Table of Contents

[Bagging 1](#_Toc124690241)

[Bagged Classifier 1](#_Toc124690242)

[Bagged Regressors 4](#_Toc124690243)

[Grid Searching 5](#_Toc124690244)

[Boosting 8](#_Toc124690245)

[Adaptive Boosting 8](#_Toc124690246)

[Gradient Boosting 12](#_Toc124690247)

[Xtreme Gradient Boosting 12](#_Toc124690248)

[Ensembles 14](#_Toc124690249)

[Stacked Models 15](#_Toc124690250)

[Stacking Regression 15](#_Toc124690251)

[Stacking Classification 18](#_Toc124690252)

## 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.

A random forest is a great example of a bagging algorithm since it combines many weak decision trees with random data. Other types of learners can be bagged as well.

### 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:

* KNeighborsClassifier()
* SVC()
* RidgeClassifier()

Bagging often offers a slight improvement in accuracy with generally less variance in most cases. Overall the KNeighborsClassifier appears to gain the most from bagging. (see Table 1). However, when bagging or not, RidgeClassifier scores already are superior to KNeighborsClassifier.

Table : Comparing Stand-Alone and Bagged Classifiers

|  |
| --- |
| \*\*\* KNeighborsClassifier \*\*\*  precision recall f1-score support  0 0.55 0.72 0.62 53  1 0.35 0.23 0.28 53  2 0.64 0.67 0.66 58    \*\*\* Bagged: KNeighborsClassifier \*\*\*  precision recall f1-score support  0 0.64 0.79 0.71 53  1 0.58 0.26 0.36 53  2 0.66 0.84 0.74 58  \*\*\* SVC \*\*\*  precision recall f1-score support  0 0.57 0.81 0.67 53  1 0.54 0.13 0.21 53  2 0.63 0.81 0.71 58  \*\*\* Bagged: SVC \*\*\*  precision recall f1-score support  0 0.60 0.83 0.70 53  1 0.53 0.17 0.26 53  2 0.62 0.79 0.70 58  \*\*\* RidgeClassifier \*\*\*  precision recall f1-score support  0 0.62 0.83 0.71 53  1 0.50 0.25 0.33 53  2 0.75 0.86 0.80 58  \*\*\* Bagged: RidgeClassifier \*\*\*  precision recall f1-score support  0 0.58 0.83 0.68 53  1 0.47 0.13 0.21 53  2 0.70 0.88 0.78 58 |

Here is the code for this sample:

|  |
| --- |
| import pandas as pd  # Get the housing data  df = pd.read\_csv('housing\_classification.csv')  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  print(df.head(5))  # Split into two sets  y = df['price']  X = df.drop('price', axis=1)  from sklearn.ensemble import BaggingClassifier  from sklearn.neighbors import KNeighborsClassifier  from sklearn.linear\_model import RidgeClassifier  from sklearn.svm import SVC  from sklearn.metrics import classification\_report  # Create classifiers  knn = KNeighborsClassifier()  svc = SVC()  rg = RidgeClassifier()  # Build array of classifiers.  classifierArray = [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.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)  report = classification\_report(y\_test, predictions)  print(report)  # 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=6,  n\_estimators=100)  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 for KNeighborsClassifier improve? Please explain.

|  |
| --- |
|  |

Exercise (3 marks)

Modify Example 1 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:

|  |
| --- |
|  |

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

|  |
| --- |
|  |

### 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 2. 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)  import numpy as np  # Build linear regression ensemble.  ensembleModel = BaggingRegressor(estimator=LinearRegression(),  max\_features=max\_features,  # Can be percent (float) or actual  # total of samples (int).  max\_samples =0.5,  n\_estimators=num\_estimators).fit(X\_train, y\_train.values.ravel())  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 = pd.concat([dfStats,  pd.DataFrame.from\_records([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 3 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 [0.2, 0.4, 0.6]. Show your revised program here:

|  |
| --- |
|  |

Show your modified output here:

|  |
| --- |
|  |

## 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.

This article offers a good summary of boosting algorithms.

