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. |  |

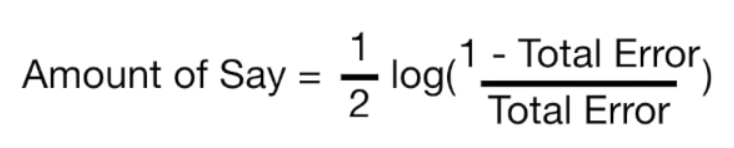
|  |  |
| --- | --- |
| Errors that are made by the first stump influence how the second stump is made and so on…. |  |

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

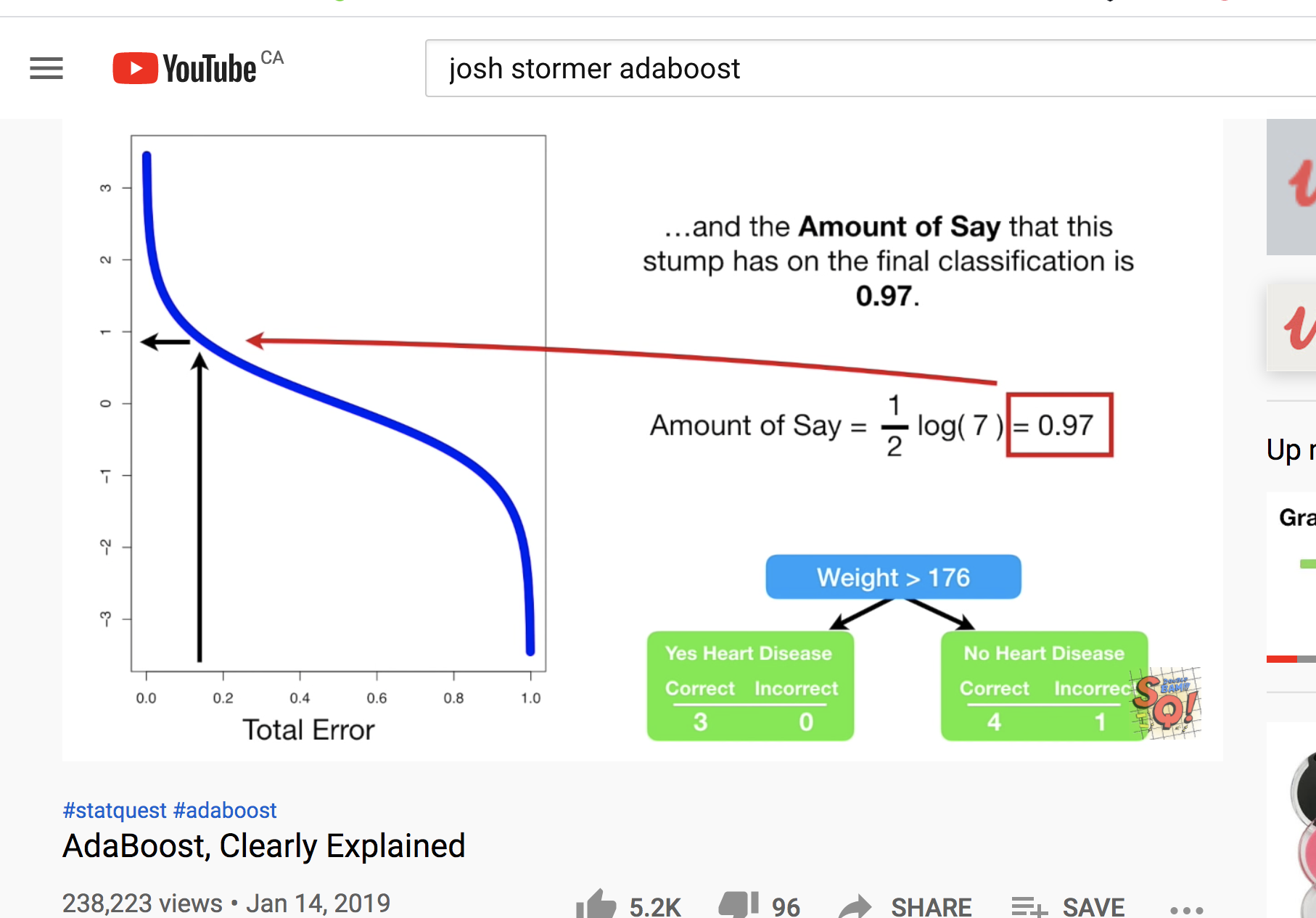
|  |  |
| --- | --- |
| Patient weight turns out to be the best variable to perform the first prediction for heart disease. Only the prediction at the very right is incorrect. It is no coincidence that Gini index is also the lowest compared to all over predictor variables. This stump becomes the first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

