CO544 - Machine Learning and Data Mining

Group Project | Final Report

Group 01: E/15/154 | E/15/179 | E/15/187

ABSTRACT

A detailed description of the methodologies which were used for an imbalanced, binary classification is discussed here in this report. Several machine learning techniques and algorithms have been used to achieve the desired outputs and the predicted result was over 88% of accuracy. Techniques used were shown to be an effective way of classifying features with a given dataset that contains both numerical and categorical data. Further improvements also can be discovered at the latter part of the report.

Table of Contents

---

1. Introduction

With a given train dataset, it was asked to predict the outcome, *whether it is a "Success" or a "Failure"*, as a classification problem using machine learning techniques. The approach to the problem is discussed below and the possible methods, procedure, difficulties and how to address the common problems when developing a solution for such a problem are also mentioned in this report.

There are several techniques and methods to address this problem and the main difficulty is to find the ideal or the most suitable approach to get the most accurate outcome. Therefore, the approach taken to find a solution to the given task is rather empirical.

Further, there can be many other solutions that can be developed to achieve the same result and therefore there is no *right* way for the solution. Important matter is to maintain and increase the accuracy of the prediction.

2. Problem Definition and Algorithm

2.1. Task Definition

As discussed before, the given problem is a simple classification task. The given dataset contains a large amount of heterogeneous data. There are both numerical and categorical data with missing values too in the dataset.

```
print(dataframe.info())
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 524 entries, 0 to 551
Data columns (total 16 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   A1      524 non-null    object
 1   A2      524 non-null    float64
 2   A3      524 non-null    object
 3   A4      524 non-null    object
 4   A5      524 non-null    float64
 5   A6      524 non-null    object
 6   A7      524 non-null    int64
 7   A8      524 non-null    bool
 8   A9      524 non-null    object
 9   A10     524 non-null    float64
 10  A11     524 non-null    bool
 11  A12     524 non-null    int64
 12  A13     524 non-null    bool
 13  A14     524 non-null    float64
 14  A15     524 non-null    object
 15  A16     524 non-null    object
dtypes: bool(3), float64(4), int64(2), object(7)
memory usage: 58.8+ KB
None
```

Figure 2.1: Train Dataset Details with Data Types

If it is observed further, it can be deduced that the task is an ***imbalanced, binary classification problem***.

```
# summarize the class distribution
target = dataframe.values[:,-1]
counter = Counter(target)
for k,v in counter.items():
    per = v / len(target) * 100
    print('Class=%s, Count=%d, Percentage=%.3f%%' % (k, v, per))
```

```
Class=Success, Count=248, Percentage=47.328%
Class=Failure, Count=276, Percentage=52.672%
```

Figure 2.2: Summary of the Feature Distribution of Target Column

2.2. Algorithm Definition

Since this is a classification problem, classification algorithms should be applied. There are many algorithms available out there to achieve this goal. Following are some of the algorithms that can be used in this project.

- Linear Classifiers: Logistic Regression, Naive Bayes Classifier
- Nearest Neighbor
- Support Vector Machines
- Decision Trees
- Boosted Trees
- Random Forest
- Neural Networks

As it is hard to select the most suitable algorithm for this problem, several algorithms were used to determine the algorithm to be used in making predictions. In the following sections the exact procedure will be discussed with the details.

3. Experimental Evaluation

3.1. Methodology

In this project the following tools were used to develop the solution.

- Python
- NumPy
- Scikit-learn

to preprocess raw data, train algorithms and make predictions on new data and

- Matplotlib
- Seaborn
- Weka

for data visualization.

Explore and Visualize the Dataset

First, the train dataset was explored and visualized. As Figure 3.1 depicts, there are sixteen columns in the given dataset. Weka Explorer was used to explore and visualize features in the dataset.

