cs109a_hw8_109_submit

November 15, 2018

1 CS109A Introduction to Data Science:

1.1 Homework 8: Ensembles: Bagging, Random Forests, and Boosting

Harvard University Fall 2018 Instructors: Pavlos Protopapas, Kevin Rader

```
In [1]: #RUN THIS CELL
        import requests
        from IPython.core.display import HTML
        styles = requests.get("https://raw.githubusercontent.com/Harvard-IACS/2018-CS109A/mast
        HTML(styles)

Out[1]: <IPython.core.display.HTML object>
```

1.1.1 INSTRUCTIONS

- To submit your assignment follow the instructions given in Canvas.
- If needed, clarifications will be posted on Piazza.
- This homework can be submitted in pairs.
- If you submit individually but you have worked with someone, please include the name of your **one** partner below.

Name of the person you have worked with goes here:

1.2 Learning Objectives

Completing this assignment will demonstrate success at the following objectives:

- Statistical
- Predict when bagging will help model performance.
- Identify how Random Forests improve over bagging.
- Predict when boosting will help model performance.
- Compare and contrast bagging and boosting.
- Coding
- Identify and fix problems in poorly written code
- Communication
- Visually explain a complex concept

```
In [2]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt

        from sklearn.model_selection import cross_val_score
        from sklearn.utils import resample
        from sklearn import tree
        from sklearn.tree import DecisionTreeClassifier
        from sklearn.ensemble import RandomForestClassifier
        from sklearn.ensemble import AdaBoostClassifier
        from sklearn.metrics import accuracy_score

        %matplotlib inline

        import seaborn as sns
        sns.set(style='whitegrid')
        pd.set_option('display.width', 1500)
        pd.set_option('display.max_columns', 100)
```

Overview: Higgs Boson Discovery

The discovery of the Higgs boson in July 2012 marked a fundamental breakthrough in particle physics. The Higgs boson particle was discovered through experiments at the Large Hadron Collider at CERN, by colliding beams of protons at high energy. A key challenge in analyzing the results of these experiments is to differentiate between collisions that produce Higgs bosons and collisions that produce only background noise. We shall explore the use of ensemble methods for this classification task.

You are provided with data from Monte-Carlo simulations of collisions of particles in a particle collider experiment. The training set is available in `Higgs_train.csv` and the test set is in `Higgs_test.csv`. Each row in these files corresponds to a particle collision described by 28 features (columns 1-28), of which the first 21 features are kinematic properties measured by the particle detectors in the accelerator, and the remaining features are derived by physicists from the first 21 features. The class label is provided in the last column, with a label of 1 indicating that the collision produces Higgs bosons (signal), and a label of 0 indicating that the collision produces other particles (background).

The data set provided to you is a small subset of the HIGGS data set in the UCI machine learning repository. The following paper contains further details about the data set and the predictors used: Baldi et al., Nature Communications 5, 2014.

```
In [3]: # Load data
        data_train = pd.read_csv('data/Higgs_train.csv')
        data_test = pd.read_csv('data/Higgs_test.csv')

        print(f"{len(data_train)} training samples, {len(data_test)} test samples")
        print("\nColumns:")
        print(', '.join(data_train.columns))
```

```
5000 training samples, 5000 test samples
```

```
Columns:
lepton pT, lepton eta, lepton phi, missing energy magnitude, missing energy phi, jet 1 pt, jet
```

```
In [4]: display(data_train.head())
        display(data_train.describe())
```

```
   lepton pT  lepton eta  lepton phi  missing energy magnitude  missing energy phi  jet 1 pt
0      0.377     -1.5800     -1.7100                     0.991               0.114     1.250
1      0.707      0.0876     -0.4000                     0.919              -1.230     1.170
2      0.617      0.2660     -1.3500                     1.150               1.040     0.955
3      0.851     -0.3810     -0.0713                     1.470              -0.795     0.692
4      0.768     -0.6920     -0.0402                     0.615               0.144     0.749
```

```
         lepton pT   lepton eta   lepton phi  missing energy magnitude  missing energy phi
count  5000.000000  5000.000000  5000.000000               5000.000000         5000.000000  50
mean      0.978645    -0.014280    -0.018956                  1.005793            0.002528
std       0.547025     1.011927     0.997945                  0.591907            1.003337
min       0.275000    -2.410000    -1.740000                  0.010000           -1.740000
25%       0.587000    -0.764250    -0.877500                  0.581000           -0.870000
50%       0.846000    -0.009305    -0.016050                  0.903500            0.001300
75%       1.220000     0.725500     0.837000                  1.300000            0.866000
max       5.330000     2.430000     1.740000                  6.260000            1.740000
```

```
In [5]: # Split into NumPy arrays
        X_train = data_train.iloc[:, data_train.columns != 'class'].values
        y_train = data_train['class'].values
        X_test = data_test.iloc[:, data_test.columns != 'class'].values
        y_test = data_test['class'].values

In [6]: X_train_columns = data_train.columns.drop('class')
        display(X_train_columns)
```

```
Index(['lepton pT', 'lepton eta', 'lepton phi', 'missing energy magnitude', 'missing energy ph
```

Question 1: A Single Model [20 pts]

We start by fitting a basic model we can compare the other models to. We'll pick a decision tree as the base model, because we'll later include random forests and want a fair comparison. We'll tune the decision tree using cross-validation. As usual, we'll be tuning the maximum tree depth; we refer to this parameter as "depth" for simplicity.

Since we will only be using tree-based methods in this homework, we do not need to standardize or normalize the predictors.

1.1: Fit a decision tree model to the training set. Choose a range of tree depths to evaluate. Plot the estimated performance +/- 2 standard deviations for each depth using 5-fold cross validation. Also include the training set performance in your plot, but set the y-axis to focus on the cross-validation performance.

Hint: use `plt.fill_between` to shade the region.

1.2 Select an appropriate depth and justify your choice. Using your cross-validation estimates, report the mean +/- 2 stdev. Then report the classification accuracy on the test set. (Store the training and test accuracies in variables to refer to in a later question.)

1.3 What is the mechanism by which limiting the depth of the tree avoids over-fitting? What is one downside of limiting the tree depth? Your answer should refer to the bias-variance trade-off.

Answers

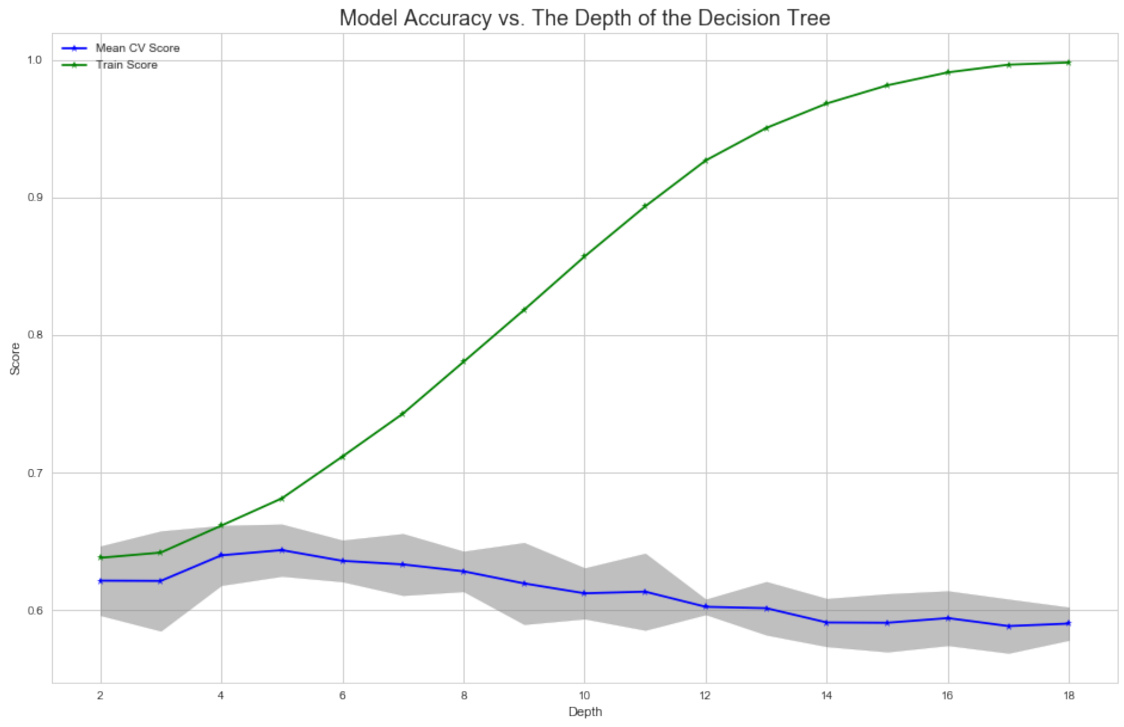
1.1

```
In [7]: # your code here
        for i in range(2,11):
            cls = DecisionTreeClassifier(max_depth = i)
            score = cross_val_score(cls, X_train, y_train, cv=5)
            mean = (score[0]+score[1]+score[2]+score[3]+score[4])/5
            words = "The mean is: "
            print(i, score, words, mean)
```

```
2 [0.61738262 0.6003996  0.638      0.62862863 0.62262262] The mean is:  0.6214066938066938
3 [0.61738262 0.58741259 0.631      0.63563564 0.63463463] The mean is:  0.621213095013095
4 [0.63736264 0.62237762 0.637      0.64864865 0.65365365] The mean is:  0.6398085124085124
5 [0.63636364 0.63236763 0.655      0.64964965 0.63963964] The mean is:  0.6426041116041116
6 [0.62637363 0.63436563 0.628      0.63863864 0.64764765] The mean is:  0.6350051094051093
7 [0.62437562 0.63136863 0.632      0.65165165 0.63063063] The mean is:  0.6340053076053076
8 [0.61138861 0.63436563 0.626      0.63863864 0.62862863] The mean is:  0.6278043026043025
9 [0.5974026  0.63536464 0.61      0.60760761 0.63763764] The mean is:  0.6176024956024956
10 [0.60839161 0.6023976  0.611      0.60860861 0.61261261] The mean is:  0.6086020864020865
```

```
In [8]: # your code here
        depths = [2,3,4,5,6,7,8,9,10, 11, 12, 13, 14,15, 16, 17,18]
        scores_train = []
        mean_CV_score = []
        CV_sd = []
        for depth in depths:
            dt = tree.DecisionTreeClassifier(criterion = 'gini',max_depth = depth)
            dt.fit(X_train, y_train)
            cv_score = cross_val_score(estimator=dt, X=X_train, y=y_train, cv=5)
            mean_score = np.mean(cv_score)
            mean_CV_score.append(mean_score)
            stdev = np.std(cv_score)
            CV_sd.append(stdev)
            scores_train.append(dt.score(X_train, y_train))

        upper = []
        lower = []
        for i in range(len(depths)):
            up = mean_CV_score[i]+2*CV_sd[i]
            low = mean_CV_score[i]-2*CV_sd[i]
```

```
    upper.append(up)
    lower.append(low)


fig, ax = plt.subplots(figsize=(16,10))
plt.plot(depths, mean_CV_score, 'b*-', label = 'Mean CV Score')
plt.plot(depths, scores_train, 'g*-', label = 'Train Score')
plt.fill_between(depths,upper, lower, color = 'grey', alpha = 0.5)
plt.xlabel('Depth')
plt.ylabel('Score')
plt.title('Model Accuracy vs. The Depth of the Decision Tree', size = 18)
plt.legend();
```



