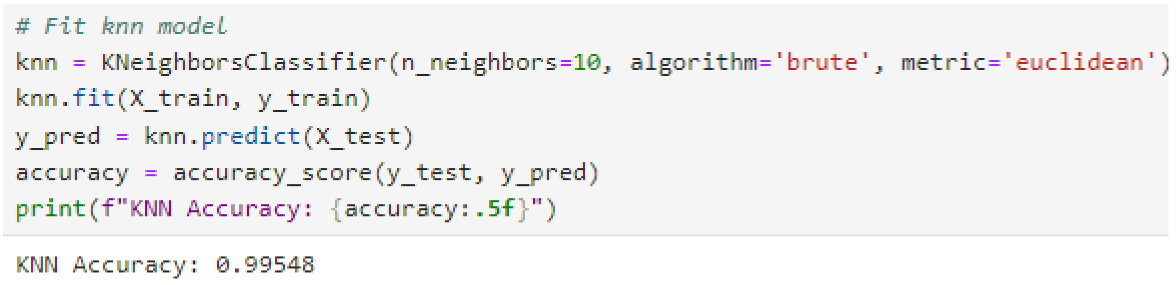
0. Preprocess data

Before I can fit data into any models, I need to clean it first. I notice that we have an abundance of data, so I remove all the missing values. Then I label encode the categorical data type columns. I was planning on oversample the target since we have imbalanced classes, but after analysing I realized that oversampled causes the model to train very slow because of too much data, so I will just have to work with imbalanced classes. Next, I feature scaling the data using quantile scaling. One thing to note about feature scaling is that I only scale the continuous data (data that have more than 10 unique values). Finally, was splitting the data into training and testing set.

1. The differences between hyperparameter and parameter

Hyperparameters are settings we tweak like learning rate (gradient-based model), Regularization, number of trees (tree models), layers (NN). These values are set before we train our model. Parameters are the final output like the slope and intercept line of linear regression or weights and biases in Neural Networks. I have used KNN and SVM as the two machine learning models for demonstration.

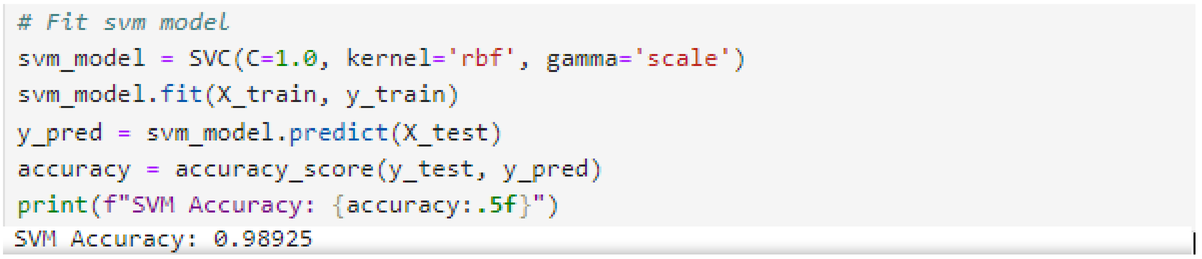
For KNN, it doesn't have parameters since it works by determining the class labels by its nearest k. It does have hyperparameters like `n_neighbors`: number of neighbors to query; `algorithm`: the query algorithm; `metric`: distance metric to use. Here is an example.



```
# Fit knn model
knn = KNeighborsClassifier(n_neighbors=10, algorithm='brute', metric='euclidean')
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"KNN Accuracy: {accuracy:.5f}")
```

```
KNN Accuracy: 0.99548
```

For SVM, the parameters are weights (coefficients), and bias (intercept) learned during model training to define the decision boundary. Hyperparameters `C`: Regularization; `kernel`: kernel type; `gamma`: kernel coefficient. Here is an example.



```
# Fit svm model
svm_model = SVC(C=1.0, kernel='rbf', gamma='scale')
svm_model.fit(X_train, y_train)
y_pred = svm_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"SVM Accuracy: {accuracy:.5f}")
```

```
SVM Accuracy: 0.98925
```

