Table of Contents

[Boosting 1](#_Toc86129196)

[Data Preparation 6](#_Toc86129197)

[qcut() 6](#_Toc86129198)

[map() 6](#_Toc86129199)

[Bagging 8](#_Toc86129200)

[Bagged Classifier 8](#_Toc86129201)

[Bagged Regressors 11](#_Toc86129202)

[Grid Searching 12](#_Toc86129203)

[Stacked Models 15](#_Toc86129204)

[Stacking Regression 15](#_Toc86129205)

[Stacking Classification 18](#_Toc86129206)

## Boosting

Boosting is an ensemble technique which can often improve model performance. With boosting, models are added to an ensemble sequentially instead of all at once. At each iteration, a new model is created and the new base-learner model is trained while using the errors of the previous learners. Josh Stormer provides an excellent overview of how boosting (with AdaBoost) is implemented at:

<https://www.youtube.com/watch?v=LsK-xG1cLYA&t=1020s>

|  |  |
| --- | --- |
| With AdaBoost, the trees usually only include a root node and two leaves. These short trees are called stumps. Stumps on their own though are not great for classifications. |  |

|  |  |
| --- | --- |
| A stump generally uses one variable to make decisions. |  |

|  |  |
| --- | --- |
| With AdaBoost some stumps are weighted more in the classification. |  |

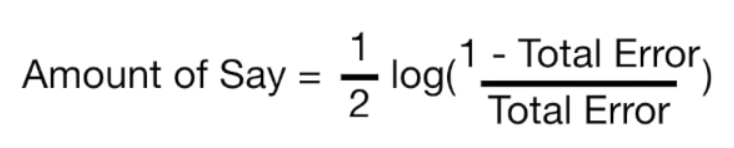
|  |  |
| --- | --- |
| Errors that are made by the first stump influence how the second stump is made and so on…. |  |

|  |  |
| --- | --- |
|  | For the heart disease prediction example, when the first stump is created, all samples are given the same weight.  Sample Weight = = |

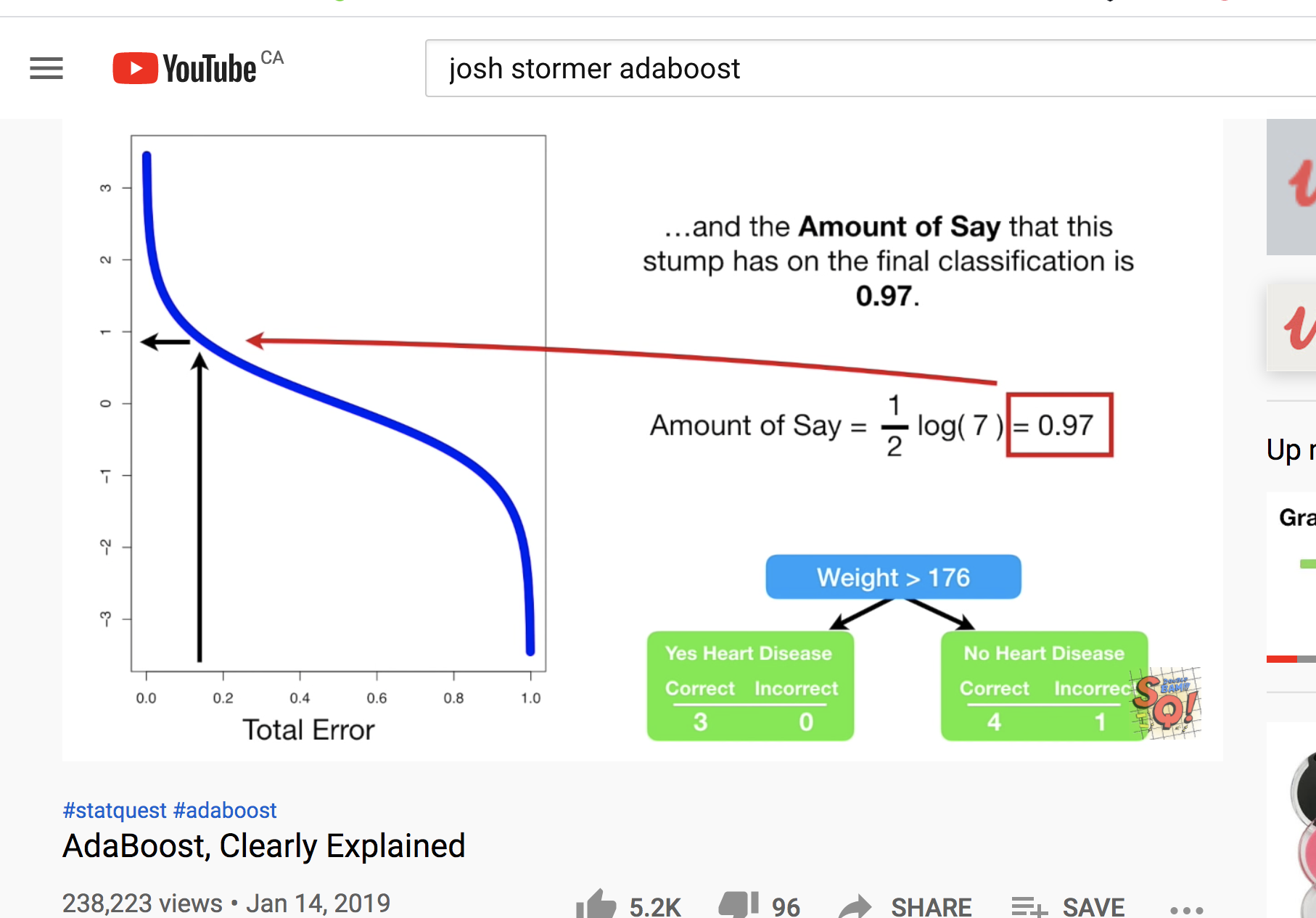
|  |  |
| --- | --- |
| Patient weight turns out to be the best variable to perform the first prediction for heart disease. Only the prediction at the very right is incorrect. It is no coincidence that Gini index is also the lowest compared to all over predictor variables. This stump becomes the first stump in our forest. |  |

|  |  |
| --- | --- |
| The patient who is incorrectly classified by the stump actually has heart disease but the stump shown above says they do not. The total error for the stump is therefore . The total error always ranges between 0 for a good stump and 1 for a bad stump. |  |

The amount of say that a stump has in the prediction is:

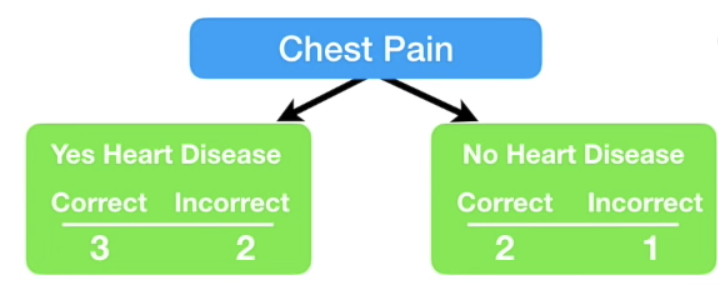


When total error is small, the amount of say for the sample is large. When the total error is large the amount of say is small. When the total error is 1/8 the amount of say is 0.97.



Exercise (4 marks)

The patient weight is definitely the correct classifier for the first stump. However, if we had chosen chest pain to be the classifier in the first stump 3 classifications would have been incorrect.