<https://aws.amazon.com/what-is/boosting/>

Adaptive Boosting

With adaptive boosting, when a new model is created, samples that are incorrectly predicted are weighted higher in the sample pool. Subsequent models that are built have a better chance to learn from past mistakes. Adaptive boosting is suitable for classification. 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:

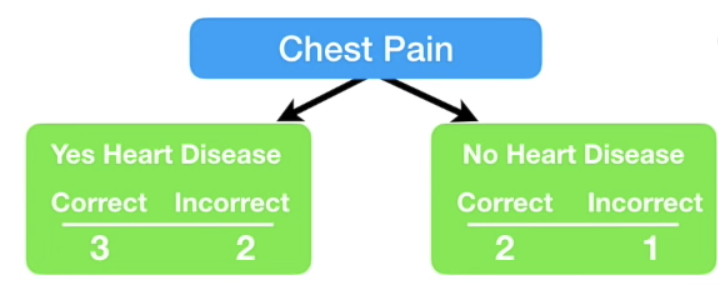
Amount of Say =

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:

|  |
| --- |
|  |

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

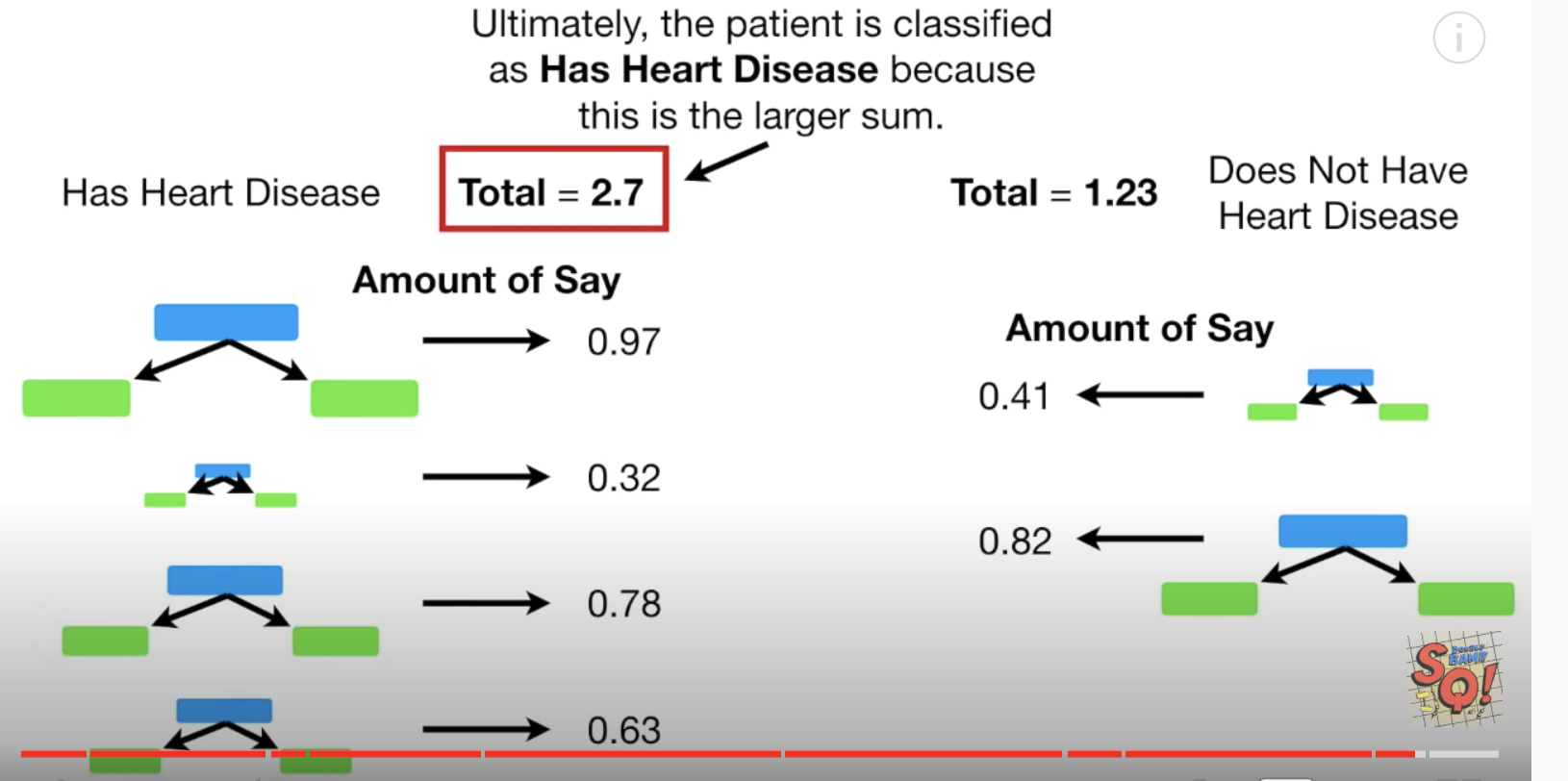
= = 0.0736

= = 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.