2. Prove that Elastic net can be used as either LASSO or Ridge regulariser

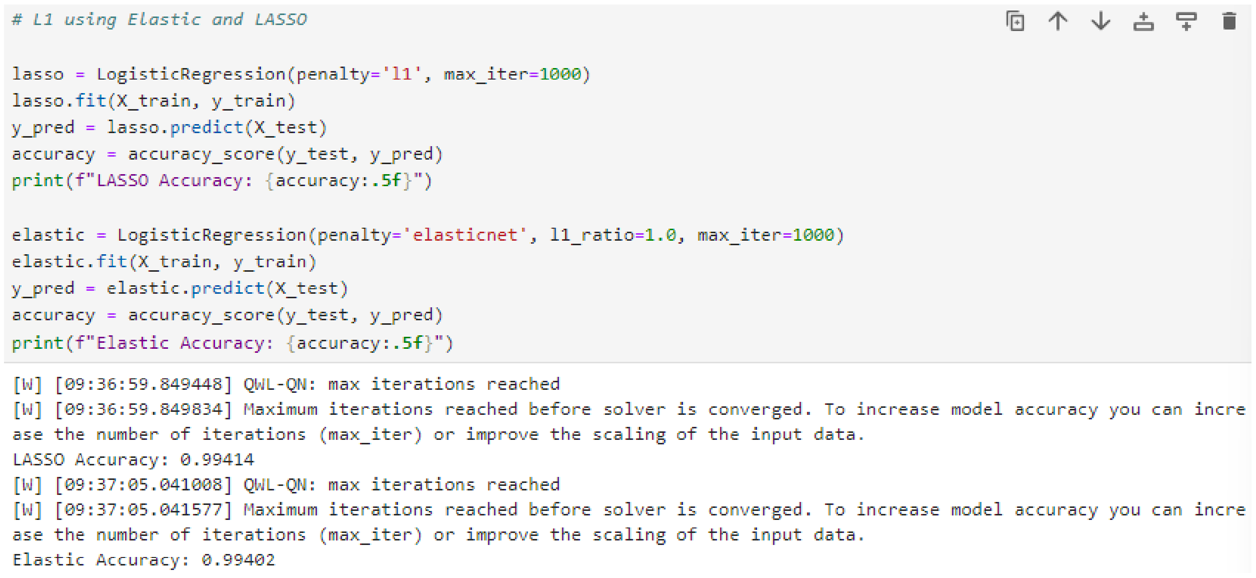
Elastic net is a combination of both LASSO and Ridge, and why can tune it to behave like either of the two or as a mix.

- Elastic net with `l1_ratio=1.0` behaves like LASSO (`L1` regularization)

- Elastic net with `l1_ratio=0.0` behaves like Ridge (`L2` regularization)

- Elastic net with `l1_ratio=0.5` is the combination of both

Here is the code to prove.

