**3035345306**

**Agrawal Shubhankar**

**COMP3314: Machine Learning**

**Prof. Yizhou Yu**

**Programming Assignment 1**

|  |  |
| --- | --- |
| **Model** | Random Forest Classifier |
| **Datasets Tested** | Cancer, Car |
| **Language** | Python 3.6 |

**Code Structure**

1. rf.py
   1. Random Forest Classifier
      1. fit()
      2. \_bootstrap()
      3. \_grow\_decision\_tree()
      4. \_feature\_selection()
      5. \_bag\_pred()
      6. \_calc\_param()
      7. \_to\_numpy()
      8. predict()
      9. score()
      10. confusion()
      11. metrics()
2. rf\_node.py
   1. Node
      1. \_gini()
      2. \_entropy()
      3. \_error()
      4. info\_gain()
      5. print\_node()
      6. calc\_criterion()
      7. split\_node()
3. rf\_tree.py
   1. Tree
      1. build\_tree()
      2. print\_tree()
      3. get\_expandable\_node()
      4. predict()
      5. depth()
4. rf\_preprocess.py
   1. MinMaxScaler
      1. fit\_transform()
      2. transform()
   2. OneHotEncoder
      1. fit\_transform()
      2. transform()
   3. CarsTransformer
      1. fit\_transform()
      2. transform()
5. test.py
   1. get\_datasets()
   2. rf\_test()
   3. scikit\_test() (Check results using scikit estimator)
   4. consolidate\_results()

**Additional Files:**

* Report.docx
* README.md

**Dependencies:**

1. abc
2. numpy
3. pandas
4. sklearn (Only for comparing results with Scikit-learn model)

**Design:**

An object-oriented design has been followed to build the Random Forest Classifier to ensure modularity. The use of classes with access modifiers ensures functions are only called within the relevant spaces.

**Implementation:**

The Random Forest Classifier is implemented as follows:

**Pre-processing:**

The rf\_preprocess.py file contains the required code for the processing. This has three classes which enable us to process the features, which are the MinMaxScaler, the OneHotEncoder and the Cars Transformer. All of these classes created do not take in any initializing parameters.

**MinMaxScaler:**

The MinMaxScaler class is used to normalize the data to a range of 0-1. This class stores the minimum and maximum value for each column in a dictionary when fit and applies those on transformation.

**OneHotEncoder:**

The OneHotEncoder converts categorical features to One-Hot encoded features. To generate the one-hot encoded features, the pandas get\_dummies function is used. The names of the transformed columns are stored using a dictionary when fit and are then used to fill in missing columns when transformation happens.

**CarsTransformer:**

The CarsTransformer converts the categorical features to numeric features based on a predecided ordering of the categories in order of bad to good. These convert the features from categorical to numerical. The features are converted using a dictionary that contains the feature names mapped to a relevant number.

**Random Forest Classifier:**

The random forest classifier within the rf.py file contains several functions to provide the necessary functionalities. In order to facilitate more configurability, the class also takes in several parameters to the user.

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Function** | **Default** |
| n\_iter | The number of individual decision trees built | 10 |
| n\_feat | The number of features to be selected for each decision tree. Can be integer/whole number | 0.5 |
| metric | The impurity metric to be used for the decision trees. Options available are: gini, entropy, error (classification error) | gini |
| max\_depth | The maximum depth of the decision tree to be built | 6 |
| min\_samples\_split | The minimum number of samples required in a node to be considered for a split | 2 |
| min\_samples\_leaf | The minimum number of samples required for a node to be considered a leaf node | 1 |
| Num\_bootstrap samples | The number of bootstrap samples to be selected for each decision tree. Can be integer/whole number | 1.0 |
| Num\_splits | Number of random splits to consider at a node | 5 |

The functions in the RandomForestClassifier class are as follows:

**\_calc\_param():**

This function is used to calculate the number of features and bootstrap samples to be used if the user passes in a fraction as the parameters. It is called from within the \_bootstrap and \_feature\_selection functions.

**\_bootstrap():**

The bootstrap functions selects a sample from the training data set with replacement of records using the random.choices function. Since the classes might be skewed, this function, performs weighted sampling by weighting each class by the inverse of its frequency in the data set.

