**Homework 2**

**Kavana Manvi Krishnamurthy**

**ID: 2158984**

**Pima Diabetes**

Open up HW3\_Diabetes.py

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.
   1. Note on line 191, the test/training data is already split for you into separate arrays for both the features (data) and the target
   2. 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:
      1. Set number of trees to 100
      2. Set maximum depth = None
      3. Set minimum # of samples for split to occur = 3
      4. Set splitting criterion = 'entropy'
      5. Set random\_state variable to rand\_st
   3. 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).
   4. Note that performance is already be calculated here for you on lines 199 and 201, using the test set (data\_test and target\_test)

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

|  |  |  |
| --- | --- | --- |
| *Run* | *Random Forest Accuracy* | *Random Forest AUC* |
| *1* | *0.7472* | *0.7988* |
| *2* | *0.7732* | *0.8385* |
| *3* | *0.8141* | *0.8546* |
| *4* | *0.7584* | *0.8311* |
| *5* | *0.7621* | *0.8197* |

*The Random Forest Accuracy varies from 0.7472 to 0.8141. There's some variability, but it's generally consistent within this range. Accuracy represents the proportion of correct predictions (both true positives and true negatives) among the total number of cases examined.*

*The AUC scores range from 0.7988 to 0.8546. This shows a similar level of consistency as the accuracy scores. AUC (Area Under the Curve) measures the model's ability to distinguish between classes across all possible classification thresholds. An AUC of 0.8 indicates that the model has an 80% chance of correctly ranking a random positive instance higher than a random negative instance.*

*The variability in scores could be attributed to the randomness inherent in the Random Forest algorithm, including:*

* *Random selection of features at each split*
* *Bootstrap sampling of the training data*
* *Random initialization of the model*

*Additionally, differences in train-test splits across runs could contribute to the observed variations in performance metrics.*

*Overall, while there are fluctuations, the model's performance appears relatively stable across multiple runs.*

*\*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?*

The fit() method of a RandomForestClassifier in Scikit-Learn requires three main parameters:

* **X (Feature Data):** This is an array-like or sparse matrix of shape (n\_samples, n\_features), containing the training input samples. It can be passed as a NumPy array or a Pandas DataFrame.
* **y (Target Labels):** This is an array-like structure of shape (n\_samples,) or (n\_samples, n\_outputs), containing the target values (class labels for classification). It can be passed as a NumPy array or a Pandas Series.
* **sample\_weight (Optional):** This is an array-like structure of shape (n\_samples,), used to assign weights to individual samples. If provided, it helps adjust the classifier's sensitivity to certain samples.

1. 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.
   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. 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:
      1. Set number of trees = 100
      2. Set maximum depth = None
      3. Set minimum # of samples for split to occur = 3
      4. Set splitting criterion = 'entropy'
      5. Set random\_state variable to rand\_st
   3. 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:
      1. Set the estimator = clf
      2. Set the data to be fit = data\_np
      3. Set the target variable = target\_np
      4. Set scoring = scorers
      5. Set the # of cv folds = 5
   4. Note we are passing in the WHOLE dataset here, not just training data or training target values
   5. 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.

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

*Random Forest Acc: 0.77 (+/- 0.08)*

*Random Forest AUC: 0.83 (+/- 0.07)*

*CV Runtime: 1.979461908340454*

*The Random Forest Accuracy has an average of 0.77 with a standard deviation of 0.08. This means the accuracy typically falls between 0.69 and 0.85 (one standard deviation from the mean).The AUC score averages 0.83 with a standard deviation of 0.07. So the AUC usually ranges from 0.76 to 0.90.These ranges indicate good overall performance with some variability. The CV Runtime of about 1.98 seconds suggests the model trains and evaluates relatively quickly, which is good for iterative improvements and testing.*

1. Let’s explore how the number of trees affects performance of a Random Forest.
   1. On line 211, change the number of trees from 100 to 5
   2. Now set the number of trees to 10
   3. Now set the number of trees to 20
   4. Now set the number of trees to 50
   5. Now set the number of trees to 200
   6. Now set the number of trees to 500
   7. Now set the number of trees to 1000

