K-Nearest Neighbor

(3) Select and train models

This code is using the KNeighborsClassifier class from the scikit-learn library to create a K-Nearest Neighbors (KNN) classifier. It first creates an instance of the class with the number of neighbors to consider set to 5 (n\_neighbors=5).

The fit method is then called on this classifier with the training data (X\_train and y\_train) as the input. This trains the KNN classifier on the training data.

After that, the predict method is called on the trained classifier with the test data (X\_test) as the input. This generates the predictions for the test set and the results are stored in the variable 'KNN\_prediction'.

The KNN algorithm is a non-parametric method used for classification and regression. The idea is to find a fixed number (k) of training examples that are closest to the new data point, and output the most common output value among those k training examples. It's a simple algorithm that's easy to implement and understand, and it can be useful in situations where the data doesn't have a clear linear boundary.

(4) Fine-tuning Model

This code is using the GridSearchCV function from the scikit-learn library to tune the hyperparameters of a KNeighborsClassifier model. It starts by creating two lists of values for the 'leaf\_size' and 'n\_neighbors' hyperparameters and one list of possible value for the 'p' hyperparameter. These lists of values are then used to create a dictionary of hyperparameters that will be passed to the GridSearchCV function.

The GridSearchCV function takes in an estimator (in this case, the KNeighborsClassifier() object), the hyperparameter grid (created from the dictionary), and a value for the number of cross-validation folds (cv=10).

The GridSearchCV object is then fit to the training data (X\_train, y\_train) and the best hyperparameter values are obtained by calling the best\_estimator\_ method on the fitted GridSearchCV object. The best values of 'leaf\_size', 'p' and 'n\_neighbors' are printed.

This technique is called grid search and it is used to find the best combination of hyperparameter values for a model. It works by training the model multiple times with different combinations of hyperparameter values and evaluating the performance of each combination. The combination that produces the best performance is chosen as the best set of hyperparameters.

(5) evaluate the outcomes

This code uses the K-Nearest Neighbors (KNN) algorithm, which is a supervised machine learning algorithm used for classification and regression. The KNN algorithm is based on the idea that similar instances tend to have similar labels.

* The first line imports the KNeighborsClassifier class from the sklearn.neighbors module.
* The next line creates an instance of the KNeighborsClassifier class, setting the number of neighbors to 5.
* Then we fit the model using the fit() function on the X\_train and y\_train data.

Then we make predictions using the predict() function on the X\_test data.

Then, it uses several metrics to evaluate the performance of the model:

* The first line of the print statement calculates the accuracy of the model on the training data using the score() function and prints it.
* The second line of the print statement calculates the accuracy of the model on the test data using the score() function and prints it.
* The third line of the print statement uses the classification\_report() function from the metrics module to print a more detailed report on the model's performance, including precision, recall, and F1-score.
* The fourth line of the print statement uses the accuracy\_score() function to print the accuracy of the model on the test data.

The accuracy of KNN with train dataset and test dataset are calculated. The classification report is generated which includes precision, recall, f1-score and support. Finally, the accuracy\_score of the model is printed.

Decision Tree Classifier

This code is using the DecisionTreeClassifier class from the scikit-learn library to train a decision tree model on the input training data (X\_train, y\_train) and then make predictions on the input test data (X\_test, y\_test).

1. The first line imports the DecisionTreeClassifier class from the sklearn.tree module.
2. Next, an instance of the DecisionTreeClassifier class is created and assigned to the variable 'clf'.
3. The fit() method is called on 'clf' to fit the classifier to the training data (X\_train, y\_train)
4. The predict() method is called on 'clf' to make predictions on the test data (X\_test). The predictions are saved in the variable 'y\_pred'.
5. Next, the precision\_recall\_fscore\_support() function from the sklearn.metrics module is used to compute the precision, recall and f1-score, and support for the test data.
6. The score() method is called on 'clf' to measure the accuracy of the model on the test data (X\_test, y\_test).
7. Finally, the values of precision, recall, f1-score and accuracy are printed.

Fine-tuning Decision Tree classifier

This code is using the **RandomizedSearchCV** class from the **sklearn.model\_selection** module to perform a randomized search for the best hyperparameters for a decision tree classifier.

The decision tree classifier is first created using the **DecisionTreeClassifier()** class. Then, a parameter grid is defined with a range of values for the **max\_depth**, **min\_samples\_split**, **min\_samples\_leaf**, and **criterion** hyperparameters.

The **RandomizedSearchCV** class is then initialized with the classifier, parameter grid, a number of iterations (n\_iter) , 5-fold cross-validation (cv=5) and random\_state=0 to make sure the output is reproducible.

