Homework #2
Alex Teboul

Casey Bennett, PhD
DSC540, Winter 2019
DePaul University

Overview

There are two python scripts, implementing Scikit using the same basic template you were shown in class. The first one uses a dataset related to Diabetes from Pima Indians for classification, the target variable being the presence/absence of diabetes in individuals. The second script uses a dataset of quality ratings for various red wines rated 1-10, for regression. We will explore ways of using Random Forests and Bagging to create models, as well as talk about the effects of feature normalization.

For classification, we will be creating an object we'll name 'clf', and for regression we'll name 'rgr'. These are objects we can call methods on (such as fitting a model to some data), and access their internal variables (such as getting predicted class labels). Scikit API links for Random Forests describing methods and variables available can be found in the included links document.

**Follow the steps below, record answers to questions in a word document, and turn in both your completed code and the word doc.*

Pima Diabetes

Open up HW2_Diabetes.py - ******Code at Link Below******
https://colab.research.google.com/drive/1rWD7opvLA1HSA6ttZZbWh0OH2E9wVrld?authuser=1#scrollTo=AZsv74jbp2O8&line=159&uniqifier=1

1) First, let's run a simple test/train split using a random forest. To do so, we need to do two things, first create a Random Forest Classifier object (clf), then "fit" some data using that object.
    a. Note on line 191, the test/training data is already split for you into separate arrays for both the features (data) and the target
    b. On line 196, we need to create a Random Forest Classifier. We can do this using RandomForestClassifier(). Using the API link in the accompanying document, call that function, and pass in the following parameters:
        i. Set number of trees to 100
            1. `n_estimators`
        ii. Set maximum depth = None
            1. `max_depth`
        iii. Set minimum # of samples for split to occur = 3
            1. `min_samples_split`
        iv. Set splitting criterion = 'entropy'
            1. `criterion`
        v. Set random_state variable to rand_st
            1. rand_st=1

c. Now we need to fit data. On line 197, call a fit() function on the clf object. It takes two arrays, the first for the training set feature data (data_train) and the second for the training set target data (target_train).
d. Note that performance is already be calculated here for you on lines 199 and 201, using the test set (data_test and target_test)

```
195    #SciKit Random Forest
196    clf = RandomForestClassifier( n_estimators=100, max_depth=None, min_samples_split=3,criterion='entropy', random_state=rand_st)
197    clf.fit(data_train,target_train)
198
```

Question #1a: Run the code 5 times, record the accuracy and AUC score. What do you notice about the scores?

```
--ML Model Output--

Random Forest Acc: 0.7509293680297398
Random Forest AUC: 0.8043382722250126
```

```
--ML Model Output--

Random Forest Acc: 0.7732342007434945
Random Forest AUC: 0.8359604105571847
```

```
--ML Model Output--

Random Forest Acc: 0.7211895910780669
Random Forest AUC: 0.7979733817301875
```

```
--ML Model Output--

Random Forest Acc: 0.7546468401486989
Random Forest AUC: 0.8017561735794773
```

```
--ML Model Output--

Random Forest Acc: 0.7211895910780669
Random Forest AUC: 0.7979289940828402
```

- I notice that accuracy is in the range of 0.72-0.77 and AUC is between 0.79-0.83. This would suggest to me that this classifier is not yet reliable for medical diagnosis of diabetes. AUC up towards 0.95 would be desirable. Also AUC is higher than accuracy here, so the classifier is slightly better at predicting the positive (has diabetes) class at the cost of some false negatives or slightly lower true negatives.

Question #1b: For the fit() method of a RandomForestClassifier, it lists three possible parameters on the API webpage, what are they? Define what you could pass in to each one?

fit(self, X, y[, sample_weight])	Build a forest of trees from the training set (X, y).

Build a forest of trees from the training set (X, y).