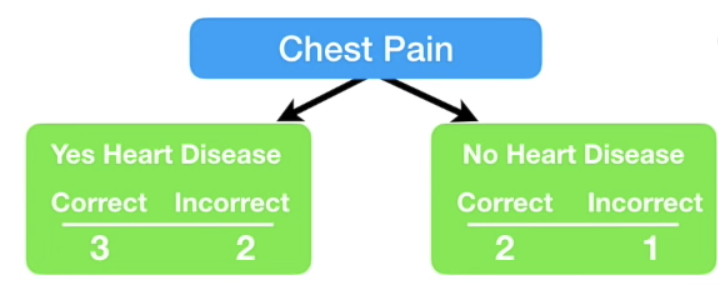


When total error is small, the amount of say for the sample is large. When the total error is large the amount of say is small. When the total error is 1/8 the amount of say is 0.97.



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 = ½ \* ln((1-total error)/total error)  Amount of say = 1\2 \* ln((1-0.375)/0.375) = 0.25509 🡪 The amount of say this stump would have had is 0. 25509 |

|  |  |
| --- | --- |
| 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.

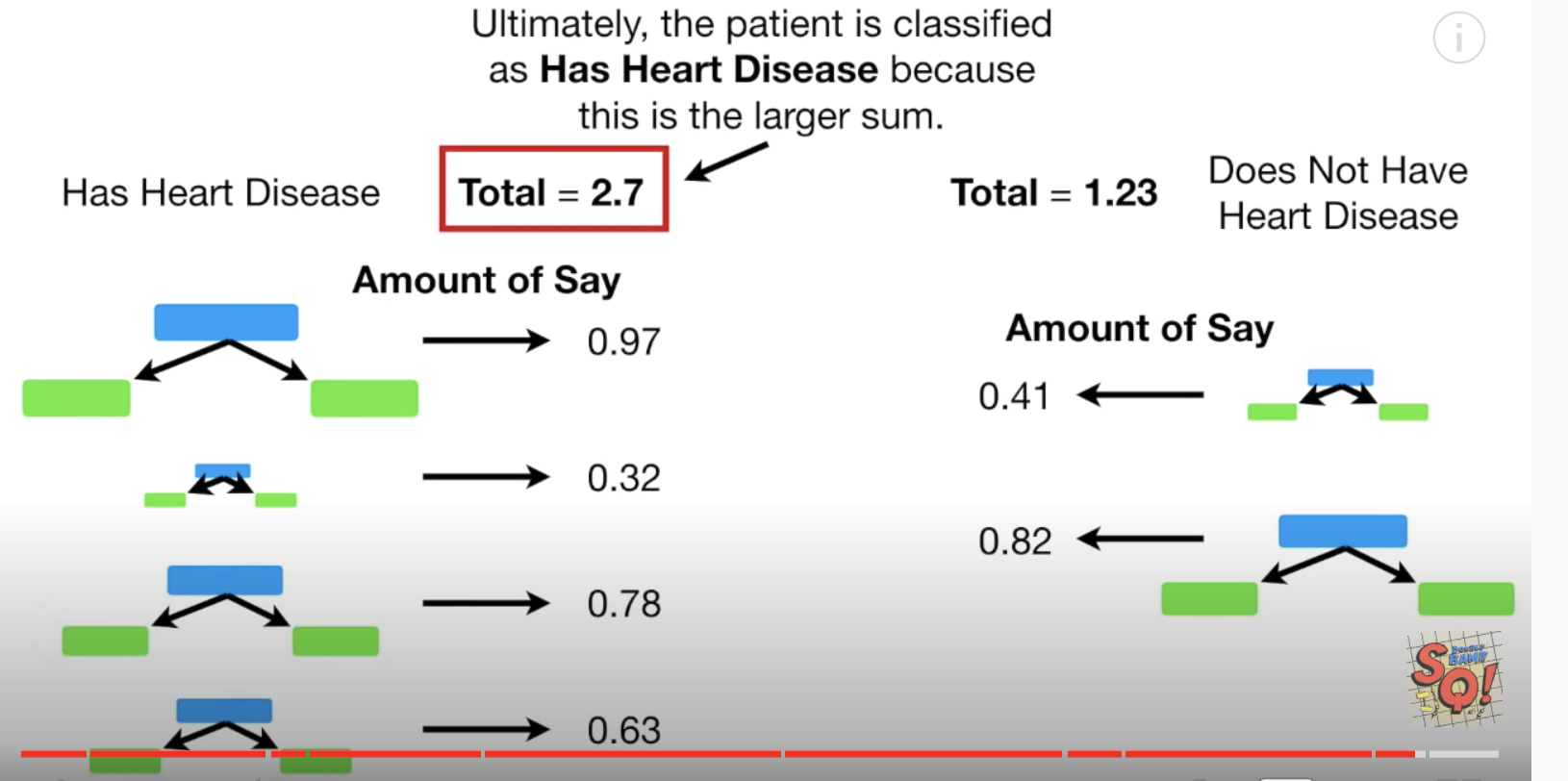
= = 0.0736

= = 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* pydataset *import* data  *import* pandas  *as* pd  *# Pandas is used for data manipulation*  *from* sklearn *import* datasets  *# Get the housing data*  *# df = data('Housing')*  iris = datasets.load\_iris()  *import* numpy *as* np  *# Creating a DataFrame of given iris dataset.*  *import* pandas *as* pd  df=pd.DataFrame({      'sepal length':iris.data[:,0],      'sepal width':iris.data[:,1],      'petal length':iris.data[:,2],      'petal width':iris.data[:,3],      'species':iris.target  })  iris['target\_names']  *# Import train\_test\_split function*  *# Show all columns.