Calculate the amount of say that this stump would have had. Show your calculations here:

|  |
| --- |
| Amount of say =1/2 log(1-3/8) / 3/8 |

|  |  |
| --- | --- |
| Continuing on with our example, since the first stump incorrectly classified the heart disease status for the person on the right, we need to adjust the sample weight. |  |
|  |  |

The **incorrectly classified** sample weight becomes:

New sample weight = sample weight = = 0.33

The **correctly classified** sample weights on the other hand must be decreased.

New sample weight = sample weight = = 0.05

|  |  |
| --- | --- |
| These are the weights after at the very right. |  |

Next, we need to normalize the weights so they sum up to 1.

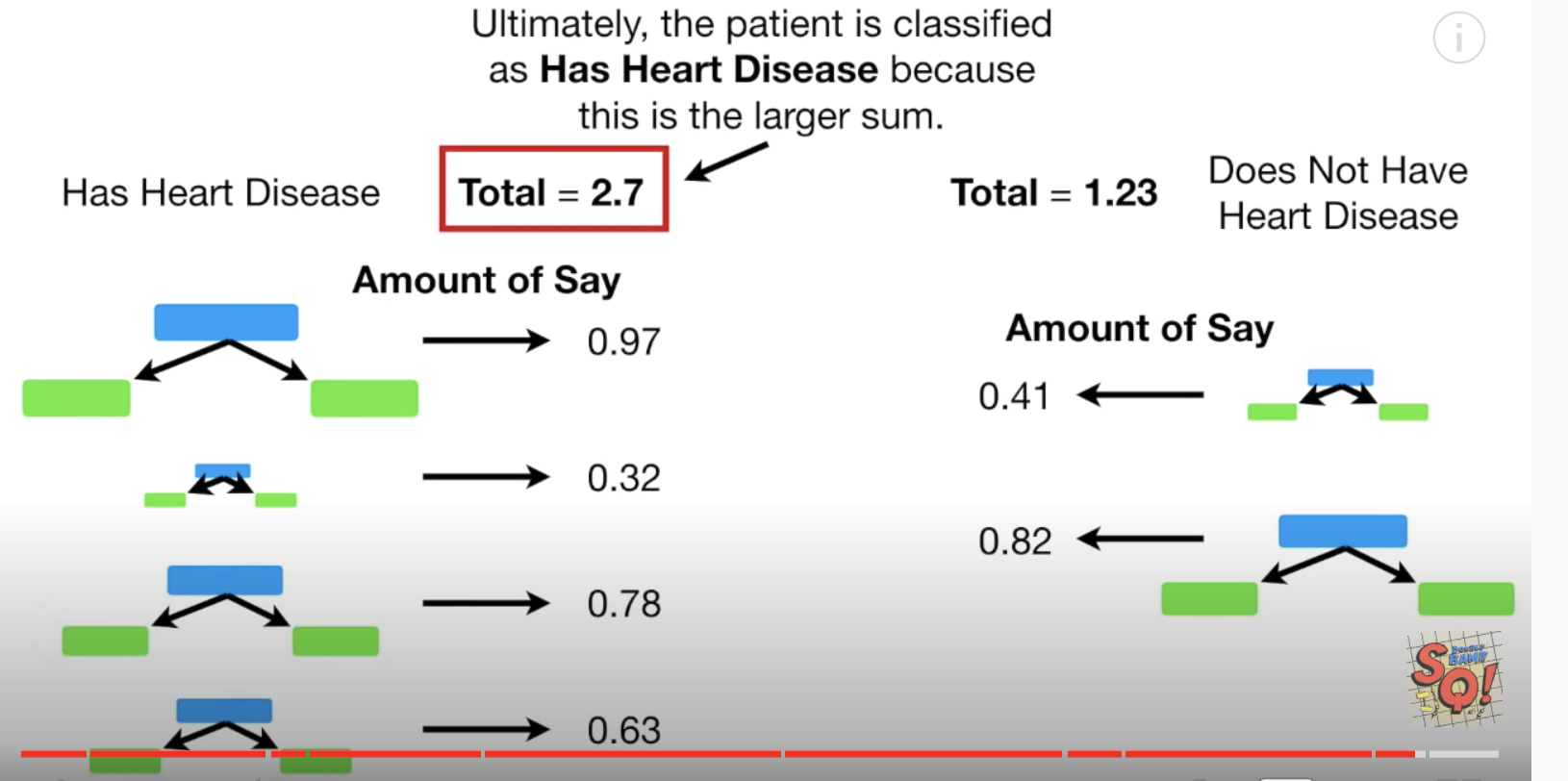
= = 0.0736

= = 0.4853

|  |  |
| --- | --- |
| Next, we throw out the old weights and use the new weights to build our new sample set. To generate the new sample, we take random picks between 0 and 1. If the pick falls between 0 and 0.07 we choose sample 1. If we pick 0.1 we choose sample 2. If we pick between 0.22 and 0.71 we choose sample 4 and so on. |  |

|  |  |
| --- | --- |
| Note that we are likely to choose sample 4 several times as outlined at the right but this is okay since selection is with replacement. |  |

We continue these steps to build more stumps. To make a prediction we sum the ‘amounts of say’ for each stump. The largest sum represents the prediction.



Example : Boosting

Thankfully the boosting classifiers hide all of the detail for us. We can just use them in our code. Add this code onto the end of Example 2.

|  |
| --- |
| from sklearn.model\_selection import cross\_val\_score  from mlxtend.classifier import EnsembleVoteClassifier  from xgboost import XGBClassifier, plot\_importance  from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier  ada\_boost = AdaBoostClassifier()  grad\_boost = GradientBoostingClassifier()  xgb\_boost = XGBClassifier()  boost\_array = [ada\_boost, grad\_boost, xgb\_boost]  eclf = EnsembleVoteClassifier(clfs=[ada\_boost, grad\_boost,  xgb\_boost], voting='hard')  labels = ['Ada Boost', 'Grad Boost', 'XG Boost', 'Ensemble']  for clf, label in zip([ada\_boost, grad\_boost, xgb\_boost, eclf], labels):  scores = cross\_val\_score(clf, X, y, cv=10, scoring='accuracy')  print("Mean: {0:.3f}, std: (+/-) {1:.3f} [{2}]".format(scores.mean(),  scores.std(), label)) |

The output is:

|  |
| --- |
| Name: price, Length: 546, dtype: int8  Mean: 0.641, std: (+/-) 0.082 [Ada Boost]  Mean: 0.630, std: (+/-) 0.103 [Grad Boost]  Mean: 0.635, std: (+/-) 0.099 [XG Boost]  Mean: 0.626, std: (+/-) 0.091 [Ensemble] |

Exercise (4 marks)