3 Parameters:

- **X:** array-like or sparse matrix of shape (n_samples, n_features)
  - This is the data to train the classifier on, in terms of features/attributes. We pass in data_train which contains features like 'Times Pregnant' and 'Age'.
- **y:** array-like of shape (n_samples,) or (n_samples, n_outputs)
  - This is target or class data that is passed in as an array. In our case classes are 1s or 0s for 'has diabetes' and 'doesn't have diabetes'. We passed in target_train here.

- **sample_weight:** array-like of shape (n_samples,), default=None
    - In our case, the samples get weighted equally so the default of None taken, but you could pass in an array to specify weights to a number of samples. This would impact splits in the decision trees created in the RF.

2) Let's repeat step 1 above, this time using cross-validation. To do so, we need to do two things, first create a Random Forest Classifier object (clf), then second pass that object and some data arrays into a "cross-validate" function.
    a. To turn on cross-validation, we need to first on line 32 change the cross_val flag to equal 1 instead of 0
    b. On line 211, we need to create a Random Forest Classifier. We can do this using RandomForestClassifier(). Using the API link in the accompanying document, call that function, and pass in the following parameters:
        i. Set number of trees = 100
        ii. Set maximum depth = None
        iii. Set minimum # of samples for split to occur = 3
        iv. Set splitting criterion = 'entropy'
        v. Set random_state variable to rand_st
    c. Now we need to run cross-validation. On line 212, call a cross_validate() function and return the value to a scores object (you can do this by setting the scores = cross_validate()). For the cross_validate function, pass in the following parameters:
        i. Set the estimator = clf
        ii. Set the data to be fit = data_np
        iii. Set the target variable = target_np
        iv. Set scoring = scorers
        v. Set the # of cv folds = 5
    d. Note we are passing in the WHOLE dataset here, not just training data or training target values
    e. Note that performance is already be calculated here for you with the scorers being setup on line 207, and the results pulled out and printed below.

```
210    start_ts=time.time()
211    clf = RandomForestClassifier( n_estimators=100, max_depth=None, min_samples_split=3,criterion='entropy', random_state=rand_st)
212    scores = cross_validate(clf, data_np, target_np, scoring=scorers, cv=5)
```

Question #2: Run the code once, record the accuracy and AUC score.

```
--ML Model Output--

Random Forest Acc: 0.77 (+/- 0.08)
Random Forest AUC: 0.83 (+/- 0.07)
CV Runtime: 1.2498631477355957
```

- **Accuracy:** 0.77 (+/- 0.08)
- **AUC:** 0.83 (+/- 0.07)

3) Let's explore how the number of trees affects performance of a Random Forest.
   a. On line 211, change the number of trees from 100 to 5

```
--ML Model Output--

Random Forest Acc: 0.74 (+/- 0.06)
Random Forest AUC: 0.77 (+/- 0.07)
CV Runtime: 0.08236861228942871
```

   b. Now set the number of trees to 10

```
--ML Model Output--

Random Forest Acc: 0.74 (+/- 0.05)
Random Forest AUC: 0.80 (+/- 0.08)
CV Runtime: 0.14772868156433105
```

   c. Now set the number of trees to 20

```
--ML Model Output--

Random Forest Acc: 0.76 (+/- 0.06)
Random Forest AUC: 0.81 (+/- 0.07)
CV Runtime: 0.265122652053833
```

   d. Now set the number of trees to 50

```
--ML Model Output--

Random Forest Acc: 0.77 (+/- 0.06)
Random Forest AUC: 0.83 (+/- 0.07)
CV Runtime: 0.6512598991394043
```

   e. Now set the number of trees to 200

```
--ML Model Output--

Random Forest Acc: 0.77 (+/- 0.07)
Random Forest AUC: 0.83 (+/- 0.06)
CV Runtime: 2.5171823501586914
```

   f. Now set the number of trees to 500

```
--ML Model Output--

Random Forest Acc: 0.78 (+/- 0.07)
Random Forest AUC: 0.83 (+/- 0.06)
CV Runtime: 6.252587795257568
```

   g. Now set the number of trees to 1000

```
--ML Model Output--

Random Forest Acc: 0.77 (+/- 0.07)
Random Forest AUC: 0.83 (+/- 0.07)
CV Runtime: 12.162900924682617
```

