**Homework #3**

**Kavana Manvi Krishnamurthy**

**ID: 2158984**

**Pima Diabetes**

Open up HW4\_Diabetes.py

1. First, let’s run a Gradient Boosting Model, then an Ada Boosting Model, and compare.
   1. First we need to import the functions, on line 11, replace the comment with a call to import GradientBoostingClassifier() and AdaBoostClassifier() from the sklearn “ensemble” package.
   2. On line 278, create a GradientBoostingClassifier(). Using the API link in the accompanying document, call that function, and pass in the following parameters:
      1. Set number of estimators to 100
      2. Set loss = ‘deviance’
      3. Set the learning rate = 0.1
      4. Set maximum depth =3
      5. Set minimum # of samples for split to occur = 3
      6. Set random\_state variable to rand\_st
   3. Add in a cross\_validate function on line 279 (use previous homework as an example) with 5 folds, and pass in the clf object.
   4. You may want to edit the print statements, so they say “Gradient Boosting” when printing scores, to make the output easier to see.
   5. Repeat B, C, and D above for Ada Boost. Copy the block of code between lines 277-285. Paste it down under the section header “#SciKit Ada Boosting - Cross Val” on line 288.
   6. Change the clf to AdaBoostClassifier() and pass in the following parameters:
      1. Set number of estimators to 100
      2. Set base\_estimator = None
      3. Set the learning rate = 0.1
      4. Set random\_state variable to rand\_st

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

Gradient Boosting Acc: 0.76 (+/- 0.06)

Gradient Boosting AUC: 0.82 (+/- 0.06)

CV Runtime: 1.3922979831695557

AdaBoost Acc: 0.76 (+/- 0.07)

AdaBoost AUC: 0.83 (+/- 0.08)

CV Runtime: 1.674124002456665

Both Gradient Boosting and AdaBoost hit the same accuracy of 76%, meaning they classify correctly 76% of the time. AdaBoost edges out slightly in AUC (0.83 vs. 0.82), so it might be a bit better at distinguishing between classes, but the difference is tiny. Gradient Boosting, though, is more consistent (lower standard deviation) and runs faster (1.39s vs. 1.67s). So, if you need efficiency, go with Gradient Boosting, but if you care about squeezing out that extra class separation, AdaBoost has a slight edge. Honestly, they’re super close, so the choice depends on what matters more—speed or a tiny boost in AUC.

\*Question #1b: In the Scikit API for Ada Boost Classifier, it tells us that when the base\_estimator parameter is set to None, it uses a particular estimator by default. What is this default estimator, and why is it significant?

The default estimator for AdaBoostClassifier in Scikit-learn, when base\_estimator is set to None, is a DecisionTreeClassifier with max\_depth=1. This is significant because a decision tree with a depth of 1, also known as a **decision stump**, is a weak learner that helps AdaBoost focus on misclassified samples by adjusting their weights iteratively, improving overall classification performance.

1. Now let’s try a Neural Network and compare.
   1. First we need to import the function, on line 12, replace the comment with a call to import MLPClassifier() from the sklearn “neural\_network” package.
   2. Repeat what you did for Ada Boost in Question #1. Copy the block of code for Gradient Boosting (somewhere around line 277, though may have shifted down now that you’ve added lines to the code), paste down under the section header “#SciKit Neural Network - Cross Val”
   3. Change the clf to MLPClassifier() and using theAPI link in the accompanying document, pass in the following parameters:
      1. Set activation = 'logistic'
      2. Set solver = 'lbfgs'
      3. Set alpha = 0.0001
      4. Set the max # of iterations = 1000
      5. Set the hidden layers sizes = (10,)
      6. Set random\_state variable to rand\_st

\*Question #2a: Run the code once, record the accuracy and AUC score. What do you notice about the scores? How do they compare to boosting methods? What about run times?

|  |  |  |  |
| --- | --- | --- | --- |
| Classifiers | Accuracy (+/-) | AUC (+/-) | CV Runtime (s) |
| Gradient Boosting | 0.76 (+/- 0.06) | 0.82 (+/- 0.06) | 1.26 |
| AdaBoost | 0.76 (+/- 0.07) | 0.83 (+/- 0.08) | 1.63 |
| MLP | 0.70 (+/- 0.06) | 0.73 (+/- 0.06) | 2.65 |

Gradient Boosting and AdaBoost clearly outperform MLP in both accuracy and AUC. Both boosting models hit 76% accuracy, while MLP lags behind at 70%. When it comes to class separation, AdaBoost slightly edges out with an AUC of 0.83, followed closely by Gradient Boosting at 0.82, whereas MLP struggles at 0.73. All models have similar standard deviations, meaning their performance is consistent across cross-validation folds. The key takeaway? Boosting methods are better suited for this dataset, delivering stronger predictive power compared to MLP.

When it comes to runtime, Gradient Boosting is the fastest at 1.26s, followed by AdaBoost at 1.63s, while MLP drags behind at 2.65s—more than twice as slow as Gradient Boosting. This makes boosting models not just more accurate but also way more efficient. MLP’s underperformance could be due to hyperparameter settings or the dataset not being complex enough to justify its extra computational cost. Overall, if you’re choosing between these models, boosting is the way to go—better accuracy, stronger AUC, and much faster training time.

\*Question #2b: In the Scikit API for MLP Classifier, there are different solvers described. When might we use the ‘adam’ solver?

The 'adam' solver in Scikit-learn's MLPClassifier is a popular choice for many scenarios, particularly when dealing with larger datasets. Here are the key situations where using the 'adam' solver is beneficial:

1. Large datasets: The 'adam' solver works well on relatively large datasets with thousands of training samples or more. It's efficient in terms of both training time and validation score for such datasets.
2. Complex problems: Adam is effective for problems with noisy or sparse gradients, making it suitable for complex optimization tasks often encountered in deep learning.
3. Default choice: As the default solver in MLPClassifier, 'adam' is a good starting point for most problems, especially when you're unsure which solver to use.
4. Robustness to hyperparameters: Adam is generally regarded as being fairly robust to the choice of hyperparameters, though the learning rate sometimes needs adjustment.
5. Adaptive learning rates: Adam adapts the learning rate for each parameter, which can be beneficial for problems where different parameters require different learning rates.
6. Faster convergence: In many cases, Adam converges faster than traditional stochastic gradient descent (SGD), especially in the early stages of training.
7. Memory efficiency: Adam has little memory requirements, making it suitable for problems with a large number of parameters.

