Performance Comparison of Multi-Class Classification Algorithms

This article comprises the application and comparison of supervised multi-class classification algorithms to a dataset, which involves the chemical compositions (features) and types (four major types – target) of stainless steels. The dataset is quite small in numbers, but very accurate.

Stainless steel alloy datasets are commonly limited in size, thus restraining applications of Machine Learning (ML) techniques for classification. I explored the potential of 6 different classification algorithms, in the context of a small dataset of 62 samples, for outcome prediction in type classification.

In this article, multi-class classification was analyzed using various algorithms, with the target of classifying the stainless steels using their chemical compositions (15 elements). There are four basic types are of stainless steels and some alloys have very close compositions. Hyperparameter tuning by Grid Search was also applied for Random Forest and XGBoost algorithms in order to observe possible improvements on the metrics. Finally, the performances of the algorithms were compared. After the application of these algorithms, the successes of the models were evaluated with appropriate performance metrics.

The dataset was prepared using "High-Temperature Property Data: Ferrous Alloys".

Wikipedia's definition for multi-class classification is: "In machine learning, multiclass or multinomial classification is the problem of classifying instances into one of three or more classes (classifying instances into one of two classes is called binary classification)."

The following algorithms were used for classification analysis:

- Decision Tree Classifier,
- Random Forest Classifier,
- XGBoost Classifier,
- Naïve Bayes,
- Support Vector Machines (SVM),
- AdaBoost.

The following issues were the scope of this study:

- Which algorithm provided the best results for multi-class classification?
- Was hyperparameter tuning successful in improving the metrics?
- Reason for poor (if any) metrics.
- Is it safe to use these methods for multi-class classification of alloys.

Data Cleaning

The first step is to import and clean the data (if needed) using pandas before starting the analysis.

```
df = pd.read_csv('SS-Compositions.csv')
```

```
df.sample(8)
```

Carbon	Manganese	Silicon	Chromium	Nickel	Molybdenum	Phosphorus	Nitrogen	Sulphur	Niobium	Aluminium	Titanium	Copper	Vanadium	Tungsten	Type
0.15	1.25	1.0	13.00	0.00	0.60	0.060	0.00	0.150	0.0	0.00	0.00	0.0	0.00	0.00	M
0.15	2.00	2.5	18.00	9.00	0.00	0.045	0.00	0.030	0.0	0.00	0.00	0.0	0.00	0.00	A
0.05	0.10	0.1	12.75	8.00	2.25	0.010	0.01	0.008	0.0	1.13	0.00	0.0	0.00	0.00	P
0.23	0.75	0.5	12.00	0.75	1.12	0.025	0.00	0.025	0.0	0.00	0.00	0.0	0.25	1.75	M
0.08	1.00	1.0	11.13	0.50	0.00	0.045	0.00	0.045	0.0	0.00	0.75	0.0	0.00	0.00	F
0.08	2.00	1.0	18.00	10.50	0.00	0.045	0.10	0.030	0.0	0.00	0.40	0.0	0.00	0.00	A
0.13	0.88	0.5	15.50	4.50	2.88	0.040	0.10	0.030	0.0	0.00	0.00	0.0	0.00	0.00	P
0.07	1.00	0.0	18.00	0.50	0.00	0.040	0.00	0.030	0.0	0.15	0.00	0.0	0.00	0.00	F

```
df.shape
```

```
(62, 17)
```

There are 25 austenitic (A), 17 martensitic (M), 11 ferritic (F) and 9 precipitation-hardening (P) stainless steels in the dataset.

```
df.Type.value_counts()
```

```
A    25
M    17
F    11
P     9
```

There are 62 rows (stainless steels) and 17 columns (attributes) of data. 15 columns cover the chemical composition information of the alloys. The first column is the AISI designation and the last column is the type of the alloy. Our target is to estimate the type of the steel.