- **Number of trees:** 5, 10, 20, 50, 200, 500, 1000
- **Accuracy**: 0.74 (+/- 0.06), 0.74 (+/- 0.05), 0.76 (+/- 0.06), 0.77 (+/- 0.06), 0.77 (+/- 0.07), 0.78 (+/- 0.07), 0.77 (+/- 0.07)
- **AUC:** 0.77 (+/- 0.07), 0.80 (+/- 0.08), 0.81 (+/- 0.07), 0.83 (+/- 0.07), 0.83 (+/- 0.06), 0.83 (+/- 0.06), 0.83 (+/- 0.06)

Question #3: Run the code once for each setting of the number of trees (5,10,20,50,100,200,500, 1000), record the accuracy and AUC scores. What do you notice about the scores? How do they change as the number of trees increases?

Pima Question 3: Results **Scores with Different #trees** **Number of trees:** 5, 10, 20, 50, 200, 500, 1000		
# trees	**Acc**	**AUC**
5	0.74 (+/- 0.06)	0.77 (+/- 0.07)
10	0.74 (+/- 0.05)	0.80 (+/- 0.08)
20	0.76 (+/- 0.06)	0.81 (+/- 0.07)
50	0.77 (+/- 0.06)	0.83 (+/- 0.07)
200	0.77 (+/- 0.07)	0.83 (+/- 0.06)
500	0.78 (+/- 0.07)	0.83 (+/- 0.06)
1000	0.77 (+/- 0.07)	0.83 (+/- 0.06)

- I notice that the accuracy scores increase slightly with increasing number of trees, but the range of scores also increases. For the small dataset we are working with of only 768 instances, such large numbers of trees don't make the computational expense too great. That said, if we had many thousands or millions of rows of data it would take a huge time to run and any insignificant gains in average accuracy are simply not worth it for adding more trees.
- AUC sees a slight increase in it's average, but it tapers off at 50 trees. Again, for so few samples, these gains in accuracy displayed here from the added trees may be insignificant if we were to apply this model to a larger dataset. Doubling the trees pretty much doubles computational time which is important to keep in mind. Anything over 500 trees seems to be really pushing it in terms of what is worth it even for this small dataset.

4) Now let's try applying feature selection method we used for the wine dataset in Homework #1 to the diabetes dataset. We will turn on the Wrapper-Based Feature Selection, which essentially builds lots of models with different subsets of features, and picks the subset that performs the best. For simplicity here though, we will just build a single subset and select the top variables. We will use the same Random Forest model for this.
    a. First, on line 211, change the number of trees back to 100
    b. To turn on feature selection, we need to first on line 37 change the feat_select flag to equal 1 instead of 0
    c. Note that there is an option to change the feature selection type on line 38, but the homework code is hard-coded to only use wrapper-based, so this doesn't matter for now
    d. You will need to add a RandomForestClassifier() call to pass to the clf object on line 148, you can use something similar to the calls used elsewhere in the code (e.g. line 196). Don't forget to set the parameters, particularly the random_state and number of trees to 100.
    e. Note the SelectFromModel() function being called on line 149, this is where the actual feature selection occurs, with the clf object being passed in

```
['Class', 'Times Pregnant', 'Blood Glucose', 'Blood Pressure', 'Skin Fold Thickness', '2-Hour Insulin', 'BMI', 'Family History', 'Age']
768 768


--FEATURE SELECTION ON--

Wrapper Select:
Selected: ['Blood Glucose', 'BMI', 'Family History', 'Age']
Features (total/selected): 8 4


--ML Model Output--

Random Forest Acc: 0.76 (+/- 0.05)
Random Forest AUC: 0.82 (+/- 0.06)
CV Runtime: 1.434706687927246
```

Question #4a: Run the code once, record the accuracy and AUC scores. What do you notice about the scores? How do they compare to the performance above in question #2?