1.2

```
In [9]: # Selecting a depth of 5
        cls = DecisionTreeClassifier(max_depth = 5)
        cls.fit(X_train, y_train)
        train_score = cross_val_score(cls, X_train, y_train, cv=5)
        train_CV_mean = np.mean(train_score)
        stdev = np.std(score)
        print(train_score, train_CV_mean, stdev)

[0.63436563 0.63136863 0.656      0.64964965 0.63963964] 0.6422047110047111 0.0034748892184455
```

```
In [10]: test_score = cls.score(X_test, y_test)
         stdev = np.std(score)
         print(test_score, stdev)
```

```
0.6478 0.003474889218445574
```

Answer We selected a depth of 5. It appears to give us the highest mean CV score with the smallest standard deviations. With a depth of 5, the model achieves a score of 0.647 on the test data.

1.3

The mechanism we use is the manual inspection of the CV score and standard deviation displayed on the figure in (1.1). If we had only considered the training set performance, the tree would continue to grow until it had enough perturbations to capture every training set data point correctly. This would result in overfitting.
We could continue to add depth to our tree, which would result in a low-bias/high variance model. We don't want our model to be overfit to our training data (high variance), so we accept a certain level of bias in order to avoid overfitting.

Question 2: Bagging [25 pts]

Bagging is the technique of building the same model on multiple bootstraps from the data and combining each model's prediction to get an overall classification. In this question we build an example by hand and study how the number of bootstrapped datasets impacts the accuracy of the resulting classification.

2.1 Choose a tree depth that will overfit the training set. What evidence leads you to believe that this depth will overfit? Assign your choice to a variable here. (You may want to explore different settings for this value in the problems below.)

2.2 Create 45 bootstrapped replications of the original training data, and fit a decision tree to each. Use the tree depth you just chose in 2.1. Record each tree's prediction. In particular, produce a dataset like those below, where each row is a training (or test) example, each column is one of the trees, and each entry is that tree's prediction for that example. (Labeling the rows and columns is optional.)

Store these results as `bagging_train` and `bagging_test`. Don't worry about visualizing these results yet.

2.3 *Aggregate* all 45 *bootstrapped* models to get a combined prediction for each training and test point: predict a 1 if and only if a majority of the models predict that example to be from class 1. What accuracy does this *bagging* model achieve on the test set? Write an assertion that verifies that this test-set accuracy is at least as good as the accuracy for the model you fit in Question 1.

2.4 We want to know how the number of bootstraps affects our bagging ensemble's performance. Use the `running_predictions` function (given below) to get the model's accuracy score when using only 1,2,3,4,... of the bootstrapped models. Make a plot of training and test set accuracies as a function of number of bootstraps.

On your plot, also include horizontal lines for two baselines: - the test accuracy of the best model from question 1 - the test accuracy of a single tree with the tree depth you chose in 2.1, trained on the full training set.

2.5 Referring to your graph from 2.4, compare the performance of bagging against the baseline of a single depth from 2.1 tree. Explain the differences you see.

2.6 Bagging and limiting tree depth both affect how much the model overfits. Compare and contrast these two approaches. Your answer should refer to your graph in 2.4 and may duplicate something you said in your answer to 1.5.

Bagging variance reduction, limiting depth alters both bias and variance

2.7: In what ways might our bagging classifier be overfitting the data? In what ways might it be underfitting?

Hints - Use `resample` from sklearn to easily bootstrap the x and y data. - use `np.mean` to easily test for majority. If a majority of models vote 1, what does that imply about the mean?

Answers:

2.1

Answer: A tree depth of 18 would overfit the data. We know this because at a depth of 18, the accuracy score on the training data is 1.

```
In [11]: # Selecting a depth of 18
         cls = DecisionTreeClassifier(max_depth = 18)
         cls.fit(X_train, y_train)
         train_score = cls.score(X_train, y_train)
         train_CV_mean = np.mean(train_score)
         stdev = np.std(score)
         print(train_score, train_CV_mean, stdev)
```

```
0.9984 0.9984 0.003474889218445574
```

```
In [12]: test_score = cls.score(X_test, y_test)
         stdev = np.std(score)
         print(test_score, stdev)
```

```
0.6 0.003474889218445574
```

```
In [13]: #Set 18 as the tree depth
         depth_over = 18
```

2.2

Structure of `bagging_train` and `bagging_test`:

`bagging_train`:

	bootstrap model 1's prediction	bootstrap model 2's prediction	...
training row 1	binary value	binary value	...
training row 2	binary value	binary value	...
...	...	...	...

`bagging_test`:

	bootstrap model 1's prediction	bootstrap model 2's prediction	...
test row 1	binary value	binary value	...
test row 2	binary value	binary value	...
...	...	...	...

```
In [14]: # your code here
         samples = 5000
         train_rows = []
         test_rows = []
         columns = []
         for i in range (1, samples+1):
             words_1 = "training row %s" %i
             words_2 = "test row %s" %i
             train_rows.append(words_1)
             test_rows.append(words_2)
         for i in range (1,46):
             words_3 = "bootstrap model %s's prediction" %i
             columns.append(words_3)

         #Comment
         data_train = np.zeros(shape=(samples,45))
         data_test = np.zeros(shape=(samples,45))
         for i in range (0,45):
             X, y = resample(X_train, y_train, n_samples = samples)
             dt = tree.DecisionTreeClassifier(max_depth = depth_over)
             dt.fit(X,y)
             y_pred_train = dt.predict(X_train[0:samples])
             y_pred_test = dt.predict(X_test[0:samples])
             data_train[:,i]=y_pred_train
             data_test[:,i]=y_pred_test
         bagging_train = pd.DataFrame(data_train, index = train_rows, columns = columns)
         bagging_test = pd.DataFrame(data_test, index = test_rows, columns = columns)

In [15]: display(bagging_train, bagging_test)
```

```
                   bootstrap model 1's prediction  bootstrap model 2's prediction  bootstrap mo
training row 1                                0.0                             1.0
training row 2                                1.0                             1.0
training row 3                                0.0                             1.0
training row 4                                1.0                             1.0
training row 5                                1.0                             0.0
training row 6                                1.0                             1.0
training row 7                                1.0                             0.0
training row 8                                1.0                             1.0
training row 9                                1.0                             1.0
training row 10                               1.0                             0.0
training row 11                               0.0                             1.0
training row 12                               1.0                             1.0
training row 13                               0.0                             0.0
training row 14                               0.0                             0.0
training row 15                               1.0                             1.0
training row 16                               0.0                             0.0
training row 17                               0.0                             1.0
training row 18                      0.0                0.0
training row 19                      1.0                1.0
training row 20                      0.0                0.0
training row 21                      1.0                1.0
training row 22                      0.0                0.0
training row 23                      0.0                1.0
training row 24                      1.0                1.0
training row 25                      0.0                1.0
training row 26                      1.0                0.0
training row 27                      0.0                1.0
training row 28                      1.0                1.0
training row 29                      1.0                1.0
training row 30                      1.0                1.0
...                                  ...                ...
training row 4971                    1.0                1.0
training row 4972                    0.0                0.0
training row 4973                    1.0                0.0
training row 4974                    0.0                0.0
training row 4975                    0.0                0.0
training row 4976                    1.0                1.0
training row 4977                    1.0                1.0
training row 4978                    0.0                0.0
training row 4979                    0.0                0.0
training row 4980                    1.0                1.0
training row 4981                    1.0                1.0
training row 4982                    0.0                0.0
training row 4983                    0.0                0.0
training row 4984                    0.0                0.0
training row 4985                    1.0                0.0
training row 4986                    1.0                0.0
training row 4987                    1.0                1.0
training row 4988                    1.0                1.0
training row 4989                    1.0                1.0
training row 4990                    1.0                1.0
training row 4991                    1.0                1.0
training row 4992                    1.0                1.0
training row 4993                    1.0                1.0
training row 4994                    1.0                1.0
training row 4995                    1.0                1.0
training row 4996                    1.0                1.0
training row 4997                    1.0                1.0
training row 4998                    0.0                1.0
training row 4999                    1.0                0.0
training row 5000                    1.0                1.0

[5000 rows x 45 columns]
```

```
               bootstrap model 1's prediction  bootstrap model 2's prediction  bootstrap model
test row 1                                0.0                             1.0
test row 2                                1.0                             1.0
test row 3                                0.0                             0.0
test row 4                                1.0                             1.0
test row 5                                1.0                             1.0
test row 6                                1.0                             1.0
test row 7                                0.0                             0.0
test row 8                                0.0                             0.0
test row 9                                0.0                             0.0
test row 10                               0.0                             1.0
test row 11                               1.0                             1.0
test row 12                               1.0                             1.0
test row 13                               0.0                             0.0
test row 14                               1.0                             1.0
test row 15                               1.0                             1.0
test row 16                               1.0                             1.0
test row 17                               1.0                             0.0
test row 18                               1.0                             0.0
test row 19                               1.0                             1.0
test row 20                               1.0                             1.0
test row 21                               1.0                             1.0
test row 22                               1.0                             1.0
test row 23                               1.0                             1.0
test row 24                               1.0                             1.0
test row 25                               1.0                             0.0
test row 26                               0.0                             0.0
test row 27                               1.0                             0.0
test row 28                               1.0                             0.0
test row 29                               1.0                             1.0
test row 30                               0.0                             0.0
...                                       ...                             ...
test row 4971                             0.0                             1.0
test row 4972                             1.0                             0.0
test row 4973                             0.0                             0.0
test row 4974                             0.0                             1.0
test row 4975                             1.0                             0.0
test row 4976                             1.0                             0.0
test row 4977                             0.0                             0.0
test row 4978                             1.0                             1.0
test row 4979                             1.0                             0.0
test row 4980                             1.0                             1.0
test row 4981                             1.0                             0.0
test row 4982                             1.0                             1.0
test row 4983                             0.0                             1.0
test row 4984                             1.0                             1.0
test row 4985                             1.0                             1.0
test row 4986                             1.0                             1.0
test row 4987                                  1.0                            1.0
test row 4988                                  1.0                            1.0
test row 4989                                  1.0                            1.0
test row 4990                                  0.0                            0.0
test row 4991                                  0.0                            0.0
test row 4992                                  1.0                            1.0
test row 4993                                  1.0                            0.0
test row 4994                                  0.0                            0.0
test row 4995                                  0.0                            1.0
test row 4996                                  1.0                            1.0
test row 4997                                  0.0                            0.0
test row 4998                                  0.0                            1.0
test row 4999                                  1.0                            0.0
test row 5000                                  1.0                            0.0

[5000 rows x 45 columns]
```