The **fit()** method is then called on the **random\_search** object with the training data to perform the randomized search. The best parameters and best score from the search are then printed using the **best\_params\_** and **best\_score\_** attributes of the **random\_search** object.

Random Forest Classifier

A random forest is an ensemble machine learning model that is composed of multiple decision trees. The idea behind a random forest is to create multiple decision trees, each trained on a different subset of the data, and then average or take a majority vote of the predictions made by each decision tree.

Random Forest Classifier is used for this dataset because it is a powerful machine learning algorithm that can handle large datasets with many features and can handle categorical variables well. It is also good at identifying important features in the dataset and can be used to estimate the feature importance. Additionally, it can be used for both classification and regression problems, making it a versatile algorithm that is well-suited for this dataset.

This code uses the RandomForestClassifier from the scikit-learn library to train a random forest classifier model on a given training dataset and then makes predictions on a test dataset. It then evaluates the performance of the model by calculating various metrics such as accuracy, precision, recall, F1-score and AUC.

It starts by importing the necessary libraries, such as RandomForestClassifier, accuracy\_score, precision\_score, recall\_score, f1\_score, roc\_auc\_score, classification\_report, confusion\_matrix from sklearn package.

Then, it defines a function named 'random\_forest\_classifier\_accuracy' which takes two parameters 'train\_data' and 'test\_data'. Inside the function, the dataset is split into features and labels for both the training and test sets. The features are stored in the X\_train and X\_test variables, and the labels are stored in the y\_train and y\_test variables.

Then, it creates an object of the RandomForestClassifier class and sets the number of trees in the forest to 100. Then, the fit() method is called on the classifier object with the X\_train and y\_train variables as arguments. This trains the classifier on the training data.

After that, the predict() method is called on the classifier object with the X\_test variable as an argument. This returns an array of predictions for the test data.

Then, the accuracy, precision, recall, F1-score, AUC and confusion matrix are calculated using the test labels (y\_test) and the predictions (y\_pred) obtained from the classifier. Finally, it prints the results of all the metrics obtained.

**Accuracy**: measures the proportion of correct predictions out of all predictions made by the model. In this case, the accuracy is 86.06%, which means that the model correctly predicted the class for 86.06% of the instances in the test set.

**Precision**: measures the proportion of true positive predictions out of all positive predictions made by the model. In this case, the precision is 95.78%, which means that out of all instances the model predicted as positive, 95.78% of them were actually positive.

**Recall**: measures the proportion of true positive predictions out of all actual positive instances. In this case, the recall is 83.19%, which means that the model correctly identified 83.70% of all actual positive instances.

**F1 Score**: the mean of precision and recall. In this case, the F1 score is 89.04%. F1 score gives the balance between precision and recall.

**AUC**: "Area Under the ROC Curve." it is a probability curve and AUC represents degree or measure of separability. It tells how much model is capable of distinguishing between classes. In this case, the AUC is 87.69%. A higher AUC indicates a better model.

(4)Fine-tuning random forest classifier using GridSearch CV

This code is performing a grid search for hyperparameter tuning on a RandomForestClassifier model. It first imports the GridSearchCV class from sklearn.model\_selection, which allows for an exhaustive search for the best combination of hyperparameters for a given model.

The function **perform\_grid\_search\_attack\_cat** takes four inputs: the training data and labels for the 'attack\_cat' column, and the test data and labels for the 'attack\_cat' column.

It then defines a dictionary, **param\_grid**, that lists the hyperparameters and the possible values for each hyperparameter that will be searched through in the grid search. The hyperparameters being tuned in this case are:

* n\_estimators: The number of trees in the forest.
* max\_depth: The maximum depth of the tree.
* min\_samples\_split: The minimum number of samples required to split an internal node.
* min\_samples\_leaf: The minimum number of samples required to be at a leaf node.

It then creates an instance of the RandomForestClassifier class with a random\_state of 42 and assigns it to the variable **rf**. It then creates an instance of the GridSearchCV class and assigns it to the variable **grid\_search**. The GridSearchCV class takes the following input:

* estimator: The model to be used in the grid search
* param\_grid: The hyperparameters and their possible values
* cv: Number of folds to use in the cross-validation.
* n\_jobs: Number of CPU cores used when parallel computing. -1 means using all available cores.

It then fits the grid\_search object to the training data and labels using the **fit()** method. It then finds the best parameters and best score of the grid search using the **best\_params\_** and **best\_score\_** attributes of the grid\_search object. It then finds the test score of the grid search using the **score()** method of the grid\_search object and the test data and labels. Finally, it prints out the best parameters, best cross-validation score, and test score for the 'attack\_cat' column.