- **Accuracy:** 0.76 (+/- 0.05)
- **AUC:** 0.82 (+/- 0.06)
- I notice that the scores are fairly similar to the scores we observed in question 2. So given the same parameter settings, the model using fewer features (4 vs. 8) achieved comparable performance. We like models that are simpler that can give the same results so this is of benefit. Runtime was a little higher for this model, but feature selection was performed so it's okay in the end. We could just choose those features to plug into the model in the future.

Question #4b: What features were selected, and which were removed?

- **Selected Features:** Blood Glucose, BMI, Family History, and Age
- **Removed Features:** Times Pregnant, Blood Pressure, Skin Fold Thickness, 2-Hour Insulin
- It is interesting that blood pressure was removed as a feature given that about 25% of people with Type 1 diabetes and 80% of people with Type 2 diabetes have high blood pressure.

5) Random Forests and similar tree methods also produce a “feature importance” score that can also be used for feature selection. Let’s try manually setting up a feature selection section for that.
    a. Feature selection should already be turned on, with line 37 having feat_select flag set to equal 1
    b. On line 38, change fs_type from 2 to 4
    c. You will need to add a RandomForestClassifier() call to pass to the clf object on line 156 (under where it says “if fs_type=4”), you can use something similar to the calls used elsewhere in the code. Don’t forget to set the parameters, particularly the random_state and number of trees to 100.
    d. You will then need a line to fit the data. On line 157, call a fit() function on the clf object.
    e. On the line below that, add a line creating an empty array, and call it sel_idx.
    f. Now we need to create a For loop, taking each value out of the feature_importances_ in the clf object in turn. Call this value ‘x’.
    g. Within the For loop, create an If-Else statement. Set “If” so that if x >= mean of the feature_importances_ array, then append the value of 1 to sel_idx. For “Else”, append a 0.
    h. Hint, you can use numpy to calculate means and other statistics of arrays, e.g. np.mean (numpy is given the alias of ‘np’ in the code).
    i. Try to run the code.

```
155     if fs_type==4:
156         clf= RandomForestClassifier( n_estimators=100, max_depth=None, min_samples_split=3,criterion='entropy', random_state=rand_st)
157         clf.fit(data_np,target_np)
158         sel_idx = []
159         print('clf.feature_importances_ = ', clf.feature_importances_)
160         for x in clf.feature_importances_:
161           if x >= np.mean(clf.feature_importances_):
162             sel_idx.append(1)
163           else:
164             sel_idx.append(0)
```

```
['Class', 'Times Pregnant', 'Blood Glucose', 'Blood Pressure', 'Skin Fold Thickness', '2-Hour Insulin', 'BMI', 'Family History', 'Age']
768 768


--FEATURE SELECTION ON--

clf.feature_importances_ =  [0.08170225 0.24451985 0.09324714 0.07462812 0.07193837 0.16977238
 0.12788971 0.13630217]
Selected: ['Blood Glucose', 'BMI', 'Family History', 'Age']
Features (total/selected): 8 4


--ML Model Output--

Random Forest Acc: 0.76 (+/- 0.05)
Random Forest AUC: 0.82 (+/- 0.06)
CV Runtime: 1.3329291343688965
```

**** Question #5: Run the code once, record the accuracy and AUC scores. What features were selected, and which were removed? How do the selected features compare to what you saw in Question #4 above? Was the performance (accuracy, AUC) different than in Question #4?***

- **Accuracy:** 0.76 (+/- 0.05)
- **AUC:** 0.82 (+/- 0.06)
- **Selected Features:** Blood Glucose, BMI, Family History, and Age
- **Removed Features:** Times Pregnant, Blood Pressure, Skin Fold Thickness, 2-Hour Insulin
- The same features were selected and removed as in Question #4 suggesting that this was done correctly. Accuracy and AUC were the same in both cases as well. Double checking the clf.feature_importances_ from the print statement confirms that selection is working properly.

