Boosting and Stacking

Introduction

We will be using the Human Activity Recognition with Smartphones database, which was built from the recordings of study participants performing activities of daily living (ADL) while carrying a smartphone with an embedded inertial sensors. The objective is to classify activities into one of the six activities (walking, walking upstairs, walking downstairs, sitting, standing, and laying) performed.

For each record in the dataset it is provided:

- Triaxial acceleration from the accelerometer (total acceleration) and the estimated body acceleration.
- Triaxial angular velocity from the gyroscope.
- A 561-feature vector with time and frequency domain variables.
- Its activity label.

More information about the features is available on the website above.

In [1]:
```
import pandas as pd, numpy as np, matplotlib.pyplot as plt, os, sys, seaborn as sns
```

- Importing the data from the file `Human_Activity_Recognition_Using_Smartphones_Data.csv` and examine the shape and data types. For the data types, there will be too many to list each column separately. Rather, aggregate the types by count.
- Determine if the float columns need to be scaled.

In [2]:
```
filepath = 'Human_Activity_Recognition_Using_Smartphones_Data.csv'
data = pd.read_csv("C:\\Users\\rsnen\\Desktop\\IBM\\Human_Activity_Recognition_Using_Smartphones_Data.csv", sep=',')
```

The data has quite a few predictor columns.

In [3]:
```
data.shape
```

Out[3]:
```
(10299, 562)
```

And they're all float values. The only non-float is the categories column, which is what's being predicted.

In [4]:
```
data.dtypes.value_counts()
```

Out[4]:
```
float64    561
object       1
dtype: int64
```

The minimum and maximum value for the float columns is -1.0 and 1.0, respectively. However, scaling is never required for tree-based methods.

In [9]:
```
# create a panda series that will just tell us for each column whether or not they are a float.
data.dtypes == np.float
```

Out[9]:
```
tBodyAcc-mean()-X                     True
tBodyAcc-mean()-Y                     True
tBodyAcc-mean()-Z                     True
tBodyAcc-std()-X                      True
tBodyAcc-std()-Y                      True
                                     ...
angle(tBodyGyroJerkMean,gravityMean)     True
angle(X,gravityMean)                     True
angle(Y,gravityMean)                     True
angle(Z,gravityMean)                     True
Activity                                False
Length: 562, dtype: bool
```

locate every single row that's going to be our cos-linear, and then the float columns, only the columns that were already a float, and then check the max value, and this will by default check the max value for each one of those different columns.So once we get that max value, we then check that it's equal to 1. So now we have true or false for every single one of our different columns that are floats that is, and then we call.all, and that will just say, is every single value true? So if you have one false in there, this will return false. If they are all true, then it will return true.

In [12]:
```
float_columns = (data.dtypes == np.float)
(data.loc[:,float_columns].max()==1.0)
```

Out[12]:
```
tBodyAcc-mean()-X                     True
tBodyAcc-mean()-Y                     True
tBodyAcc-mean()-Z                     True
tBodyAcc-std()-X                      True
tBodyAcc-std()-Y                      True
                                     ...
angle(tBodyGyroMean,gravityMean)      True
angle(tBodyGyroJerkMean,gravityMean)     True
angle(X,gravityMean)                     True
angle(Y,gravityMean)                     True
angle(Z,gravityMean)                     True
Length: 561, dtype: bool
```

In [13]:
```
#same like above do with min(-1)
float_columns = (data.dtypes == np.float)
(data.loc[:,float_columns].min()==1.0)
```

Out[13]:
```
tBodyAcc-mean()-X                     False
tBodyAcc-mean()-Y                     False
tBodyAcc-mean()-Z                     False
tBodyAcc-std()-X                      False
tBodyAcc-std()-Y                      False
                                      ...
angle(tBodyGyroMean,gravityMean)      False
angle(tBodyGyroJerkMean,gravityMean)     False
angle(X,gravityMean)                     False
angle(Y,gravityMean)                     False
angle(Z,gravityMean)                     False
Length: 561, dtype: bool
```

In [14]:
```
# above 3 steps in cell

# Mask to select float columns
float_columns = (data.dtypes == np.float)

# Verify that the maximum of all float columns is 1.0
print( (data.loc[:,float_columns].max()==1.0).all() )

# Verify that the minimum of all float columns is -1.0
print( (data.loc[:,float_columns].min()==-1.0).all() )
```

```
True
True
```

- Integer encode the activities.
- Split the data into train and test data sets. Decide if the data will be stratified or not during the train/test split.

In [16]:
```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

data['Activity'] = le.fit_transform(data['Activity'])

le.classes_
```

Out[16]:
```
array([0, 1, 2, 3, 4, 5])
```

In [17]:
```
data.Activity.unique()
```

Out[17]:
```
array([2, 1, 0, 3, 4, 5], dtype=int64)
```

We are about to create training and test sets from `data`. On those datasets, we are going to run grid searches over many choices of parameters. This can take some time. In order to shorten the grid search time,scale down the size of your train and test sets, or especially your train sets, in order to ensure that it trains a bit quicker. (i.e) downsample `data` and create `X_train, X_test, y_train, y_test` from the downsampled dataset. split the data into train and test data sets.

In [19]:
```
from sklearn.model_selection import train_test_split

# Alternatively, we could stratify the categories in the split, as was done previously
feature_columns = [x for x in data.columns if x != 'Activity']

X_train, X_test, y_train, y_test = train_test_split(data[feature_columns], data['Activity'],
                    test_size=0.3, random_state=42)
```

In [21]:
```
X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

Out[21]:
```
((7209, 561), (7209,), (3090, 561), (3090,))
```

- Fitting gradient boosted tree models with all parameters set to their defaults the following tree numbers (`n_estimators = [15, 25, 50, 100, 200, 400]`) and evaluate the accuracy on the test data for each of these models.
- Plotting the accuracy as a function of estimator number.

Simply create the model inside the `for` loop and set the number of estimators at this time. This will make the fitting take a little longer. Additionally, boosting models tend to take longer to fit than bagged ones because the decision stumps must be fit successively.

In [22]:
```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score

error_list = list()

# Iterate through various possibilities for number of trees
tree_list = [15, 25, 50, 100, 200, 400] #takes too long to run
for n_trees in tree_list:

    # Initialize the gradient boost classifier
    GBC = GradientBoostingClassifier(n_estimators=n_trees, random_state=42)

    # Fit the model
    print(f'Fitting model with {n_trees} trees')
    GBC.fit(X_train.values, y_train.values)
    y_pred = GBC.predict(X_test)

    # Get the error
    error = 1.0 - accuracy_score(y_test, y_pred)

    # Store it
    error_list.append(pd.Series({'n_trees': n_trees, 'error': error}))

error_df = pd.concat(error_list, axis=1).T.set_index('n_trees')

error_df
```

```
Fitting model with 15 trees
Fitting model with 25 trees
Fitting model with 50 trees
Fitting model with 100 trees
Fitting model with 200 trees
Fitting model with 400 trees
```

Out[22]:

	error
n_trees	
15.0	0.051133
25.0	0.033981
50.0	0.019417
100.0	0.013592
200.0	0.011003
400.0	0.010356

plotting the result.

In [25]:
```
sns.set_context('talk')
sns.set_style('white')


# Create the plot
ax = error_df.plot(marker='o', figsize=(12, 8), linewidth=5)

# Set parameters
ax.set(xlabel='Number of Trees', ylabel='Error')
ax.set_xlim(0, max(error_df.index)*1.1);
```