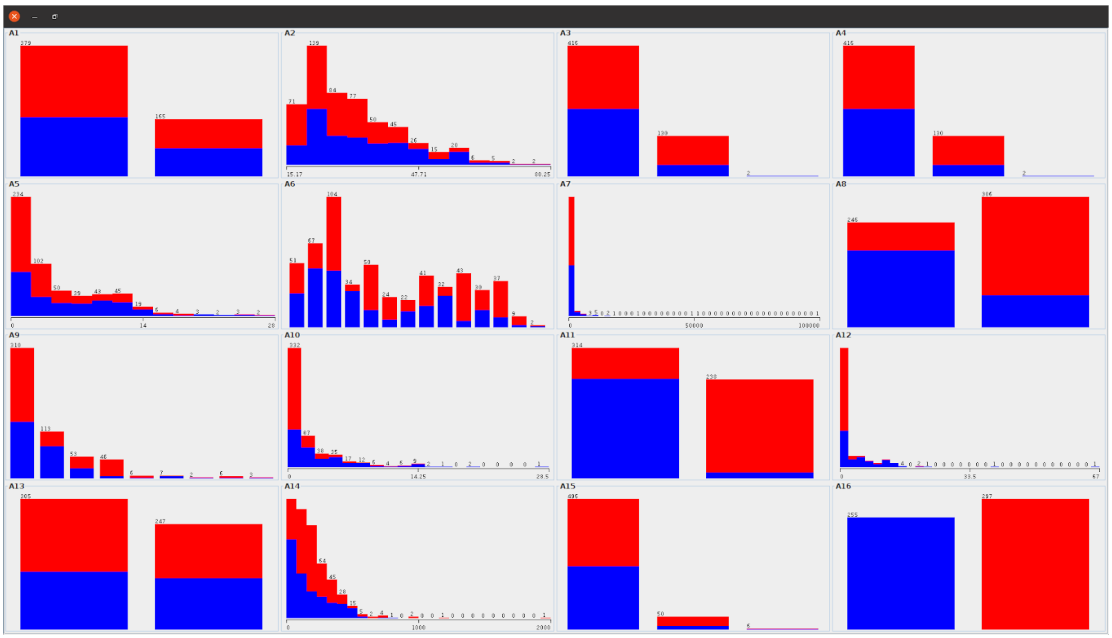


Figure 3.1: Overall Data Visualization using Weka

Selected attribute
Name: A1 **Type: Nominal**
Missing: 8 (1%) **Distinct: 2** **Unique: 0 (0%)**

No.	Label	Count
1	b	379
2	a	165

Selected attribute
Name: A2 **Type: Numeric**
Missing: 10 (2%) **Distinct: 314** **Unique: 174 (32%)**

Statistic	Value
Minimum	15.17
Maximum	80.25
Mean	31.977
StdDev	12.212

Selected attribute
Name: A3 **Type: Nominal**
Missing: 4 (1%) **Distinct: 3** **Unique: 0 (0%)**

No.	Label	Count
1	u	416
2	y	130
3	l	2

Selected attribute

Name: A4 | **Missing: 4 (1%)** | **Distinct: 3** | **Type: Nominal** | **Unique: 0 (0%)**

No.	Label	Count
1	g	416
2	p	130
3	gg	2

Selected attribute

Name: A5 | **Missing: 0 (0%)** | **Distinct: 188** | **Type: Numeric** | **Unique: 93 (17%)**

Statistic	Value
Minimum	0
Maximum	28
Mean	4.884
StdDev	5.087

Selected attribute

Name: A6 | **Missing: 6 (1%)** | **Distinct: 14** | **Type: Nominal** | **Unique: 0 (0%)**

No.	Label	Count
1	w	51
2	q	67
3	c	104
4	x	34
5	i	50
6	d	24
7	e	22
8	aa	41
9	cc	32
10	ff	43
11	m	30
12	k	37
13	j	9
14	r	2

Selected attribute

Name: A7 | **Missing: 0 (0%)** | **Distinct: 206** | **Type: Numeric** | **Unique: 163 (30%)**