Wine Quality Dataset ***Code link below***

https://colab.research.google.com/drive/1rWD7opvLA1HSA6ttZZbWh0OH2E9wVrId?authuser=1#scrollTo=bc6YfXyAPtc0

Open up HW2_Wine.py … First, let's repeat the steps we did above for Diabetes, with some tweaks. We will skip the Train/Test version, and jump right to the Cross-Val version.

6) Let's repeat what we did for the Diabetes dataset in Question #2 above here for the Wine dataset. To do so, we need to do two things, first create a Random Forest Classifier object (clf), then second pass that object and some data arrays into a "cross-validate" function.
    a. On line 32, cross-validation is already turned on for you (set to 1)
    b. On line 211, we need to create a Random Forest Regressor. We can do this using RandomForestRegressor(). Using the API link in the accompanying document, call that function, and pass in the following parameters:
        i. Set number of trees = 100
        ii. Set max features (for each split) = .33
        iii. Set maximum depth = None
        iv. Set minimum # of samples for split to occur = 3
        v. Set splitting criterion = 'mse'
        vi. Set random_state variable to rand_st
    c. Now we need to run cross-validation. On line 212, call a cross_validate() function and return the value to a scores object (you can do this by setting the scores = cross_validate()). For the cross_validate function, pass in the following parameters:
        i. Set the estimator = rgr
        ii. Set the data to be fit = data_np
        iii. Set the target variable = target_np
        iv. Set scoring = scorers
        v. Set the # of cv folds = 5
    d. Note we are passing in the WHOLE dataset here, not just training data or training target values
    e. Note that performance is already being calculated here for you with the scorers being setup on line 207, and the results pulled out and printed below.

```
210     start_ts=time.time()
211     rgr = RandomForestRegressor(n_estimators=100, max_features=0.33, max_depth=None, min_samples_split=3,criterion='mse',random_state=rand_st)
212     scores = cross_validate(rgr, data_np, target_np, scoring=scorers,cv=5)
213
```

```
['Class', 'fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar', 'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density', 'pH', 'sulphates', 'alcohol']
1599 1599


--ML Model Output--

Random Forest RMSE:: 0.65 (+/- 0.02)
Random Forest Expl Var: 0.33 (+/- 0.11)
CV Runtime: 1.6694278717041016
```

Question #6: Run the code once, record the RMSE and Expl Variance.

- **RMSE:** 0.65 (+/- 0.02)
- **Expl Variance:** 0.33 (+/- 0.11)

7) Let's explore how the number of trees affects performance of a Random Forest.
   a. On line 211, change the number of trees from 100 to 5
   b. Now set the number of trees to 10
   c. Now set the number of trees to 20
   d. Now set the number of trees to 50
   e. Now set the number of trees to 200
   f. Now set the number of trees to 500
   g. Now set the number of trees to 1000

Wine Question 7: Results **Scores with Different #trees** **Number of trees:** 5, 10, 20, 50, 200, 500, 1000			
# trees	**RMSE**	**Expl Var**	**Runtime (s)**
5	0.70 (+/- 0.04)	0.20 (+/- 0.13)	0.095
10	0.67 (+/- 0.04)	0.28 (+/- 0.09)	0.183
20	0.66 (+/- 0.04)	0.30 (+/- 0.09)	0.355
50	0.65 (+/- 0.03)	0.32 (+/- 0.11)	0.854
200	0.64 (+/- 0.02)	0.34 (+/- 0.10)	3.478
500	0.64 (+/- 0.02)	0.34 (+/- 0.11)	8.396
1000	0.64 (+/- 0.02)	0.34 (+/- 0.10)	16.716

Question #7a: Run the code <u>once</u> for each setting of the number of trees (5,10,20,50,100,200,500, 1000), record the RMSE and Expl Variance. What do you notice about the scores? How do they change as the number of trees increases? Is this the same as you for the Diabetes dataset in Question #3?