However, it's important to note that for small datasets, the 'lbfgs' solver might converge faster and perform better. Additionally, in some cases, particularly for convolutional networks, a well-tuned SGD might outperform Adam in terms of generalization. Therefore, while 'adam' is a great default choice, it's always worth experimenting with different solvers for your specific problem.

1. Let’s explore how the depth of each tree affects performance in Gradient Boosting.
   1. Change the max\_depth parameter of the GradientBoostingClassifier() from 3 to 5
   2. Now set the max\_depth to 7
   3. Now set the max\_depth to 10

\*Question #3: Run the code once for each setting of the max depth (3,5,7,10), record the accuracy and AUC scores. What do you notice about the scores as the max depth increases? What about run-times?

|  |  |  |  |
| --- | --- | --- | --- |
| Max Depth | Accuracy (+/-) | AUC (+/-) | CV Runtime (s) |
| 3 | 0.76 (+/- 0.06) | 0.82 (+/- 0.06) | 1.44 |
| 5 | 0.77 (+/- 0.05) | 0.83 (+/- 0.07) | 2.04 |
| 7 | 0.77 (+/- 0.07) | 0.81 (+/- 0.08) | 3.62 |
| 10 | 0.73 (+/- 0.06) | 0.79 (+/- 0.07) | 7.60 |

Gradient Boosting's performance peaks at max depth 5, where it hits 0.77 accuracy and 0.83 AUC, but going beyond that backfires. At depth 7 and 10, accuracy drops to 0.73, and AUC declines to 0.79, signaling overfitting. Plus, the runtime skyrockets—from 1.44s at depth 3 to a whopping 7.60s at depth 10. Basically, deeper trees don’t just slow things down—they also make the model worse at generalizing. Depth 5 is the sweet spot, striking the best balance between performance and efficiency without overcomplicating things.

Also, looking at standard deviations, depth 5 maintains the tightest range (±0.05 for accuracy, ±0.07 for AUC), meaning it's not just performing well on average, but also keeping results stable. Meanwhile, depth 7 and 10 have wider deviations, making their performance less reliable across different validation sets.

1. Finally, let’s run feature selection again on the Diabetes dataset, but this time do it using Gradient Boosting. Just like Random Forests, Gradient Boosting is a tree-based method, so we can use it to calculate a measure of “feature importance” natively.
   1. First, on line where we call the GradientBoostingClassifier(), change the max\_depth back to 3 from where we changed it above in Question #3
   2. To turn on feature selection, we need to first on line 38 change the feat\_select flag to equal 1 instead of 0
   3. Note that there is an option to change the feature selection type is already set to 2 (wrapper-based) on line 39
   4. You will need to add a GradientBoostingClassifier(), call to pass to the clf object on line 191, 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 estimators to 100.
   5. Note there are two sub-sections under wrapper select feature selection, one for datasets with a binned target (classification) and another for datasets with a continuous target (regression). We are doing classification with diabetes here, so we using the former section (if binning=1).
   6. Note the SelectFromModel() function being called on line 192, 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 for Gradient Boosting, Ada Boosting, and Neural Networks with no feature selection? Did you notice any changes in run-times?

['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', 'Age']

Features (total/selected): 8 3

|  |  |  |  |
| --- | --- | --- | --- |
| Model | Accuracy (+/-) | AUC (+/-) | CV Runtime (s) |
| Gradient Boosting | 0.77 (+/- 0.05) | 0.83 (+/- 0.07) | 1.23 |
| AdaBoost | 0.76 (+/- 0.06) | 0.83 (+/- 0.08) | 1.42 |
| MLP | 0.77 (+/- 0.06) | 0.82 (+/- 0.07) | 2.76 |

Comparing the results with and without feature selection, we can observe the following:

1. Gradient Boosting:
   * Accuracy improved slightly from 0.76 to 0.77
   * AUC improved slightly from 0.82 to 0.83
   * Runtime decreased slightly from 1.26s to 1.23s
   * Accuracy range: 0.71-0.81 (with feature selection) vs 0.70-0.82 (without)
   * AUC range: 0.76-0.90 (with feature selection) vs 0.76-0.88 (without)
2. AdaBoost:
   * Accuracy remained the same at 0.76
   * AUC remained the same at 0.83
   * Runtime decreased from 1.63s to 1.42s
   * Accuracy range: 0.70-0.82 (with feature selection) vs 0.69-0.83 (without)
   * AUC range: 0.75-0.91 (with and without feature selection)
3. Neural Networks (MLP):
   * Accuracy improved significantly from 0.70 to 0.77
   * AUC improved significantly from 0.73 to 0.82
   * Runtime increased slightly from 2.65s to 2.76s
   * Accuracy range: 0.71-0.83 (with feature selection) vs 0.64-0.76 (without)
   * AUC range: 0.75-0.89 (with feature selection) vs 0.67-0.79 (without)

Feature selection played a game-changing role in refining model performance. Performance improvements: Gradient Boosting and MLP showed improvements in both accuracy and AUC with feature selection, while AdaBoost maintained its performance.When it comes to consistency, the performance of Gradient Boosting and AdaBoost remained more consistent across both scenarios compared to MLP.The most notable change was in the MLP model, which saw substantial improvements in both accuracy and AUC with feature selection. Variability took a nosedive—accuracy and AUC ranges tightened up, meaning the models are now more stable and reliable.

Runtime changes: Gradient Boosting and AdaBoost saw slight decreases in runtime with feature selection, while MLP had a small increase.

\*Question #4b: What features were selected, and which were removed? Were there any differences from when you did feature selection with Random Forests in HW2?