*  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  print("\n\*\*\* Before data prep.")  print(df.head(150))  print(df.tail(150))  *# Convert species variable into evenly distributed categories [0, 1, 2].*  df['species'] = pd.qcut(df['species'], 3, *labels*=[0,1,2]).cat.codes  print(df['species'].value\_counts())  y = df['species']  X = df.drop('species', 1)  *# Show prepared data.*  print("\n\*\*\* X")  print(X.head(5))  print("\n\*\*\* y")  print(y.head(5))  *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)) |

Show the output from your code:

|  |
| --- |
| \*\*\* X  sepal length sepal width petal length petal width  0 5.1 3.5 1.4 0.2  1 4.9 3.0 1.4 0.2  2 4.7 3.2 1.3 0.2  3 4.6 3.1 1.5 0.2  4 5.0 3.6 1.4 0.2  \*\*\* y  0 0  1 0  2 0  3 0  4 0  Name: species, dtype: int8  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.

|  |
| --- |
| I think the XG boost performed the best, it had a similar mean to both the Grad and Ensemble but had a lower standard deviation. |

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

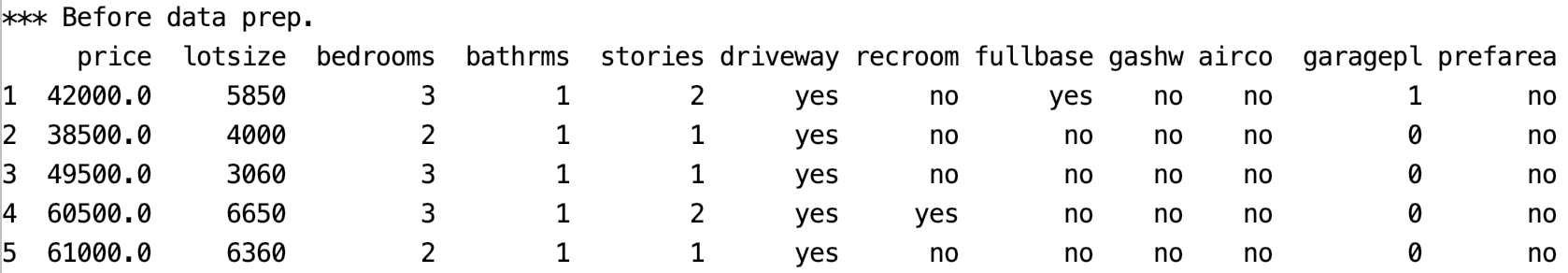
### map()

map() translates text encodings to specific numeric labels.

