**Homework #1 Instructions**

**Casey Bennett, PhD**

**Machine Learning**

**Hanyang 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. Both scripts implement a basic decision tree from Scikit to make predictions. We will explore different ways to score the predictions and evaluate results, including setting up different scorers, comparing cross-validation to simple test/training splits, and finally look at the effects of simple feature selection.

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 decision trees 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

1. First, we need to setup scorers for the simple test/train split. Accuracy can be calculated using the score method within the Decision Tree object (clf), but AUC requires us to use a function from the metrics library on predicted labels.
   1. Note on line 111, the test/training data is already split for you into separate arrays for both the features (data) and the target
   2. Note the Decision Tree classifier being created on line 116, then fitted using the training data on the line below
   3. To calculate accuracy, replace the comment placeholder on line 119 with: clf.score(data\_test, target\_test)
   4. To calculate AUC, replace the comment placeholder on lines 121 with: metrics.roc\_auc\_score(target\_test, clf.predict\_proba(data\_test)[:,1])

*\*Question #1: Run the code 5 times, record the accuracy and AUC scores of each run, as well as run times. What do you notice about the scores?* *Are they consistent?*

1. Next, let’s try changing one of the parameters of the Decision Tree.
   1. On line 116, change the criterion option from ‘gini’ to ‘entropy’

*\*Question #2: Run the code again 5 times, record the accuracy and AUC scores of each run, as well as run times. What do you notice about the scores? How do they compare to scores above in question 1?*

1. Now, let’s setup scorers for the cross-validation split. This works a bit differently, we have to set up a dictionary of scorers first, then pass that into the cross\_validate function call. The function will then return a dictionary of scores, which we can call by name.
   1. To turn on cross-validation, we need to first on line 32 change the cross\_val flag to equal 1 instead of 0
   2. To setup the scorers, replace the comment placeholder on lines 127 with:

{'Accuracy': 'accuracy', 'roc\_auc': 'roc\_auc'}

* 1. Note the cross\_validate function call on line 132, with clf object passed in, no need to change this yet
  2. To calculate accuracy, replace the comment placeholder on lines 134 with: scores['test\_Accuracy']
  3. To calculate AUC, replace the comment placeholder on lines 136 with:

scores['test\_roc\_auc']

*\*Question #3: Run the code 5 times, record the accuracy and AUC scores of each run, as well as run times. What do you notice about the scores? How do they compare to the simple test/train split scores in question 1?*

1. Let’s see how the number of cross-validation folds affects performance.
   1. On line 132, change the cv option from 5 to cv=10
   2. Now set the cv option to cv=3
   3. Now set the cv option to cv=8

*\*Question #4:* *Run the code once for each cv setting (3,8,10), record the accuracy and AUC scores, as well as run times. Line these results up in a table, and include the results with cv=5 from Q3. What do you notice about the scores? How about the run times?* *Can we choose a best CV setting?*

**Wine Quality Dataset**

Open up HW2\_Wine.py … First, let’s repeat the steps we did above for Diabetes, with some tweaks.

1. First, we need to setup scorers for the simple test/train split. For regression problems, both RMSE and Explained Variance requires us to use a function from the metrics library on predicted labels. Note that we have to take the square root of the mean\_squared\_error metric (MSE>>RMSE).
   1. Note on line 186, the test/training data is already split for you into separate arrays for both the features (data) and the target
   2. Note the Decision Tree classifier being created on line 191, then fitted using the training data on the line below
   3. To calculate RMSE, replace the comment placeholder on line 194 with:

math.sqrt(metrics.mean\_squared\_error(target\_test, rgr.predict(data\_test)))

* 1. To calculate Explained Variance, replace the comment placeholder on lines 196 with: metrics.explained\_variance\_score(target\_test, rgr.predict(data\_test))

*\*Question #5: Run the code 5 times, record the RMSE and Expl Variance scores of each run, , as well as run times. What do you notice about the scores?*

1. Next, let’s try changing one of the parameters of the Decision Tree.
   1. On line 191, change the criterion option from ‘mse’ to ‘friedman\_mse’

*\*Question #6: Run the code again 5 times, record the RMSE and Expl Variance of each run, as well as run times. What do you notice about the scores? How do they compare to scores above in question 5?*

1. Now, let’s setup scorers for the cross-validation split. This works a bit differently, we have to set up a dictionary of scorers first, then pass that into the cross\_validate function call. The function will then return a dictionary of scores, which we can call by name. For RMSE, we have to again take the square root of MSE (in this case flipping the negative sign first).
   1. To turn on cross-validation, we need to first on line 32 change the cross\_val flag to equal 1 instead of 0
   2. To setup the scorers, replace the comment placeholder on lines 202 with:

{'Neg\_MSE': 'neg\_mean\_squared\_error', 'expl\_var': 'explained\_variance'}

* 1. Note the cross\_validate function call on line 207, with rgr object passed in, no need to change this yet
  2. To calculate RMSE, replace the comment placeholder on lines 209 with: np.asarray([math.sqrt(-x) for x in scores['test\_Neg\_MSE']])
  3. To calculate Explained Variance, replace the comment placeholder on lines 210 with:

scores['test\_expl\_var']

*\*Question #7: Run the code 5 times, record the RMSE and Expl Variance scores of each run, as well as run times. What do you notice about the scores? How do they compare to the simple test/train split scores in question 5?*

1. Let’s see how the number of cross-validation folds affects performance.
   1. On line 207, change the cv option from 5 to cv=10
   2. Now set the cv option to cv=3
   3. Now set the cv option to cv=8

*\*Question #8: Run the code once for each cv setting (3,8,10), record the RMSE and Expl Variance, as well as run times. Line these up in table, and include the results from Q7 (cv=5). What do you notice about the scores? How about the run times?* *Can we determine a best CV setting?*

1. Finally let’s see how feature selection affects performance. First, let’s turn on the LV Filter, which will filter out variables with low variance, e.g. if there are 100 samples but 95 of them have the exact same value for a variable, it’s probably not that useful a predictor feature.
   1. Set the cv option on line 207 back to cv=5
   2. To turn on the LV Filter, we need to first on line 39 change the lv\_filter flag to equal 1 instead of 0

*\*Question #9: Run the code once, record the RMSE and Expl Variance. What do you notice about the scores? How do they compare to the CV performance above in question 7? What features were selected, and which were removed?*

1. Now let’s see how a more involved feature selection method affects performance. 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 Decision Tree regressor model for this.
   1. Set the lv\_filter on line 39 back to lv\_filter=0
   2. To turn on feature selection, we need to first on line 37 change the feat\_select flag to equal 1 instead of 0
   3. 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
   4. You will need to add a DecisionTreeRegressor() call to pass to the rgr object on line 147, you can use something similar to the calls used elsewhere in the code (e.g. line 191 or 206). Don’t forget to set the parameters, particularly the random\_state.
   5. Note the SelectFromModel() function being called on line 148, this is where the actual feature selection occurs, with the rgr object being passed in

*\*Question #10: Run the code once, record the RMSE and Expl Variance. What do you notice about the scores? How do they compare to the CV performance above in question 7? What features were selected, and which were removed?*

**Summary Questions**

*\*Question #11: Were there any notable differences in performance between the two datasets? Given that one was a classification problem and the other a regression problem, can we say that the Decision Tree model performed better on one of them? Why or why not?*

*\*Question #12: Based on the results you obtained, would you say that Decision Tree is a “good” model for these two datasets? If your boss or a customer asked you to build a decision tree for one of these datasets, what would you tell him/her? How would you explain the pros and cons of decisions trees?*