HW2:

Selected: ['Blood Glucose', 'BMI', 'Family History', 'Age']

HW3:

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

The key difference between HW2 and HW3’s feature selection lies in the inclusion of Family History, which was selected in HW2 but removed in HW3. HW2 identified four key features, whereas HW3 narrowed it down to three, consistently selecting Blood Glucose, BMI, and Age as the strongest predictors. This variation likely stems from differences in feature selection methods and parameters—HW2 utilized Wrapper Select with RandomForestClassifier (100 estimators and a 'mean' threshold), while HW3 applied Stepwise Recursive Backward Feature Removal with 200 estimators and a specified k\_cnt for feature selection. Additionally, potential discrepancies in random state or data splitting may have influenced the final feature set. The repeated selection of Blood Glucose, BMI, and Age across both methods reinforces their importance in prediction, while the fluctuating presence of Family History suggests it is a borderline feature whose significance depends on the specific selection technique and settings.

1. Let’s explore changing the solver method for Neural Networks (we first saw in saw in Question #2). Solvers are essentially how the Neural Network goes about searching for optimal weights between its “neurons”, so that choice plays a fundamental role in how our model learns.
   1. Change the solver parameter of the MLPClassifier() from lbgfs to ‘adam’

\*Question #5: Run the code once for each setting of the solver, record the accuracy and AUC scores. What do you notice about the scores when we change the solver? What about run-times?

Comparing the results for the MLP classifier with different solvers:

1. LBFGS solver (from previous output):  
   Accuracy: 0.77 (+/- 0.06)  
   AUC: 0.82 (+/- 0.07)  
   Runtime: 2.76s
2. Adam solver:  
   Accuracy: 0.72 (+/- 0.07)  
   AUC: 0.80 (+/- 0.08)  
   Runtime: 3.90s

Comparing the MLP classifier’s performance across different solvers, the LBFGS solver outperformed the Adam solver, achieving a higher accuracy (0.77 vs. 0.72) and a slightly better AUC (0.82 vs. 0.80). Additionally, LBFGS demonstrated lower standard deviations in both metrics, suggesting more consistent performance across folds. In terms of efficiency, LBFGS was significantly faster, completing in 2.76s compared to 3.90s for Adam. The lower variance in LBFGS results further highlights its stability for this dataset, likely due to better convergence properties. These findings indicate that LBFGS is a more suitable solver for this particular problem, providing higher accuracy, faster runtime, and more stable results.

**Wine Quality Dataset**

Open up HW4\_Wine.py … First, let’s repeat the steps we did above for Diabetes.

1. First, let’s run a Gradient Boosting Model, then an Ada Boosting Model, and compare.
   1. First we need to import the functions, on line 11, add calls for the GradientBoostingRegressor() and AdaBoostRegressor() from the sklearn “ensemble” package.
   2. On line 278, create a GradientBoostingRegressor(). Using the API link in the accompanying document, call that function, and pass in the following parameters:
      1. Set number of estimators to 100
      2. Set loss = ‘ls’
      3. Set the learning rate = 0.1
      4. Set maximum depth =3
      5. Set minimum # of samples for split to occur = 3
      6. Set random\_state variable to rand\_st
   3. Add in a cross\_validate function on line 279 (use previous homework as an example) with 5 folds, and pass in the rgr object.
   4. You may want to edit the print statements, so they say “Gradient Boosting” when printing scores, to make the output easier to see.
   5. Repeat B, C, and D above for Ada Boost. Copy the block of code between lines 277-285. Paste it down under the section header “#SciKit Ada Boosting - Cross Val” on line 288.
   6. Change the clf to AdaBoostRegressor() and pass in the following parameters:
      1. Set number of estimators to 100
      2. Set base\_estimator = None
      3. Set loss = ‘linear’
      4. Set the learning rate = 0.5
      5. Set random\_state variable to rand\_st

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

--ML Model Output--

Gradient Boosting RGR RMSE:: 0.64 (+/- 0.01)

Gradient Boosting RGR Expl Var: 0.34 (+/- 0.12)

CV Runtime: 1.4308969974517822

Ada Boosting RGR RMSE:: 0.66 (+/- 0.03)

Ada RGR Expl Var: 0.31 (+/- 0.16)

CV Runtime: 2.0129759311676025

\*Question #6b: In the Scikit API for Gradient Boost Regressor, what do you think is the purpose of the learning rate parameter (hint: do some googling)?

The learning rate in Gradient Boosting controls how much each tree contributes to the final prediction by scaling its output, preventing overly aggressive updates. It acts as a built-in regularization mechanism, reducing overfitting by limiting individual tree impact. There’s a trade-off between learning rate and the number of estimators—lower values require more trees for optimal performance, while higher values need fewer. Empirically, keeping the learning rate below 1.0 improves generalization, especially in binary classification. It also fine-tunes the boosting process, making corrections from previous trees more measured. By tweaking the learning rate, data scientists can strike a balance between model complexity, training efficiency, and predictive power.

1. Now let’s try a Neural Network and compare.
   1. First we need to import the function, on line 12, replace the comment with a call to import MLPRegressor()from the sklearn “neural\_network” package.
   2. Repeat what you did for Ada Boost in Question #6. Copy the block of code for Gradient Boosting (somewhere around line 278, though may have shifted down now that you’ve added lines to the code), paste down under the section header “#SciKit Neural Network - Cross Val”
   3. Change the clf to MLPRegressor() and using the API link in the accompanying document, pass in the following parameters:
      1. Set activation = 'logistic'
      2. Set solver = 'lbfgs'
      3. Set alpha = 0.0001
      4. Set the max # of iterations = 1000
      5. Set the hidden layers sizes = (10,)
      6. Set random\_state variable to rand\_st

\*Question #7a: Run the code once, record the RMSE and Explained Variance. What do you notice about the scores? How do they compare to boosting methods? What about run times?