2.3: *Aggregate* all 45 *bootstrapped* models to get a combined prediction for each training and test point: predict a 1 if and only if a majority of the models predict that example to be from class 1. What accuracy does this *bagging* model achieve on the test set? Write an assertion that verifies that this test-set accuracy is at least as good as the accuracy for the model you fit in Question 1.

```
In [16]: # your code here
         x_agg_train = np.array(np.round(bagging_train.sum(axis=1)/45))
         x_agg_test = np.array(np.round(bagging_test.sum(axis=1)/45))
         # display(x_agg_train)
         # display(y_train)

         train_acc_score = accuracy_score(x_agg_train, y_train[0:samples])
         test_acc_score = accuracy_score(x_agg_test, y_test[0:samples])

         train_words="Train score: "
         test_words = "Test score: "
         display(train_words, train_acc_score)
         display(test_words, test_acc_score)
         # For rows with mean value > .5, classify as 1;
```

```
'Train score: '


0.9976


'Test score: '


0.6912
```

```
In [17]: assert test_acc_score > test_score
```

Answer: The bagging model achieves an accuracy of 0.68, which is greater than the score of 0.64 when the original model from 1.1 was run on the test data.

2.4 We want to know how the number of bootstraps affects our bagging ensemble's performance. Use the running_predictions function (given below) to get the model's accuracy score when using only 1,2,3,4,... of the bootstrapped models. Make a plot of training and test set accuracies as a function of number of bootstraps.

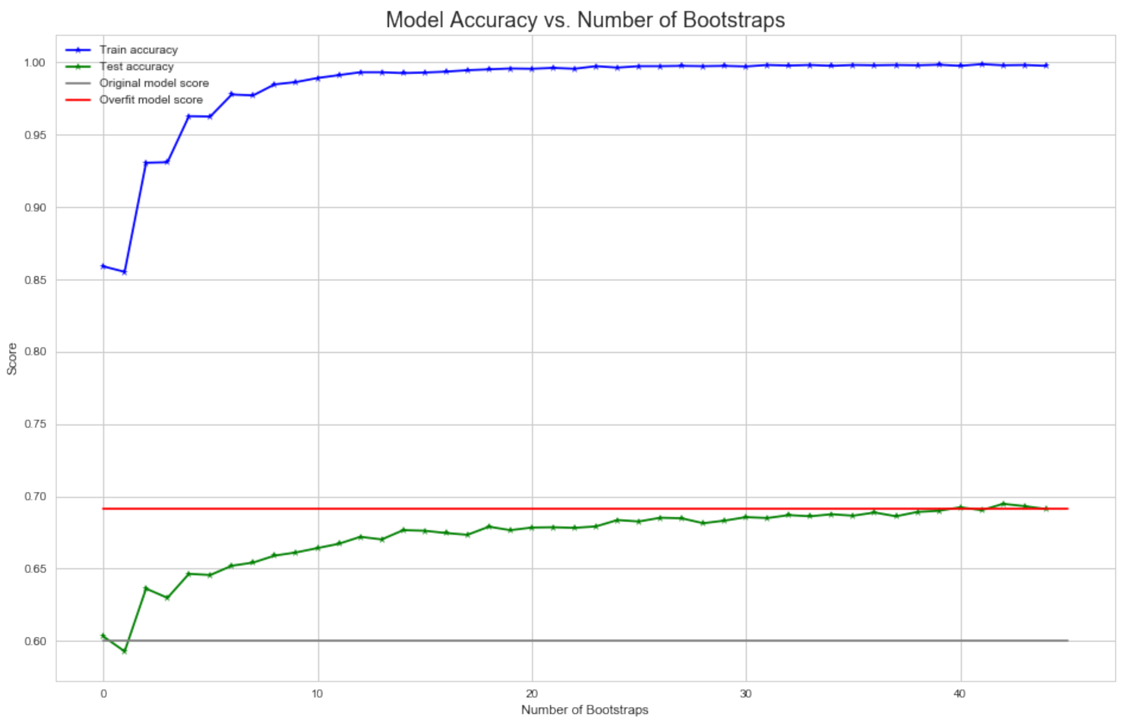
On your plot, also include horizontal lines for two baselines: - the test accuracy of the best model from question 1 - the test accuracy of a single tree with the tree depth you chose in 2.1, trained on the full training set.

```
In [18]: def running_predictions(prediction_dataset, targets):
             """A function to predict examples' class via the majority among trees (ties are p

             Inputs:
               prediction_dataset - a (n_examples by n_sub_models) dataset, where each entry [
                   for example i
               targets - the true class labels

             Returns:
               a vector where vec[i] is the model's accuracy when using just the first i+1 sub
             """

             n_trees = prediction_dataset.shape[1]

             # find the running percentage of models voting 1 as more models are considered
             running_percent_1s = np.cumsum(prediction_dataset, axis=1)/np.arange(1,n_trees+1)

             # predict 1 when the running average is above 0.5
             running_conclusions = running_percent_1s > 0.5

             # check whether the running predictions match the targets
             running_correctnesss = running_conclusions == targets.reshape(-1,1)

             return np.mean(running_correctnesss, axis=0)
             # returns a 1-d series of the accuracy of using the first n trees to predict the

In [19]: # your code here
         vals = list(range(0,45))
         train_accuracy = running_predictions(bagging_train.values, y_train)
         test_accuracy = running_predictions(bagging_test.values, y_test)

         display(train_accuracy, test_accuracy)
```

```
array([0.859 , 0.8552, 0.9306, 0.931 , 0.9628, 0.9626, 0.9778, 0.9772,
       0.9848, 0.9864, 0.9892, 0.9912, 0.9932, 0.9932, 0.9926, 0.993 ,
       0.9936, 0.9946, 0.9952, 0.9958, 0.9956, 0.9962, 0.9956, 0.9974,
       0.9964, 0.9974, 0.9974, 0.9976, 0.9974, 0.9976, 0.9972, 0.9982,
       0.9978, 0.9982, 0.9976, 0.9982, 0.998 , 0.9982, 0.998 , 0.9984,
       0.9976, 0.9988, 0.998 , 0.9982, 0.9976])
```

```
array([0.6032, 0.5928, 0.6362, 0.6298, 0.6464, 0.6456, 0.652 , 0.6542,
       0.659 , 0.6612, 0.6642, 0.6672, 0.672 , 0.6702, 0.6766, 0.6762,
       0.6746, 0.6734, 0.679 , 0.6766, 0.6784, 0.6786, 0.6782, 0.6792,
       0.6836, 0.6826, 0.6852, 0.6848, 0.6814, 0.6832, 0.6856, 0.685 ,
       0.687 , 0.6862, 0.6876, 0.6866, 0.6888, 0.6862, 0.6892, 0.69  ,
       0.6924, 0.6904, 0.6948, 0.6932, 0.6912])
```

```
In [20]: fig, ax = plt.subplots(figsize=(16,10))
         plt.plot(vals, train_accuracy, 'b*-', label = 'Train accuracy')
         plt.plot(vals, test_accuracy, 'g*-', label = 'Test accuracy')
         plt.plot((0,45), (test_score,test_score), color = 'grey', label = 'Original model sco
         plt.plot((0,45), (test_acc_score, test_acc_score), color = 'red', label = 'Overfit mod
         plt.xlabel('Number of Bootstraps')
         plt.ylabel('Score')
         plt.title('Model Accuracy vs. Number of Bootstraps', size = 18)
         plt.legend();
```



2.5 Referring to your graph from 2.4, compare the performance of bagging against the baseline of a single depth from 2.1 tree. Explain the differences you see.

Answer: Immediately, we see that the training score begins at a lower value but quicky rises to a threshold near the single-depth tree (although, interestingly, it never achieves the same 0.98

score!). However, the improved accuracy on the test score is significant. After two bootstraps, the bagging model's test score is already higher than the single-depth score. This tells us that bagging reduces variance and improves the model's overall performance.

2.6 Bagging and limiting tree depth both affect how much the model overfits. Compare and contrast these two approaches. Your answer should refer to your graph in 2.4 and may duplicate something you said in your answer to 1.5.

Answere Bagging results in variance reduction, while limiting depth alters both bias and variance. We can see in the graph in 2.4 that bagging resulted in a slightly lower training score, but a much higher test score. Limiting tree depth will result in a high-bias/low variance model. While we want to achieve lower variance, we do not want high bias.

2.7: In what ways might our bagging classifier be overfitting the data? In what ways might it be underfitting?