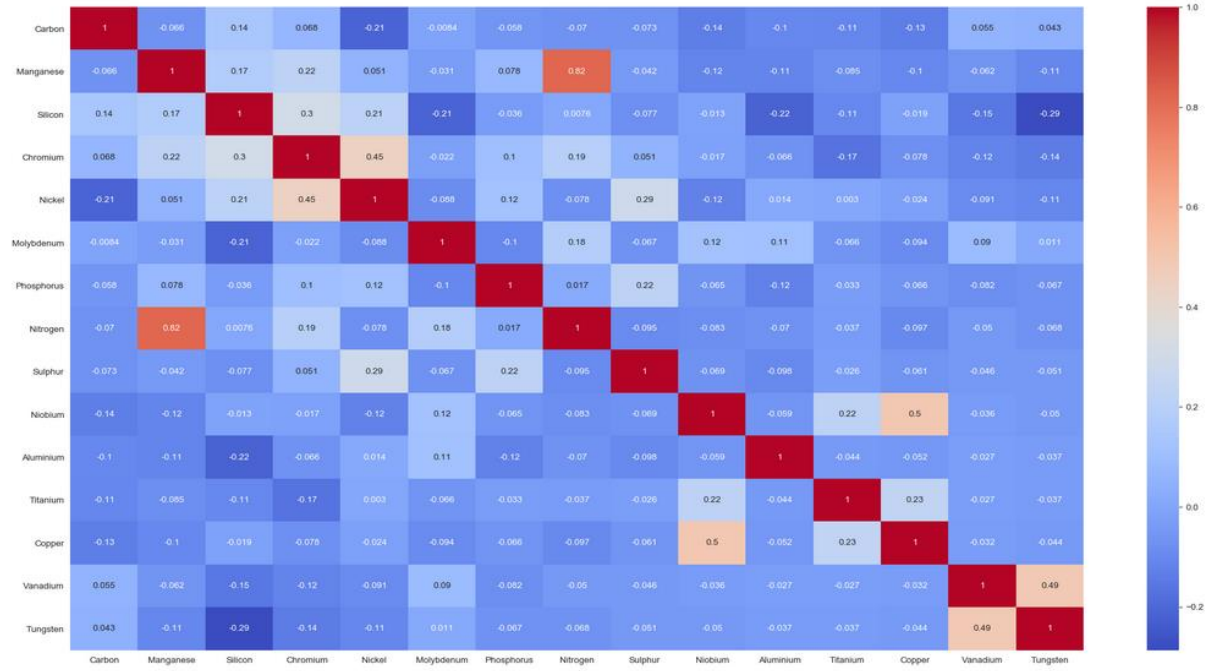
Descriptive statistics of the dataset are shown below. Some element percentages are almost stable (Sulphur), but Chromium and Nickel percentages have a very wide range (and these two elements are the defining elements of stainless steels).

```
df.describe().T
```

	count	mean	std	min	25%	50%	75%	max
Carbon	62.0	0.156339	0.173878	0.030	0.08	0.12	0.1500	1.08
Manganese	62.0	1.838871	2.261588	0.100	1.00	1.00	2.0000	14.75
Silicon	62.0	0.939194	0.369928	0.000	1.00	1.00	1.0000	2.50
Chromium	62.0	16.379194	4.677556	5.000	13.00	17.00	18.0000	30.00
Nickel	62.0	6.080968	7.549874	0.000	0.00	4.50	9.1875	35.50
Molybdenum	62.0	0.514355	0.870893	0.000	0.00	0.00	0.7500	3.50
Phosphorus	62.0	0.048387	0.036862	0.010	0.04	0.04	0.0450	0.28
Nitrogen	62.0	0.032823	0.085510	0.000	0.00	0.00	0.0000	0.38
Sulphur	62.0	0.040290	0.042555	0.008	0.03	0.03	0.0300	0.30
Niobium	62.0	0.031290	0.111832	0.000	0.00	0.00	0.0000	0.60
Aluminium	62.0	0.042097	0.202704	0.000	0.00	0.00	0.0000	1.13
Titanium	62.0	0.036290	0.174189	0.000	0.00	0.00	0.0000	1.10
Copper	62.0	0.180645	0.730111	0.000	0.00	0.00	0.0000	4.00
Vanadium	62.0	0.004032	0.031750	0.000	0.00	0.00	0.0000	0.25
Tungsten	62.0	0.076613	0.437927	0.000	0.00	0.00	0.0000	3.00

Correlation can be defined as a measure of the dependence of one variable on the other one. Two features being highly correlated with each other will provide too much and useless information on finding the target. The heatmap below shows that the highest correlation is between manganese and nitrogen. Manganese is present in every steel, but nitrogen is not even present in most of the alloys; so, I kept both.

```
plt.figure(figsize=(25, 13))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm');
```



The dataset is clean (there are no NaNs, Dtype are correct), so we will directly start by Train-Test-Split and then apply the algorithms.