--ML Model Output--

Neural network RGR RMSE:: 0.66 (+/- 0.05)

Neural Network RGR Expl Var: 0.30 (+/- 0.08)

CV Runtime: 3.9588401317596436

Answer:

**Root Mean Square Error (RMSE):**

* **Neural Network Regressor:** 0.66 (+/- 0.05)
* **Gradient Boosting Regressor:** 0.64 (+/- 0.01)
* **AdaBoost Regressor:** 0.66 (+/- 0.03)

The Neural Network Regressor exhibits a Root Mean Square Error identical to the AdaBoost Regressor and slightly higher than the Gradient Boosting Regressor. This suggests that its predictive accuracy is comparable to the AdaBoost Regressor but slightly less precise than the Gradient Boosting Regressor. The lower variance in the Root Mean Square Error of the Gradient Boosting Regressor further indicates its stability across runs.

**Explained Variance:**

* **Neural Network Regressor:** 0.30 (+/- 0.08)
* **Gradient Boosting Regressor:** 0.34 (+/- 0.12)
* **AdaBoost Regressor:** 0.31 (+/- 0.16)

The Neural Network Regressor captures slightly less variability in the data compared to both boosting methods. The Gradient Boosting Regressor achieves the highest explained variance, with values ranging between 0.22 and 0.46, indicating stronger predictive power. The Neural Network Regressor and the AdaBoost Regressor exhibit similar performance, but the higher variance in the AdaBoost Regressor suggests less consistency across different runs.

**Runtime Performance:**

* **Neural Network Regressor:** 3.96 seconds
* **Gradient Boosting Regressor:** 1.43 seconds
* **AdaBoost Regressor:** 2.01 seconds

The Neural Network Regressor has the longest runtime, taking approximately 2.8 times longer than the Gradient Boosting Regressor and nearly twice as long as the AdaBoost Regressor. This indicates a significantly higher computational cost, which may not be justified given its comparable accuracy.

Conclusion:

While the Neural Network Regressor performs similarly to the AdaBoost Regressor in terms of accuracy and explained variance, it requires significantly more computational resources. The Gradient Boosting Regressor outperforms both, offering the best trade-off between accuracy, variance, and efficiency, making it the most optimal choice for this dataset.

\*Question #7b: In the Scikit API for MLP Regressor, if you wanted to create a neural network to have two hidden layers of 10 and 10, instead of just a single hidden layer of 20, how would you set the hidden\_layers parameter equal to in the function call?

MLPRegressor(hidden\_layer\_sizes=(10, 10))

This setup defines a neural network with two hidden layers, each consisting of 10 neurons. The tuple (10, 10) indicates the structure of the hidden layers, where the first value represents the number of neurons in the first hidden layer, and the second value corresponds to the number of neurons in the second hidden layer.

1. A fundamental question you will deal with as data scientists when it comes to regression problems, is whether you should try to predict the target “as-is” as a continuous variable, or discretize the target into bins and then treat it as a classification problem. Let’s try this here with the Wine Quality dataset and see what happens.
   1. We’re gonna do this using a function called KBinsDiscretizer, notice that it is already imported for you on line 19, and down in the Preprocessing section it is setup to run. You have to do a little maneuvering with numpy arrays to make sure everything stays in the proper shape, so I’ve done that for you.
   2. To turn on target discretization, we need to change the binning flag on line 36 to equal 1 instead of 0
   3. Note on line 37 below that, the bin\_cnt is already set to equal 2, so we will be creating 2 bins for the wine ratings
   4. Now we need to add classifier versions of Gradient Boosting, Ada Boosting, and Neural Networks underneath the Section labeled ####Cross-Val Classifiers#### … in the original code this was on line 296, but it’s probably shifted down since you added things. Easiest thing to do is just copy and paste your classifier code from the Diabetes Python script, should run as is.
      1. GradientBoostingClassifier()
      2. AdaBoostClassifier()
      3. MLPClassifier()
      4. Don’t forget to copy the scorers line!
   5. You will also need to add those functions to your import statements back up around Line 11, before you can call them. Note that you can import multiple functions from the same Scikit module on each line, by making a comma-separated list.

\*Question #8a: Run the code once, record the accuracy and AUC score. What do you notice about the scores? How do they compare to the regression scores (or can you compare them)?

Bin 0 : 3.0 5.0 744

Bin 1 : 6.0 8.0 855

|  |  |  |  |
| --- | --- | --- | --- |
| Model | RMSE (± Std) | Explained Variance (± Std) | Cross-Validation Runtime (seconds) |
| **Gradient Boosting Regressor** | 0.64 (± 0.01) | 0.34 (± 0.12) | 1.43 |
| **AdaBoost Regressor** | 0.66 (± 0.03) | 0.31 (± 0.16) | 2.01 |
| **Neural Network Regressor** | 0.66 (± 0.05) | 0.30 (± 0.08) | 3.96 |

|  |  |  |  |
| --- | --- | --- | --- |
| Model | Accuracy (± Std) | AUC (± Std) | Cross-Validation Runtime (seconds) |
| **Gradient Boosting Classifier** | 0.73 (± 0.05) | 0.81 (± 0.06) | 2.39 |
| **AdaBoost Classifier** | 0.73 (± 0.05) | 0.82 (± 0.07) | 1.83 |
| **Multi-Layer Perceptron Classifier** | 0.73 (± 0.06) | 0.81 (± 0.06) | 6.22 |

**Classifiers:** All three models—Gradient Boosting Classifier, AdaBoost Classifier, and Multi-Layer Perceptron Classifier—deliver identical accuracy at **73%**, with minimal variation (±0.05 to ±0.06). In terms of **Area Under the Curve (AUC)**, the scores fall within a narrow range of **0.81 to 0.82**, meaning all models are nearly equally effective at distinguishing between classes. AdaBoost edges out slightly with **0.82 (±0.07)**, but given the overlapping standard deviations, the difference isn't likely significant.

Now, when it comes to **runtime**, that's where things get real. AdaBoost is the fastest, clocking in at just **1.83 seconds**, while Gradient Boosting takes **2.39 seconds**—not a huge jump. But the Multi-Layer Perceptron? **6.22 seconds**—**more than 3x slower** than AdaBoost, without delivering any noticeable performance boost.