Answer Bagging may result in overfitting if the trees contained in the model are too large. When there are multiple large trees, it is easy for models to become correlated. It may be underfitting if the trees are too small. This is why it is important to explore optimal tree depth before creating a bagging model.

Question 3: Random Forests [15 pts]

Random Forests are closely related to the bagging model we built by hand in question 2. In this question we compare our by-hand results with the results of using `RandomForestClassifier` directly.

3.1 Fit a `RandomForestClassifier` to the original `X_train` data using the same tree depth and number of trees that you used in Question 2.2. Evaluate its accuracy on the test set.

3.2 For each of the decision trees you fit in the bagging process, how many times is each feature used at the top node? How about for each tree in the random forest you just fit? What about the process of training the Random Forest causes this difference? What implication does this observation have on the accuracy of bagging vs Random Forest?

Hint: A decision tree's top feature is stored in `model.tree_.feature[0]`. A random forest object stores its decision trees in its `.estimators_` attribute.

3.3: Make a table of the training and test accuracy for the following models:

- Single tree with best depth chosen by cross-validation (from Question 1)
- A single overfit tree trained on all data (from Question 2, using the depth you chose there)
- Bagging 45 such trees (from Question 2)
- A Random Forest of 45 such trees (from Question 3.1)

(This problem should not require fitting any new models, though you may need to go back and store the accuracies from models you fit previously.)

What is the relative performance of each model on the training set? On the test set? Comment on how these relationships make sense (or don't make sense) in light of how each model treats the bias-variance trade-off.

Answers:

3.1

```
In [21]: # your code here
         # your code here
         train_rows = []
         test_rows = []
         columns = []
```

```
        for i in range (1, 5001):
            words_1 = "training row %s" %i
            words_2 = "test row %s" %i
            train_rows.append(words_1)
            test_rows.append(words_2)
        for i in range (1,46):
            words_3 = "bootstrap model %s's prediction" %i
            columns.append(words_3)

        #Comment
        data_train = np.zeros(shape=(5000,45))
        data_test = np.zeros(shape=(5000,45))
        for i in range (0,45):
            X, y = resample(X_train, y_train, n_samples = 5000)
            rf = RandomForestClassifier(max_depth = 17)
            rf.fit(X,y)
            y_pred_train = dt.predict(X)
            y_pred_test = dt.predict(X_test)
            data_train[:,i]=y_pred_train
            data_test[:,i]=y_pred_test

        rf_train = pd.DataFrame(data_train, index = train_rows, columns = columns)
        rf_test = pd.DataFrame(data_test, index = test_rows, columns = columns)


        train_score = cross_val_score(rf, rf_train, y, cv=5)
        test_score = cross_val_score(rf, rf_test, y_test, cv=5)

        rf_train_accuracy = (train_score[0]+train_score[1]+train_score[2]+train_score[3]+trai
        rf_test_accuracy = (test_score[0]+test_score[1]+test_score[2]+test_score[3]+test_scor

        display(rf_train_accuracy, rf_test_accuracy)
```

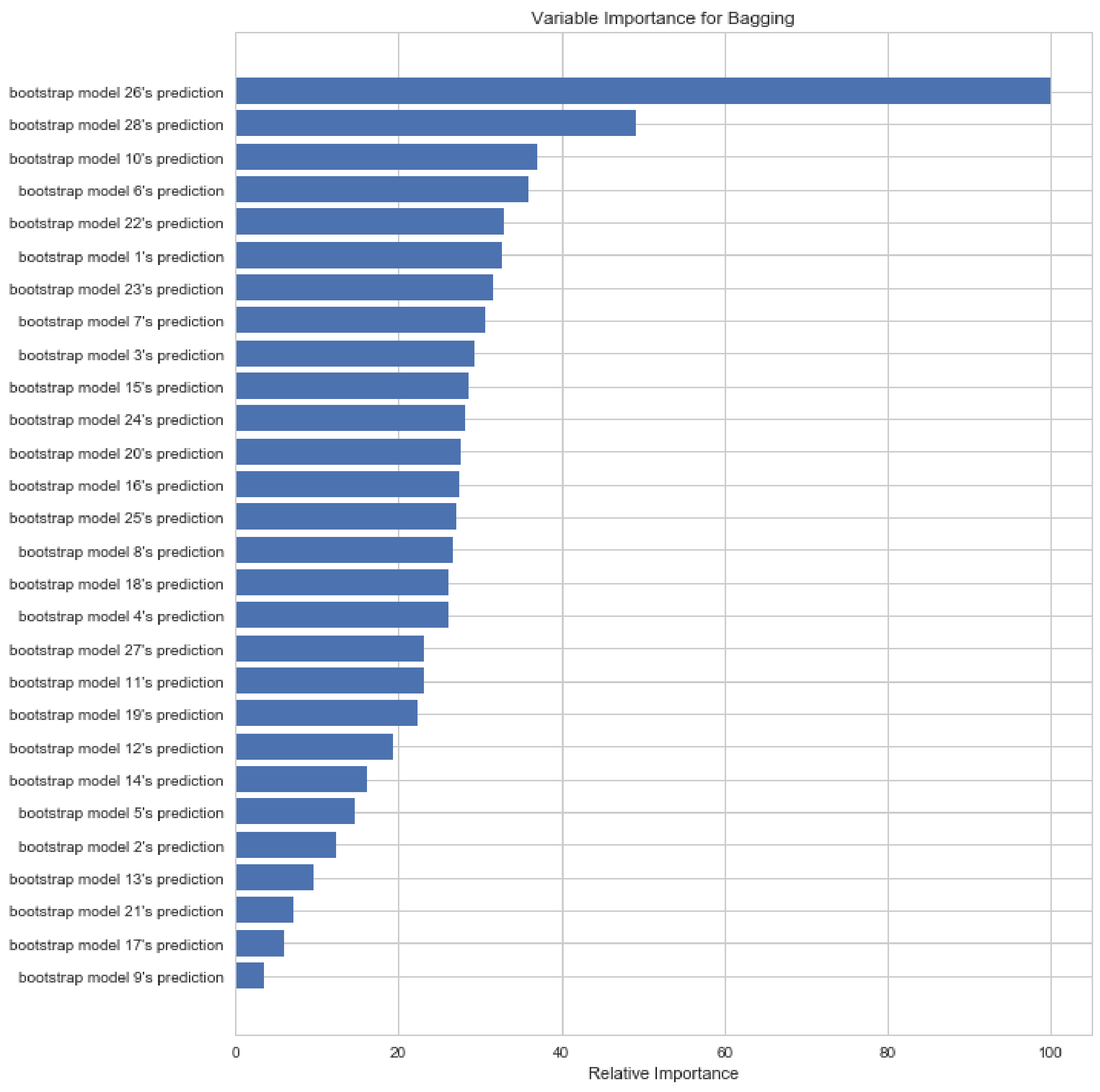
```
0.7997987187987188


0.5819916643916644
```

```
In [22]: display(rf_train, rf_test)
```