### Gradient Boosting

Gradient Boosting (GB) is similar to AdaBoost in that it, too, is a sequential training technique. The difference between AdaBoost and GB is that GB does not give incorrectly classified items more weight. Instead, GB software optimizes the loss function by generating base learners sequentially so that the present base learner is always more effective than the previous one. This method attempts to generate accurate results initially instead of correcting errors throughout the process, like AdaBoost. For this reason, GB software can lead to more accurate results. Gradient Boosting can help with both classification and regression-based problems.

### Xtreme Gradient Boosting

Extreme Gradient Boosting (XGBoost) improves gradient boosting for computational speed and scale in several ways. XGBoost uses multiple cores on the CPU so that learning can occur in parallel during training. It is a boosting algorithm that can handle extensive datasets, making it attractive for big data applications. The key features of XGBoost are parallelization, distributed computing, cache optimization, and out-of-core processing.

Example : Boosting

This example compares the performance of several boosting classifiers.

ada\_boost = AdaBoostClassifier()

grad\_boost = GradientBoostingClassifier()

xgb\_boost = XGBClassifier()

|  |
| --- |
| AdaBoostClassifier  precision recall f1-score support  0 0.68 0.69 0.69 52  1 0.54 0.61 0.57 54  2 0.88 0.76 0.81 58  GradientBoostingClassifier  precision recall f1-score support  0 0.73 0.69 0.71 52  1 0.52 0.59 0.55 54  2 0.79 0.72 0.76 58  XGBClassifier  precision recall f1-score support  0 0.65 0.65 0.65 52  1 0.49 0.56 0.52 54  2 0.84 0.74 0.79 58 |

Here is the code:

|  |
| --- |
| import pandas as pd  from sklearn.metrics import classification\_report  # Get the housing data  df = pd.read\_csv('housing\_classification.csv')  # Show all columns.  pd.set\_option('display.max\_columns', None)  pd.set\_option('display.width', 1000)  print(df.head(5))  # Split into two sets  y = df['price']  X = df.drop('price', axis=1)  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)  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()  classifiers = [ada\_boost, grad\_boost, xgb\_boost]  for clf in classifiers:  print(clf.\_\_class\_\_.\_\_name\_\_)  clf.fit(X\_train, y\_train)  predictions = clf.predict(X\_test)  report = classification\_report(y\_test, predictions)  print(report) |

Exercise (2 marks)

Add a reference for LogisticRegression to Example 4 and include it in the classifier array.

|  |
| --- |
| from sklearn.linear\_model import LogisticRegression  lr = LogisticRegression(fit\_intercept=True, solver='liblinear') |

Show the output for Logistic Regression here:

|  |
| --- |
|  |

How do the results for LogisticRegression compare with the other classifier algorithms?

|  |
| --- |
|  |

Exercise (4 marks)

Using code from Example 4, compare the different boosting algorithm performances for the Iris data set. You could use the following code to load and prepare the Iris data.

|  |
| --- |
| # Get the housing data  df = pd.read\_csv('/Users/pm/Downloads/iris\_v2.csv')  dict\_map = {'Iris-setosa':0,'Iris-versicolor':1, 'Iris-virginica':2}  df['target'] = df['iris\_type'].map(dict\_map)  y = df['target']  X = df.copy()  del X['target']  del X['iris\_type'] |

Show your working program here:

|  |
| --- |
|  |

Show the output from your code:

|  |
| --- |
|  |

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

|  |
| --- |
|  |

## Ensembles

Ensembles allow you to average the predictions from multiple models to provide a more stable and accurate model.

eclf = EnsembleVoteClassifier(clfs=[ada\_boost, grad\_boost,

xgb\_boost], voting='hard')

* ‘hard’ voting chooses the majority of votes
* ‘soft’ voting averages the percentages of votes for each class.

Ensemble models do not usually end up being the best model but they do weight results from the best models more heavily. As a result ensemble models tend to be very competitive.

