**Homework 1**

**Kavana MK**

**ID: 2158984**

**kmanvikr@depaul.edu**

**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?***

Accuracy and AUC for Decision Tree using Gini Index:

|  |  |  |  |
| --- | --- | --- | --- |
| **Run** | **Accuracy** | **AUC** | **Runtime** |
| 1 | 0.7212 | 0.6694 | 0.0074 |
| 2 | 0.6989 | 0.7056 | 0.0073 |
| 3 | 0.7175 | 0.6993 | 0.0086 |
| 4 | 0.7323 | 0.7041 | 0.0093 |
| 5 | 0.7026 | 0.6686 | 0.0066 |

The Accuracy of the decision tree varies from 0.699 to 0.732. So it’s generally consistent.

The AUC ranges from 0.669 to 0.706. So it’s safe to that it’s pretty consistent as well.

The runtimes are somewhat consistent, ranging from 0.0066 to 0.0093 seconds. It’s maybe because of the randomness of train-test splitting that could have affected the entropy calculation, thus changes in the decision tree structure.

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?***

Accuracy and AUC for Decision Tree using Entropy:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | **Run** | **Accuracy** | **AUC** | **Runtime** | | 1 | 0.7398 | 0.7073 | 0.0077 | | 2 | 0.6729 | 0.6392 | 0.0073 | | 3 | 0.7175 | 0.7032 | 0.0065 | | 4 | 0.6580 | 0.6474 | 0.0075 | | 5 | 0.6617 | 0.6248 | 0.0061 | |

The Accuracy of the decision tree varies from 0.658 to 0.740. So it’s relatively consistent.  
The AUC ranges from 0.625 to 0.707. So we can say that it’s moderately consistent as well.  
The runtimes are fairly consistent, ranging from 0.0061 to 0.0077 seconds. Like I mentioned in the previous question, the slight variations might be due to the randomness in train-test splitting, which could have influenced the entropy calculation and caused changes in the decision tree structure.

Score Comparison Gini vs Entropy:

* **Accuracy:** For the **Gini Index** it ranges from 0.699 to 0.732, with an average of 0.714, so I would say it is more consistent compared to entropy. As for **Entropy**, the accuracy ranges from 0.658 to 0.740, an average of around 0.690, so I would say it has a wider range hence less consistent than Gini.
* **AUC:** For the Gini Index, the AUC ranges from 0.669 to 0.706, with an average of 0.689, so I would say it is more consistent compared to Entropy. As for Entropy, the AUC ranges from 0.625 to 0.707, with an average of around 0.664, so I would say it has a wider range and is therefore less consistent than Gini.

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?***

*Cv=5*

|  |  |  |  |
| --- | --- | --- | --- |
| Run | Accuracy | AUC | CV Runtime |
| 1 | 0.71 (+/- 0.08) | 0.69 (+/- 0.07) | 0.04099 |
| 2 | 0.71 (+/- 0.08) | 0.69 (+/- 0.07) | 0.04144 |
| 3 | 0.71 (+/- 0.08) | 0.69 (+/- 0.07) | 0.04120 |
| 4 | 0.71 (+/- 0.08) | 0.69 (+/- 0.07) | 0.03983 |
| 5 | 0.71 (+/- 0.08) | 0.69 (+/- 0.07) | 0.04085 |

The accuracy and AUC are stable across the runs, with only slight variations in runtime.

Accuracy: Cross-validation stayed solid at 0.71 (±0.08) every time. Simple split ranged from 0.699 to 0.732, averaging ~0.714. Both methods pretty much match up.

AUC: Cross-validation held steady at 0.69 (±0.07). Simple split ranged from 0.669 to 0.706, averaging ~0.689. Same deal here—cross-validation and simple split are super close.

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?***

Cv=3

|  |  |  |  |
| --- | --- | --- | --- |
| Run | Accuracy | AUC | CV Runtime (seconds) |
| 1 | 0.69 (+/- 0.06) | 0.67 (+/- 0.05) | 0.02476 |
| 2 | 0.69 (+/- 0.06) | 0.67 (+/- 0.05) | 0.02415 |
| 3 | 0.69 (+/- 0.06) | 0.67 (+/- 0.05) | 0.08142 |
| 4 | 0.69 (+/- 0.06) | 0.67 (+/- 0.05) | 0.04007 |
| 5 | 0.69 (+/- 0.06) | 0.67 (+/- 0.05) | 0.02971 |