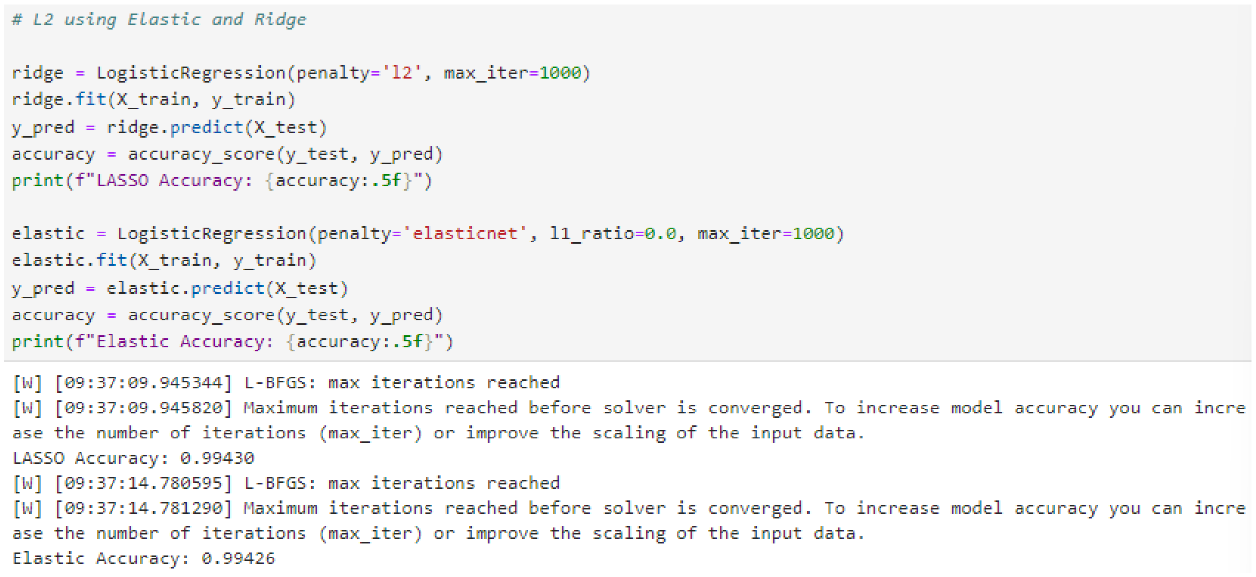


```
# L1 using Elastic and LASSO

lasso = LogisticRegression(penalty='l1', max_iter=1000)
lasso.fit(X_train, y_train)
y_pred = lasso.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"LASSO Accuracy: {accuracy:.5f}")

elastic = LogisticRegression(penalty='elasticnet', l1_ratio=1.0, max_iter=1000)
elastic.fit(X_train, y_train)
y_pred = elastic.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Elastic Accuracy: {accuracy:.5f}")
```

```
[W] [09:36:59.849448] QWL-QN: max iterations reached
[W] [09:36:59.849834] Maximum iterations reached before solver is converged. To increase model accuracy you can incre
ase the number of iterations (max_iter) or improve the scaling of the input data.
LASSO Accuracy: 0.99414
[W] [09:37:05.041008] QWL-QN: max iterations reached
[W] [09:37:05.041577] Maximum iterations reached before solver is converged. To increase model accuracy you can incre
ase the number of iterations (max_iter) or improve the scaling of the input data.
Elastic Accuracy: 0.99402
```



```
# L2 using Elastic and Ridge

ridge = LogisticRegression(penalty='l2', max_iter=1000)
ridge.fit(X_train, y_train)
y_pred = ridge.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"LASSO Accuracy: {accuracy:.5f}")

elastic = LogisticRegression(penalty='elasticnet', l1_ratio=0.0, max_iter=1000)
elastic.fit(X_train, y_train)
y_pred = elastic.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
print(f"Elastic Accuracy: {accuracy:.5f}")
```

```
[W] [09:37:09.945344] L-BFGS: max iterations reached
[W] [09:37:09.945820] Maximum iterations reached before solver is converged. To increase model accuracy you can incre
ase the number of iterations (max_iter) or improve the scaling of the input data.
LASSO Accuracy: 0.99430
[W] [09:37:14.780595] L-BFGS: max iterations reached
[W] [09:37:14.781290] Maximum iterations reached before solver is converged. To increase model accuracy you can incre
ase the number of iterations (max_iter) or improve the scaling of the input data.
Elastic Accuracy: 0.99426
```

We can see that the accuracy for both L1 using LASSO, and Elastic tune to be LASSO is the same. The same can be said for Ridge, and Elastic tune to be Ridge.