- I notice that RMSE is in the range of 0.64 (+/- 0.02) to 0.70 (+/- 0.04) and Explained Variance is in the range of 0.20 (+/- 0.13) to 0.34 (+/- 0.10). This shows that as the number of trees increases, the average RMSE and range of RMSE values decreases. It also shows that as the number of trees increases, the average explained variance increases and range slightly decreases.
- I also notice that RMSE > Expl Var, with Explained Variance actually being quite low. A low explained variance with high a range of +/- 0.1 is pretty poor performance. A high RMSE plus a low explained variance with all features selected together suggests to me that the features included in this model are not explaining or predicting the wine quality well. Edge cases in Wine quality are likely a big source of the problem for this model to handle.

- The changes for both metrics are negligible after about 50 trees. This is the same as with the Diabetes dataset. Seems that using too many trees is overkill.

Question #7b: What about run-times, how do those change as you change the number of trees? What do the changes in scores and run-times tell us about choosing the right number of trees?

- Doubling the number of trees roughly doubles the run-time.
- This tells us that we do not want to increase the number of trees for marginal or potentially insignificant gains in performance, as there is a large computational expense. A similar case is made in the Pima dataset question. If we had a huge dataset with thousands-millions of rows and many features, doubling run-time becomes a serious concern.
- We want a model with the fewest trees that still exhibits good performance, where good is relative to the specific modelling task, dataset we are working with, and its ability to consistently perform on key metrics.

# trees	Runtime (s)
500	8.4
1000	16.7

8) Now let's turn on Wrapper-Based Feature Selection, which essentially builds lots of models with different subsets of features, and picks the subset that performs the best. For simplicity here though, we will just build a single subset and select the top variables. We will use the same Random Forest model for this.
    a. First, on line 211, change the number of trees back to 100
    b. To turn on feature selection, we need to first on line 37 change the feat_select flag to equal 1 instead of 0
    c. Note that there is an option to change the feature selection type on line 38, but the homework code is hard-coded to only use wrapper-based, so this doesn't matter for now
    d. You will need to add a RandomForestRegressor() call to pass to the clf object on line 148, you can use something similar to the calls used elsewhere in the code (e.g. line 196). Don't forget to set the parameters, particularly the random_state and number of trees to 100.
    e. Note the SelectFromModel() function being called on line 149, this is where the actual feature selection occurs, with the clf object being passed in

```
['Class', 'fixed acidity', 'volatile acidity', 'citric acid', 'residual sugar', 'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density', 'pH', 'sulphates', 'alcohol']
1599 1599
```

```
--FEATURE SELECTION ON--

Wrapper Select:
Selected: ['volatile acidity', 'sulphates', 'alcohol']
Features (total/selected): 11 3


--ML Model Output--

Random Forest RMSE:: 0.68 (+/- 0.03)
Random Forest Expl Var: 0.25 (+/- 0.11)
CV Runtime: 1.040503978729248
```

Question #8a: Run the code once, record the RMSE and Expl Variance. What do you notice about the scores? How do they compare to the performance above in question #6?

- **RMSE:** 0.68 (+/- 0.03)
- **Expl Variance:** 0.25 (+/- 0.11)

Metric	Q8 - w/feat_select	Q6 - w/o feat_select
RMSE	0.68 (+/- 0.03)	0.65 (+/- 0.02)
Expl Variance	0.25 (+/- 0.11)	0.33 (+/- 0.11)

- I notice that the RMSE scores are comparable and there is a drop in explained variance for the Q8 scores. Given that explained variance is already quite low, and we are using fewer features here, I would argue that the model with fewer features is better. A model with 3 versus 11 features that only sees an average Expl Variance decrease of 0.08 is arguable better or at least more efficient.

Question #8b: What features were selected, and which were removed?