Cv=8

|  |  |  |  |
| --- | --- | --- | --- |
| Run | Accuracy | AUC | CV Runtime (seconds) |
| 1 | 0.69 (+/- 0.06) | 0.66 (+/- 0.07) | 0.06515 |
| 2 | 0.69 (+/- 0.06) | 0.66 (+/- 0.07) | 0.06589 |
| 3 | 0.69 (+/- 0.06) | 0.66 (+/- 0.07) | 0.06609 |
| 4 | 0.69 (+/- 0.06) | 0.66 (+/- 0.07) | 0.06533 |
| 5 | 0.69 (+/- 0.06) | 0.66 (+/- 0.07) | 0.06547 |

Cv=10

|  |  |  |  |
| --- | --- | --- | --- |
| Run | Accuracy | AUC | CV Runtime (seconds) |
| 1 | 0.71 (+/- 0.14) | 0.67 (+/- 0.16) | 0.08264 |
| 2 | 0.71 (+/- 0.14) | 0.67 (+/- 0.16) | 0.08401 |
| 3 | 0.71 (+/- 0.14) | 0.67 (+/- 0.16) | 0.08238 |
| 4 | 0.71 (+/- 0.14) | 0.67 (+/- 0.16) | 0.08442 |
| 5 | 0.71 (+/- 0.14) | 0.67 (+/- 0.16) | 0.08081 |

*Cv=5*

|  |  |  |  |
| --- | --- | --- | --- |
| Run | Accuracy | AUC | CV Runtime |
| 1 | 0.71 (+/- 0.08) | 0.69 (+/- 0.07) | 0.04099 |
| 2 | 0.71 (+/- 0.08) | 0.69 (+/- 0.07) | 0.04144 |
| 3 | 0.71 (+/- 0.08) | 0.69 (+/- 0.07) | 0.04120 |
| 4 | 0.71 (+/- 0.08) | 0.69 (+/- 0.07) | 0.03983 |
| 5 | 0.71 (+/- 0.08) | 0.69 (+/- 0.07) | 0.04085 |

For **accuracy**, both CV=3 and CV=8 consistently give a score of 0.69 (±0.06), while CV=5 and CV=10 both give a slightly higher accuracy of 0.71 (with CV=5 having a ±0.08 and CV=10 having a ±0.14). When it comes to **AUC**, CV=3 is at 0.67 (±0.05), CV=5 leads with 0.69 (±0.07), CV=8 is a bit lower at 0.66 (±0.07), and CV=10 stays at 0.67 (±0.16). You can see that CV=5 pulls ahead with the highest AUC score, while the others are pretty close, mostly hanging between 0.66 and 0.67. The **standard deviations** also increase as the number of folds increases, especially for CV=10, which shows some higher variability.

In terms of **consistency**, CV=3, CV=5, and CV=8 show stable results across their runs. CV=10, while consistent in the scores, does show higher standard deviations. When it comes to **runtime**, the trend is clear: higher CV values lead to longer runtimes. CV=3 is a bit all over the place, with runtimes ranging from 0.02 to 0.08 seconds, while CV=5 stays pretty stable at around 0.04 seconds. CV=8 runs at a consistent 0.065 seconds, and CV=10 is the slowest at 0.082 seconds.

Given all of this, **CV=5** stands out as the best choice. It balances **accuracy** (0.71) and **AUC** (0.69) with reasonable **standard deviations** (±0.08 for accuracy, ±0.07 for AUC). It also maintains a **consistent runtime** of about 0.04 seconds, which is super reasonable. This setup gives a good mix of performance and efficiency, avoiding the potential overfitting of CV=10 (with its higher standard deviations) and the slightly lower performance of CV=3 and CV=8. So, yeah, CV=5 is the optimal pick here.

**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?***

|  |  |  |
| --- | --- | --- |
| Run | RMSE | Explained Variance |
| 1 | 0.7832 | 0.0786 |
| 2 | 0.7973 | 0.0165 |
| 3 | 0.7843 | 0.0268 |
| 4 | 0.8606 | -0.0523 |
| 5 | 0.8227 | 0.0215 |

The RMSE of the Decision Tree Regressor ranges from 0.7832 to 0.8606, with a mean of around 0.8096. This shows the errors are relatively consistent across runs, although run 4 stands out with a slightly higher value. On the other hand, the Explained Variance scores are all super low, ranging from -0.0523 to 0.0786, with an average of 0.0182. Most of the runs barely explain any variance, and run 4 even dips into negative territory, meaning the model did worse than just predicting the mean.