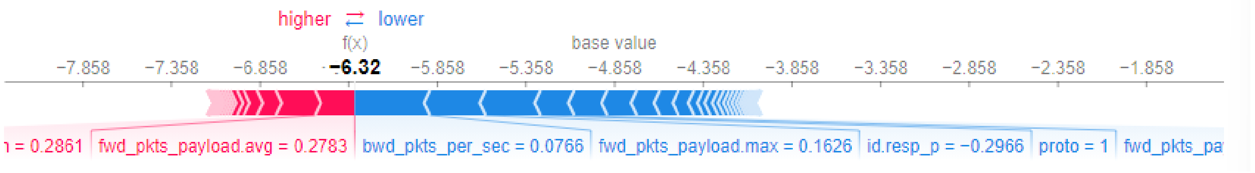
3. Feature importance

I have used the power of Random Forest model to show me the feature importance of the columns. Géron (2019) said that Random Forest looks at how much the tree nodes used a particular feature to reduce the impurity on average across all trees in the forest. I have printed out features that have above or equal 1% importance.

```
Important features: ['fwd_data_pkts_tot', 'flow_pkts_per_sec', 'down_up_ratio', 'fwd_header_size_tot', 'fwd_header_si
ze_min', 'fwd_header_size_max', 'bwd_header_size_min', 'bwd_PSH_flag_count', 'fwd_URG_flag_count', 'bwd_URG_flag_coun
t', 'fwd_pkts_payload.min', 'fwd_pkts_payload.max', 'fwd_pkts_payload.tot', 'fwd_pkts_payload.avg', 'bwd_pkts_payloa
d.max', 'bwd_pkts_payload.tot', 'bwd_pkts_payload.avg', 'flow_pkts_payload.max', 'flow_pkts_payload.tot', 'flow_pkts_
payload.avg', 'fwd_iat.min', 'fwd_iat.max', 'fwd_iat.tot', 'fwd_iat.avg', 'flow_iat.min', 'flow_iat.max', 'flow_iat.t
ot', 'flow_iat.avg', 'fwd_subflow_bytes', 'active.min', 'active.max', 'active.tot', 'active.avg', 'fwd_init_window_si
ze', 'fwd_last_window_size']
Total score of important features: 0.86
```

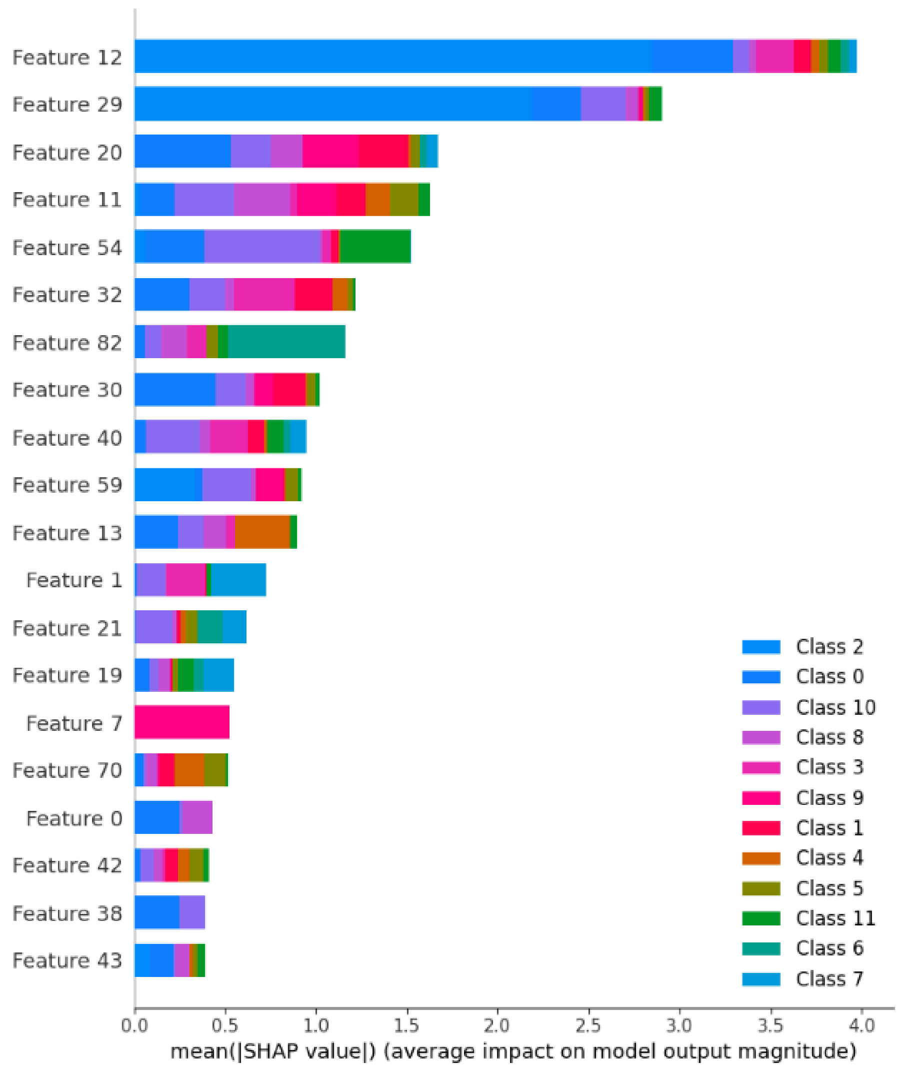
Another method I used is with SHAP (SHapley Additive exPlanations). SHAP measures how much each feature contributes to the model's prediction (Awan, 2023). SHAP uses game theory to explain the output of any prediction. Here is the SHAP values for the first instance.

```
# Explanation object
explanation = shap.Explanation(
    values=shap_values[..., 0][0],   # SHAP values for the first instance
    base_values=explainer.expected_value[0],  # Corresponding base value
    data=X_test.iloc[0],  # Feature values
    feature_names=X_test.columns
)
shap.force_plot(explanation)
```