Example : Ensembles

Starting with Example 4, replace the definition for the *classifiers* list with the following code which groups the classifiers.

|  |
| --- |
| eclf = EnsembleVoteClassifier(clfs=[ada\_boost, grad\_boost,  xgb\_boost], voting='hard')  classifiers = [ada\_boost, grad\_boost, xgb\_boost, eclf] |

The output:

|  |
| --- |
| AdaBoostClassifier  precision recall f1-score support  0 0.60 0.63 0.61 54  1 0.39 0.47 0.43 51  2 0.80 0.63 0.70 59    GradientBoostingClassifier  precision recall f1-score support  0 0.59 0.65 0.62 54  1 0.45 0.49 0.47 51  2 0.82 0.68 0.74 59  XGBClassifier  precision recall f1-score support  0 0.60 0.61 0.61 54  1 0.46 0.53 0.49 51  2 0.84 0.71 0.77 59  EnsembleVoteClassifier  precision recall f1-score support  0 0.61 0.67 0.64 54  1 0.45 0.49 0.47 51  2 0.82 0.68 0.74 59 |

Exercise (2 marks)

Describe how the ensemble model in Example 5 performs compared to the individual base models for each of the classes that are being predicted.

|  |
| --- |
|  |

## 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=200))  models.append(ExtraTreesRegressor(n\_estimators=200))  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\_val, unfitModels)  stackedModel = fitStackedModel(dfPredictions, y\_val)  # 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\_test)  colName = str(i)  dfValidationPredictions[colName] = predictions  evaluateModel(y\_test, predictions, models[i])  # Evaluate stacked model with validation data.  stackedPredictions = stackedModel.predict(dfValidationPredictions)  print("\n\*\* Evaluate Stacked Model \*\*")  evaluateModel(y\_test, stackedPredictions, stackedModel) |

Exercise (3 marks)

Show a screenshot of the data frame that is used for input that is used to fit the stacked model in this exercise. Please make it reader-friendly by whatever means you feel is appropriate.

|  |
| --- |
|  |

Explain what the columns of the data frame referenced above within this exercise represent.

|  |
| --- |
|  |

Exercise (3 marks)

Replace the code that retrieves the data in Example 6 with the following code which imports and prepares the wine quality data set. Make any other adjustments as required.

|  |
| --- |
| 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:

|  |
| --- |
|  |

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

|  |
| --- |
|  |

### Stacking Classification

Example : Stacking for Classification

This example shows how to implement stacking for classification.

|  |
| --- |
| \*\* Evaluate Base Models \*\*  \*\*\* LogisticRegression  precision recall f1-score support  0 0.64 1.00 0.78 89  1 0.00 0.00 0.00 51  \*\*\* DecisionTreeClassifier  precision recall f1-score support  0 0.84 0.88 0.86 89  1 0.77 0.71 0.73 51  \*\*\* AdaBoostClassifier  precision recall f1-score support  0 0.80 0.89 0.84 89  1 0.76 0.61 0.67 51  \*\*\* RandomForestClassifier  precision recall f1-score support  0 0.84 0.88 0.86 89  1 0.77 0.71 0.73 51  \*\* Evaluate Stacked Model \*\*  \*\*\* LogisticRegression  precision recall f1-score support  0 0.83 0.88 0.85 89  1 0.76 0.69 0.72 51 |

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 classification\_report  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):  print("\n\*\*\* " + model.\_\_class\_\_.\_\_name\_\_)  report = classification\_report(y\_test, predictions)  print(report)  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\_val, unfitModels)  stackedModel = fitStackedModel(dfPredictions, y\_val)  # 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\_test)  colName = str(i)  dfValidationPredictions[colName] = predictions  evaluateModel(y\_test, predictions, models[i])  # Evaluate stacked model with validation data.  stackedPredictions = stackedModel.predict(dfValidationPredictions)  print("\n\*\* Evaluate Stacked Model \*\*")  evaluateModel(y\_test, stackedPredictions, stackedModel) |

Exercise (3 marks)

Show a screenshot of the data frame that is used for input that is used to fit the stacked model in this exercise. Please make it reader-friendly by whatever means you feel is appropriate.

|  |
| --- |
|  |

Explain what the columns of the data frame referenced above within this exercise represent.

|  |
| --- |
|  |

Exercise (3 marks)

Replace the code that retrieves the data in Example 6 with the following code which imports and prepares the iris data set. Make any other adjustments as required.

|  |
| --- |
| # Get the housing data  df = pd.read\_csv('/Users/pm/Downloads/iris\_v2.csv')  dict\_map = {'Iris-setosa':0,'Iris-versicolor':1, 'Iris-virginica':2}  df['target'] = df['iris\_type'].map(dict\_map)  y = df['target']  X = df.copy()  del X['target']  del X['iris\_type'] |

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

|  |
| --- |
|  |

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

|  |
| --- |
|  |