Decision Tree Classifier

First algorithm is the Decision Tree Classifier. It uses a decision tree (as a predictive model) to go from observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves).

Decision Tree Classifier

```
y_pred = modelTree.predict(X_test)
confusion_matrix(y_test, y_pred)
```

```
array([[8, 0, 0, 0],
       [0, 2, 1, 0],
       [0, 0, 5, 0],
       [0, 0, 0, 3]], dtype=int64)
```

```
df_f1 = f1_score(y_test, y_pred, average='macro')
df_f1
```

```
0.9272727272727272
```

```
dt_accuracy = accuracy_score(y_test, y_pred)
dt_accuracy
```

```
0.9473684210526315
```

```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           A       1.00      1.00      1.00         8
           F       1.00      0.67      0.80         3
           M       0.83      1.00      0.91         5
           P       1.00      1.00      1.00         3

    accuracy                           0.95        19
   macro avg       0.96      0.92      0.93        19
weighted avg       0.96      0.95      0.94        19
```

The results are very good; actually, only one alloy type was classified mistakenly.

Random Forest Classifier

Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks that operate by constructing a multitude of decision trees at training time and outputting

the class that is the mode of the classes (classification) or mean/average prediction (regression) of the individual trees.

Random forests generally outperform decision trees, but their accuracy is lower than gradient boosted trees. However, data characteristics can affect their performance [ref].

Random Forest Classifier

```
rf_model=RandomForestClassifier().fit(X_train, y_train)
```

```
y_pred_rf = rf_model.predict(X_test)
confusion_matrix(y_test, y_pred_rf)
```

```
array([[8, 0, 0, 0],
       [0, 3, 0, 0],
       [0, 1, 4, 0],
       [0, 0, 0, 3]], dtype=int64)
```

```
print(classification_report(y_test, y_pred_rf))
```

```
              precision    recall  f1-score   support

           A       1.00      1.00      1.00         8
           F       0.75      1.00      0.86         3
           M       1.00      0.80      0.89         5
           P       1.00      1.00      1.00         3

    accuracy                           0.95        19
   macro avg       0.94      0.95      0.94        19
weighted avg       0.96      0.95      0.95        19
```

```
rf_accuracy = accuracy_score(y_test, y_pred_rf)
rf_accuracy
```

```
0.9473684210526315
```

```
rf_f1 = f1_score(y_test, y_pred_rf, average='macro')
rf_f1
```

```
0.9365079365079365
```

Hyperparameter Tuning with Grid Search

Even though I got satisfactory results with Random Forest Analysis, I applied hyperparameter tuning with Grid Search. Grid search is a common method for tuning a model's hyperparameters. The grid search algorithm is simple: feed it a set of hyperparameters and the values to be tested for each hyperparameter, and then run an exhaustive search over all possible combinations of these values, training one model for each set of values. The algorithm then compares the scores of each model it trains and keeps the best one. Here are the results:

Random Forest Tuning

```
rf = RandomForestClassifier()
```

```
rf_params = {"n_estimators":[50, 100, 300],
             "max_depth":[3,5,7],
             "max_features": [2,4,6,8],
             "min_samples_split": [2,4,6]}
```

```
from sklearn.model_selection import GridSearchCV
rf_cv_model = GridSearchCV(rf, rf_params, cv = 5, n_jobs = -1, verbose = 2).fit(X_train, y_train)
```

```
Fitting 5 folds for each of 108 candidates, totalling 540 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  17 tasks      | elapsed:    2.9s
[Parallel(n_jobs=-1)]: Done 138 tasks      | elapsed:    7.3s
[Parallel(n_jobs=-1)]: Done 341 tasks      | elapsed:   14.8s
[Parallel(n_jobs=-1)]: Done 540 out of 540 | elapsed:   22.6s finished
```

```
rf_cv_model.best_params_
```

```
{'max_depth': 7,
 'max_features': 6,
 'min_samples_split': 4,
 'n_estimators': 300}
```

```
rf_tuned = RandomForestClassifier(max_depth = 7,
                                  max_features = 6,
                                  min_samples_split = 4,
                                  n_estimators = 300).fit(X_train, y_train)
```

```
y_pred = rf_tuned.predict(X_test)
confusion_matrix(y_test, y_pred)
```

```
array([[8, 0, 0, 0],
       [0, 3, 0, 0],
       [0, 0, 5, 0],
       [0, 0, 0, 3]], dtype=int64)
```

```
rf_f1_tuned = f1_score(y_test, y_pred, average='macro')
rf_f1_tuned
```

```
1.0
```

```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           A       1.00      1.00      1.00         8
           F       1.00      1.00      1.00         3
           M       1.00      1.00      1.00         5
           P       1.00      1.00      1.00         3

    accuracy                           1.00        19
   macro avg       1.00      1.00      1.00        19
weighted avg       1.00      1.00      1.00        19
```

Hyperparameter tuning with Grid Search took the results to the perfect level – or overfitting.

XGBoost Classifier

XGBoost is well known to provide better solutions than other machine learning algorithms. In fact, since its inception, it has become the "state-of-the-art" machine learning algorithm to deal with structured data.

The results of the XGBoost Classifier provided the best results for this classification study.

XGBoost Classifier

```
y_pred = xgb_classifier.predict(X_test)
```

```
confusion_matrix(y_test, y_pred)
```

```
array([[8, 0, 0, 0],
       [0, 3, 0, 0],
       [0, 0, 5, 0],
       [1, 0, 0, 2]], dtype=int64)
```

```
xgb_accuracy = accuracy_score(y_test, y_pred)
xgb_accuracy
```

```
0.9473684210526315
```

```
xgb_f1 = f1_score(y_test, y_pred, average='macro')
xgb_f1
```

```
0.9352941176470588
```

```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           A       0.89      1.00      0.94         8
           F       1.00      1.00      1.00         3
           M       1.00      1.00      1.00         5
           P       1.00      0.67      0.80         3

    accuracy                           0.95        19
   macro avg       0.97      0.92      0.94        19
```

Hyperparameter Tuning with Grid Search

Once again, I applied the hyperparameter tuning with Grid Search, even though the results were near perfect.

Tuning XGBoost

```
xgb = XGBClassifier()
```

```
xgb_params = {"n_estimators": [50, 100, 300],
              "subsample":[0.5,0.8,1],
              "max_depth":[3,5,7],
              "learning_rate":[0.1,0.01,0.3]}
```

```
from sklearn.model_selection import train_test_split, GridSearchCV
```

```
xgb_cv_model = GridSearchCV(xgb, xgb_params, cv = 3,
                            n_jobs = -1, verbose = 2).fit(X_train, y_train)
```

```
Fitting 3 folds for each of 81 candidates, totalling 243 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  17 tasks      | elapsed:    8.2s
[Parallel(n_jobs=-1)]: Done 243 out of 243 | elapsed:    9.9s finished
```

```
xgb_cv_model.best_params_
```

```
{'learning_rate': 0.3, 'max_depth': 3, 'n_estimators': 300, 'subsample': 0.5}
```

```
xgb_tuned = XGBClassifier(learning_rate= 0.3,
                                max_depth= 3,
                                n_estimators= 300,
                                subsample= 0.5).fit(X_train, y_train)
```

```
y_pred = xgb_tuned.predict(X_test)
confusion_matrix(y_test, y_pred)
```

```
array([[8, 0, 0, 0],
       [0, 3, 0, 0],
       [0, 0, 5, 0],
       [0, 0, 0, 3]], dtype=int64)
```

```
xgb_f1_tuned = f1_score(y_test, y_pred, average='macro')
xgb_f1_tuned
```

```
1.0
```

```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           A       1.00      1.00      1.00         8
           F       1.00      1.00      1.00         3
           M       1.00      1.00      1.00         5
           P       1.00      1.00      1.00         3

    accuracy                           1.00        19
   macro avg       1.00      1.00      1.00        19
weighted avg       1.00      1.00      1.00        19
```

Naïve Bayes Classifier

Naïve Bayes algorithm is a supervised learning algorithm, which is based on Bayes theorem and used for solving classification problems. Naïve Bayes Classifier is one of the simple and most effective Classification algorithms which helps in building the fast machine learning models that can make quick predictions.