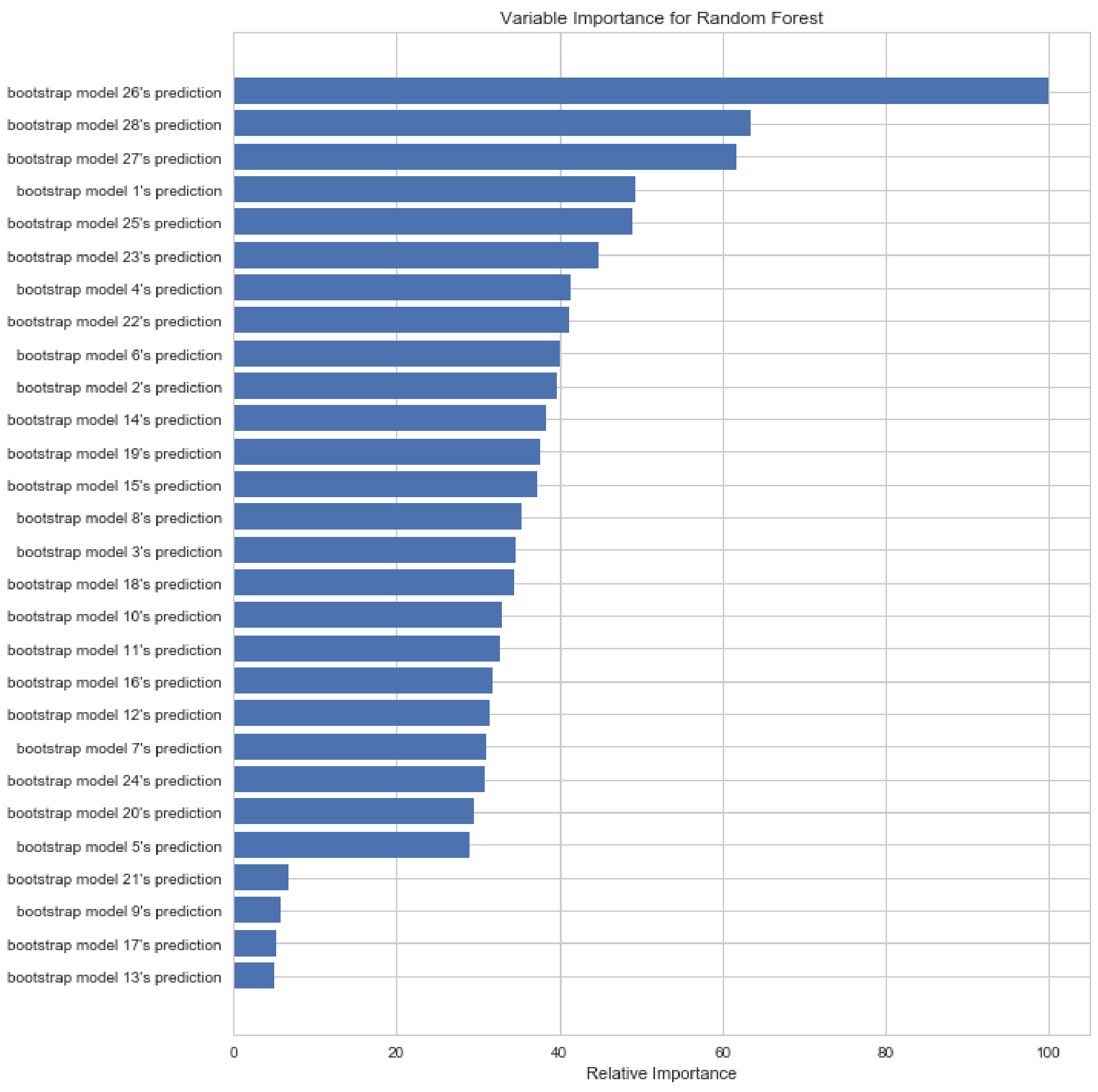
```
                   bootstrap model 1's prediction  bootstrap model 2's prediction  bootstrap mo
training row 1                                1.0                             1.0
training row 2                                0.0                             0.0
training row 3                                0.0                             0.0
training row 4                                1.0                             1.0
training row 5                                0.0                             1.0
training row 6                                0.0                             1.0
training row 7                                0.0                             0.0
training row 8                                1.0                             0.0
training row 9                                1.0                             0.0
training row 10                               1.0                             0.0
training row 11                               0.0                             0.0
training row 12                               0.0                             0.0
training row 13                               0.0                             0.0
training row 14                               0.0                             1.0
training row 15                               0.0                             1.0
training row 16                               0.0                             0.0
training row 17                               0.0                             0.0
training row 18                               0.0                             0.0
training row 19                               1.0                             0.0
training row 20                               1.0                             0.0
training row 21                               1.0                             1.0
training row 22                               0.0                             1.0
training row 23                               0.0                             0.0
training row 24                               0.0                             0.0
training row 25                               1.0                             0.0
training row 26                               0.0                             0.0
training row 27                               1.0                             1.0
training row 28                               1.0                             0.0
training row 29                               1.0                             1.0
training row 30                               1.0                             1.0
...                                           ...                             ...
training row 4971                             0.0                             0.0
training row 4972                             0.0                             0.0
training row 4973                             1.0                             1.0
training row 4974                             0.0                             0.0
training row 4975                             0.0                             0.0
training row 4976                             0.0                             1.0
training row 4977                             1.0                             1.0
training row 4978                             1.0                             0.0
training row 4979                             1.0                             1.0
training row 4980                             0.0                             1.0
training row 4981                             1.0                             1.0
training row 4982                             1.0                             1.0
training row 4983                             1.0                             0.0
training row 4984                             1.0                             1.0
training row 4985                             0.0                             0.0
training row 4986                             0.0                             1.0
training row 4987                             1.0                             0.0
training row 4988                             1.0                             0.0
training row 4989                             0.0                             0.0
training row 4990                             1.0                             1.0
training row 4991                             0.0                             1.0
training row 4992                             0.0                             0.0
training row 4993                             1.0                             1.0
training row 4994                             1.0                             0.0
training row 4995                             0.0                             0.0
training row 4996                             1.0                             0.0
training row 4997                             0.0                             0.0
training row 4998                             1.0                             1.0
training row 4999                             1.0                             1.0
training row 5000                             1.0                             0.0

[5000 rows x 45 columns]


               bootstrap model 1's prediction  bootstrap model 2's prediction  bootstrap model
test row 1                                0.0                             0.0
test row 2                                0.0                             0.0
test row 3                                0.0                             0.0
test row 4                                1.0                             1.0
test row 5                                1.0                             1.0
test row 6                                1.0                             1.0
test row 7                                0.0                             0.0
test row 8                                0.0                             0.0
test row 9                                0.0                             0.0
test row 10                               1.0                             1.0
test row 11                               1.0                             1.0
test row 12                               1.0                             1.0
test row 13                               0.0                             0.0
test row 14                               1.0                             1.0
test row 15                               1.0                             1.0
test row 16                               1.0                             1.0
test row 17                               0.0                             0.0
test row 18                               1.0                             1.0
test row 19                               0.0                             0.0
test row 20                               1.0                             1.0
test row 21                               1.0                             1.0
test row 22                               0.0                             0.0
test row 23                               1.0                             1.0
test row 24                               0.0                             0.0
test row 25                               0.0                             0.0
test row 26                               1.0                             1.0
test row 27                               0.0                             0.0
test row 28                               1.0                             1.0
test row 29                               1.0                             1.0
test row 30                               1.0                             1.0
...                                       ...                             ...
test row 4971                             1.0                             1.0
test row 4972                             0.0                             0.0
test row 4973                             1.0                             1.0
test row 4974                             0.0                             0.0
test row 4975                             0.0                             0.0
test row 4976                             0.0                             0.0
test row 4977                                      1.0                             1.0
test row 4978                                      0.0                             0.0
test row 4979                                      0.0                             0.0
test row 4980                                      1.0                             1.0
test row 4981                                      1.0                             1.0
test row 4982                                      0.0                             0.0
test row 4983                                      1.0                             1.0
test row 4984                                      1.0                             1.0
test row 4985                                      0.0                             0.0
test row 4986                                      1.0                             1.0
test row 4987                                      1.0                             1.0
test row 4988                                      1.0                             1.0
test row 4989                                      0.0                             0.0
test row 4990                                      0.0                             0.0
test row 4991                                      0.0                             0.0
test row 4992                                      1.0                             1.0
test row 4993                                      1.0                             1.0
test row 4994                                      0.0                             0.0
test row 4995                                      1.0                             1.0
test row 4996                                      1.0                             1.0
test row 4997                                      0.0                             0.0
test row 4998                                      0.0                             0.0
test row 4999                                      1.0                             1.0
test row 5000                                      1.0                             1.0

[5000 rows x 45 columns]
```

3.2

```
In [23]: # your code here
         #Top Features for Random Forest
         feature_importance = dt.feature_importances_
         feature_importance = 100.0 * (feature_importance / feature_importance.max())
         sorted_idx = np.argsort(feature_importance)
         pos = np.arange(sorted_idx.shape[0]) + .5

         #Plot
         plt.figure(figsize=(10,12))
         plt.barh(pos, feature_importance[sorted_idx], align='center')
         plt.yticks(pos, bagging_train.columns[sorted_idx])
         plt.xlabel('Relative Importance')
         plt.title('Variable Importance for Bagging')

Out[23]: Text(0.5,1,'Variable Importance for Bagging')
```



```
In [24]: #Top Features for Random Forest
         feature_importance = rf.feature_importances_
         feature_importance = 100.0 * (feature_importance / feature_importance.max())
         sorted_idx = np.argsort(feature_importance)
         pos = np.arange(sorted_idx.shape[0]) + .5

         #Plot
         plt.figure(figsize=(10,12))
         plt.barh(pos, feature_importance[sorted_idx], align='center')
         plt.yticks(pos, rf_train.columns[sorted_idx])
         plt.xlabel('Relative Importance')
         plt.title('Variable Importance for Random Forest')

Out[24]: Text(0.5,1,'Variable Importance for Random Forest')
```



your answer here
3.3
Fill in the following table (ideally in code, but ok to fill in this Markdown cell).

classifier	training accuracy	test accuracy
single tree with best depth chosen by CV	0.643	0.648
single depth-18 tree	0.998	0.598
bagging 45 depth-18 trees	0.997	**0.679**
Random Forest of 45 depth-X trees	0.994	0.603

your answer here
Question 4: Boosting [15 pts]
In this question we explore a different kind of ensemble method, boosting, where each new

model is trained on a dataset weighted towards observations that the current set of models predicts incorrectly.

We'll focus on the AdaBoost flavor of boosting and examine what happens to the ensemble model's accuracy as the algorithm adds more predictors to the ensemble.

4.1 We'll motivate AdaBoost by noticing patterns in the errors that a single classifier makes. Fit `tree1`, a decision tree with depth 3, to the training data. For each predictor, make a plot that compares two distributions: the values of that predictor for examples that `tree1` classifies correctly, and the values of that predictor for examples that `tree1` classifies incorrectly. Do you notice any predictors for which the distributions are clearly different?

4.2 The following code attempts to implement a simplified version of boosting using just two classifiers (described below). However, it has both stylistic and functionality flaws. First, imagine that you are a grader for a Data Science class; write a comment for the student who submitted this code. Then, imagine that you're the TF writing the solutions; make an excellent example implementation. Finally, use your corrected code to compare the performance of `tree1` and the boosted algorithm on both the training and test set.

4.3 Now let's use the sklearn implementation of AdaBoost: Use `AdaBoostClassifier` to fit another ensemble to `X_train`. Use a decision tree of depth 3 as the base learner and a learning rate 0.05, and run the boosting for 800 iterations. Make a plot of the effect of the number of estimators/iterations on the model's train and test accuracy.

Hint: The `staged_score` method provides the accuracy numbers you'll need. You'll need to use `list()` to convert the "generator" it returns into an ordinary list.

4.4 Repeat the plot above for a base learner with depth of (1, 2, 3, 4). What trends do you see in the training and test accuracy?

(It's okay if your code re-fits the depth-3 classifier instead of reusing the results from the previous problem.)

4.5 Based on the plot you just made, what combination of base learner depth and number of iterations seems optimal? Why? How does the performance of this model compare with the performance of the ensembles you considered above?

Answers

4.1 We'll motivate AdaBoost by noticing patterns in the errors that a single classifier makes. Fit `tree1`, a decision tree with depth 3, to the training data. For each predictor, make a plot that compares two distributions: the values of that predictor for examples that `tree1` classifies correctly, and the values of that predictor for examples that `tree1` classifies incorrectly. Do you notice any predictors for which the distributions are clearly different?