Compare the different boosting algorithm performances for the Iris data set. Use the code at the start of Exercise 6 to load and prepare the Iris data set. Show your working program here:

|  |
| --- |
| from sklearn.model\_selection import cross\_val\_score from mlxtend.classifier import EnsembleVoteClassifier from xgboost import XGBClassifier, plot\_importance from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier  import pandas as pd from sklearn.preprocessing import LabelEncoder   PATH = "/Users/hyerimshin/PycharmProjects/MachineLearning/datasets/" df = pd.read\_csv(PATH + 'iris\_old.csv') df.columns = ['Sepal L', 'Sepal W', 'Petal L', 'Petal W', 'Iris Type']  # Convert text to numeric category. # 0 is setosa, 1 is versacolor and 2 is virginica df['y'] = LabelEncoder().fit\_transform(df['Iris Type'])  # Prepare the data. X = df[['Sepal L', 'Sepal W', 'Petal L', 'Petal W']] print(type(df)) y = df['y'] print(type(y))  ada\_boost = AdaBoostClassifier() grad\_boost = GradientBoostingClassifier() xgb\_boost = XGBClassifier() boost\_array = [ada\_boost, grad\_boost, xgb\_boost] eclf = EnsembleVoteClassifier(clfs=[ada\_boost, grad\_boost,  xgb\_boost], voting='hard') labels = ['Ada Boost', 'Grad Boost', 'XG Boost', 'Ensemble']  for clf, label in zip([ada\_boost, grad\_boost, xgb\_boost, eclf], labels):  scores = cross\_val\_score(clf, X, y, cv=10, scoring='accuracy')  print("Mean: {0:.3f}, std: (+/-) {1:.3f} [{2}]".format(scores.mean(),  scores.std(), label)) |

Show the output from your code:

|  |
| --- |
| <class 'pandas.core.series.Series'>  Mean: 0.953, std: (+/-) 0.052 [Ada Boost]  Mean: 0.960, std: (+/-) 0.044 [Grad Boost]  Mean: 0.960, std: (+/-) 0.033 [XG Boost]  Mean: 0.960, std: (+/-) 0.044 [Ensemble] |

Explain which boosting algorithm performed best and why you made that selection.

|  |
| --- |
| from sklearn.model\_selection import cross\_val\_score from mlxtend.classifier import EnsembleVoteClassifier from xgboost import XGBClassifier, plot\_importance from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier  import pandas as pd from sklearn.preprocessing import LabelEncoder  PATH = "/Users/hyerimshin/PycharmProjects/MachineLearning/datasets/" df = pd.read\_csv(PATH + 'loan\_default.csv') print(df.head())  # Prepare the data. X = df.copy() del X['default'] y = df[['default']]   ada\_boost = AdaBoostClassifier() grad\_boost = GradientBoostingClassifier() xgb\_boost = XGBClassifier() boost\_array = [ada\_boost, grad\_boost, xgb\_boost] eclf = EnsembleVoteClassifier(clfs=[ada\_boost, grad\_boost,  xgb\_boost], voting='hard') labels = ['Ada Boost', 'Grad Boost', 'XG Boost', 'Ensemble']  for clf, label in zip([ada\_boost, grad\_boost, xgb\_boost, eclf], labels):  scores = cross\_val\_score(clf, X, y, cv=10, scoring='accuracy')  print("Mean: {0:.3f}, std: (+/-) {1:.3f} [{2}]".format(scores.mean(),  scores.std(), label)) |

Exercise (2 marks)

Indicate if the following statement is true or false. Boosting for classification attempts to improve learning by assigning extra learning focus towards samples that are initially incorrectly classified.

True or False

## Data Preparation

This section discusses two additional functions which will be used to help with data preparation.

### qcut()

qcut() converts continuous data into evenly distributed categories. This line of code shows how qcut() converts continuous price values into one of three price categories.

df['price'] = pd.qcut(df['price'], 3, labels=[0, 1, 2]).cat.codes

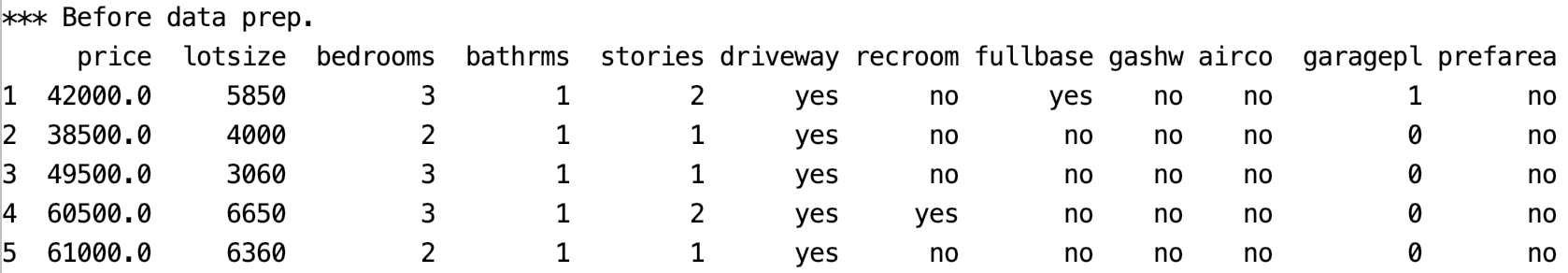
### map()

map() translates text encodings to specific numeric labels.

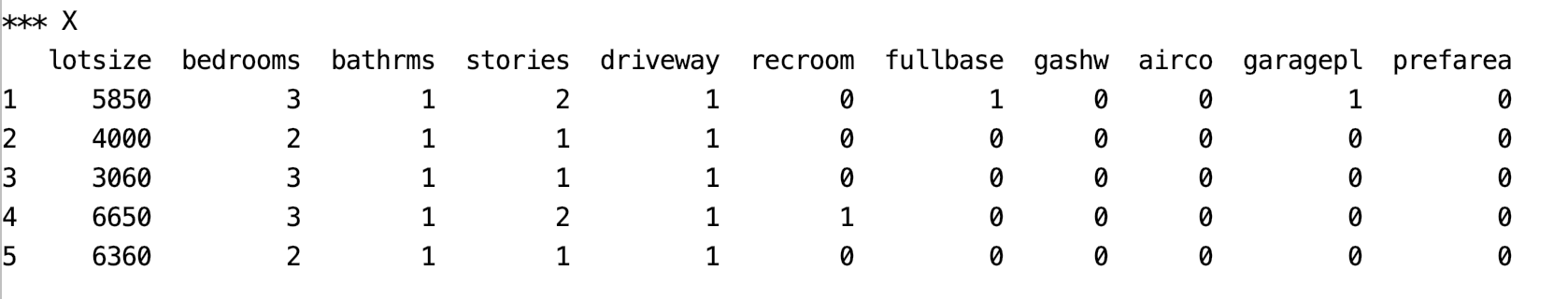
df[columns[i]] = df[columns[i]].map({'yes': 1, 'no': 0})

Example : Data Preparation

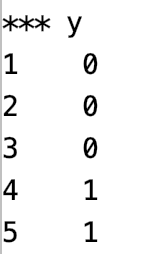
This example prepares data for modelling. The prepared data will be used in Example 3. Initially, the data looks like the following:



After using the map() function to convert the ‘no’ and ‘yes’ values to 0 and 1, the X feature set becomes:



After using the qcut() function to convert prices into categories, the price target becomes:



Here is the program that loads and prepares the housing price data for categorical price prediction:

|  |
| --- |
| from pydataset import data  import pandas as pd  # Get the housing data  df = data('Housing')  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  print("\n\*\*\* Before data prep.")  print(df.head(5))  # Convert continues price variable into evenly distributed categories.  df['price'] = pd.qcut(df['price'], 3, labels=[0,1,2]).cat.codes  # Show new price variable cateogories.  print("\nNewly categorized target (price) values")  print(df['price'].value\_counts())  def convertToBinaryValues(df, columns):  for i in range(0, len(columns)):  df[columns[i]] = df[columns[i]].map({'yes': 1, 'no': 0})  return df  df = convertToBinaryValues(df, ['driveway', 'recroom', \  'fullbase', 'gashw', 'airco', 'prefarea'])  # Split into two sets  y = df['price']  X = df.drop('price', 1)  # Show prepared data.  print("\n\*\*\* X")  print(X.head(5))  print("\n\*\*\* y")  print(y.head(5)) |

The output from this data set will be used a little later in bagging and stacking examples.