The force plot shows us how each feature contributes to the prediction of each sample. To see the overall feature importance, we can plot a summary plot on the mean absolute value of each feature over all the instances.

```
shap.summary_plot(shap_values, X_test, plot_type="bar")
```



Comparing the two, in terms of computational wise, Random Forest feature importance is faster compared to SHAP (not by much). In terms of how the work, for SHAP, it is based on game theory and explains the contribution of each feature on the prediction. Random Forest calculates how frequently a feature is used to split the data across all trees. We can see that tree-based models like Random Forest calculate the feature importance quite bias compared to the robustness of SHAP.

4. Three supervised none ensemble models to predict the target

a) Some metrics to measure the performance of a classifier can are accuracy and classification report (includes precision, recall, f1-score, support). If we tested these metrics on both the train and test set and see a large differences, we can conclude that our model is overfitting. But from the looks of it, all of the models perform really well (at least 98%). I have tried to oversample and re-fit the data, but the overall results are pretty much the same. I don't think I can show an overfit model based on the preprocessed data.

b) I have used SVM, KNN and Decision as the 3 models. The designs for each model differ in the hyperparameter I set for each one.

c) I have used Grid Search to search for the most optimal features. For SVM the hyperparameters are `C`, `kernel`, `gamma`. For KNN are `n_neighbors` and ` algorithm`. To choose the hyperparameters for searching, I just look up on the library documentation. For Decision Tree I didn't use Grid Search because there doesn't seem to be a library with GPU Accelerated on Decision Tree, but I did use post pruning. Post pruning is used after constructing the Decision Tree model to control the depth of the tree.

d) The accuracy of SVM is 0.996, Decision Tree is 0.9976795, KNN is 0.996. It seems like Decision Tree output the best score with 0.9977. The score could be a little bit higher if I had used grid search on Decision Tree, but it took so long so I just settled for this value.