Hints: - If you have `fig, axs = plt.subplots(...)`, then `axs.ravel()` gives a list of each plot in reading order. - `sns.kdeplot` takes `ax` and `label` parameters.

```
In [25]: data_train = pd.DataFrame(data_train)

         tree1 = DecisionTreeClassifier(max_depth = 3)
         tree1 = tree1.fit(X_train, y_train)
         y_pred_new = pd.DataFrame(tree1.predict(X_train))
         display(y_pred_new)
         display(y_train)
```

```
        0
0     1.0
1     0.0
```

```
2       0.0
3       0.0
4       0.0
5       0.0
6       0.0
7       0.0
8       1.0
9       0.0
10      0.0
11      0.0
12      0.0
13      0.0
14      1.0
15      0.0
16      0.0
17      0.0
18      1.0
19      0.0
20      0.0
21      0.0
22      1.0
23      1.0
24      0.0
25      0.0
26      1.0
27      0.0
28      0.0
29      1.0
...     ...
4970    0.0
4971    0.0
4972    0.0
4973    0.0
4974    0.0
4975    1.0
4976    1.0
4977    1.0
4978    0.0
4979    0.0
4980    0.0
4981    1.0
4982    0.0
4983    0.0
4984    0.0
4985    1.0
4986    1.0
4987    0.0
4988    1.0
```

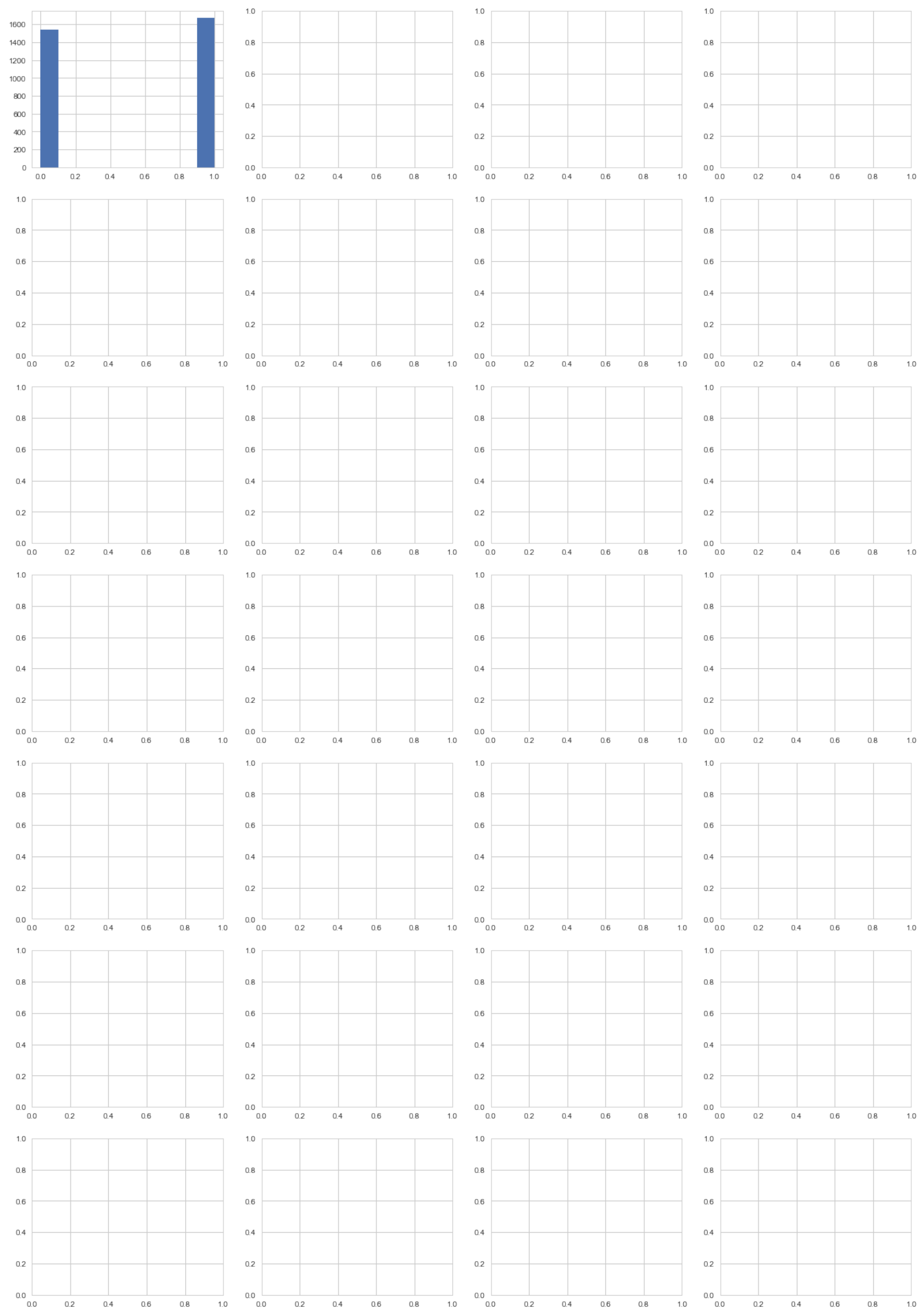
```
4989  1.0
4990  1.0
4991  0.0
4992  0.0
4993  1.0
4994  1.0
4995  1.0
4996  1.0
4997  0.0
4998  1.0
4999  1.0

[5000 rows x 1 columns]


array([1., 1., 1., ..., 1., 1., 1.])
```

```
In [26]: y_pred_new['true'] = y_train
         y_pred_new['correct'] = y_pred_new['true']==y_pred_new[0]
         correct_index = y_pred_new.index[y_pred_new['correct'] == True].tolist()
         incorrect_index = y_pred_new.index[y_pred_new['correct'] == False].tolist()
```

```
In [29]: nrow = 7
         ncol = 4

         fig, axs = plt.subplots(nrow, ncol, figsize = (20, 30))
         for i, ax in enumerate(fig.axes):
             col = data_train.columns[i]
             data_correct = data_train[col][correct_index]
             data_incorrect = data_train[col][incorrect_index]
             ax.hist(data_correct)
             ax.hist(data_incorrect)
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-29-112ff105c781> in <module>()
          7     data_correct = data_train[col][correct_index]
          8     data_incorrect = data_train[col][incorrect_index]
    ----> 9     ax.hist(data_correct)
         10     ax.hist(data_incorrect)


        ~\Anaconda3\lib\site-packages\matplotlib\__init__.py in inner(ax, *args, **kwargs)
       1853                         "the Matplotlib list!)" % (label_namer, func.__name__),
       1854                         RuntimeWarning, stacklevel=2)
```

```
-> 1855             return func(ax, *args, **kwargs)
   1856
   1857         inner.__doc__ = _add_data_doc(inner.__doc__,


    ~\Anaconda3\lib\site-packages\matplotlib\axes\_axes.py in hist(***failed resolving argu
   6711             labels = [label]
   6712         else:
-> 6713             labels = [six.text_type(lab) for lab in label]
   6714
   6715         for patch, lbl in zip_longest(patches, labels, fillvalue=None):


    TypeError: 'int' object is not iterable
```



your answer here

4.2

The intended functionality is the following: 1. Fit `tree1`, a decision tree with max depth 3. 2. Construct an array of sample weights. Give a weight of 1 to samples that `tree1` classified correctly, and 2 to samples that `tree1` misclassified. 3. Fit `tree2`, another depth-3 decision tree, using those sample weights. 4. To predict, compute the probabilities that `tree1` and `tree2` each assign to the positive class. Take the average of those two probabilities as the prediction probability.

The following code attempts to implement a simplified version of boosting using just two classifiers (described below). However, it has both stylistic and functionality flaws. First, imagine that you are a grader for a Data Science class; write a comment for the student who submitted this code. Then, imagine that you're the TF writing the solutions; make an excellent example implementation. Finally, use your corrected code to compare the performance of `tree1` and the boosted algorithm on both the training and test set.

```
In [30]: def boostmeup():
             tree = DecisionTreeClassifier(max_depth=3)
             tree1 = tree.fit(X_train, y_train)
             sample_weight = np.ones(len(X_train))
             q = 0
             for idx in range(len(X_train)):
               if tree1.predict([X_train[idx]]) != y_train[idx]:
                  sample_weight[idx] = sample_weight[idx] * 2
                  q = q + 1
             print("tree1 accuracy:", q / len(X_train))
             tree2 = tree.fit(X_train, y_train, sample_weight=sample_weight)

         # Train
             q = 0
             for idx in range(len(X_train)):
                 t1p = tree1.predict_proba([X_train[idx]])[0][1]
                 t2p = tree2.predict_proba([X_train[idx]])[0][1]
                 m = (t1p + t2p) / 2
                 if m > .5:
                     if y_train[idx] == True:
                         q = q + 0
                     else:
                         q = q + 1
                 else:
                     if y_train[idx] == True:
                         q = q + 1
                     else:
                         q = 0
             print("Boosted accuracy:", q / len(X_train))

         # Test
             q = 0
             for idx in range(len(X_test)):
                 t1p = tree1.predict_proba([X_test[idx]])[0][1]
                 t2p = tree2.predict_proba([X_test[idx]])[0][1]
```

```
            m = (t1p + t2p) / 2
            if m > .5:
                if y_train[idx] == True:
                    q = q + 0
                else:
                    q = q + 1
            else:
                if y_train[idx] == True:
                    q = q + 1
                else:
                    q = 0
        print("Boosted accuracy:", q / len(X_test))

    boostmeup()
```

```
tree1 accuracy: 0.3582
Boosted accuracy: 0.0008
Boosted accuracy: 0.002
```

Your answer here

```
In [31]: def boostmeup():
        tree = DecisionTreeClassifier(max_depth=3)
        tree1 = tree.fit(X_train, y_train)
        sample_weight = np.ones(len(X_train))
        q = 0
        for idx in range(len(X_train)):
          if tree1.predict([X_train[idx]]) != y_train[idx]:
             sample_weight[idx] = sample_weight[idx] * 2
             q = q + 1
        print("tree1 accuracy:", 1-q / len(X_train))
        tree2 = tree.fit(X_train, y_train, sample_weight=sample_weight)

    # Train
        q = 0
        for idx in range(len(X_train)):
            t1p = tree1.predict_proba([X_train[idx]])[0][1]
            t2p = tree2.predict_proba([X_train[idx]])[0][1]
            m = (t1p + t2p) / 2
            if m > .5:
                if y_train[idx] == True:
                    q = q + 0
                else:
                    q = q + 1
            else:
                if y_train[idx] == True:
                    q = q + 1
                else:
                    q = q + 0
        print("Boosted accuracy:", 1-q / len(X_train))

    # Test
        q = 0
        for idx in range(len(X_test)):
            t1p = tree1.predict_proba([X_test[idx]])[0][1]
            t2p = tree2.predict_proba([X_test[idx]])[0][1]
            m = (t1p + t2p) / 2
            if m > .5:
                if y_test[idx] == True:
                    q = q + 0
                else:
                    q = q + 1
            else:
                if y_test[idx] == True:
                    q = q + 1
                else:
                    q = q + 0
        print("Boosted accuracy:", 1-q / len(X_test))

    boostmeup()
```