- **Selected Features:** volatile acidity, sulphates, alcohol
- **Removed Features:** fixed acidity, citric acid, residual sugar, chlorides, free sulphur dioxide, total sulfur dioxide, density, pH

9) Let's see if normalizing the features changes performance. To do this we are going to use the simple scaling function, but there are many different types of normalization available in Scikit (see the preprocessing link in the accompanying API document).
    a. First, turn off feature selection, on line 37 change the feat_select flag back to 0
    b. To turn on feature normalization, on line 34 change the norm_features flag to equal 1
    c. In the Preprocessing Section, on line 96, we need to call the scale() function, and pass into it the WHOLE dataset of features, which is data_np. See the API link in the accompanying document.
    d. Note that we are simply returning the output of the scale function (i.e. normalized features) back to the same object.

```
--ML Model Output--

Random Forest RMSE:: 0.65 (+/- 0.02)
Random Forest Expl Var: 0.33 (+/- 0.11)
CV Runtime: 1.6752452850341797
```

Question #9: Run the code once and record the RMSE and Expl Variance. What do you notice about the scores? How do they differ from question #6?

- The scores are exactly the same as in Question #6. Random forest is a tree-based model as thus does not require feature scaling. If we were building a model that uses a distance function, like KNN does, we would need to normalize.

Metric	Q9 w/normalization	Q6
RMSE	0.65 (+/- 0.02)	0.65 (+/- 0.02)
Expl Variance	0.33 (+/- 0.11)	0.33 (+/- 0.11)

10) Finally, let's try to setup Bagging, using Decision Tree Regressors. Remember though, Bagging can use any sort of classifier or regressor as an ensemble, but generally simple fast methods (decision trees, K-nearest neighbors, naïve bayes, etc) are best because you are going to build a whole bunch of them. A neural network, for instance, would probably take too long on any decent sized dataset.
    a. First, turn off feature normalization, on line 34 change the norm_features flag back to 0
    b. We'll do this in the single test/train split section, so turn cross-validation off on line 32 by setting the cross_val flag to 0
    c. We need to import the function into our script before we can call it. On line 11, add BaggingRegressor to the comma-separated list of packages imported from the sklearn.ensemble module.
    d. Let you do the rest yourself. You can see between lines 195-202 I've left space for you code You'll need to add a DecisionTreeRegressor() on line 195. You can use your code from HW1 if you want, with the same parameters. Make sure you set random_state= rand_st.
    e. Create a BaggingRegressor() on the line below, pass it to the bag object, then fit the bag to data. Make sure you set parameters:
        i. Set max_samples =0.6
        ii. Set random_state = rand_st
    f. You will need to create some scoring on the lines down below replacing the commented lines. Refer back to Homework 1, but protip: we need to score/predict using the bag object, not the rgr object.

```
rgr =  DecisionTreeRegressor(criterion='friedman_mse', splitter='best', max_depth=None, min_samples_split=3, min_samples_leaf=1, max_features=None, random_state=rand_st)
bag = BaggingRegressor(base_estimator= rgr, max_samples=0.6,random_state=rand_st)
bag.fit(data_train, target_train)

scores_RMSE = math.sqrt(metrics.mean_squared_error(target_test, bag.predict(data_test)))
print('Decision Tree RMSE:', scores_RMSE)
scores_Expl_Var = metrics.explained_variance_score(target_test, bag.predict(data_test))
print('Decision Tree Expl Var:', scores_Expl_Var)
```

Question #10a: Run the code 5 times and record the RMSE and Expl Variance. What do you notice about the scores? How stable are they? How do they differ from question #6?

Wine Question 10a: Results **5 runs of BaggingRegressor(DecisionTreeRegressor())**		
Run	**RMSE**	**Expl Var**
1	0.628	0.399
2	0.662	0.361
3	0.635	0.335
4	0.612	0.387
5	0.614	0.432

- I notice that the RMSE values are between 0.61 and 0.66. This is close to the performance we saw previously in Question #6. The Explained variance is consistently higher than before, with the range 0.33 to 0.43. They both seem stable in these ranges. Another 20 runs confirmed the stability as RMSE and Expl Var stayed pretty cleanly within those ranges. The boost in explained variance score and stability makes sense because bagging helps reduce variance.
- **Q6 - RMSE:** 0.65 (+/- 0.02)
- **Q6 - Expl Variance:** 0.33 (+/- 0.11)

Question #10b: Based on the API webpage for the BaggingRegressor() in the accompanying API links document, what two parameters do we need to change to create a Random Subspaces model?

