracy_score(y_test, log.predict(X_test))
# get the error using the accuracy from the test data
error = 1 - accuracy
# get the precision using the test data
precision = precision_score(y_test, log.predict(X_test), average = 'macro')
# get the recall using the test data
recall = recall_score(y_test, log.predict(X_test), average = 'macro')
print("Testing Data: \n Accuracy: ", accuracy, "\n", "Error: ", error, "\n Precision: "
```

```
Logistic Regression:
Training Data:
 Accuracy:  0.9888888888888889
 Error:  0.011111111111111072
```

```
 Precision:  0.989068100358423
 Recall:  0.9888888888888889

Testing Data:
 Accuracy:  0.9666666666666667
 Error:  0.033333333333333326
 Precision:  0.9722222222222223
 Recall:  0.9666666666666667
```