*\*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 and run times. What do you notice about the scores? How do they change as the number of trees increases?*

|  |  |  |  |
| --- | --- | --- | --- |
| Trees | Random Forest Accuracy (+/-) | Random Forest AUC (+/-) | CV Runtime (s) |
| 5 | 0.74 (+/- 0.06) | 0.77 (+/- 0.07) | 0.11669421195983887 |
| 10 | 0.74 (+/- 0.05) | 0.80 (+/- 0.08) | 0.2758598327636719 |
| 20 | 0.76 (+/- 0.06) | 0.81 (+/- 0.07) | 0.3361058235168457 |
| 50 | 0.77 (+/- 0.06) | 0.83 (+/- 0.07) | 0.8980779647827148 |
| 200 | 0.77 (+/- 0.07) | 0.83 (+/- 0.06) | 3.0684430599212646 |
| 500 | 0.78 (+/- 0.07) | 0.83 (+/- 0.06) | 7.251321077346802 |
| 1000 | 0.77 (+/- 0.07) | 0.83 (+/- 0.07) | 13.675472021102905 |

The accuracy of the Random Forest model fluctuates between 0.74 and 0.78, showing a slight boost as the number of trees increases, but it pretty much levels off around 50 to 100 trees. The variability stays consistent, hovering between ±0.05 and ±0.07, which means adding more trees doesn’t really introduce more unpredictability. It’s like, yeah, throwing in more trees helps at first, but after a certain point, it’s just extra effort for the same results.

Now, the AUC tells a more interesting story—it starts at 0.77 and climbs up to 0.83, with the biggest gains happening early on, especially between 5 and 50 trees. But once it hits 50 trees, it just vibes there at 0.83, no major jumps. The variability stays within ±0.06 to ±0.08, so while the model becomes more confident, the added complexity isn’t necessarily making it perform significantly better.

But here’s where things get real—runtime blows up fast. At 5 trees, it’s barely over 0.12 seconds, but by 1000 trees, we’re talking a massive jump to 13.68 seconds. The relationship here is almost linear, meaning the more trees you throw in, the longer it takes, without any real payoff in performance. So yeah, if you're trying to optimize for both efficiency and performance, 50-100 trees is the sweet spot. Anything more is just flexing computational power without real gains.

1. 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.
   1. First, on line 211, change the number of trees back to 100
   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 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.
   5. Note the SelectFromModel() function being called on line 149, this is where the actual feature selection occurs, with the clf object being passed in

*\*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?*

*['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: 1.8472809791564941*

***ANSWER:***

*Accuracy:*

* *Before feature selection: 0.77 (+/- 0.08), range of 0.69 to 0.85*
* *After feature selection: 0.76 (+/- 0.05), range of 0.71 to 0.81*

*AUC:*

* *Before feature selection: 0.83 (+/- 0.07), range of 0.76 to 0.90*
* *After feature selection: 0.82 (+/- 0.06), range of 0.76 to 0.88*

*Before feature selection, the model’s accuracy sat at* ***0.77 (±0.08)****, bouncing between* ***0.69 and 0.85****. After feature selection, it dipped slightly to* ***0.76 (±0.05)****, with a tighter range of* ***0.71 to 0.81****. The fact that the accuracy range shrank suggests the model is delivering more* ***consistent results across different data subsets****. While the* ***highest accuracy took a small hit (0.85 → 0.81)****, the* ***worst-case scenario actually improved (0.69 → 0.71)****, making the model more stable overall.*

*For* ***AUC****, the trend is pretty similar. Before feature selection, it sat at* ***0.83 (±0.07)****, ranging from* ***0.76 to 0.90****. After feature selection, it barely changed, dropping slightly to* ***0.82 (±0.06)****, with a range of* ***0.76 to 0.88****. The lower bound stayed the same, but the* ***upper bound dropped slightly from 0.90 to 0.88****. Basically, the model lost a bit of its highest potential but kept* ***solid performance even in worst-case scenarios****.*

*So overall, feature selection made the model a little less flashy at its peak, but way more* ***consistent and reliable****, which is often* ***a better trade-off*** *in real-world applications. Instead of chasing max performance, it’s giving* ***stable, repeatable results****—which is clutch when working with real data.*