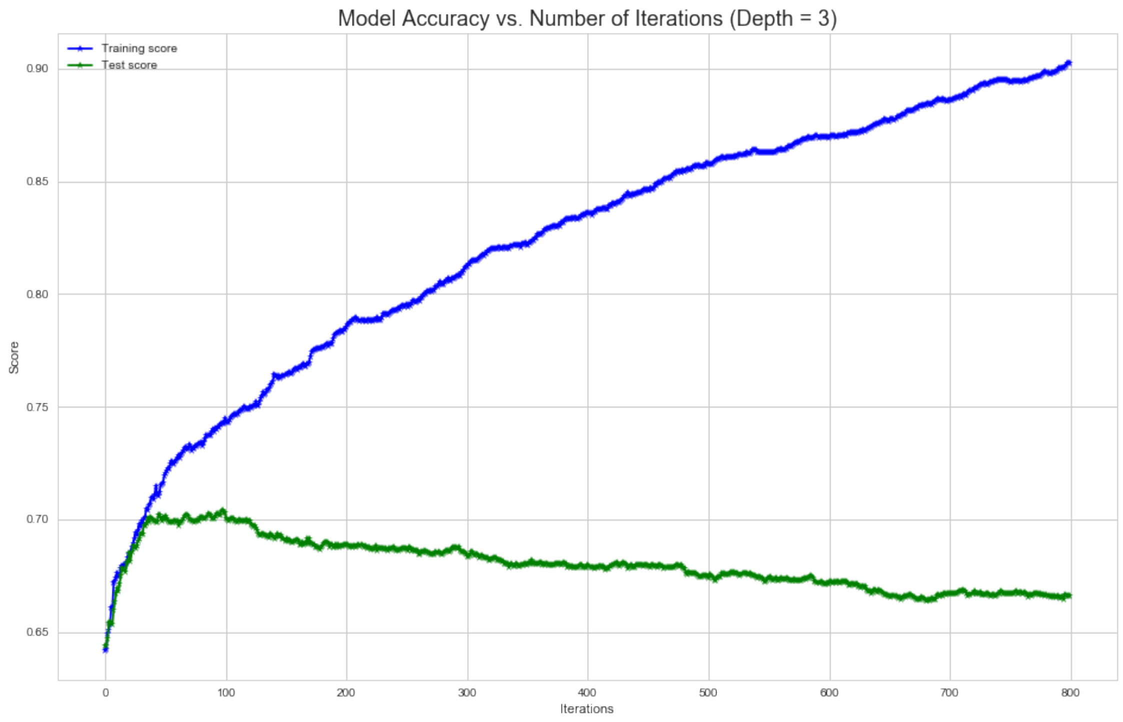
```
tree1 accuracy: 0.6417999999999999
Boosted accuracy: 0.6134
Boosted accuracy: 0.6088
```

4.3 Now let's use the sklearn implementation of AdaBoost: Use `AdaBoostClassifier` to fit another ensemble to `X_train`. Use a decision tree of depth 3 as the base learner and a learning rate 0.05, and run the boosting for 800 iterations. Make a plot of the effect of the number of estimators/iterations on the model's train and test accuracy.

Hint: The `staged_score` method provides the accuracy numbers you'll need. You'll need to use `list()` to convert the "generator" it returns into an ordinary list.

```
In [32]: # your code here
         boost_model = AdaBoostClassifier(base_estimator = DecisionTreeClassifier(max_depth=3)
         boost_model.fit(X_train, y_train)

         train_scores = list(boost_model.staged_score(X_train,y_train))
         test_scores = list(boost_model.staged_score(X_test, y_test))
```

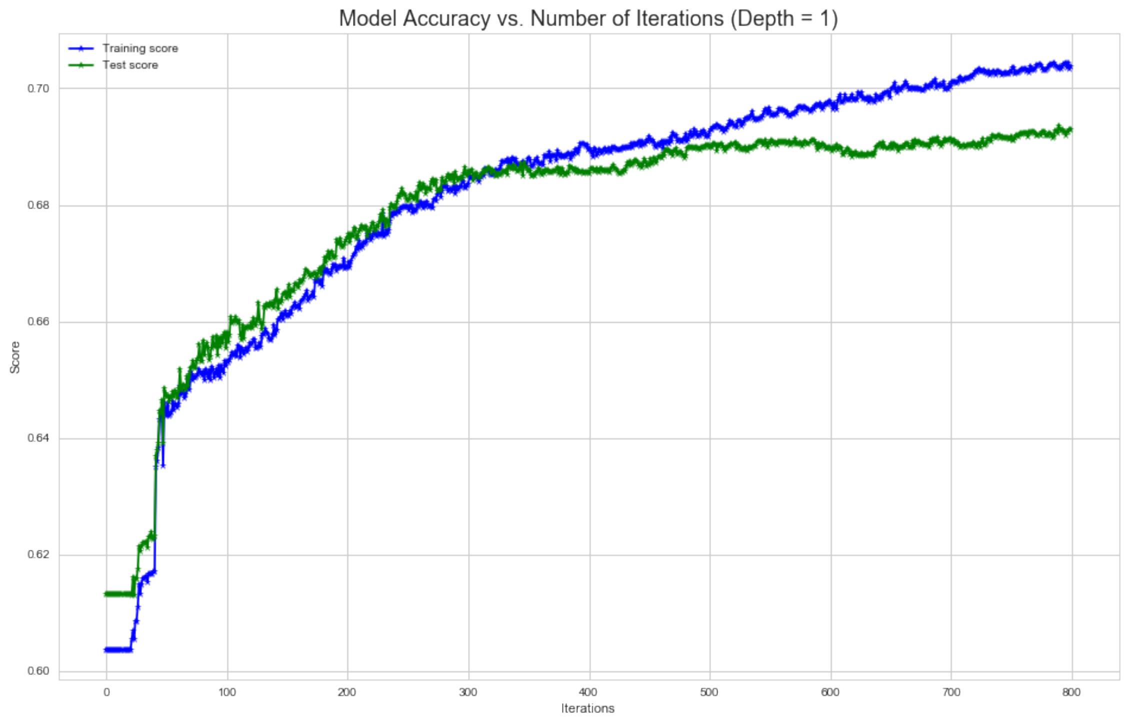
```
In [33]: values = list(range(0,800))

         fig, ax = plt.subplots(figsize=(16,10))
         plt.plot(values, train_scores,'b*-', label = 'Training score')
         plt.plot(values, test_scores,'g*-', label = 'Test score')
         plt.xlabel('Iterations')
         plt.ylabel('Score')
         plt.title('Model Accuracy vs. Number of Iterations (Depth = 3)', size = 18)
         plt.legend();
```



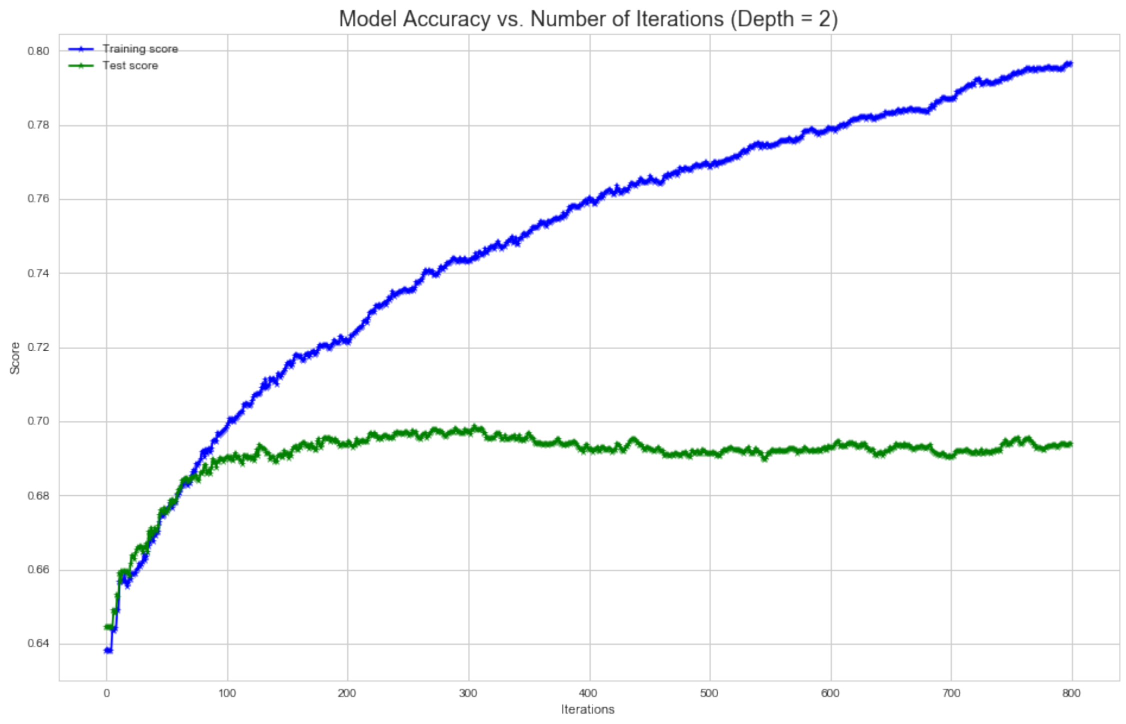
4.4 Repeat the plot above for a base learner with depth of (1, 2, 3, 4). What trends do you see in the training and test accuracy?

```
In [34]: # your code here
         boost_model1 = AdaBoostClassifier(base_estimator = DecisionTreeClassifier(max_depth=1
         boost_model1.fit(X_train, y_train)

         train_scores_1 = list(boost_model1.staged_score(X_train,y_train))
         test_scores_1 = list(boost_model1.staged_score(X_test, y_test))

         boost_model2 = AdaBoostClassifier(base_estimator = DecisionTreeClassifier(max_depth=2
         boost_model2.fit(X_train, y_train)

         train_scores_2 = list(boost_model2.staged_score(X_train,y_train))
         test_scores_2 = list(boost_model2.staged_score(X_test, y_test))

         boost_model4 = AdaBoostClassifier(base_estimator = DecisionTreeClassifier(max_depth=4
         boost_model4.fit(X_train, y_train)

         train_scores_4 = list(boost_model1.staged_score(X_train,y_train))
         test_scores_4 = list(boost_model1.staged_score(X_test, y_test))
```