Overall, the model's performance is underwhelming. The RMSE suggests pretty significant prediction errors, and the low (or negative) Explained Variance scores confirm the model isn’t capturing much of the target’s variability. The results also vary quite a bit depending on the train-test split, especially for Explained Variance maybe because of the randomness.

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?***

|  |  |  |
| --- | --- | --- |
| **Run** | **Tree RMSE** | **Explained Variance** |
| 1 | 0.8270 | 0.0087 |
| 2 | 0.8326 | -0.0139 |
| 3 | 0.8523 | -0.0567 |
| 4 | 0.8095 | -0.1114 |
| 5 | 0.8475 | -0.0739 |

The RMSE with Friedman MSE ranges from 0.8095 to 0.8523, with a mean of around 0.8338, which is slightly worse compared to the original MSE criterion. The range is narrower, so the results seem more consistent, but the overall performance took a hit. Meanwhile, the Explained Variance dropped significantly, ranging from -0.1114 to 0.0087, with most runs showing negative scores. Compared to the original criterion, this is a step down, as it now consistently underperforms even a simple mean predictor.

Switching to the Friedman MSE criterion didn’t help the model. The consistently lower Explained Variance and slightly worse RMSE mean it’s not the right fit for this dataset or problem. While the original MSE criterion wasn’t great either, it performed better overall. The results suggest that the model might need better tuning or even a different algorithm to improve its performance.

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?***

Cv=5

|  |  |  |  |
| --- | --- | --- | --- |
| **Run** | **RMSE** | **Explained Variance** | **CV Runtime (seconds)** |
| 1 | 0.90 (± 0.10) | -0.31 (± 0.17) | 0.0653 |
| 2 | 0.90 (± 0.10) | -0.31 (± 0.17) | 0.0842 |
| 3 | 0.90 (± 0.10) | -0.31 (± 0.17) | 0.0628 |
| 4 | 0.90 (± 0.10) | -0.31 (± 0.17) | 0.0670 |
| 5 | 0.90 (± 0.10) | -0.31 (± 0.17) | 0.0627 |

The results from the simple train/test split and cross-validation (CV=5) highlight some key differences. For RMSE, the simple split ranged from 0.7832 to 0.8606 with an average of 0.8096, while CV was steady at 0.90 (± 0.10). On explained variance, the simple split averaged 0.0182, with some positive scores, but CV showed a consistent -0.31 (± 0.17). Basically, CV gives more stable results but paints a worse picture of the model’s performance. CV runtimes were also pretty quick, between 0.0627 and 0.0842 seconds.

Overall, the decision tree regressor is not it. The consistently negative explained variance in CV shows the model is doing worse than just predicting the mean. While the simple split looked a bit better with some positive variance, CV calls it out for what it is—bad.

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?***

Cv=3

|  |  |  |  |
| --- | --- | --- | --- |
| **Run** | **RMSE** | **Explained Variance** | **CV Runtime (seconds)** |
| 1 | 0.98 (+/- 0.09) | -0.46 (+/- 0.27) | 0.0340 |
| 2 | 0.98 (+/- 0.09) | -0.46 (+/- 0.27) | 0.0336 |
| 3 | 0.98 (+/- 0.09) | -0.46 (+/- 0.27) | 0.0335 |
| 4 | 0.98 (+/- 0.09) | -0.46 (+/- 0.27) | 0.0332 |
| 5 | 0.98 (+/- 0.09) | -0.46 (+/- 0.27) | 0.0328 |

Cv=8

|  |  |  |  |
| --- | --- | --- | --- |
| **Run** | **RMSE** | **Explained Variance** | **CV Runtime (seconds)** |
| 1 | 0.93 (+/- 0.11) | -0.50 (+/- 0.64) | 0.1170 |
| 2 | 0.93 (+/- 0.11) | -0.50 (+/- 0.64) | 0.1168 |
| 3 | 0.93 (+/- 0.11) | -0.50 (+/- 0.64) | 0.1166 |
| 4 | 0.93 (+/- 0.11) | -0.50 (+/- 0.64) | 0.1230 |
| 5 | 0.93 (+/- 0.11) | -0.50 (+/- 0.64) | 0.1116 |

Cv=10

|  |  |  |  |
| --- | --- | --- | --- |
| **Run** | **RMSE** | **Explained Variance** | **CV Runtime (seconds)** |
| 1 | 0.91 (+/- 0.16) | -0.47 (+/- 0.85) | 0.1597 |
| 2 | 0.91 (+/- 0.16) | -0.47 (+/- 0.85) | 0.1443 |
| 3 | 0.91 (+/- 0.16) | -0.47 (+/- 0.85) | 0.1420 |
| 4 | 0.91 (+/- 0.16) | -0.47 (+/- 0.85) | 0.1649 |
| 5 | 0.91 (+/- 0.16) | -0.47 (+/- 0.85) | 0.1753 |

When evaluating the cross-validation settings (CV=3, CV=5, CV=8, and CV=10), there are several key points to consider regarding the performance, consistency, and runtime of the decision tree model.