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

Selected features: 'Blood Glucose', 'BMI', 'Family History', 'Age'  
Removed features: 'Class', 'Times Pregnant', 'Blood Pressure', 'Skin Fold Thickness', '2-Hour Insulin'

Features (total/selected): 8 4

1. 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.
   1. Feature selection should already be turned on, with line 37 having feat\_select flag set to equal 1
   2. On line 38, change fs\_type from 2 to 4
   3. 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.
   4. You will then need a line to fit the data. On line 157, call a fit() function on the clf object.
   5. On the line below that, add a line creating an empty array, and call it sel\_idx.
   6. 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’.
   7. 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.
   8. 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).
   9. Try to run the code.

*\* Question #5: Run the code once, record the accuracy and AUC scores. What features were selected, and which were removed? How do the results compare with Q4? Based on this, what do you think the SelectFromModel() function was doing in Q4? (hint: check out the Scikit API)*

OUTPUT:

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

768 768

--FEATURE SELECTION ON--

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: 2.1269869804382324

ANSWER:

Comparison: Overall, the performance metrics:

Accuracy: 0.76 (+/- 0.05)

AUC: 0.82 (+/- 0.06)

are exactly the same between Q5 and Q4. The only noticeable difference is a slight increase in runtime for Q5. This suggests that the different feature selection methods used in Q5 and Q4 resulted in very similar model performance, with Q5's method being a tad bit more computationally expensive.

Selected features: 'Blood Glucose', 'BMI', 'Family History', 'Age'

Removed Features: 'Class', 'Times Pregnant', 'Blood Pressure', 'Skin Fold Thickness', '2-Hour Insulin'

*SelectFromModel() function: In Q4* — *Since the threshold parameter is set to ‘mean’, the f*eatures whose absolute importance value is greater or equal to the mean of the feature importances are kept while the others are discarded. The maximum number of features to select is None so there is no limit.  The prefit parameter value is set to False, so the estimator is fitted and updated by calling fit and partial\_fit, respectively.

In Q5— The feature selection process is implemented manually but follows a similar logic to the SelectFromModel() function used in Q4. This approach is equivalent to setting the threshold to 'mean' in SelectFromModel(). Features with importance greater than or equal to the mean are kept, while others are discarded.

**Wine Quality Dataset**

Open up HW3\_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.

1. 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 Regressor object (rgr), then second pass that object and some data arrays into a “cross-validate” function.
   1. On line 32, cross-validation is already turned on for you (set to 1)
   2. 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:
      1. Set number of trees = 100
      2. Set max features (for each split) = .33
      3. Set maximum depth = None
      4. Set minimum # of samples for split to occur = 3
      5. Set splitting criterion = 'mse'
      6. Set random\_state variable to rand\_st
   3. 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:
      1. Set the estimator = rgr
      2. Set the data to be fit = data\_np
      3. Set the target variable = target\_np
      4. Set scoring = scorers
      5. Set the # of cv folds = 5
   4. Note we are passing in the WHOLE dataset here, not just training data or training target values
   5. 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.

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

*OUTPUT:*

--ML Model Output--

Random Forest RMSE:: 0.65 (+/- 0.02)

Random Forest Expl Var: 0.33 (+/- 0.11)

CV Runtime: 2.2290499210357666

1. Let’s explore how the number of trees affects performance of a Random Forest.
   1. On line 211, change the number of trees from 100 to 5
   2. Now set the number of trees to 10
   3. Now set the number of trees to 20
   4. Now set the number of trees to 50
   5. Now set the number of trees to 200
   6. Now set the number of trees to 500
   7. Now set the number of trees to 1000

*\*Question #7a: Run the code once for each setting of the number of trees (5,10,20,50,100,200,500, 1000), record the RMSE and Expl Variance and run times. 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?*

|  |  |  |  |
| --- | --- | --- | --- |
| Trees | RMSE (±) | Explained Variance (±) | CV Runtime (s) |
| 5 | 0.70 (± 0.04) | 0.20 (± 0.13) | 0.116 |
| 10 | 0.67 (± 0.04) | 0.28 (± 0.09) | 0.205 |
| 20 | 0.66 (± 0.04) | 0.30 (± 0.09) | 0.407 |
| 50 | 0.65 (± 0.03) | 0.32 (± 0.11) | 1.103 |
| 200 | 0.64 (± 0.02) | 0.34 (± 0.10) | 4.065 |
| 500 | 0.64 (± 0.02) | 0.34 (± 0.11) | 10.355 |
| 1000 | 0.64 (± 0.02) | 0.34 (± 0.10) | 26.025 |