**\_feature\_selection():**

The feature selection function selects a subset of features from the training data set without replacement using the random.sample function.

**\_to\_numpy():**

The to\_numpy function converts the object from a pandas DataFrame to a numpy ndarray for faster processing in subsequent functions.

**\_grow\_decision\_tree():**

The grow decision tree calls in the feature selection function on the bootstrapped data. It then creates a new tree with a new node calls a build tree function.

**\_bag\_pred():**

The bag\_pred function iterates through the trees built and obtains the prediction from each tree for a record provided. The most common prediction is thus returned.

**fit():**

The fit function is called by the user along with the X and y training data sets. This function first converts the dataframes to numpy ndarrays and then iterates through the number of iterations. Within each iteration, a bootstrap is performed and the decision tree is grown and appended to the list of decision trees.

**predict()**

The predict function is called by the user along with the X test data sets. The bag\_pred function is applied to each record in the data and the consolidated results are returned.

**score():**

The score function is called by the use along with the X and y test data sets. After obtaining predictions, they are compared with the true data and the accuracy is calculated.

**confusion()**

This function prints out a confusion matrix taking in the test set for the features and target after obtaining predictions.

**metrics()**

This function returns a table containing precision, recall and F1 score statistics for the different classes. It takes in the test sets for the features and targets.

**Tree:**

The workings of an individual Decision Tree Classifier are within the Tree class of the rf\_tree.py file. The parameters from the Random Forest Classifier are passed into here to build an efficient tree. The functions implemented here are as follows:

**get\_expandable\_node():**

Starting from the root, this function checks all the nodes to find the node with the highest impurity which hasn’t already been split up.

**build\_tree():**

This function builds the decision tree by obtaining the next expandable node and splitting it while the conditions for the maximum tree depth and the minimum number of samples are met.

**print\_tree():**

This function helps visualize the tree by printing out the nodes at different levels.

**predict():**

Given a tree and one record, this function returns one prediction.

**depth():**

This function calculates the depth of the decision tree built.

**Node:**

This is the class for the individual nodes in the Decision Tree Classifier and can be found in the rf\_node.py file. The constructor takes in the parameters for the X and y datasets present in the node along with the features and the impurity metric. The functions implemented here are as follows:

**\_gini():**

Function to calculate the gini impurity given the probability.

**\_entropy():**

Function to calculate the entropy given the probability.

**\_error():**

Function to calculate the classification error given the probability.

**calc\_criterion():**

Calculates criterion for each node by summing the impurity measure for each of the classes present.

**info\_gain():**

Calculates the information gain using the impurity values of the node and its children.

**get\_values():**

This function generates a dictionary of key value pairs for the different classes and the count of each of the classes.

**split\_node():**

This function performs the split for the node. It iterates over all the features in the X data for the node and selects the points after which the class changes in the y data. The split is considered at each of these points and the information gain is calculated. The split with the highest information gain is then considered as the best split for the node.

**Test:**

The test.py file contains the code to test out the random forest model and check its performance against the scikit-learn model which performs the same classification. In order to remove redundancies, the preprocessing for both the models is performed using the scalers built by me.

**get\_datasets():**

This function loads the datasets from the location specified.

**rf\_test():**

This function performs the RandomForestClassification using my built approach and returns the accuracy of the model.

**scikit\_test():**

This function performs the classification using the method from scikit-learn and returns the accuracy of the model.

**consolidate\_results():**

This function prints out a dataframe with the consolidated results comparing performance of the RandomForestClassifier between my model and the scikit-learn model, for both the datasets with and without data preprocessing.

**Results:**

The results obtained are using the default parameters provided with MinMaxScaling performed on the Cancer dataset and OneHotEncoding on the Cars dataset.

Once test.py is run, we can see detailed results of the scores for the test data set.

* To begin with, the dataset is printed.
* The fitting of trees is split into multiple iterations. Each iteration fits 10 trees and takes ~2 seconds as no parallel processing is used at this stage.
* Once the model has been fit, it is evaluated on both the training and testing dataset and the accuracy is printed out for both.
* Following, a confusion matrix is printed for the different target classes.
* At the end, a table of scores is printed which contains the precision, recall and F1 values for the different classes predicted.