## Bagging

Bagging involves averaging the predictions from the weak learners to generate a more stable result with less variance. ‘Bagging’ is a term that is derived from the terms “bootstrapping” and “aggregating”. Bootstrapping involves sampling with replacement. Weak learners of the same type are built with the bootstrapped data.

### Bagged Classifier

We have looked at bagging with random forests. We can perform bagging with other model types as well.

Example : Bagging

Next in this example, several stand-alone and bagged classifiers are compared. These classifiers include:

* RandomForestClassifier()
* ExtraTreesClassifier()
* KNeighborsClassifier()
* SVC()
* RidgeClassifier()

Bagging often offers a slight improvement in accuracy with generally less variance in most cases. Overall the Bagged RandomForest classifier appears to have the best accuracy, precision and recall (see Table 1).

Table : Comparing Stand-Alone and Bagged Classifiers

|  |  |
| --- | --- |
| \*\*\* RandomForestClassifier \*\*\*  Accuracy: 0.6158536585365854  Precision: 0.6236235578117348  Recall: 0.6158536585365854  F1: 0.6162779662773046  \*\*\* Bagged: RandomForestClassifier \*\*\*  Accuracy: 0.6951219512195121  Precision: 0.6989793281926248  Recall: 0.6951219512195121  F1: 0.6962455650810152 | \*\*\* ExtraTreesClassifier \*\*\*  Accuracy: 0.6219512195121951  Precision: 0.6246320390556507  Recall: 0.6219512195121951  F1: 0.6195969265924138  \*\*\* Bagged: ExtraTreesClassifier \*\*\*  Accuracy: 0.6463414634146342  Precision: 0.652869134573478  Recall: 0.6463414634146342  F1: 0.6476985461972846 |

To build this example, add this code to the code in Example 2:

|  |
| --- |
| from sklearn.ensemble import BaggingClassifier, \  ExtraTreesClassifier, RandomForestClassifier  from sklearn.neighbors import KNeighborsClassifier  from sklearn.linear\_model import RidgeClassifier  from sklearn.svm import SVC  # Create classifiers  rf = RandomForestClassifier()  et = ExtraTreesClassifier()  knn = KNeighborsClassifier()  svc = SVC()  rg = RidgeClassifier()  # Build array of classifiers.  classifierArray = [rf, et, knn, svc, rg]  def showStats(classifier, scores):  print(classifier + ": ", end="")  strMean = str(round(scores.mean(),2))  strStd = str(round(scores.std(),2))  print("Mean: " + strMean + " ", end="")  print("Std: " + strStd)  from sklearn import metrics  from sklearn.model\_selection import train\_test\_split  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.30)  def evaluateModel(model, X\_test, y\_test, title):  print("\n\*\*\* " + title + " \*\*\*")  predictions = model.predict(X\_test)  accuracy = metrics.accuracy\_score(y\_test, predictions)  recall = metrics.recall\_score(y\_test, predictions, average='weighted')  precision = metrics.precision\_score(y\_test, predictions, average='weighted')  f1 = metrics.f1\_score(y\_test, predictions, average='weighted')  print("Accuracy: " + str(accuracy))  print("Precision: " + str(precision))  print("Recall: " + str(recall))  print("F1: " + str(f1))  # Search for the best classifier.  for clf in classifierArray:  modelType = clf.\_\_class\_\_.\_\_name\_\_  # Create and evaluate stand-alone model.  clfModel = clf.fit(X\_train, y\_train)  evaluateModel(clfModel, X\_test, y\_test, modelType)  # max\_features means the maximum number of features to draw from X.  # max\_samples sets the percentage of available data used for fitting.  bagging\_clf = BaggingClassifier(clf, max\_samples=0.4, max\_features=11,  n\_estimators=10)  baggedModel = bagging\_clf.fit(X\_train, y\_train)  evaluateModel(baggedModel, X\_test, y\_test, "Bagged: " + modelType) |

Exercise (2 marks)

Compared to when max\_features is 11, what happens if you set max\_features to 3. Do the results improve? Please explain.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Max feature is the maximum number of features to draw from X. So it affects the accuracy. ncreasing max\_features generally **improves the performance of the model** as at each node now we have a higher number of options to be considered. But in this case, less max features gives us the better accuracy.     |  |  | | --- | --- | | Max feature:11 | Max features 3 | | \*\*\* RandomForestClassifier \*\*\*  Accuracy: 0.6646341463414634  Precision: 0.6586214344116876  Recall: 0.6646341463414634  F1: 0.6602221542891117  \*\*\* Bagged: RandomForestClassifier \*\*\*  Accuracy: 0.7195121951219512  Precision: 0.7247920810461614  Recall: 0.7195121951219512  F1: 0.7216538451954594  \*\*\* ExtraTreesClassifier \*\*\*  Accuracy: 0.5792682926829268  Precision: 0.5820845651638334  Recall: 0.5792682926829268  F1: 0.5790242387752242  \*\*\* Bagged: ExtraTreesClassifier \*\*\*  Accuracy: 0.6219512195121951  Precision: 0.6264452011403231  Recall: 0.6219512195121951  F1: 0.6222015859576289  \*\*\* KNeighborsClassifier \*\*\*  Accuracy: 0.5304878048780488  Precision: 0.522309682187731  Recall: 0.5304878048780488  F1: 0.5187947005081566  \*\*\* Bagged: KNeighborsClassifier \*\*\*  Accuracy: 0.5792682926829268  Precision: 0.5706937595834574  Recall: 0.5792682926829268  F1: 0.5387280058942692  Accuracy: 0.5548780487804879  Precision: 0.35872841874486167  Recall: 0.5548780487804879  F1: 0.43566021867115234  \*\*\* Bagged: SVC \*\*\*  Accuracy: 0.5670731707317073  Precision: 0.5450649554843655  Recall: 0.5670731707317073  F1: 0.4738506160846508  \*\*\* RidgeClassifier \*\*\*  Accuracy: 0.6341463414634146  Precision: 0.6090298266576815  Recall: 0.6341463414634146  F1: 0.6042214439639596  \*\*\* Bagged: RidgeClassifier \*\*\*  Accuracy: 0.6402439024390244  Precision: 0.6214153019031068  Recall: 0.6402439024390244  F1: 0.6166463486683785 | \*\*\* RandomForestClassifier \*\*\*  Accuracy: 0.6158536585365854  Precision: 0.6258153047632171  Recall: 0.6158536585365854  F1: 0.6200407288821922  \*\*\* Bagged: RandomForestClassifier \*\*\*  Accuracy: 0.6402439024390244  Precision: 0.61699495272484  Recall: 0.6402439024390244  F1: 0.6222316544048293  \*\*\* ExtraTreesClassifier \*\*\*  Accuracy: 0.5853658536585366  Precision: 0.6016770562246566  Recall: 0.5853658536585366  F1: 0.5915147539536698  \*\*\* Bagged: ExtraTreesClassifier \*\*\*  Accuracy: 0.5548780487804879  Precision: 0.5704224789126113  Recall: 0.5548780487804879  F1: 0.5592839957944085  \*\*\* KNeighborsClassifier \*\*\*  Accuracy: 0.5  Precision: 0.5192534599465938  Recall: 0.5  F1: 0.5012413791792084  \*\*\* Bagged: KNeighborsClassifier \*\*\*  Accuracy: 0.6524390243902439  Precision: 0.6432926829268293  Recall: 0.6524390243902439  F1: 0.6451857797721513  \*\*\* SVC \*\*\*  Accuracy: 0.5182926829268293  Precision: 0.4721780414340287  Recall: 0.5182926829268293  F1: 0.46612774545317814  \*\*\* Bagged: SVC \*\*\*  Accuracy: 0.5853658536585366  Precision: 0.5905526391653861  Recall: 0.5853658536585366  F1: 0.5573852403575824  \*\*\* RidgeClassifier \*\*\*  Accuracy: 0.6280487804878049  Precision: 0.6024404754227342  Recall: 0.6280487804878049  F1: 0.6021132305754049  \*\*\* Bagged: RidgeClassifier \*\*\*  Accuracy: 0.5548780487804879  Precision: 0.4782353868797309  Recall: 0.5548780487804879  F1: 0.49086691410392363 | |