*The RMSE (Root Mean Square Error) starts off at 0.70 with just 5 trees and drops to 0.64 as we crank up the forest size. It's like, the more trees we throw in, the better we get at predicting wine quality, but there's a point of diminishing returns. We see the biggest improvements early on, going from 5 to 50 trees, but after that, it's like trying to squeeze water from a stone - not much changes.*

*Now, the Explained Variance is telling us a similar story, but it's more dramatic. We start at a measly 0.20 with 5 trees (meaning we're explaining just 20% of the variance in wine quality), and this jumps up to 0.34 by the time we hit 200 trees. That's a solid improvement, but again, after 200 trees, we're just spinning our wheels - no real gains.*

*The variability in both metrics (those ± numbers) tends to decrease as we add more trees, which is cool. It's like the forest is becoming more sure of itself, less prone to wild guesses.*

*Comparison to Diabetes dataset: Performance improves fast up to 50 trees, then hits diminishing returns (50-200 trees) with minimal gains. Runtime scales linearly with tree count. Wine (regression) sees smoother RMSE drops & Explained Variance gains, while Diabetes (classification) has Accuracy/AUC fluctuations. Variability shrinks more in Wine, making it more stable. Wine also takes longer to run, especially at high tree counts.*

*\*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?*

*I f you look at that runtime, it starts off innocently enough at 0.116 seconds with 5 trees, but by the time we hit 1000 trees, we're waiting a whopping 26 seconds for our results. That's a massive jump, and the relationship looks pretty linear. So we're paying a hefty time tax for those extra trees, even when they're not really improving our predictions anymore.*

*The sweet spot? Looks like somewhere around 50-200 trees. After that, we're just burning CPU cycles for bragging rights, not better wine predictions. It's all about finding that balance between accuracy and not waiting forever for your model to run, you know?*

1. 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.
   1. First, on line 211, change the number of trees back to 100
   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 RandomForestRegressor() call to pass to the rgr object on line 150, you can use something similar to the calls used elsewhere in the code (e.g. line 211). Don’t forget to set the parameters, particularly the random\_state and number of trees to 100.
   5. Note the SelectFromModel() function being called on line 151, this is where the actual feature selection occurs, with the rgr object being passed in

*\*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?*

*--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: 1.634425163269043*

*Comparing the results from Question #6 and Question #8a:Question #6 (without feature selection):*

* *RMSE: 0.65 (± 0.02)*
* *Explained Variance: 0.33 (± 0.11)*
* *Runtime: 2.229 seconds*

*Question #8a (with feature selection):*

* *RMSE: 0.68 (± 0.03)*
* *Explained Variance: 0.25 (± 0.11)*
* *Runtime: 1.634 seconds*

*Observations:*

1. *Performance: The model with feature selection (Q8a) shows slightly worse performance.*
   * *RMSE increased from 0.65 to 0.68 (higher is worse)*
   * *Explained Variance decreased from 0.33 to 0.25 (lower is worse)*
2. *Variability: The variability in RMSE slightly increased (from ±0.02 to ±0.03), while Explained Variance variability remained the same (±0.11).*
3. *Runtime: The model with feature selection ran faster (1.634 seconds vs 2.229 seconds), likely due to the reduced number of features (3 vs 11).*
4. *Feature Reduction: The feature selection process dramatically reduced the number of features from 11 to 3, retaining only 'volatile acidity', 'sulphates', and 'alcohol'.*

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

*Removed features:*  *'fixed acidity', 'citric acid', 'residual sugar', 'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density', 'pH'*

Selected features: *'volatile acidity', 'sulphates', 'alcohol'*

1. 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).
   1. First, turn off feature selection, on line 37 change the feat\_select flag back to 0
   2. To turn on feature normalization, on line 34 change the norm\_features flag to equal 1
   3. 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.
   4. Note that we are simply returning the output of the scale function (i.e. normalized features) back to the same object.