5. Three ensemble models

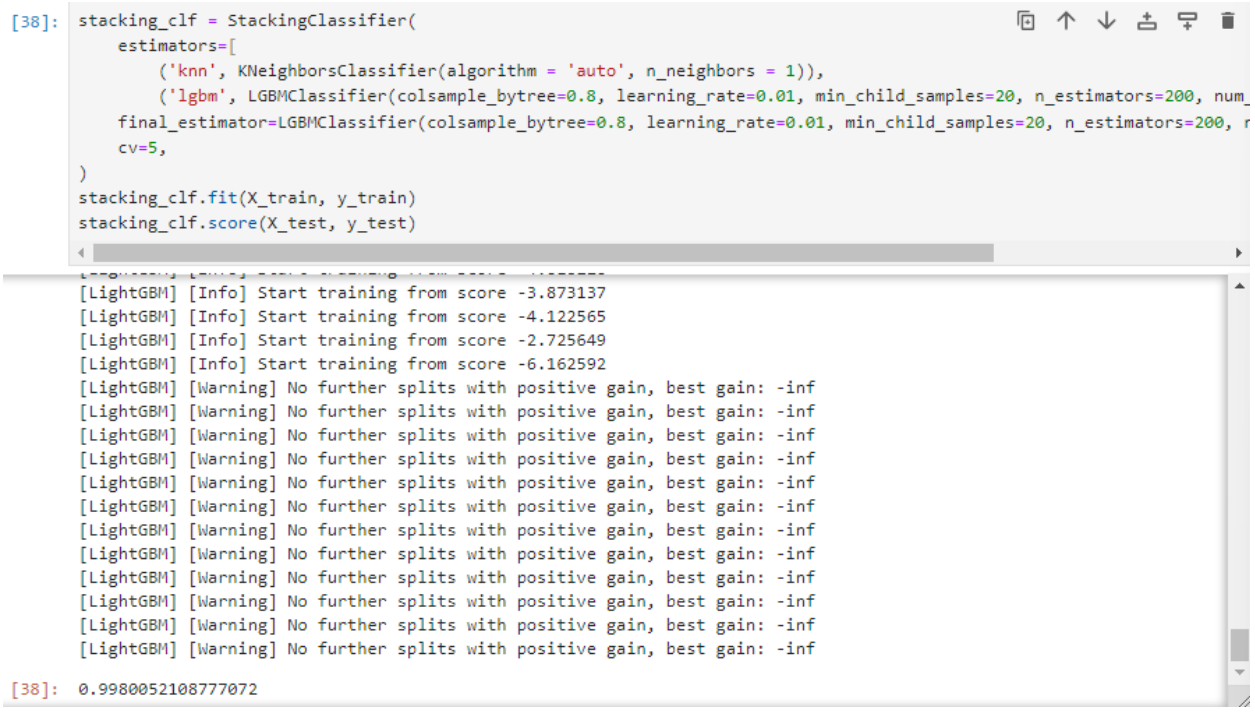
a) I think for most real-world applications, we need to use ensemble models because of the complexity of the data. We don't have the luxury of having data suitable for any given type of model, so we need models that are powerful and can capture complex patterns in the data.

b) I have used Random Forest, Gradient Boost and Stacking as the 3 ensemble models. The Random Forest is an ensemble of decision trees; Gradient Boost is a model work by sequentially adding predictions to an ensemble, with each correcting its predecessors' residual errors; Stacking is a model trained on the aggregation of other models (Géron, 2019).

c) It is quite hard to say when to use what ensemble models. We need to try different models to see which one is the most suitable for our data. One rule of thumb from Géron (2019) is to use Random Forest, AdaBoost and Gradient Boost first, then use Voting and Stacking Classifiers to squeeze the model performance to its limits.

d) I have used Grid Search on Random Forest and Gradient Boost. Random Forest has an accuracy of 0.9969 and Gradient Boost has an accuracy of 0.9982. Decision Tree and Gradient Boost are the 2 best models, so I combined these two into a Stacking Classifiers which gives an accuracy of 0.9983. This is the final model and the best one.

e) Yes, we can use Voting or Stacking Classifier to combine different models like KNN and Gradient Boost. Here is an example of that.



```
[38]: stacking_clf = StackingClassifier(
          estimators=[
              ('knn', KNeighborsClassifier(algorithm = 'auto', n_neighbors = 1)),
              ('lgbm', LGBMClassifier(colsample_bytree=0.8, learning_rate=0.01, min_child_samples=20, n_estimators=200, num_
          final_estimator=LGBMClassifier(colsample_bytree=0.8, learning_rate=0.01, min_child_samples=20, n_estimators=200, r
          cv=5,
      )
      stacking_clf.fit(X_train, y_train)
      stacking_clf.score(X_test, y_test)
```

```
[LightGBM] [Info] Start training from score -3.873137
[LightGBM] [Info] Start training from score -4.122565
[LightGBM] [Info] Start training from score -2.725649
[LightGBM] [Info] Start training from score -6.162592
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf

[38]: 0.9980052108777072
```

References

Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition* [Book]. [online] www.oreilly.com. Available at: https://learning.oreilly.com/library/view/hands-on-machine-learning/9781492032632/.

Awan, A. A. (2023, June). *Introduction to SHAP values for machine learning interpretability*. DataCamp. https://www.datacamp.com/tutorial/introduction-to-shap-values-machine-learning-interpretability