Exercise (3 marks)

Modify Example 3 by importing the LogisticRegression class.

|  |
| --- |
| from sklearn.linear\_model import LogisticRegression |

Define the logistic regression classifier as shown here:

|  |
| --- |
| lr = LogisticRegression(fit\_intercept=True, solver='liblinear') |

Modify the classifier array to also include your logistic regressor in the grid search. Show the output here:

|  |
| --- |
| \*\*\* RandomForestClassifier \*\*\*  Accuracy: 0.6524390243902439  Precision: 0.6731348775315399  Recall: 0.6524390243902439  F1: 0.6584614212756355  \*\*\* Bagged: RandomForestClassifier \*\*\*  Accuracy: 0.6646341463414634  Precision: 0.6893710018604888  Recall: 0.6646341463414634  F1: 0.6707288462913955  \*\*\* ExtraTreesClassifier \*\*\*  Accuracy: 0.6036585365853658  Precision: 0.62022885121737  Recall: 0.6036585365853658  F1: 0.6085834226915294  \*\*\* Bagged: ExtraTreesClassifier \*\*\*  Accuracy: 0.6585365853658537  Precision: 0.680574912891986  Recall: 0.6585365853658537  F1: 0.6647157456886686  \*\*\* KNeighborsClassifier \*\*\*  Accuracy: 0.5060975609756098  Precision: 0.48742733793874626  Recall: 0.5060975609756098  F1: 0.48913259895686806  \*\*\* Bagged: KNeighborsClassifier \*\*\*  Accuracy: 0.49390243902439024  Precision: 0.47317543302313775  Recall: 0.49390243902439024  F1: 0.46731358266864753  \*\*\* SVC \*\*\*  Accuracy: 0.5182926829268293  Precision: 0.49827556122643724  Recall: 0.5182926829268293  F1: 0.44309714444393017  \*\*\* Bagged: SVC \*\*\*  Accuracy: 0.5182926829268293  Precision: 0.4850174216027875  Recall: 0.5182926829268293  F1: 0.444263604104446  \*\*\* RidgeClassifier \*\*\*  Accuracy: 0.5914634146341463  Precision: 0.5551266904294658  Recall: 0.5914634146341463  F1: 0.5529567220538217  \*\*\* Bagged: RidgeClassifier \*\*\*  Accuracy: 0.6036585365853658  Precision: 0.5770486610550585  Recall: 0.6036585365853658  F1: 0.5742219893125893  \*\*\* LogisticRegression \*\*\*  Accuracy: 0.6280487804878049  Precision: 0.607014778834229  Recall: 0.6280487804878049  F1: 0.6088102277126668  \*\*\* Bagged: LogisticRegression \*\*\*  Accuracy: 0.6158536585365854  Precision: 0.6009535021658249  Recall: 0.6158536585365854  F1: 0.6045150987224158 |

How does the logistic regressor compare to the best performing classifier?

|  |
| --- |
| For the best performing classifier in this case( RandomForestClassifier), the model improved with begged but logistic regression has the better model without begging.  \*\*\* LogisticRegression \*\*\*  Accuracy: 0.6280487804878049  Precision: 0.607014778834229  Recall: 0.6280487804878049  F1: 0.6088102277126668  \*\*\* Bagged: LogisticRegression \*\*\*  Accuracy: 0.6158536585365854  Precision: 0.6009535021658249  Recall: 0.6158536585365854  F1: 0.6045150987224158  \*\*\* RandomForestClassifier \*\*\*  Accuracy: 0.6524390243902439  Precision: 0.6731348775315399  Recall: 0.6524390243902439  F1: 0.6584614212756355  \*\*\* Bagged: RandomForestClassifier \*\*\*  Accuracy: 0.6646341463414634  Precision: 0.6893710018604888  Recall: 0.6646341463414634  F1: 0.6707288462913955 |

Exercise (6 marks)

Starting with the following code:

|  |
| --- |
| import pandas as pd  from sklearn.preprocessing import LabelEncoder  PATH = "/Users/pm/Desktop/DayDocs/data/"  df = pd.read\_csv(PATH + 'loan\_default.csv')  print(df.head())  # Prepare the data.  X = df.copy()  del X['default']  y = df[['default']] |

Modify it to compare bagged and stand alone classifiers with the loan\_default data set. Show the output here:

|  |
| --- |
| \*\*\* RandomForestClassifier \*\*\*  Accuracy: 0.7619047619047619  Precision: 0.7460317460317462  Recall: 0.7619047619047619  F1: 0.7488095238095237  \*\*\* Bagged: RandomForestClassifier \*\*\*  Accuracy: 0.7523809523809524  Precision: 0.7319047619047618  Recall: 0.7523809523809524  F1: 0.7058201058201059  \*\*\* Bagged: ExtraTreesClassifier \*\*\*  Accuracy: 0.7523809523809524  Precision: 0.7319047619047618  Recall: 0.7523809523809524  F1: 0.7058201058201059  \*\*\* KNeighborsClassifier \*\*\*  Accuracy: 0.719047619047619  Precision: 0.6739805803744423  Recall: 0.719047619047619  F1: 0.6728514720648447  \*\*\* SVC \*\*\*  Accuracy: 0.7238095238095238  Precision: 0.5239002267573696  Recall: 0.7238095238095238  F1: 0.6078400420941857  \*\*\* Bagged: SVC \*\*\*  Accuracy: 0.7333333333333333  Precision: 0.7163398692810458  Recall: 0.7333333333333333  F1: 0.6444756554307116  \*\*\* RidgeClassifier \*\*\*  Accuracy: 0.7857142857142857  Precision: 0.7758484609313339  Recall: 0.7857142857142857  F1: 0.7593307593307593  \*\*\* Bagged: RidgeClassifier \*\*\*  Accuracy: 0.7285714285714285  Precision: 0.6853227174835215  Recall: 0.7285714285714285  F1: 0.6543009325618021  \*\*\* LogisticRegression \*\*\*  Accuracy: 0.7238095238095238  Precision: 0.5239002267573696  Recall: 0.7238095238095238  F1: 0.6078400420941857  \*\*\* Bagged: LogisticRegression \*\*\*  Accuracy: 0.7380952380952381  Precision: 0.7140681763797342  Recall: 0.7380952380952381  F1: 0.666430724401739 |

State which classifier performs best and explain why you made that choice.