Big takeaway? These models are all hitting a performance ceiling with this dataset. The neural network might sound fancy, but it’s just dragging its feet without adding value. Boosting methods, especially AdaBoost, get the job done faster with the same accuracy and AUC.

**Classifier vs Regressors:**

**Performance Metrics: Regression vs. Classification:** Alright, so regression and classification use totally different metrics, so we can’t compare them one-to-one. Looking at both regression and classification models, we see some clear performance trends. For regression, RMSE values range from 0.64 to 0.66, with Gradient Boosting Regressor performing best at 0.64 (± 0.01), while AdaBoost and Neural Network sit at 0.66 with slightly higher standard deviations (± 0.03 to ± 0.05). Explained Variance follows a similar trend, ranging from 0.30 to 0.34, where Gradient Boosting leads at 0.34 (± 0.12), AdaBoost follows at 0.31 (± 0.16), and Neural Network lags at 0.30 (± 0.08). Runtime shows more spread, with Gradient Boosting being the fastest at 1.43s, AdaBoost at 2.01s, and Neural Network the slowest at 3.96s, giving a 2.53s gap between the fastest and slowest. For classification, accuracy is stable across all models at 0.73 (± 0.05 to ± 0.06), while AUC scores range narrowly from 0.81 to 0.82, with AdaBoost slightly ahead at 0.82 (± 0.07). However, the biggest difference is in runtime—AdaBoost is the fastest at 1.83s, followed by Gradient Boosting at 2.39s, while Multi-Layer Perceptron (MLP) drags at 6.22s, making it over three times slower than AdaBoost. In both regression and classification, Neural Networks take significantly longer without offering a real performance advantage, while Boosting methods (Gradient Boosting and AdaBoost) consistently strike the best balance between accuracy, variance, and efficiency.

**Consistency Check:** Regression models show slight variation, with Gradient Boosting performing best in terms of RMSE and Explained Variance. Classification models are highly consistent, with all three achieving 0.73 accuracy and AUC differences being minimal (AdaBoost at 0.82, others at 0.81). In classification, model choice has little impact on performance, whereas in regression, Gradient Boosting holds a clear advantage.

**Speed Comparison:** Regression models are generally faster than classifiers, but there is still variation.vGradient Boosting Regressor is the fastest (1.43s), while Neural Network Regressor is the slowest (3.96s, over 2.5x slower).For classification, AdaBoost is the fastest (1.83s), while MLP is significantly slower (6.22s, over 3x AdaBoost’s runtime).If efficiency matters, Neural Networks consistently take the longest across both tasks without added performance benefits.

**Complexity vs. Performance:** Neural Networks have the longest runtimes but do not offer superior accuracy, AUC, RMSE, or Explained Variance. Boosting methods (Gradient Boosting and AdaBoost) perform just as well while being significantly more efficient. Unless a Neural Network is necessary for a specific reason, it is not the best choice in this case.

**Boosting Models are the Strongest:** Gradient Boosting and AdaBoost perform well in both regression and classification, with strong accuracy and efficient runtimes. They offer the best balance between performance and computational efficiency, making them the ideal choice.

**Conclusion:**

While regression and classification models use different evaluation methods, we can observe clear patterns. Classification models tend to be more consistent across different algorithms, whereas regression models exhibit slightly more variation in performance. Additionally, classification models generally take longer to train, and in both cases, Neural Networks require the most computational time without offering significant performance benefits. For practical use, boosting methods strike the best balance between accuracy and efficiency.

\*Question #8b: Look at the bins that were created (some info should be printed out about the # of samples in each bin, and min and max values). How would you explain what you did to your boss or customer? What are we actually predicting here?

Bin 0 : 3.0 5.0 744

Bin 1 : 6.0 8.0 855

So, instead of working with exact quality scores, we grouped the wines into two broader categories to make the prediction task simpler and more reliable. Wines rated 3 to 5 are in the 'lower quality' group (Bin 0), and those rated 6 to 8 are in the 'higher quality' group (Bin 1). This gives us a dataset with 744 lower-quality wines and 855 higher-quality wines—which is pretty balanced, making it easier for the model to learn patterns effectively.

Rather than predicting a specific quality score (which can be subjective and noisy), we’re now predicting which category a wine falls into. This shifts the problem from regression (guessing an exact number) to classification (sorting into two groups)—which is often more accurate and interpretable.

Now, how well is this working? Pretty solid! Our models are 73% accurate, and the AUC scores (0.81-0.82) show they’re doing a great job distinguishing between the two categories. This means that based on a wine’s characteristics, we can predict with high confidence whether it’s on the higher or lower end of the quality scale.

In short, by simplifying the problem into two meaningful categories, we get a cleaner, more useful prediction—without getting lost in minor score differences that might not actually matter in real-world decision-making

1. Another question might be if the target variable has a weird distribution or a bunch of outliers, which might affect our results. That could affect your discretization of it. So let’s see what happens when we normalize the target variable before we discretize it.
   1. On line 34, change the norm\_target flag to equal 1 instead of 0

Question #9: Run the code once, record the accuracy and AUC score. What do you notice about the scores? How do they compare to results in Question #8a? What do these results suggest?

Bin 0 : -3.265164632733176 -0.787822640922809 744

Bin 1 : 0.4508483549823745 2.9281903467927415 855

|  |  |  |  |
| --- | --- | --- | --- |
| Model | Accuracy (± Std) | AUC (± Std) | CV Runtime (seconds) |
| **Gradient Boosting Classifier** | 0.73 (± 0.05) | 0.81 (± 0.06) | 2.27 |
| **AdaBoost Classifier** | 0.73 (± 0.05) | 0.82 (± 0.07) | 1.82 |
| **MLP Classifier** | 0.73 (± 0.06) | 0.81 (± 0.06) | 7.40 |

Looking at the results for the three classifiers (Gradient Boosting, AdaBoost, and MLP), we can observe some interesting patterns in their performance metrics. The accuracy scores are remarkably consistent across all three models, with each achieving 0.73 (± 0.05 or ± 0.06). This consistency suggests that all three models are performing similarly well in correctly classifying wines into their respective quality bins. The AUC scores also show very little variation, ranging from 0.81 to 0.82, with standard deviations between ± 0.06 and ± 0.07. These high and consistent AUC scores indicate that all three models have good discriminative power in distinguishing between the two wine quality categories. The narrow range of both accuracy (0.73 for all) and AUC (0.81-0.82) scores across different model types suggests that we might be approaching the limit of predictive performance possible with the current feature set and data. Despite the differences in model complexity and approach, they all seem to extract similar levels of predictive power from the available data.