The results are shown below:

Naive Bayes

```
model = MultinomialNB()
model.fit(X_train, y_train)

nb_count_acc = cross_val_score(model, X_test, y_test,cv = 8).mean()
print(nb_count_acc)
```

```
0.6666666666666666
```

```
y_pred = model.predict(X_test)
confusion_matrix(y_test,y_pred)
```

```
array([[8, 0, 0, 0],
       [0, 3, 0, 0],
       [0, 1, 4, 0],
       [1, 0, 0, 2]], dtype=int64)
```

```
nb_f1 = f1_score(y_test, y_pred, average='weighted')
nb_f1
```

```
0.8918570937146789
```

```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           A       0.89      1.00      0.94         8
           F       0.75      1.00      0.86         3
           M       1.00      0.80      0.89         5
           P       1.00      0.67      0.80         3

    accuracy                           0.89        19
   macro avg       0.91      0.87      0.87        19
weighted avg       0.91      0.89      0.89        19
```

Support Vector Machines (SVM)

Support-vector machines (SVMs, also support-vector networks) are supervised learning models with associated learning algorithms that analyze data for classification and regression analysis. An SVM maps training examples to points in space to maximize the width of the gap between the two categories. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall.

The results are shown below:

Support Vector Machine (SVM)

```
model = SVC()
model.fit(X_train, y_train)

svm_acc = cross_val_score(model, X_test, y_test,cv = 8).mean()
print(svm_acc)
```

```
0.6875
```

```
y_pred = model.predict(X_test)
confusion_matrix(y_test,y_pred)
```

```
array([[8, 0, 0, 0],
       [0, 1, 2, 0],
       [0, 0, 5, 0],
       [2, 0, 1, 0]], dtype=int64)
```

```
svm_f1 = f1_score(y_test, y_pred, average='weighted')
svm_f1
```

```
0.6556455240665767
```

```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           A       0.80      1.00      0.89         8
           F       1.00      0.33      0.50         3
           M       0.62      1.00      0.77         5
           P       0.00      0.00      0.00         3

    accuracy                           0.74        19
   macro avg       0.61      0.58      0.54        19
weighted avg       0.66      0.74      0.66        19
```

AdaBoost

AdaBoost, short for Adaptive Boosting, is a machine learning meta-algorithm, which can be used in conjunction with many other types of learning algorithms to improve performance. The output of the other learning algorithms ('weak learners') is combined into a weighted sum that represents the final output of the boosted classifier.

The results are shown below:

ADA Boosting

```
model = AdaBoostClassifier()
model.fit(X_train, y_train)

ada_acc = cross_val_score(model, X_test, y_test,cv = 8).mean()
print(ada_acc)
```

```
0.6875
```

```
y_pred = model.predict(X_test)
confusion_matrix(y_test,y_pred)
```

```
array([[8, 0, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 5, 0],
       [0, 0, 0, 3]], dtype=int64)
```

```
ada_f1 = f1_score(y_test, y_pred, average='weighted')
ada_f1
```

```
0.7813765182186235
```

```
print(classification_report(y_test, y_pred))
```