|  |
| --- |
| \*\*\* RandomForestClassifier \*\*\*  Accuracy: 0.7619047619047619  Precision: 0.7460317460317462  Recall: 0.7619047619047619  F1: 0.7488095238095237 |

Show your code here:

|  |
| --- |
| import pandas as pd from sklearn.preprocessing import LabelEncoder  PATH = "/Users/hyerimshin/PycharmProjects/MachineLearning/datasets/" df = pd.read\_csv(PATH + 'loan\_default.csv') print(df.head())  # Prepare the data. X = df.copy() del X['default'] y = df[['default']]  from sklearn.ensemble import BaggingClassifier, \  ExtraTreesClassifier, RandomForestClassifier from sklearn.neighbors import KNeighborsClassifier from sklearn.linear\_model import RidgeClassifier from sklearn.svm import SVC from sklearn.linear\_model import LogisticRegression  # Create classifiers rf = RandomForestClassifier() et = ExtraTreesClassifier() knn = KNeighborsClassifier() svc = SVC() rg = RidgeClassifier() lr = LogisticRegression(fit\_intercept=True, solver='liblinear')  # Build array of classifiers. classifierArray = [rf, et, knn, svc, rg, lr]  def showStats(classifier, scores):  print(classifier + ": ", end="")  strMean = str(round(scores.mean(),2))   strStd = str(round(scores.std(),2))  print("Mean: " + strMean + " ", end="")  print("Std: " + strStd)  from sklearn import metrics from sklearn.model\_selection import train\_test\_split X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.30)  def evaluateModel(model, X\_test, y\_test, title):  print("\n\*\*\* " + title + " \*\*\*")  predictions = model.predict(X\_test)  accuracy = metrics.accuracy\_score(y\_test, predictions)  recall = metrics.recall\_score(y\_test, predictions, average='weighted')  precision = metrics.precision\_score(y\_test, predictions, average='weighted')  f1 = metrics.f1\_score(y\_test, predictions, average='weighted')   print("Accuracy: " + str(accuracy))  print("Precision: " + str(precision))  print("Recall: " + str(recall))  print("F1: " + str(f1))  # Search for the best classifier. for clf in classifierArray:  modelType = clf.\_\_class\_\_.\_\_name\_\_   # Create and evaluate stand-alone model.  clfModel = clf.fit(X\_train, y\_train)  evaluateModel(clfModel, X\_test, y\_test, modelType)   # max\_features means the maximum number of features to draw from X.  # max\_samples sets the percentage of available data used for fitting.  bagging\_clf = BaggingClassifier(clf, max\_samples=0.4, max\_features=3,  n\_estimators=10)  baggedModel = bagging\_clf.fit(X\_train, y\_train)  evaluateModel(baggedModel, X\_test, y\_test, "Bagged: " + modelType) |

Exercise (2 marks)

In general, how do the stand-alone classifiers compare with the bagged classifiers?

|  |
| --- |
| t is used **to deal with bias-variance trade-offs and reduces the variance of a prediction model**. Bagging avoids overfitting of data and is used for both regression and classification models, specifically for decision tree algorithms. Begged classifiers usually improved the models. |

### Bagged Regressors

In addition to classification problems, you can use bagging to create an ensemble of regressors.

Example : Bagging with Linear Regression

This example uses a bagged regressor for linear regression to predict the total petrol consumption. The results show that the learning ensemble does more poorly but we will try to fix this in the next example

\*\*\*\*\*\* Ensemble

Root Mean Squared Error: 65.7520989586188

\*\*\*\*\*\* Linear Regression

Root Mean Squared Error: 70.4921615920565

For now, here is how to implement a basic ensemble for linear regression:

|  |
| --- |
| import pandas as pd  from sklearn.ensemble import BaggingRegressor  from sklearn.linear\_model import LinearRegression  from sklearn.model\_selection import train\_test\_split  import numpy as np  from sklearn.metrics import mean\_squared\_error  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  # Load and prepare data.  FOLDER = '/users/pm/desktop/daydocs/data/'  FILE = 'petrol\_consumption.csv'  dataset = pd.read\_csv(FOLDER + FILE)  print(dataset)  X = dataset.copy()  del X['Petrol\_Consumption']  y = dataset[['Petrol\_Consumption']]  # Create random split.  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2)  def evaluateModel(model, X\_test, y\_test, title):  print("\n\*\*\*\*\*\* " + title)  predictions = model.predict(X\_test)  print('Root Mean Squared Error:',  np.sqrt(mean\_squared\_error(y\_test, predictions)))  # Build linear regression ensemble.  ensembleModel = BaggingRegressor(base\_estimator=LinearRegression(),max\_features=4,  max\_samples =0.5,  n\_estimators=10).fit(X\_train, y\_train)  evaluateModel(ensembleModel, X\_test, y\_test, "Ensemble")  # Build stand alone linear regression model.  model = LinearRegression()  model.fit(X\_train, y\_train)  evaluateModel(model, X\_test, y\_test, "Linear Regression") |

## Grid Searching

With countless parameter combinations for ensembles and potential models it helps to be able to grid search different combinations to find the optimal combination.

Example : Grid Searching Bagging Regressor Parameters

This example actually shows that it is possible to reduce the variance with an ensemble. The optimal parameter combination is found through a grid search.

Since the data split and sampling is random, the results will vary each run. However, it appears that an ensemble with a maximum of 4 features and 800 estimators can out-perform linear regression.

|  |
| --- |
| estimators features rmse type  6 800 4 30.910344 Ensemble  4 800 3 47.448549 Ensemble  10 900 4 52.893641 Ensemble  2 750 4 53.472123 Ensemble  8 900 3 70.127002 Ensemble  14 1000 4 73.186699 Ensemble  12 1000 3 110.820228 Ensemble  0 750 3 122.607095 Ensemble  7 None None 33.178561 Linear Regression  5 None None 45.119426 Linear Regression  3 None None 51.786050 Linear Regression  11 None None 52.931385 Linear Regression  9 None None 60.539137 Linear Regression  15 None None 72.666123 Linear Regression  13 None None 97.148495 Linear Regression  1 None None 116.922646 Linear Regression |

The code that performs the grid search is similar to the code in Example 4. This code however loops through different combinations of numbers of estimators and maximum features.

|  |
| --- |
| import pandas as pd  from sklearn.ensemble import BaggingRegressor  from sklearn.linear\_model import LinearRegression  from sklearn.model\_selection import train\_test\_split  import numpy as np  from sklearn.metrics import mean\_squared\_error  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  # Load and prepare data.  FOLDER = '/users/pm/desktop/daydocs/data/'  FILE = 'petrol\_consumption.csv'  dataset = pd.read\_csv(FOLDER + FILE)  print(dataset)  X = dataset.copy()  del X['Petrol\_Consumption']  y = dataset[['Petrol\_Consumption']]  feature\_combo\_list = []  def evaluateModel(model, X\_test, y\_test, title, num\_estimators, max\_features):  print("\n\*\*\*\*\*\* " + title)  predictions = model.predict(X\_test)  rmse = np.sqrt(mean\_squared\_error(y\_test, predictions))  # Store statistics and add to list.  stats = {"type":title, "rmse":rmse,  "estimators":num\_estimators, "features":max\_features}  feature\_combo\_list.append(stats)  num\_estimator\_list = [750, 800, 900, 1000]  max\_features\_list = [3, 4]  for num\_estimators in num\_estimator\_list:  for max\_features in max\_features\_list:  # Create random split.  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2)  # Build linear regression ensemble.  ensembleModel = BaggingRegressor(base\_estimator=LinearRegression(),  max\_features=max\_features,  max\_samples =0.5,  n\_estimators=num\_estimators).fit(X\_train, y\_train)  evaluateModel(ensembleModel, X\_test, y\_test, "Ensemble",  num\_estimators, max\_features)  # Build stand alone linear regression model.  model = LinearRegression()  model.fit(X\_train, y\_train)  evaluateModel(model, X\_test, y\_test, "Linear Regression", None, None)  # Build data frame with dictionary objects.  dfStats = pd.DataFrame()  print(dfStats)  for combo in feature\_combo\_list:  dfStats = dfStats.append(combo, ignore\_index=True)  # Sort and show all combinations.  # Show all rows  pd.set\_option('display.max\_rows', None)  dfStats = dfStats.sort\_values(by=['type', 'rmse'])  print(dfStats) |