*\*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?*

--ML Model Output--

Random Forest RMSE:: 0.65 (+/- 0.02)

Random Forest Expl Var: 0.33 (+/- 0.11)

CV Runtime: 2.2456722259521484

1. 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.
   1. First, turn off feature normalization, on line 34 change the norm\_features flag back to 0
   2. 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
   3. 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.
   4. 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.
   5. Create a BaggingRegressor() on the line below, pass it to the bag object, then fit the bag to data. Make sure you set parameters:
      1. Set max\_samples =0.6
      2. Set random\_state = rand\_st
   6. 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.

*\*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?*

|  |  |  |  |
| --- | --- | --- | --- |
| Run # | RMSE | Explained Variance | Runtime (s) |
| 1 | 0.6294 | 0.4219 | 0.6926 |
| 2 | 0.5918 | 0.4461 | 0.6820 |
| 3 | 0.6002 | 0.4661 | 0.6570 |
| 4 | 0.6022 | 0.4603 | 0.6566 |
| 5 | 0.5974 | 0.4999 | 0.6545 |

*Overall, the bagging approach in #10a seems to offer improved performance (lower RMSE, higher explained variance) and faster runtime compared to the random forest with cross-validation in #6. However, the trade-off is slightly less stable estimates across runs.*

*The results from Question #10a show some variability across runs, but overall, they’re pretty stable. The RMSE values range from 0.5918 to 0.6294, with a small range of 0.0376, while the explained variance falls between 0.4219 and 0.4999, spanning 0.078. Compared to Question #6, the model in #10a performs better. The average RMSE in #10a is around 0.6042, which is lower than the 0.65 in #6, meaning the predictions are more accurate. The explained variance also jumps from 0.33 in #6 to about 0.4589 in #10a, showing the model now explains more of the variance in the data. There’s still some run-to-run fluctuation in #10a due to the random nature of bagging, whereas #6 used cross-validation, which gives a more stable estimate with confidence intervals. Another big win for #10a is speed—its average runtime is just 0.6685 seconds, way faster than the 2.229 seconds in #6. This is probably because #10a skips cross-validation, making it much more efficient.*

*\*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?*

*When random subsets of the dataset are drawn as random subsets of the features, then the method is known as Random Subspaces. To create a Random Subspaces model using BaggingRegressor(), we need to change two parameters:*

1. *bootstrap = False*
2. *bootstrap\_features = True*

*Setting bootstrap=False ensures samples are not drawn with replacement, while bootstrap\_features=True allows for random feature subsets to be selected for each base estimator.*

*\*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)?*

*The out-of-bag (OOB) error is a useful way to estimate model performance when working with BaggingRegressor models, but it shouldn't completely replace traditional test/train splits or cross-validation. OOB error gives an internal performance estimate by using the samples that weren’t included in each bootstrap iteration, making it an efficient way to assess prediction error without needing a separate validation set. It’s also computationally cheaper than cross-validation, especially for large datasets. That said, OOB error isn’t perfect—it can slightly underestimate performance since it relies on fewer trees for predictions, and it may not be as reliable for small or imbalanced datasets. Cross-validation, on the other hand, ensures that every data point gets tested exactly once, offering a more systematic evaluation that can catch issues OOB might miss. The best approach? Use both. OOB error is great for quick checks and comparing models, while cross-validation provides a deeper, more reliable assessment, especially when fine-tuning hyperparameters. Combining them gives a more complete picture of model performance.*

**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?*

***i. Comparing the cross-validation scores of both the models for the Diabetes dataset:***

*Decision Tree Classifier (HW1):  
Accuracy: 0.71 (+/- 0.08) (range: 0.63 to 0.79)  
AUC: 0.69 (+/- 0.07) (range: 0.62 to 0.76)  
CV Runtime: ~0.041 seconds*

*Random Forest (HW2):  
Accuracy: 0.77 (+/- 0.06) (50 trees)(Range: 0.71 to 0.83)*

*AUC:* 0.83 (+/- 0.07) *(plateaus at 50+ trees)(Range: 0.76 to 0.90)  
CV Runtime:0* .8980 *(increasing with tree count)*