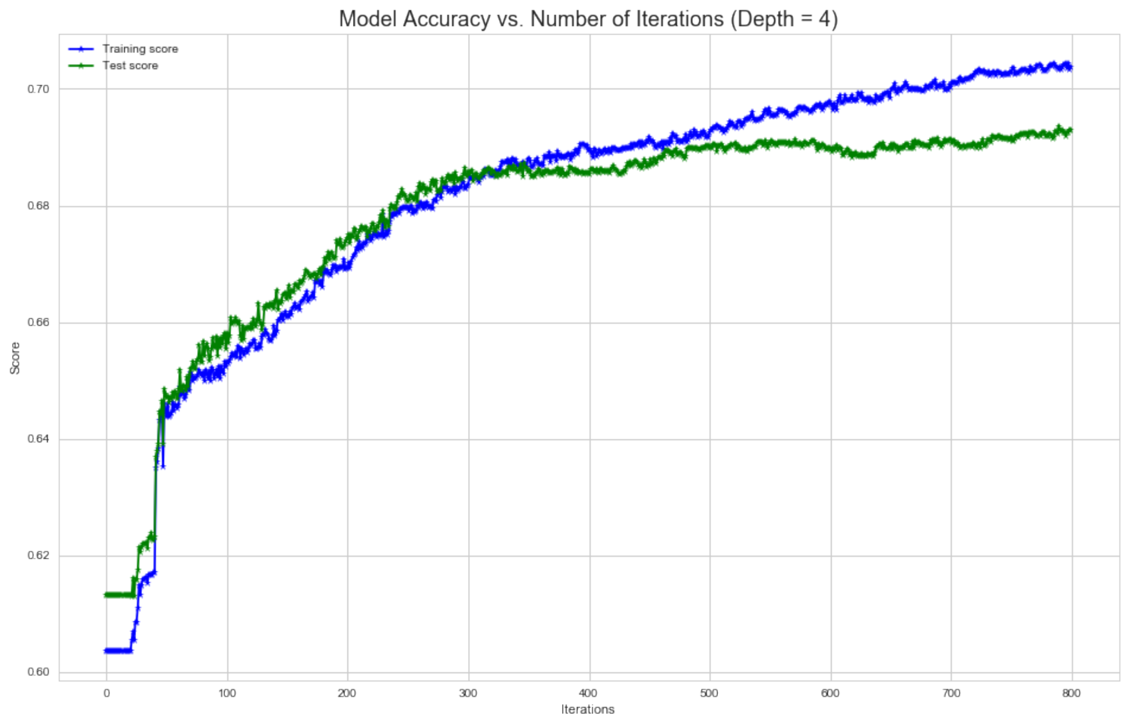
```
In [35]: # your code here
         fig, ax = plt.subplots(figsize=(16,10))
         plt.plot(values, train_scores_1,'b*-', label = 'Training score')
         plt.plot(values, test_scores_1,'g*-', label = 'Test score')
         plt.xlabel('Iterations')
         plt.ylabel('Score')
         plt.title('Model Accuracy vs. Number of Iterations (Depth = 1)', size = 18)
         plt.legend();
```



```
In [36]: fig, ax = plt.subplots(figsize=(16,10))
         plt.plot(values, train_scores_2,'b*-', label = 'Training score')
         plt.plot(values, test_scores_2,'g*-', label = 'Test score')
         plt.xlabel('Iterations')
         plt.ylabel('Score')
         plt.title('Model Accuracy vs. Number of Iterations (Depth = 2)', size = 18)
         plt.legend();
```



```
In [37]: fig, ax = plt.subplots(figsize=(16,10))
         plt.plot(values, train_scores_4,'b*-', label = 'Training score')
         plt.plot(values, test_scores_4,'g*-', label = 'Test score')
         plt.xlabel('Iterations')
         plt.ylabel('Score')
         plt.title('Model Accuracy vs. Number of Iterations (Depth = 4)', size = 18)
         plt.legend();
```



Your answer here

```
In [38]: train_list = (train_scores_1, train_scores_2, train_scores, train_scores_4)
         for i in train_list:
             max_iteration = i.index(max(i))
             max_index = max(i)
             print(max_index, max_iteration)
```

```
0.7044 785
0.7966 796
0.9026 798
0.7044 785
```

```
In [39]: test_list = (test_scores_1, test_scores_2, test_scores, test_scores_4)
         for i in test_list:
             max_iteration = i.index(max(i))
             max_index = max(i)
             print(max_index, max_iteration)
```

```
0.6936 789
0.6986 305
0.704 97
0.6936 789
```

```
In [40]: train_scores[97]
```

```
Out[40]: 0.7428
```

Answer For depth = 2 and depth = 3, we see a huge divergence between training and test accuracy beginning somewhere below 100 iterations. After this point, the training score becomes significantly higher than the test score (indicative of overfitting). For depth = 1 and depth =4, we don't see a significant divergence. However, our overall value for the training score is lower (below 0.74) and our testing score does not exceed 0.69.

4.5 Based on the plot you just made, what combination of base learner depth and number of iterations seems optimal? Why? How does the performance of this model compare with the performance of the ensembles you considered above?

Answer:The optimal depth appears to be at depth = 3 at 98 iterations. At this value, we achieve the highest test score of 0.7042. While the train score will continue to improve with more iterations, the test score drops; this indicates that our data begins to overfit to the train data. This is the highest test score we have achieved with any ensemble method considered.

Question 5: Understanding [15 pts]

This question is an overall test of your knowledge of this homework's material. You may need to refer to lecture notes and other material outside this homework to answer these questions.

5.1 How do boosting and bagging relate: what is common to both, and what is unique to each?

5.2 Reflect on the overall performance of all of the different classifiers you have seen throughout this assignment. Which performed best? Why do you think that may have happened?

5.3 What is the impact of having too many trees in boosting and in bagging? In which instance is it worse to have too many trees?

5.4 Which technique, boosting or bagging, is better suited to parallelization, where you could have multiple computers working on a problem at the same time?

5.5 Which of these techniques can be extended to regression tasks? How?

Answers:

5.1 Boosting and bagging are both ensemble methods, meaning they use multiple learning alogrithms to obtain better predictive power than a single alogrithm could. Both methods typically rely on bootstrapping of data to generate multiple models. Bagging is the aggregation of models, each created on a different set of bootstrapped data, to generate an overall model. It uses voting for classification and averaging for regression.
Boosting involves weighting averages to make weak learners into strong learners. Once a model runs, it evaluates which data has been misclassified; during the next iteration of model generation, that data is given a higher weight so it is more likely to be classified correctly. Through iteration, boosting arrives at an ideal model.

5.2

The various depth 18 trees clearly performed best on our training model. However, these were specifically created to overfit to the training data, and thus were able to achieve near perfect performance. The AdaBoost model with a depth of 3 and 98 iterations performed best on our test and train data holistically. While the training score of 0.74 wasn't the best performer, the test score of 0.704 was higher than any other method we applied. This is likely because Boosting is a learning model; each iteration considers feedback (in the form of residuals) from the previous to make a more appropriate model.

5.3

Typically, bagging and boosting produce a better result than a single model. However, having too many trees can be a problem when there aren't many relevant preductors. This is because in bagging, the chances of selecting relevant predictors on each model is low, so most of the trees will be weak. The aggregation of weak models yeilds another weak model. Additionally, if the number of trees is too large, the trees may become correlated. This results in increased variance.

5.4

Bagging is widely regarded as easy to parallelize, since all the bootstrapped models can be run independently before aggregation. However, it is possible to parallelize boosting despite it's sequential nature. Parallel boosting has been shown to result in greater accuracy than parallel bagging. Additionally, it is less computationally expensive than parallel bagging.

5.5

Bagging can be used for regression by using averaging. It takes models created from each bootstrapped sample and averages them for the final model.
Boosting can be applied to regressions through additive regression or gradient boosting. Gradient boosting simply adding models. The new models compensate for the weaknesses in the current model, and once aggregate create an ideal model.

Sources

In addition to lecture notes, we consulted the following sources:

Rorojan, Ben, "Boosting and Bagging: How to Develop a Robust Machine Learning Algorithm," https://hackernoon.com/how-to-develop-a-robust-algorithm-c38e08f32201

Yu, C., and David Skillcorn, "Parallelizing Boosting and Bagging," https://www.researchgate.net/publication/2366625_Parallelizing_Boosting_and_Bagging

Li, Jia. "Bagging and Boosting: A Brief Introduction," http://personal.psu.edu/jol2/course/stat597e/notes2/bagging.pdf

Friedman, et al., "Additive Logistic Regresson: A Statistical View of Boosting," https://projecteuclid.org/download/pdf_1/euclid.aos/1016218223

Question 6: Explaining Complex Concepts Clearly [10 pts]

One of the core skills of a data scientist is to be able to explain complex concepts clearly. To practice this skill, you'll make a short presentation of one of the approaches we have recently studied.

Choose one of the following topics:

- Decision Trees
- Random Forests
- Bagging
- Boosting
- Simple Neural Nets (like the MLP we saw in Homework 6)
- (other topics are possible, but get staff approval first)

Make 3 slides explaining the concept.

- Focus on **clear explanations**, NOT aesthetic beauty. Photos of pen-and-paper sketches are fine if they're legible.
- For your audience, choose **future CS109A students**.
- You may take inspiration from anywhere, but explain in **your own words** and **make your own illustrations**.

Submit your slides as a PDF and the source format (`.pptx`, Google Slides, etc.)

NOTE: If you would be okay with us using your slides for future classes (with attribution, of course), please include a note to that effect. This will not affect your grade either way.