**Comparison to 8a:** All models in both cases show identical accuracy of 0.73, with the same standard deviations. The AUC scores are also identical for all models between the two questions (0.81 for Gradient Boosting and MLP, 0.82 for AdaBoost). The models' performance remained unaffected by normalizing the target variable before discretization, indicating that the original binning strategy effectively captured relevant quality distinctions. The discretization process, applied to both the original and normalized target variable, yielded similar class distributions and consistent performance metrics. This suggests that the predictive power of the model is likely maximized with the current feature set and data, regardless of target variable pre-processing. Additionally, potential outliers did not significantly impact model performance, as normalization didn’t lead to improvements. Overall, the original binning approach appears to be robust and sufficient for classifying wine quality in this dataset.

1. Finally, let’s run feature selection again on the Wine dataset, just like we did for Diabetes in Question#5 above (using Gradient Boosting). This time though, we’ll do it for both the binned target and the un-binned target, and look at the effects.
   1. First turn target normalization off. On line 34, change the norm\_target flag to back equal to 0 as it was originally, instead of 1
   2. To turn on feature selection, we need to first on line 38 change the feat\_select flag to equal 1 instead of 0
   3. Note that there is an option to change the feature selection type is already set to 2 (wrapper-based) on line 39
   4. You will need to add a GradientBoostClassifier(), call to pass to the clf object on line 191, 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 estimators to 100.
   5. Since we are going to compare feature selection for both binned and non-binned targets, we also need to add a GradientBoostRegressor() on line 195. So now you should have a version under both sub-sections (binning=1 and binning=0).
   6. Note the SelectFromModel() function being called on line 192/196, this is where the actual feature selection occurs, with the clf/rgr object being passed in
   7. To run the code with the target binned and unbinned, we will toggle the binning flag on line 36 to either 0 (unbinned) or 1 (binned)

\*Question #10a: Run the code once for both settings of target discretization (binning either 0 or 1). Record the accuracy and AUC scores for binned data, and the RMSE and Explained Variance Scores for un-binned data. What do you notice about the scores? How do they compare to the performance above for Gradient Boosting, Ada Boosting, and Neural Networks with no feature selection? Did you notice any changes in run-times?

**Regression Models (Binning = 0)**

|  |  |  |  |
| --- | --- | --- | --- |
| Model | RMSE (+/-) | Explained Variance (+/-) | CV Runtime (s) |
| Gradient Boosting RGR | 0.66 (+/- 0.02) | 0.29 (+/- 0.14) | 1.04 |
| AdaBoost RGR | 0.67 (+/- 0.04) | 0.30 (+/- 0.13) | 0.67 |
| Neural Network RGR | 0.65 (+/- 0.03) | 0.31 (+/- 0.10) | 4.74 |

**Classification Models (Binning = 1)**

|  |  |  |  |
| --- | --- | --- | --- |
| Model | Accuracy (+/-) | AUC (+/-) | CV Runtime (s) |
| Gradient Boosting Clf | 0.73 (+/- 0.04) | 0.81 (+/- 0.05) | 1.48 |
| AdaBoost Clf | 0.74 (+/- 0.06) | 0.82 (+/- 0.07) | 1.89 |
| MLP Clf | 0.73 (+/- 0.06) | 0.81 (+/- 0.06) | 7.08 |

With feature selection enabled, we observe notable differences in model performance, particularly in runtime and variance.

**Regression models (Binning = 0):**

* Gradient Boosting showed a slight increase in RMSE (0.64 to 0.66) and a decrease in explained variance (0.34 to 0.29), but its runtime was significantly reduced (1.43s to 1.04s).
* AdaBoost exhibited a similar trend, with nearly unchanged RMSE (0.66 to 0.67) and a slight drop in explained variance (0.31 to 0.30), yet a substantial runtime reduction (2.01s to 0.67s).
* The Neural Network regressor slightly improved its RMSE (0.66 to 0.65) and explained variance (0.30 to 0.31), but its runtime increased (3.96s to 4.74s).

**Classification models (Binning = 1):**

* Gradient Boosting maintained its accuracy (0.73) and AUC (0.81) while reducing runtime (2.39s to 1.48s).
* AdaBoost showed a small improvement in accuracy (0.73 to 0.74), unchanged AUC (0.82), and a minor runtime increase (1.83s to 1.89s).
* MLP retained its accuracy (0.73) and AUC (0.81) but saw a noticeable runtime increase (6.22s to 7.08s).
* Additionally, standard deviations were slightly lower for some models with feature selection, suggesting improved stability, particularly in the Gradient Boosting regressor (0.12 to 0.14 in explained variance) and AdaBoost regressor (0.16 to 0.13 in explained variance).

Question #10b: What features were selected, and which were removed? How do those features differ between binned vs. un-binned runs?