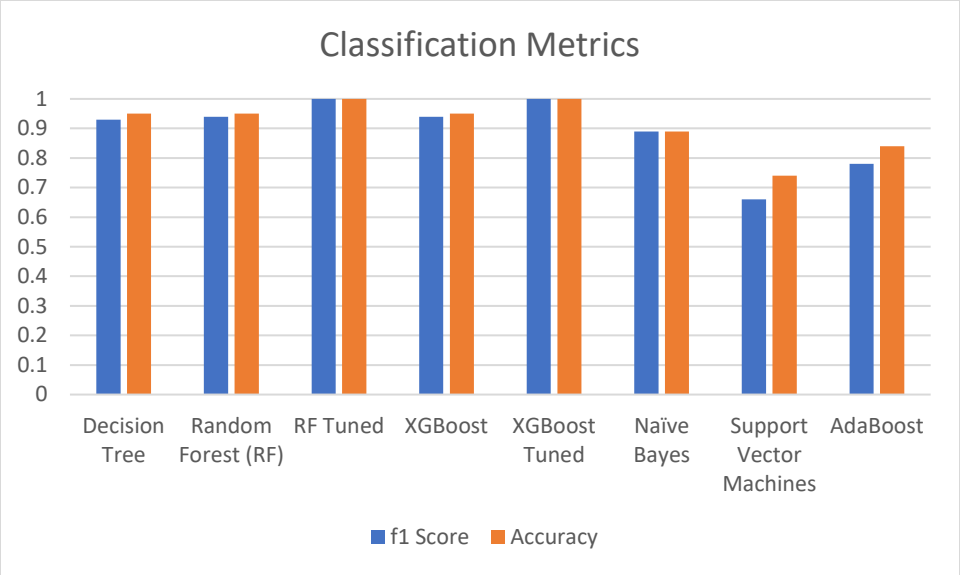
```
              precision    recall  f1-score   support

           A       1.00      1.00      1.00         8
           F       0.00      0.00      0.00         3
           M       0.62      1.00      0.77         5
           P       1.00      1.00      1.00         3

    accuracy                           0.84        19
   macro avg       0.66      0.75      0.69        19
weighted avg       0.74      0.84      0.78        19
```

Conclusion

In this article, I used six different Supervised Machine Learning (Classification) algorithms with the purpose of classifying four types of stainless steels (multi-class) according to their chemical compositions comprised of 15 elements in the alloy. The dataset included 62 alloys; which made it a small, but a very accurate dataset (all the information was taken from ASM International Sources (formerly known as American Society of Metals)).



The analysis provides evidence that:

- Considering the f1 scores, Random Forest and XGBoost methods produced the best results (0.94).
- After hyperparameter tuning by Grid Search, RF and XGBoost f1 scores jumped to 100 %.
- Multiple tries of the same algorithm resulted different results with a huge gap – most probably due to the limited data size.
- The poorest f1 scores were mostly for the classification of the types that have the least-numbered groups; which were ferritic and precipitation-hardened steels.
- Finally, test classification accuracy of 95% achieved by 3 models (DT, RF and XGBoost) and 100% by 2 tuned models demonstrates that the ML approach can be effectively applied to steel classification despite the small number of alloys and heterogeneous input parameters (chemical compositions). Based on only 62 cases, the models achieved a very high level of performance for multi-class alloy type classification.

You can access to this article and similar ones here.

Photo by Ben Wicks on Unsplash

I run the classifier many times and even got the results below. I know that it is a very small dataset, but still Random Forest got the 100 %.!!

Random Forest Classifier

```
rf_model=RandomForestClassifier().fit(X_train, y_train)
```

```
y_pred_rf = rf_model.predict(X_test)
confusion_matrix(y_test, y_pred_rf)
```

```
array([[8, 0, 0, 0],
       [0, 3, 0, 0],
       [0, 0, 5, 0],
       [0, 0, 0, 3]], dtype=int64)
```

```
print(classification_report(y_test, y_pred_rf))
```

```
              precision    recall  f1-score   support

           A       1.00      1.00      1.00         8
           F       1.00      1.00      1.00         3
           M       1.00      1.00      1.00         5
           P       1.00      1.00      1.00         3

    accuracy                           1.00        19
   macro avg       1.00      1.00      1.00        19
weighted avg       1.00      1.00      1.00        19
```

```
rf_accuracy = accuracy_score(y_test, y_pred_rf)
rf_accuracy
```

```
1.0
```

```
rf_f1 = f1_score(y_test, y_pred_rf, average='macro')
rf_f1
```

```
1.0
```