- The API webpage states that the Random Subspaces model is, “When random subsets of the dataset are drawn as **random subsets of the features**, then the method is known as Random Subspaces.
- **max_features**: int or float, optional (default=1.0) - The number of features to draw from X to train each base estimator.
- **bootstrap_features:** boolean, optional (default=False) - Whether features are drawn with replacement.

- We need to change max_features and boostrap_features to get the Random Subspaces model working as we need to pull random subsets of the features to train each time in the decision tree regressors.

Question #10c: Based on the API webpage for the BaggingRegressor(), notice that the function when called also calculates the out-of-bag error (oob_score). Should we be using that metric then rather than a test/train split or cross-validation with bagging, or can we use them together (hint: do some googling)?

- OOB or Out of Bag score is used in Random Forests as a means of validating the model. It is essentially the number of correctly predicted rows from the out of bag sample for a subset of the Decision Trees. The out of bag sample is in turn the leftover row(s) of data that the models did not see in a given bootstrap sample.
- Doesn't seem like it would be worth it to use both at the same time when trying to improve model performance. Each seem to have certain benefits and from what I can see online, most people tend to tune model performance around either oob for smaller datasets or other metrics with validation versus both together. In cases where there isn't enough data for a pure validation set like our wine dataset, in which all the data is used to train the model using cross-validation, there is a good amount of aggregation that happens in the bagging process. With the oob score however, because only some on the DTs are involved, that aggregation effect is somewhat diminished, meaning we retain some more of the variance in the data. I don't think it actually helps our model performance but shows a different aspect of performance.
- Notes: oob_score won't use the full ensemble, but validation will use the full forest.

Summary Questions

Question #11: Compare the performance of Random Forests here to the Decision Tree models for both datasets in Homework #1. Did Random Forest perform better, worse, or the same? If your boss or customer asked why that might be, how would you explain?

- The random forest had higher scores for the diabetes dataset and worse scores for the Wine Dataset. This doesn't necessarily mean that the random forest is worse for these situations as it also depends what would happen when trying to apply the model to a larger dataset or deploy it out in the real world.

Question 11: Compare RF and DT				
	Diabetes		Wine	
Model	Accuracy	AUC	RMSE	Expl Var
H2: RF	0.76 (+/- 0.05)	0.82 (+/- 0.06)	0.65 (+/- 0.02)	0.33 (+/- 0.11)

H1: DT	0.71 (+/- 0.14)	0.67 (+/- 0.16)	0.91 (+/- 0.08)	-0.35 (+/- 0.26)

- If my boss or a customer asked why this would be I could explain that decision trees are simpler models and more easily interpretable. We can go in and visualize how the tree operated, which in many cases is more useful in business cases where prediction isn't always the only goal of the modeling process. Bosses and customers also want to know what was important to the model in its ability to predict and it's consistency.
- I would also explain that the random forest may be more complex but it is still made from decision trees and may not be as susceptible to variance in future data.

Question #12: We've now seen several different sampling and evaluation techniques. When it comes to evaluating model performance, what is the "gold standard" approach?

- It seems that the "gold standard" approach to evaluating model performance is to look at performance from a number of metrics which depend on the.
- A holistic, robust approach to evaluating model performance ensures that the quality of performance you believe you are getting is backed up by multiple metrics that agree with each other.
- From our explorations, we covered metrics of Accuracy and AUC for classification and RMSE and Explained Variance for Regression. Looking and both metrics together for a single evaluation of performance tells us more meaningful information about our model than a single metric could. This is also helpful for when we are trying to improve performance, as there are trade-offs with different techniques, and accuracy isn't always what we are trying to optimize.