**Binnig=0**

Wrapper Select:

Selected ['volatile acidity', 'sulphates', 'alcohol']

Features (total/selected): 11 / 3

Features not selected: [ 'fixed acidity', 'citric acid', 'residual sugar', 'chlorides', 'free sulfur dioxide', 'total sulfur dioxide', 'density', 'pH']

**Binng=1**

Bin 0 : 3.0 5.0 744

Bin 1 : 6.0 8.0 855

Wrapper Select:

Selected ['volatile acidity', 'total sulfur dioxide', 'sulphates', 'alcohol']

Features (total/selected): 11/ 4

Features not selected : ['fixed acidity', 'citric acid', 'residual sugar', 'chlorides', 'free sulfur dioxide', 'density', 'pH']

Comparing the feature selection results for binned (classification) and un-binned (regression) runs:

* Features selected in both cases: volatile acidity, sulphates and alcohol. Difference: The binned (classification) run additionally selected 'total sulfur dioxide'. The un-binned (regression) run also removed 'total sulfur dioxide'

The analysis reveals that volatile acidity, sulphates, and alcohol were consistently identified as key predictors for wine quality, whether treated as a continuous or categorical variable. In the classification scenario, 'total sulfur dioxide' emerged as a significant feature, unlike in the regression case, suggesting its relevance in distinguishing broader quality categories. Both approaches reduced the feature set from 11 to 3 or 4, highlighting that a few variables hold the majority of the predictive power. Notably, seven features were consistently excluded in both scenarios, indicating limited predictive value. The inclusion of 'total sulfur dioxide' in classification underscores how binning the target variable can shift feature importance, capturing threshold effects more effectively in categorical predictions. This comparison illustrates how framing the problem as regression or classification can subtly alter which features are deemed most important, even with the same data.

**Summary Questions**

\*Question #11: Create a table to compare the performance results of Boosting Methods and Neural Networks here with previous methods (decision trees, random forests) from prior Homeworks. Do this for both datasets. Did the results in this homework perform better, worse, or the same in terms of both evaluation scores and run-times? If your boss or customer asked why that might be, how would you explain (based on the mechanisms of each modeling method)?

**Diabetes Dataset:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Classifiers** | **Accuracy (+/-)** | **AUC (+/-)** | **CV Runtime (s)** |
| **Decision Tree (HW1)** | 0.71 (+/- 0.08) | 0.69 (+/- 0.07) | 0.04099 |
| **Random Forest (HW2)** | 0.77 (+/- 0.06) | 0.83 (+/- 0.07) | 0.8980 |
| **Gradient Boosting (HW3)** | 0.76 (+/- 0.06) | 0.82 (+/- 0.06) | 1.26 |
| **AdaBoost (HW3)** | 0.76 (+/- 0.07) | 0.83 (+/- 0.08) | 1.63 |
| **MLP (HW3)** | 0.70 (+/- 0.06) | 0.73 (+/- 0.06) | 2.65 |

Comparing the results:

* **Performance:** Random Forest, Gradient Boosting, and AdaBoost performed similarly, with the highest accuracy and AUC scores. The lower standard deviations in accuracy (± 0.06) and AUC (± 0.07) for these models suggest consistent results across runs, reinforcing their stability and reliability. Decision Tree and MLP (Neural Network) had lower performance. The **Decision Tree** showed moderate accuracy and AUC, with a slightly higher **standard deviation** in accuracy (**± 0.08**) and AUC (**± 0.07),** indicating a bit more variability in its performance. **MLP** had the lowest performance and also exhibited a wider variability in accuracy and AUC **(± 0.06**), suggesting that it may be more sensitive to hyperparameter tuning or data characteristics.

Explanation for a boss or customer:

The ensemble methods (Random Forest, Gradient Boosting, and AdaBoost) outperformed individual Decision Trees and Neural Networks in this dataset for a few key reasons:

* Ensemble Power: These methods stack multiple decision trees to create a more powerful model, reducing the risk of overfitting and helping the model generalize better. This makes them stronger compared to a single decision tree.
* Feature Flexibility: Tree-based methods are great at handling different types of features and can capture complex relationships within the data, which was important for this dataset.
* Dataset Size: For smaller datasets like this one, ensemble tree models generally do better than neural networks, which usually need larger datasets to show their full potential.
* Boosting vs Bagging: Random Forest (bagging) and boosting methods performed similarly here, which shows that both approaches—reducing variance with RF and reducing bias with boosting—were effective for this particular problem.
* Computational Trade-off: While ensemble methods took a bit more time to run than a single decision tree, they delivered a solid performance boost. The extra runtime is worth it for the better accuracy.
* Neural Network Limits: The MLP didn’t perform as well, probably because the dataset was smaller or the model needed more fine-tuning, which neural networks often require to really excel.

To sum it up: The tree-based ensemble methods—Random Forest, Gradient Boosting, and AdaBoost—strike the best balance between performance and efficiency for this dataset. They gave us the accuracy we wanted without the huge computational cost, making them the go-to choice here.

**Wine Dataset:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Model** | **RMSE (± Std)** | **Explained Variance (± Std)** | **CV Runtime (seconds)** | **Accuracy (± Std)** | **AUC (± Std)** |
| **Decision Tree Regressor (HW1)** | 0.90 (± 0.10) | -0.31 (± 0.17) | 0.0684 | - | - |
| **Random Forest Regressor (HW2) at 50 Trees** | 0.65 (± 0.03) | 0.32 (± 0.11) | 1.103 | - | - |
| **Gradient Boosting Regressor** | 0.64 (± 0.01) | 0.34 (± 0.12) | 1.43 | - | - |
| **AdaBoost Regressor** | 0.66 (± 0.03) | 0.31 (± 0.16) | 2.01 | - | - |
| **Neural Network Regressor** | 0.66 (± 0.05) | 0.30 (± 0.08) | 3.96 | - | - |
| **Gradient Boosting Classifier** | - | - | 2.39 | 0.73 (± 0.05) | 0.81 (± 0.06) |
| **AdaBoost Classifier** | - | - | 1.83 | 0.73 (± 0.05) | 0.82 (± 0.07) |
| **Multi-Layer Perceptron Classifier** | - | - | 6.22 | 0.73 (± 0.06) | 0.81 (± 0.06) |

### Performance Comparison:

* **Regression Tasks**: The **Gradient Boosting Regressor** showed the best performance overall, with the **lowest RMSE** (**0.64 ± 0.01**) and the **highest Explained Variance** (**0.34 ± 0.12**). This suggests it effectively captured the underlying patterns in the data. The **Random Forest Regressor** followed closely with similar performance metrics, showing its strength as an ensemble method. Other methods like **AdaBoost** and **Neural Network** performed decently but with slightly higher RMSE and lower explained variance.
  + The **Decision Tree Regressor** performed poorly in comparison, with a higher **RMSE** (**0.90 ± 0.10**) and negative **explained variance** (**-0.31 ± 0.17**), indicating that it was unable to capture the relationships in the data effectively.
* **Classification Tasks**: For the **classification** models, all three methods—**Gradient Boosting**, **AdaBoost**, and **MLP**—performed similarly, with **Accuracy** values around **0.73** and **AUC** ranging between **0.81 and 0.82**. The **standard deviations** of **accuracy** (**± 0.05** for Gradient Boosting and AdaBoost, **± 0.06** for MLP) and **AUC** (**± 0.06 to ± 0.07**) are relatively tight, suggesting stable and reliable performance across runs. However, the slight variation indicates that model tuning or the dataset could impact the results in practice.