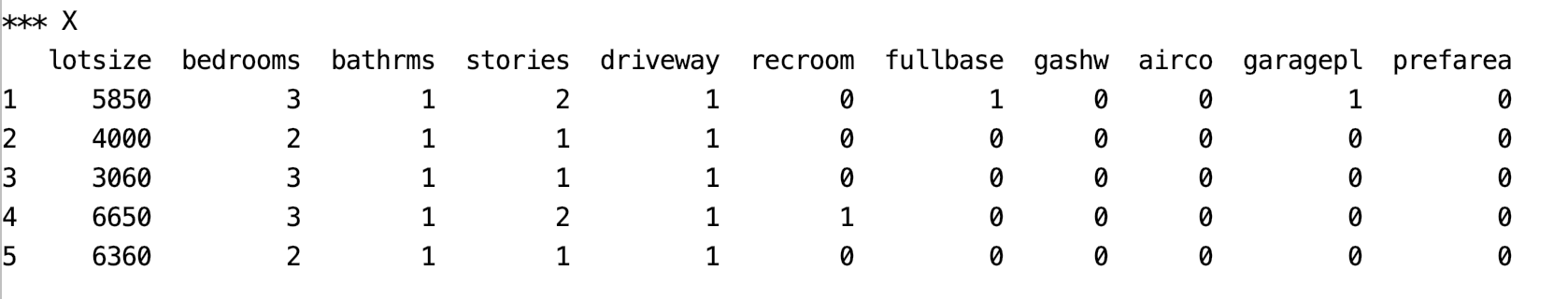
df[columns[i]] = df[columns[i]].map({'yes': 1, 'no': 0})

Example : Data Preparation

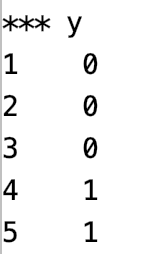
This example prepares data for modelling. The prepared data will be used in Example 3. Initially, the data looks like the following:



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.

|  |
| --- |
| The results do not appear to improve when changing the max features to 3. They are actually about the same compared to the original (11 max features)  comparing the RidgeClassifier scores and Bagged Ridge Classifier Scores 11 max features: \*\*\* Bagged: RidgeClassifier \*\*\*  Accuracy: 0.6707317073170732  Precision: 0.6651097650308136  Recall: 0.6707317073170732  F1: 0.6676851603680872  \*\*\* RidgeClassifier \*\*\*  Accuracy: 0.6707317073170732  Precision: 0.6602523492767395  Recall: 0.6707317073170732  F1: 0.6641046958311096  **3 max features:**  \*\*\* Bagged: RidgeClassifier \*\*\*  Accuracy: 0.6158536585365854  Precision: 0.4944689904786748  Recall: 0.6158536585365854  F1: 0.5284386759601903  \*\*\* RidgeClassifier \*\*\*  Accuracy: 0.7195121951219512  Precision: 0.7067220832464735  Recall: 0.7195121951219512  F1: 0.696166228579579  It looks like the scores are comparable with the 3 max features performing better in the un-bagged ridge classifier. |

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:

|  |
| --- |
| \*\*\* LogisticRegression \*\*\*  Accuracy: 0.6341463414634146  Precision: 0.6406118839014102  Recall: 0.6341463414634146  F1: 0.6037967353036208  \*\*\* Bagged: LogisticRegression \*\*\*  Accuracy: 0.5975609756097561  Precision: 0.5786849439328746  Recall: 0.5975609756097561  F1: 0.5740998838559814 |

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

|  |
| --- |
| The logistic regressor appears to perform a little worse compared to the best performing classifier (Bagged Random Forest Classifier)  \*\*\* Bagged: RandomForestClassifier \*\*\*  Accuracy: 0.6829268292682927  Precision: 0.6877200424178156  Recall: 0.6829268292682927  F1: 0.6769778258324808 |

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:

|  |
| --- |
| Name: default, dtype: int64  Mean: 0.776, std: (+/-) 0.038 [Ada Boost]  Mean: 0.789, std: (+/-) 0.035 [Grad Boost]  Mean: 0.784, std: (+/-) 0.046 [XG Boost]  Mean: 0.796, std: (+/-) 0.039 [Ensemble]  \*\*\* RandomForestClassifier \*\*\*  Accuracy: 0.7904761904761904  Precision: 0.7868942814533808  Recall: 0.7904761904761904  F1: 0.7886054421768708  \*\*\* Bagged: RandomForestClassifier \*\*\*  Accuracy: 0.8095238095238095  Precision: 0.8001949100214997  Recall: 0.8095238095238095  F1: 0.8040616246498599  \*\*\* ExtraTreesClassifier \*\*\*  Accuracy: 0.7857142857142857  Precision: 0.7838718820861678  Recall: 0.7857142857142857  F1: 0.7847736109369119  \*\*\* Bagged: ExtraTreesClassifier \*\*\*  Accuracy: 0.8095238095238095  Precision: 0.7852458172458173  Recall: 0.8095238095238095  F1: 0.7891838553603261  \*\*\* KNeighborsClassifier \*\*\*  Accuracy: 0.7619047619047619  Precision: 0.7367779694325018  Recall: 0.7619047619047619  F1: 0.7467285617574635  \*\*\* Bagged: KNeighborsClassifier \*\*\*  Accuracy: 0.8  Precision: 0.7601718078875439  Recall: 0.8  F1: 0.7546703296703297  \*\*\* SVC \*\*\*  \_warn\_prf(average, modifier, msg\_start, len(result))  Accuracy: 0.7952380952380952  Precision: 0.6324036281179138  Recall: 0.7952380952380952  F1: 0.7045345459138563  \*\*\* Bagged: SVC \*\*\*  \_warn\_prf(average, modifier, msg\_start, len(result))  Accuracy: 0.7952380952380952  Precision: 0.6324036281179138  Recall: 0.7952380952380952  F1: 0.7045345459138563  \*\*\* RidgeClassifier \*\*\*  Accuracy: 0.8380952380952381  Precision: 0.8236835474930714  Recall: 0.8380952380952381  F1: 0.8152705323702515  \*\*\* Bagged: RidgeClassifier \*\*\*  Accuracy: 0.8380952380952381  Precision: 0.8228606857762868  Recall: 0.8380952380952381  F1: 0.8181377944089808  \*\*\* LogisticRegression \*\*\*  \_warn\_prf(average, modifier, msg\_start, len(result))  Accuracy: 0.7952380952380952  Precision: 0.6324036281179138  Recall: 0.7952380952380952  F1: 0.7045345459138563  \*\*\* Bagged: LogisticRegression \*\*\*  Accuracy: 0.7952380952380952  Precision: 0.6324036281179138  Recall: 0.7952380952380952  F1: 0.7045345459138563 |

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

|  |
| --- |
| The best performing classifier is the RidgeClassifier, which just slightly beats out the Bagged RidgeClassifier. I picked this because it has the highest accuracy, precision and recall out of all of the classifiers. |

Show your code here:

|  |
| --- |
| *from* pydataset *import* data  *import* pandas  *as* pd  *import* pandas                *as*     pd  *from* sklearn.preprocessing   *import* LabelEncoder  df         = pd.read\_csv('loan\_default.csv')  print(df.head())  *# Prepare the data.*  X = df.copy()  *del* X['default']  y = df[['default']]  *# 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['default'] = pd.qcut(df['default'], 3, labels=[0,1,2]).cat.codes*  *# Show new price variable cateogories.*  print("\nNewly categorized target (price) values")  print(df['default'].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['default']  X = df.drop('default', 1)  *# Show prepared data.*  print("\n\*\*\* X")  print(X.head(5))  print("\n\*\*\* y")  print(y.head(5))  *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))  *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*=7,  *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?

|  |
| --- |
| In general, stand-alone classifiers are supposed to be inferior to bagged calssifers in term of accuracy. |

### 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:

|  |
| --- |
| *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(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* =1,  *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) |

Show your modified output here:

|  |
| --- |
| **type rmse estimators features**  **2 Ensemble 66.193908 750 3**  **22 Ensemble 75.309278 1000 4**  **16 Ensemble 80.040956 900 4**  **10 Ensemble 81.057339 800 4**  **6 Ensemble 88.753033 800 2**  **12 Ensemble 97.284010 900 2**  **18 Ensemble 100.417759 1000 2**  **8 Ensemble 102.570488 800 3**  **14 Ensemble 118.895068 900 3**  **20 Ensemble 122.795285 1000 3**  **0 Ensemble 128.675312 750 2**  **4 Ensemble 166.479670 750 4**  **17 Linear Regression 41.852370 None None**  **1 Linear Regression 45.062384 None None**  **3 Linear Regression 49.815306 None None**  **9 Linear Regression 52.777975 None None**  **13 Linear Regression 54.329817 None None**  **7 Linear Regression 57.892409 None None**  **15 Linear Regression 59.804450 None None**  **11 Linear Regression 66.177207 None None**  **21 Linear Regression 68.631918 None None**  **19 Linear Regression 70.993945 None None**  **23 Linear Regression 73.014241 None None**  **5 Linear Regression 98.072368 None None** |

## 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:

|  |
| --- |
| \*\* Evaluate Base Models \*\*  RMSE:0.783 ElasticNet  RMSE:0.755 SVR  RMSE:0.876 DecisionTreeRegressor  RMSE:0.639 AdaBoostRegressor  RMSE:0.65 RandomForestRegressor  RMSE:0.617 ExtraTreesRegressor  \*\* Evaluate Stacked Model \*\*  RMSE:0.6 LinearRegression |

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

|  |
| --- |
| On its own, the ExtraTreesRegressor model appears to have performed the best. This is because its RMSE is the lowest. |

### 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) |