This code defines a function called "perform\_grid\_search\_label" that takes in four inputs: X\_train, y\_train\_label, X\_test, and y\_test\_label.

The function first defines a parameter grid, which is a dictionary that contains several key-value pairs. These key-value pairs correspond to different hyperparameters that will be used in the grid search. The grid search will try different combinations of these hyperparameters to find the best combination that results in the highest accuracy. In this case, the grid search will try different values for 'n\_estimators', 'max\_depth', 'min\_samples\_split', and 'min\_samples\_leaf'.

The function then creates an instance of the RandomForestClassifier class, which is a type of ensemble learning method that can be used for classification and regression. It sets the random\_state to 42 to ensure reproducibility.

Next, the function creates an instance of the GridSearchCV class, which is a type of meta-estimator that performs an exhaustive search over a specified parameter space. It passes the RandomForestClassifier object as the estimator, the param\_grid as the parameter grid, and cv=5, n\_jobs=-1 which will use all available cores and perform a 5-fold cross-validation.

The function then fits the grid search to the training data (X\_train and y\_train\_label) and finds the best parameters and best score by calling the best\_params\_ and best\_score\_ attributes of the grid search object respectively. It then finds the test score by calling the score method on the grid search object and passing in the test data (X\_test and y\_test\_label).

Finally, the function prints out the best parameters, best CV score, and test score for the label column.

Logistic Regression

(3)Select and train models

This code imports the necessary libraries for building a logistic regression model and evaluating its performance.

First, it creates an instance of the LogisticRegression class and fits it to the training data (X\_train, y\_train). Then, it uses the trained model to make predictions on the test data (X\_test) and store it in the variable "LR\_prediction".

Then, it uses the .score() method to calculate the accuracy of the logistic regression model on the test data and print it. The accuracy score is a number between 0 and 1, representing the proportion of correct predictions made by the model.

It also uses the classification\_report() function from the sklearn.metrics library, which provides a detailed breakdown of the model's performance, including precision, recall, f1-score and support for each class.

Finally it print the classification report which contain the evaluation metric like precision, recall, f1-score and support for each class.

(4)Fine-tuning Model using GridSearchCV

This code is using the GridSearchCV function from scikit-learn to perform a grid search for hyperparameter tuning of a logistic regression model. The logistic regression model is first created and fit to the training data (X\_train, y\_train) using the LogisticRegression class from scikit-learn. Then, a pipeline is created which includes a StandardScaler to standardize the data, and the logistic regression model.

The grid search is performed on the following hyperparameters:

* 'solver': This is the algorithm to use in the optimization problem. The options being considered are 'newton-cg', 'lbfgs', 'liblinear', and 'sag'
* 'multi\_class': This is the option for multi-class problems. The options being considered are 'ovr' and 'multinomial'
* 'class\_weight': This is the option to balance the weight of different classes. The options being considered are 'None' and 'balanced'

The grid search will try all combinations of these parameters and evaluate the performance using 12-fold cross validation and accuracy as the scoring metric. The best set of hyperparameters will be stored in the 'best\_params\_' attribute of the grid object, and the best score will be stored in the 'best\_score\_' attribute.

(grid.best\_params\_)

The code is calling the **best\_params\_** attribute of the **GridSearchCV** object **grid** which is an instance of the **GridSearchCV** class. This attribute returns a dictionary that contains the best parameters found by the grid search, so the best combination of parameters that resulted in the best score. The **best\_params\_** attribute can be used to retrieve the best parameters after the grid search is completed and the **fit()** method has been called on the **GridSearchCV** object. It is a way to see the best parameters that were found by the grid search process, and we can use those parameters to improve the performance of our model.

This code is using the **sklearn** library's built-in metrics functions to evaluate the performance of a model. The **y\_test** and **y\_pred** variables are the true labels and predicted labels respectively for the test set.

The code is using the following metrics:

* **accuracy\_score**: This function calculates the proportion of correctly predicted instances (i.e., the number of correct predictions divided by the total number of predictions). It is defined as (number of correct predictions) / (total number of predictions)
* **precision\_score**: This function calculates the proportion of true positives over all instances that were predicted as positive. It is defined as (number of true positives) / (number of true positives + number of false positives)
* **recall\_score**: This function calculates the proportion of true positives over all actual positive instances. It is defined as (number of true positives) / (number of true positives + number of false negatives)
* **f1\_score**: This function calculates the harmonic mean of precision and recall. It is defined as 2 \* (precision \* recall) / (precision + recall)

Each of the function is called with the **y\_test** and **y\_pred** variables as inputs and the result is stored in a variable. The results are then printed out.