The output is split into 3 parts:

* Cancer dataset results
* Cars dataset results
* Overall results

For the Cancer dataset, the results are as follows:

* We can see a significantly high accuracy on the train dataset and a comparable accuracy on the test data set too. This indicates that our model has not been overfit or underfit for the data.
* The confusion matrix shows us that most of the elements lie along the diagonal indicating most predictions being made for the right class the elements belong too.
* The table of scores shows high values for precision and recall for both the classes as well as a high F1-score which can be considered as an alternative to the accuracy.

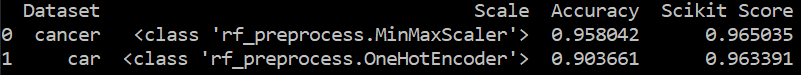
|  |  |
| --- | --- |
| **Results** | |
| **Cancer Data Set** | **Cars Data Set** |
|  |  |

For the Cars dataset, the results are as follows:

* We can see high accuracy on the train and test dataset thus showing that no significant overfit/underfitting has taken place.
* The confusion matrix isn’t as aligned as for the Cancer data set, however we can see the majority elements are predicted correctly for each of the classes.
* On closer inspection, we can also infer that several of the leaf nodes which predict **good** potentially contain an almost equal number of samples for the **acc** and **good** classes as 17 good values are predicted good whereas 12 acc values are also predicted good. We can also infer that the **unacc** classes must be easily distinguishable from the others as a very high percentage of its values have been correctly predicted.
* As mentioned above, the **good** has a low precision score owing to the fact that the values are often mixed up with acc.

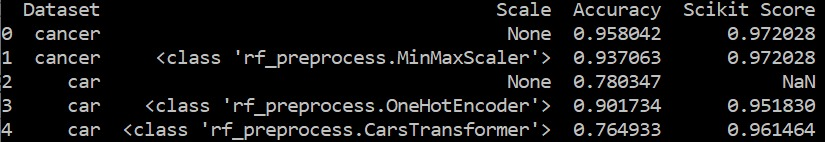
Next consolidated results are shown comparing with the Scikit-learn models.

* The accuracy for the cancer data set is around (93-95%) and for cars is (87-90%).



Comparing the results of our tests across the different types of scalers, we can see several interesting things.

* The scikit learn algorithms seem to perform better when the features are off a numeric type enabling splitting the features based on whether they are greater than or lesser than a value rather than with categorical features. This is the Scikit Score using the CarsTransformer is higher than that of with the OneHotEncoder.
* For the Cancer dataset, my Random Forest algorithm decreases in accuracy when normalized with the MinMaxScaler. This indicates that some important details are lost when the sample space for every column is reduced to [0,1]. Since this is a decision tree algorithm, scaling isn’t as important compared to other supervised learning algorithms as each split only considers one feature rather than a more interpretable model like linear regression where scaling can have a more serious effect.
* Scikit-learn produces an NaN when the Car dataset is tried on it as Scikit-learn does not perform encoding of its own and can only accept numerical features instead of categorical features.



Experimenting with different parameter settings:

* There is not much significant trend when varying the number of trees. This could be attributed to the dataset itself.
* With the max depth and feature subset parameters we can see that the graph converges with the accuracy indicating having reached an optimal level of performance.
* With respect to the bootstrap sample, we can see that the cars dataset continuously improves performance whereas the cancer dataset seems to have reached the optimal level.
* We can see accuracy rises and then slowly decreases when changing the number of random splits at every node. The picking of fewer nodes to perform splits on could explain better performance as the fewer split values may lead to an easier decision process.

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

**Advantages:**

* Building Random Forest from scratch provides us with more configurability than using an existing pre-defined method. The intricacies can be changed to suit our benefit, some of which are not allowed to be changed by the user for methods available online.
* With datasets where certain features are key in determining the classes, Random Forest does a good job as the decision-making process provides for higher predictability.
* By building a large forest of trees, we can see that there is almost no overfitting to the data as the variance has been reduced.

**Disadvantages:**

* A Random Forest algorithm can be quite complex to implement. Thus, the time complexity can be high. This is a disadvantage as certain programming languages are faster than others and code written in one can be significantly faster than in other.
* There is very little control on how the model fits itself once the parameter has been passed and can appear to be a sort of a black box approach at times. This reduces the interpretability of the model.