Starting with **RMSE**, the results show a clear trend: the error decreases as the number of folds increases from CV=3 to CV=5, with CV=5 showing the lowest RMSE of 0.90. However, for CV=8 and CV=10, the RMSE slightly increases, indicating that while adding more folds can provide more reliable estimates, it doesn't necessarily lead to better performance. This is an indication that the model might not be able to generalize well, regardless of the number of folds.

Looking at **Explained Variance**, all cross-validation settings yield negative values, suggesting that the model is performing worse than just predicting the mean. CV=5 produces the least negative value (-0.31), which further supports it as the most stable setting. It's also worth noting that as the number of folds increases, especially for CV=8 and CV=10, the standard deviation for Explained Variance increases significantly, showing more variability in the model's performance. This indicates that while higher CV values might provide more detailed estimates, they come with increased uncertainty, which can be less desirable.

In terms of **runtime**, the trend is predictable: as the number of folds increases, so does the computational cost. CV=3 is the fastest, with runtimes averaging around 0.033 seconds per fold, while CV=10 is the slowest, taking about 0.157 seconds per fold. If runtime efficiency is a priority, CV=3 is the best choice, but it comes at the expense of slightly higher RMSE and more negative Explained Variance.

In summary, **CV=5** stands out as the most balanced option. It offers the best performance with the lowest RMSE and the least negative Explained Variance, while maintaining a reasonable runtime. It also provides stability across runs, making it a reliable choice.

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?***

|  |  |  |  |
| --- | --- | --- | --- |
| Run | RMSE (±) | Explained Variance (±) | CV Runtime (seconds) |
| 1 | 0.96 (± 0.04) | -0.48 (± 0.23) | 0.03945 |

**Performance**: The RMSE with the LV filter is 0.96 (± 0.04), which is slightly worse than the RMSE without the filter (0.90 ± 0.10). This increase in RMSE suggests that the model's performance declined after applying the LV filter. Additionally, the Explained Variance with the LV filter is -0.48 (± 0.23), which is more negative compared to -0.31 (± 0.17) without the filter. A more negative Explained Variance indicates a worse fit, further supporting the idea that using the LV filter reduced the model's performance.

**Variability**: Despite the performance decline, there is a positive change in variability. The standard deviation of RMSE decreased from 0.10 to 0.04, suggesting that the results have become more consistent across the cross-validation folds when the LV filter is used. However, the standard deviation of Explained Variance has increased slightly, from 0.17 to 0.23, indicating a bit more fluctuation in the model’s performance with the LV filter.

**Runtime**: The computational efficiency is notably improved with the LV filter. The CV Runtime decreased to 0.03945 seconds, significantly faster compared to the runtime without the LV filter, which averages around 0.0684 seconds. This improvement is likely due to the reduction in the number of features, which simplifies the model and leads to faster computations.

While the LV filter improves consistency and reduces runtime, it negatively impacts the model’s performance, as evidenced by the increase in RMSE and more negative Explained Variance. This trade-off between performance and efficiency should be considered based on the specific needs of the analysis.

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?***

|  |  |  |  |
| --- | --- | --- | --- |
| Run | RMSE (± std) | Explained Variance (± std) | CV Runtime (seconds) |
| 1 | 0.91 (± 0.08) | -0.35 (± 0.26) | 0.02909 |
| 2 | 0.91 (± 0.08) | -0.35 (± 0.26) | 0.02872 |
| 3 | 0.91 (± 0.08) | -0.35 (± 0.26) | 0.02867 |
| 4 | 0.91 (± 0.08) | -0.35 (± 0.26) | 0.03978 |
| 5 | 0.91 (± 0.08) | -0.35 (± 0.26) | 0.02879 |

The results show that using Wrapper Select for feature selection has led to a slight decrease in performance, with the RMSE increasing slightly (0.91 vs 0.90) and the Explained Variance becoming more negative (-0.35 vs -0.31). However, this method has made the model more consistent, as evidenced by the reduced standard deviation in RMSE (0.08 vs 0.10). The more negative Explained Variance indicates that the model’s predictive power may have been reduced, but the performance is still poor overall, as it remains worse than a simple mean predictor.