*Random Forest outperformed Decision Tree in both metrics:*

1. *Accuracy improved by 6 percentage points (0.71 to 0.77), with the upper range increasing from 0.79 to 0.83.*
2. *AUC showed a more significant improvement, increasing by 14 points (0.69 to 0.83), with the upper range rising from 0.76 to 0.90.*

*The Random Forest model demonstrates substantially better performance across both metrics. However, this comes at the cost of increased computational time, with the runtime increasing from 0.041 seconds for the Decision Tree to 0.8980 seconds for the Random Forest with 50 trees.The performance gains for Random Forest are most pronounced up to 50 trees, after which improvements become marginal despite significantly increased runtime. This suggests that 50 trees provide a good balance between performance improvement and computational efficiency for this dataset.*

***ii. Comparing the cross-validation scores of both the models for the Wine dataset:***

*Decision Tree Regressor (HW1):  
RMSE: 0.90 (± 0.10) (range: 0.80 to 1.00)  
Explained Variance: -0.31 (± 0.17) (range: -0.48 to -0.14)  
CV Runtime: ~0.0684 seconds (average)Random Forest Regressor (HW2) at 50 trees:  
RMSE: 0.65 (± 0.03) (range: 0.62 to 0.68)  
Explained Variance: 0.32 (± 0.11) (range: 0.21 to 0.43)  
CV Runtime: 1.103 secondsRandom Forest significantly outperformed Decision Tree in both metrics:*

1. *RMSE improved by 0.25 points (0.90 to 0.65), with the upper range decreasing from 1.00 to 0.68, indicating much better prediction accuracy.*
2. *Explained Variance showed a dramatic improvement, increasing by 0.63 points (-0.31 to 0.32), with the range shifting from negative to positive values (-0.48 to -0.14 vs 0.21 to 0.43).*

*The Random Forest model demonstrates substantially better performance across both metrics. However, this comes at the cost of increased computational time, with the runtime increasing from ~0.0684 seconds for the Decision Tree to 1.103 seconds for the Random Forest with 50 trees.The performance gains for Random Forest are most pronounced up to 50 trees, after which improvements become marginal despite significantly increased runtime. This suggests that 50 trees provide a good balance between performance improvement and computational efficiency for this dataset. The negative explained variance in the Decision Tree model indicates it performed worse than a horizontal line, while the Random Forest model shows positive explained variance, capturing meaningful patterns in the data.*

***iii. If my boss or customer asked why Random Forest performed better than a single Decision Tree, I would explain it as follows:***

Ensemble Learning: Random Forest combines multiple decision trees, which helps to reduce errors that might occur from any single tree. It is like asking a panel of experts instead of trusting just one person. It combines multiple trees, so if one messes up, the others balance it out.

Decreases Overfitting: A single Decision Tree can get too obsessed with the training data, but Random Forest adds randomness and multiple perspectives, making it way better at handling new data.

Feature Importance: Each tree gets a different mix of features, so Random Forest catches patterns a single tree might totally miss.

Robustness to Noise: The averaging of predictions across many trees makes Random Forest more robust to outliers and noisy data.

Complex Relationships: The multiple trees work together to map out complex, non-linear relationships in the data, making it way more powerful than just one tree doing its thing.

*\*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?*

Cross-validation (CV) is widely recognized as the best practice for evaluating model performance. While train/test splits and out-of-bag (OOB) error estimation have their uses, CV stands out due to several key advantages:

* Thorough Evaluation: CV ensures that every data point gets a chance to be both in the training and testing sets, leading to a more complete performance assessment.
* More Reliable Results: Since CV minimizes the impact of random sampling, it tends to be more dependable, especially when working with smaller or imbalanced datasets.
* Broad Applicability: Unlike OOB error, which is specific to bagging methods, CV can be used across various models and machine learning tasks.
* Increased Stability: By leveraging multiple train-test splits, CV produces a more stable and generalizable estimate of a model’s performance.

That said, the best evaluation strategy often involves combining multiple techniques. While CV remains the go-to method, integrating it with approaches like OOB error (when applicable) and hold-out test sets can offer a more well-rounded view of model performance.