Exercise (6 marks)

Modify the code in Example 5 so you can include the maximum samples in the grid search in addition to searching on the optimal number of features. Also, expand the range of maximum features to [2,3,4]. Show your revised program here:

|  |
| --- |
| **Calendar  Description automatically generated**  **Best result with 2 features.** |

Show your modified output here:

|  |
| --- |
| import pandas as pd from sklearn.ensemble import BaggingRegressor from sklearn.linear\_model import LinearRegression  from sklearn.model\_selection import train\_test\_split import numpy as np from sklearn.metrics import mean\_squared\_error  # Show all columns. pd.set\_option('display.max\_columns', None) pd.set\_option('display.width', 1000)  # Load and prepare data. FOLDER = '//Users/hyerimshin/PycharmProjects/MachineLearning/datasets/' FILE = 'petrol\_consumption.csv' dataset = pd.read\_csv(FOLDER + FILE) print(dataset) X = dataset.copy() del X['Petrol\_Consumption'] y = dataset[['Petrol\_Consumption']]  feature\_combo\_list = [] def evaluateModel(model, X\_test, y\_test, title, num\_estimators, max\_features):  print("\n\*\*\*\*\*\* " + title)  predictions = model.predict(X\_test)  rmse = np.sqrt(mean\_squared\_error(y\_test, predictions))   # Store statistics and add to list.  stats = {"type":title, "rmse":rmse,  "estimators":num\_estimators, "features":max\_features}  feature\_combo\_list.append(stats)  num\_estimator\_list = [750, 800, 900, 1000] max\_features\_list = [2, 3, 4]  for num\_estimators in num\_estimator\_list:  for max\_features in max\_features\_list:  # Create random split.  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2)   # Build linear regression ensemble.  ensembleModel = BaggingRegressor(base\_estimator=LinearRegression(),  max\_features=max\_features,  max\_samples =0.5,  n\_estimators=num\_estimators).fit(X\_train, y\_train)  evaluateModel(ensembleModel, X\_test, y\_test, "Ensemble",  num\_estimators, max\_features)   # Build stand alone linear regression model.  model = LinearRegression()  model.fit(X\_train, y\_train)  evaluateModel(model, X\_test, y\_test, "Linear Regression", None, None)  # Build data frame with dictionary objects. dfStats = pd.DataFrame() print(dfStats) for combo in feature\_combo\_list:  dfStats = dfStats.append(combo, ignore\_index=True)  # Sort and show all combinations. # Show all rows pd.set\_option('display.max\_rows', None) dfStats = dfStats.sort\_values(by=['type', 'rmse']) print(dfStats) |

## Stacked Models

A stacked model builds a model with the output from multiple models. Stacking can be applied to a combine the outputs from a wide variety of predictive models. Often, stacking helps to improve accuracy.

### Stacking Regression

Example : Stacking Regression

This example shows how to implement stacking with outputs from the following combination of regressors:

* ElasticNet
* DecisionTreeRegressor
* SVR
* AdaBoostRegressor
* BaggingRegressor
* RandomForestRegressor
* ExtraTreesRegressor

The output when running this program shows that our super learner is competitive when compared to most models.

|  |
| --- |
| **\*\* Evaluate Base Models \*\***  **RMSE:123799.119 ElasticNet**  **RMSE:352456.005 SVR**  **RMSE:188327.974 DecisionTreeRegressor**  **RMSE:155919.974 AdaBoostRegressor**  **RMSE:136916.041 RandomForestRegressor**  **RMSE:131182.792 ExtraTreesRegressor**  **\*\* Evaluate Stacked Model \*\***  **RMSE:104680.594 LinearRegression** |

Here is the code:

|  |
| --- |
| from sklearn.linear\_model import LinearRegression  from sklearn.linear\_model import ElasticNet  from sklearn.tree import DecisionTreeRegressor  from sklearn.svm import SVR  from sklearn.ensemble import AdaBoostRegressor  from sklearn.ensemble import RandomForestRegressor  from sklearn.ensemble import ExtraTreesRegressor  from sklearn.metrics import mean\_squared\_error  from sklearn.model\_selection import train\_test\_split  import numpy as np  import pandas as pd  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  # Prep data.  PATH = "/users/pm/desktop/daydocs/data/"  CSV\_DATA = "USA\_Housing.csv"  dataset = pd.read\_csv(PATH + CSV\_DATA)  print(dataset.head())  X = dataset[['Avg. Area Income','Avg. Area House Age', 'Avg. Area Number of Rooms',\  'Avg. Area Number of Bedrooms', "Area Population"]].values  y = dataset['Price']  def getUnfitModels():  models = list()  models.append(ElasticNet())  models.append(SVR(gamma='scale'))  models.append(DecisionTreeRegressor())  models.append(AdaBoostRegressor())  models.append(RandomForestRegressor(n\_estimators=10))  models.append(ExtraTreesRegressor(n\_estimators=10))  return models  def evaluateModel(y\_test, predictions, model):  mse = mean\_squared\_error(y\_test, predictions)  rmse = round(np.sqrt(mse),3)  print(" RMSE:" + str(rmse) + " " + model.\_\_class\_\_.\_\_name\_\_)  def fitBaseModels(X\_train, y\_train, X\_test, models):  dfPredictions = pd.DataFrame()  # Fit base model and store its predictions in dataframe.  for i in range(0, len(models)):  models[i].fit(X\_train, y\_train)  predictions = models[i].predict(X\_test)  colName = str(i)  # Add base model predictions to column of data frame.  dfPredictions[colName] = predictions  return dfPredictions, models  def fitStackedModel(X, y):  model = LinearRegression()  model.fit(X, y)  return model  # Split data into train, test and validation sets.  X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(X, y, test\_size=0.70)  X\_test, X\_val, y\_test, y\_val = train\_test\_split(X\_temp, y\_temp, test\_size=0.50)  # Get base models.  unfitModels = getUnfitModels()  # Fit base and stacked models.  dfPredictions, models = fitBaseModels(X\_train, y\_train, X\_test, unfitModels)  stackedModel = fitStackedModel(dfPredictions, y\_test)  # Evaluate base models with validation data.  print("\n\*\* Evaluate Base Models \*\*")  dfValidationPredictions = pd.DataFrame()  for i in range(0, len(models)):  predictions = models[i].predict(X\_val)  colName = str(i)  dfValidationPredictions[colName] = predictions  evaluateModel(y\_val, predictions, models[i])  # Evaluate stacked model with validation data.  stackedPredictions = stackedModel.predict(dfValidationPredictions)  print("\n\*\* Evaluate Stacked Model \*\*")  evaluateModel(y\_val, stackedPredictions, stackedModel) |