Despite the minor drop in performance, Wrapper Select has provided a significant improvement in computational efficiency. The CV Runtime has been halved, going from an average of ~0.0684 seconds without feature selection to ~0.03101 seconds with Wrapper Select. The feature selection process reduced the number of features from 11 to just 3, suggesting that these features may still capture the most critical information for the model. While the reduction in features has impacted the performance slightly, the trade-off might be worthwhile in situations where runtime is crucial. Nonetheless, the model’s poor performance suggests that further optimization, such as exploring different algorithms or advanced feature engineering, could improve results.

Feature Selection:

* Selected features: ['volatile acidity', 'sulphates', 'alcohol']
* Removed features: ['fixed acidity', 'citric acid', 'residual sugar', 'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density', 'pH']
* The Wrapper Select method reduced the number of features from 11 to 3.

**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****?*

Looking at how the Decision Tree did on both the Diabetes (binary classification) and Wine (multi-class classification) datasets, it’s pretty clear the binary classification task is showing more consistent and solid results. For the Diabetes dataset, accuracy (average 0.7145) and AUC (average 0.6894) suggest the model is performing decently, with pretty steady results across different runs. These metrics are standard for binary classification, so it’s easy to gauge how well the model’s doing.

But for the Wine dataset, things get a bit tricky. The RMSE (average 0.8096) and Explained Variance (average 0.0182) aren’t typical classification metrics—they’re more for regression tasks. This makes it harder to compare the two directly. The low Explained Variance suggests the model isn’t really capturing the key patterns in the data, meaning its performance isn’t that great.

So, yeah, the Decision Tree looks like it did better with the binary classification problem, but it’s not a totally fair comparison. The Wine dataset could use more fitting classification metrics, like accuracy, precision, recall, or multi-class AUC. In short, while the Decision Tree seems to work better for the binary classification, the multi-class problem needs some tweaks before a true comparison can be made.

***\*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?***

Based on these cross-validation results, I would not consider the Decision Tree model to be particularly "good" for either of these datasets. Here's my analysis:For the binary classification (diabetes dataset):

* The accuracy is consistently 0.71 (± 0.08), which is only moderately better than random guessing (0.5).
* The AUC is consistently 0.69 (± 0.07), indicating fair but not strong discriminative ability.

For the multi-class classification (wine dataset):

* The RMSE of 0.90 (± 0.10) is relatively high.
* The Explained Variance of -0.31 (± 0.17) is negative, suggesting the model performs worse than a simple mean predictor.

If a boss or customer asked me to build a decision tree for one of these datasets, I would explain:

Decision trees have several advantages:

* They're easily interpretable, allowing us to understand the decision-making process.
* They can handle both numerical and categorical data without much pre-processing.
* They're totally fast to train and make predictions.

The cons of decision trees include:

* They're prone to overfitting, especially with deep trees.
* Randomness in train-test split can create a completely different tree, so it is somewhat unstable.
* They often don't achieve the highest accuracy compared to more advanced models.

For the Diabetes dataset, a Decision Tree could be a viable option, particularly if we prioritize model simplicity and interpretability. The moderate performance (accuracy of 0.71 ± 0.08 and AUC of 0.69 ± 0.07) suggests there's potential, but room for improvement. I would suggest:

1. Fine-tuning the model through techniques like pruning or adjusting the maximum depth.
2. Exploring feature engineering to potentially uncover more predictive patterns.
3. Comparing the Decision Tree's performance with other algorithms such as Logistic Regression or Random Forests to ensure we're using the most effective approach.

For the Wine dataset, the poor performance (negative Explained Variance) indicates that a basic Decision Tree is not suitable. I would recommend:

1. Shifting to more sophisticated algorithms designed for multi-class classification tasks.
2. Considering ensemble methods like Random Forests or Gradient Boosting, which often outperform single decision trees.
3. Re-evaluating our feature set and potentially incorporating domain knowledge to improve the model's predictive power.

In conclusion, while these specific decision tree models aren't performing well, the algorithm itself can be valuable in certain contexts. With proper tuning and understanding of its limitations, it can be a useful tool in our machine learning toolkit. However, for these specific datasets, I would recommend exploring other models or techniques to achieve better performance.