**Runtime:** The Decision Tree was by far the fastest model, while Random Forest and the boosting methods showed similar runtimes. The Neural Network (MLP) had the longest runtime for both regression and classification tasks. In regression, Gradient Boosting Regressor outperformed the others, with Random Forest following closely behind, whereas the Decision Tree performed poorly. For classification, Gradient Boosting, AdaBoost, and MLP exhibited comparable performance in terms of accuracy and AUC, with stable results across runs. Despite the performance trade-offs, the Decision Tree was the quickest, and the Neural Network required significantly more computation time.

Explanation for a boss or customer:

The results in this homework generally showed better performance than simple decision trees but were on par with Random Forests. Here’s why:

* Ensemble power: Boosting methods and Random Forests work by combining multiple decision trees to create a more powerful model. This reduces overfitting and enhances generalization, leading to stronger performance compared to single decision trees.
* Boosting advantage: Boosting techniques like Gradient Boosting and AdaBoost improve iteratively by focusing on harder-to-predict examples, which can lead to better accuracy on complex data.
* Capturing feature interactions: Tree-based methods (including ensembles) are great at capturing complex relationships between features, which is crucial for predicting wine quality.
* Bias-variance balance: Ensemble models strike a better balance between bias and variance than single decision trees, improving overall performance.
* Neural network trade-off: The MLP performed similarly to boosting methods in classification but took much longer to train. This might be because neural networks usually need more data to shine, and the wine dataset might not have been large enough to fully leverage its potential.
* Computational cost: While the ensemble methods and neural networks required more runtime than a single decision tree, they delivered notable performance improvements. The added computational cost is justified by the better accuracy.
* Diminishing returns: The similar performance between Random Forests, Gradient Boosting, and AdaBoost suggests that we may have hit the ceiling for model improvement with these methods for this dataset.

In summary, Random Forests, Gradient Boosting, and AdaBoost offer the best balance of performance and efficiency for this wine quality prediction task. The choice between them comes down to the need for accuracy, runtime, and model interpretability.

\*Question #12: Can we say anything interesting about diabetes based on the features that were selected, if we were for instance trying to create a diabetes screening program for a local healthcare organization?

Selected ['Blood Glucose', 'BMI', 'Age']

Features (total/selected): 8 3

Features not selected: ['Times Pregnant', 'Blood Pressure', 'Skin Fold Thickness', '2-Hour Insulin', 'Family History']

Based on the selected features for diabetes screening—**Blood Glucose**, **BMI**, and **Age**—we can draw some useful insights for a healthcare organization looking to create an effective screening program:

* **BMI (Body Mass Index):** Overweight or obese individuals are at a higher risk of developing diabetes due to the strain excess weight places on the body’s ability to regulate blood sugar. So, in our screening process, individuals with a high BMI should be flagged for further screening, as they are more likely to develop diabetes. This could help identify a large portion of at-risk individuals efficiently.
* **Blood Glucose Levels:** Higher blood glucose levels are a strong indicator of diabetes. People with elevated glucose are already showing signs of insulin resistance or other metabolic issues. Blood glucose levels would be a key marker to monitor in the screening program, and anyone with elevated levels would warrant further testing, such as an A1C test or an oral glucose tolerance test (OGTT).
* **Age:** As people age, their risk for developing diabetes increases, as the body becomes less efficient at regulating blood sugar. Therefore, age is a valuable feature in our screening program. Older individuals are more likely to develop type 2 diabetes, especially if they have other risk factors, so they should be flagged for further evaluation.

**Non-relevant features** such as **Times Pregnant**, **Blood Pressure**, **Skin Fold Thickness**, **2-Hour Insulin**, and **Family History** were not selected, suggesting that these may not provide as much value in this specific model for identifying diabetes risk. For example:

* **Times Pregnant:** While gestational diabetes is a known risk factor for future type 2 diabetes, the direct correlation may not be as significant in this particular dataset or context.
* **Blood Pressure:** Though high blood pressure is a known risk factor for diabetes, it may not provide as direct or strong an indication for screening purposes in this case.
* **Skin Fold Thickness:** This measure is less commonly used in diabetes screening. It may not be as predictive or relevant as other features like BMI, which is already capturing information about an individual’s weight status.
* **2-Hour Insulin:** While insulin levels are important in diagnosing diabetes, this measure might not be as effective for initial screening. It’s often used after a glucose test in clinical settings, making it less useful for initial screenings where resources are limited.
* **Family History:** While family history can influence diabetes risk, it may not be as immediately actionable for screening. Individuals without a family history can still develop diabetes, so relying solely on family history might miss individuals who are at risk.

**Additional Insights:**

* **Simplified Screening:** By focusing on **BMI**, **Blood Glucose**, and **Age**, the healthcare organization can streamline the screening process, targeting the most relevant factors. This could allow for quicker identification of high-risk individuals, saving both time and resources in the initial screening phase.
* **Early Detection & Intervention:** Screening based on these features could lead to early detection, which is crucial for preventing or delaying the onset of diabetes. For example, individuals flagged for elevated glucose levels can be advised on lifestyle changes, such as improving diet and exercise, which can prevent the progression to full-blown diabetes.

In conclusion, by using **BMI**, **Blood Glucose**, and **Age**, we can create a focused and efficient screening program for diabetes that prioritizes the most relevant risk factors. This approach allows for more accurate identification of at-risk individuals, enabling timely interventions and reducing healthcare costs in the long run.