Exercise (3 marks)

Replace the code that generates the data in Example 6 with the following code which imports and prepares the wine quality data set.

|  |
| --- |
| PATH = "/Users/pm/Desktop/DayDocs/data/"  CSV\_DATA = "winequality.csv"  dataset = pd.read\_csv(PATH + CSV\_DATA)  X = dataset[['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',  'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',  'pH', 'sulphates','alcohol']].values  y = dataset['quality'] |

Run the code after making the changes. Show the output here:

|  |
| --- |
| from sklearn.linear\_model import LinearRegression from sklearn.linear\_model import ElasticNet from sklearn.tree import DecisionTreeRegressor from sklearn.svm import SVR from sklearn.ensemble import AdaBoostRegressor from sklearn.ensemble import RandomForestRegressor from sklearn.ensemble import ExtraTreesRegressor from sklearn.metrics import mean\_squared\_error from sklearn.model\_selection import train\_test\_split import numpy as np import pandas as pd  # Show all columns. pd.set\_option('display.max\_columns', None) pd.set\_option('display.width', 1000)  # Prep data. PATH = "/Users/hyerimshin/PycharmProjects/MachineLearning/datasets/" CSV\_DATA = "winequality.csv" dataset = pd.read\_csv(PATH + CSV\_DATA) X = dataset[['fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar',  'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density',  'pH', 'sulphates','alcohol']].values y = dataset['quality']   def getUnfitModels():  models = list()  models.append(ElasticNet())  models.append(SVR(gamma='scale'))  models.append(DecisionTreeRegressor())  models.append(AdaBoostRegressor())  models.append(RandomForestRegressor(n\_estimators=10))  models.append(ExtraTreesRegressor(n\_estimators=10))  return models  def evaluateModel(y\_test, predictions, model):  mse = mean\_squared\_error(y\_test, predictions)  rmse = round(np.sqrt(mse),3)  print(" RMSE:" + str(rmse) + " " + model.\_\_class\_\_.\_\_name\_\_)  def fitBaseModels(X\_train, y\_train, X\_test, models):  dfPredictions = pd.DataFrame()   # Fit base model and store its predictions in dataframe.  for i in range(0, len(models)):  models[i].fit(X\_train, y\_train)  predictions = models[i].predict(X\_test)  colName = str(i)  # Add base model predictions to column of data frame.  dfPredictions[colName] = predictions  return dfPredictions, models  def fitStackedModel(X, y):  model = LinearRegression()  model.fit(X, y)  return model  # Split data into train, test and validation sets. X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(X, y, test\_size=0.70) X\_test, X\_val, y\_test, y\_val = train\_test\_split(X\_temp, y\_temp, test\_size=0.50)  # Get base models. unfitModels = getUnfitModels()  # Fit base and stacked models. dfPredictions, models = fitBaseModels(X\_train, y\_train, X\_test, unfitModels) stackedModel = fitStackedModel(dfPredictions, y\_test)  # Evaluate base models with validation data. print("\n\*\* Evaluate Base Models \*\*") dfValidationPredictions = pd.DataFrame() for i in range(0, len(models)):  predictions = models[i].predict(X\_val)  colName = str(i)  dfValidationPredictions[colName] = predictions  evaluateModel(y\_val, predictions, models[i])  # Evaluate stacked model with validation data. stackedPredictions = stackedModel.predict(dfValidationPredictions) print("\n\*\* Evaluate Stacked Model \*\*") evaluateModel(y\_val, stackedPredictions, stackedModel) |

What model appears to perform the best and explain why you made that choice.

|  |
| --- |
| RMSE:0.672 ExtraTreesRegressor  Text  Description automatically generated |

### Stacking Classification

Example : Stacking for Classification

This example shows how to implement stacking for classification.

|  |
| --- |
| Precision:0.0 Recall:0.0 F1:0.0 Accuracy:0.66 LogisticRegression  Precision:0.76 Recall:0.81 F1:0.79 Accuracy:0.85 DecisionTreeClassifier  Precision:0.76 Recall:0.81 F1:0.79 Accuracy:0.85 AdaBoostClassifier  Precision:0.84 Recall:0.77 F1:0.8 Accuracy:0.87 RandomForestClassifier  \*\* Evaluate Stacked Model \*\*  Precision:0.77 Recall:0.83 F1:0.8 Accuracy:0.86 LogisticRegression |

Here is the code:

|  |
| --- |
| from sklearn.linear\_model import LogisticRegression  from sklearn.tree import DecisionTreeClassifier  from sklearn.ensemble import AdaBoostClassifier  from sklearn.ensemble import RandomForestClassifier  from sklearn.metrics import precision\_score, recall\_score, f1\_score, accuracy\_score  from sklearn.model\_selection import train\_test\_split  import numpy as np  import pandas as pd  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  # Prepare the data.  PATH = "/Users/pm/Desktop/DayDocs/data/"  CSV\_DATA = "Social\_Network\_Ads.csv"  df = pd.read\_csv(PATH + CSV\_DATA)  df = pd.get\_dummies(df,columns=['Gender'])  del df['User ID']  X = df.copy()  del X['Purchased']  y = df['Purchased']  def getUnfitModels():  models = list()  models.append(LogisticRegression())  models.append(DecisionTreeClassifier())  models.append(AdaBoostClassifier())  models.append(RandomForestClassifier(n\_estimators=10))  return models  def evaluateModel(y\_test, predictions, model):  precision = round(precision\_score(y\_test, predictions),2)  recall = round(recall\_score(y\_test, predictions), 2)  f1 = round(f1\_score(y\_test, predictions), 2)  accuracy = round(accuracy\_score(y\_test, predictions), 2)  print("Precision:" + str(precision) + " Recall:" + str(recall) +\  " F1:" + str(f1) + " Accuracy:" + str(accuracy) +\  " " + model.\_\_class\_\_.\_\_name\_\_)  def fitBaseModels(X\_train, y\_train, X\_test, models):  dfPredictions = pd.DataFrame()  # Fit base model and store its predictions in dataframe.  for i in range(0, len(models)):  models[i].fit(X\_train, y\_train)  predictions = models[i].predict(X\_test)  colName = str(i)  dfPredictions[colName] = predictions  return dfPredictions, models  def fitStackedModel(X, y):  model = LogisticRegression()  model.fit(X, y)  return model  # Split data into train, test and validation sets.  X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(X, y, test\_size=0.70)  X\_test, X\_val, y\_test, y\_val = train\_test\_split(X\_temp, y\_temp, test\_size=0.50)  # Get base models.  unfitModels = getUnfitModels()  # Fit base and stacked models.  dfPredictions, models = fitBaseModels(X\_train, y\_train, X\_test, unfitModels)  stackedModel = fitStackedModel(dfPredictions, y\_test)  # Evaluate base models with validation data.  print("\n\*\* Evaluate Base Models \*\*")  dfValidationPredictions = pd.DataFrame()  for i in range(0, len(models)):  predictions = models[i].predict(X\_val)  colName = str(i)  dfValidationPredictions[colName] = predictions  evaluateModel(y\_val, predictions, models[i])  # Evaluate stacked model with validation data.  stackedPredictions = stackedModel.predict(dfValidationPredictions)  print("\n\*\* Evaluate Stacked Model \*\*")  evaluateModel(y\_val, stackedPredictions, stackedModel) |