Statistic	Value
Minimum	0
Maximum	100000
Mean	1100.828
StdDev	5628.306

Selected attribute

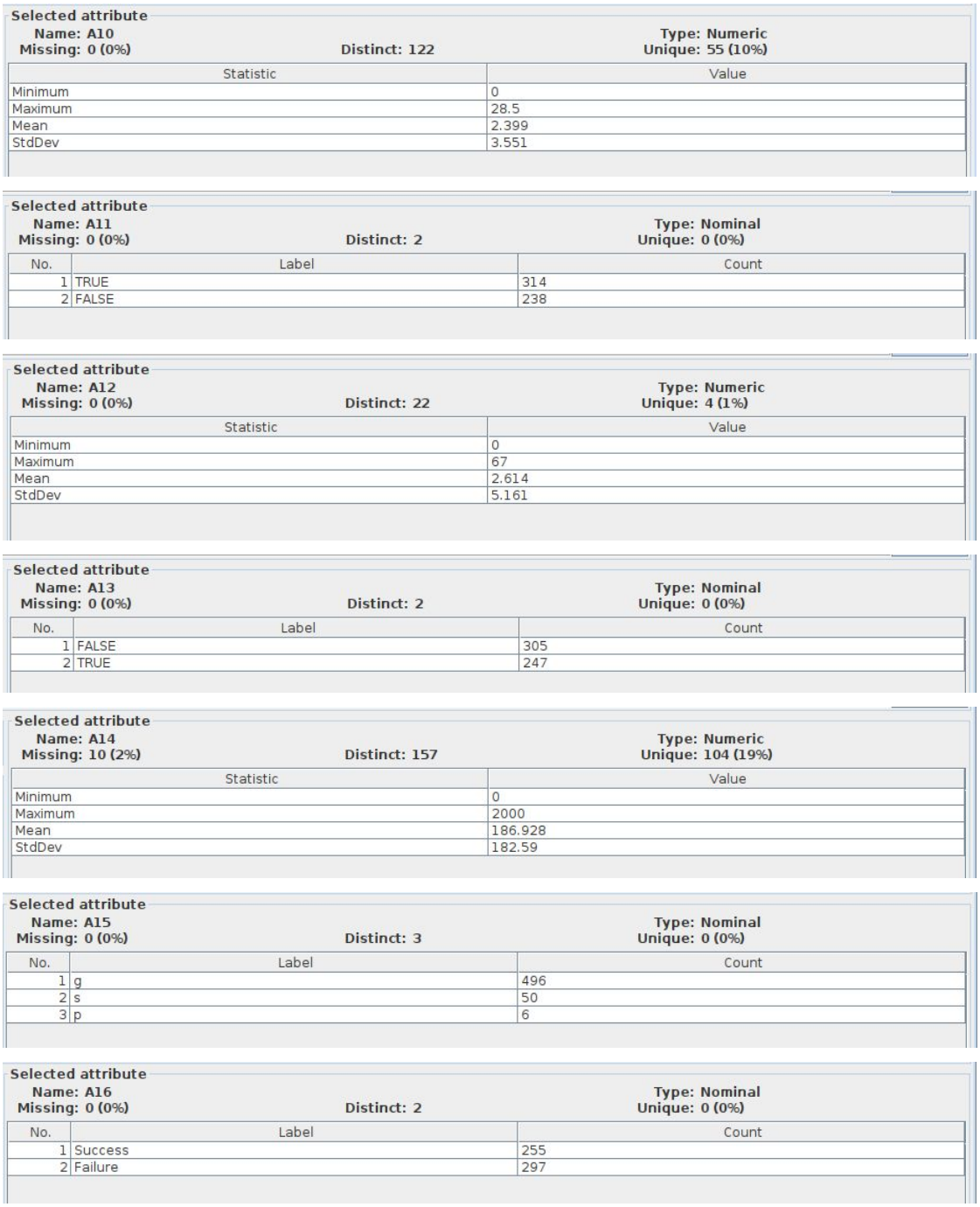
Name: A8 | **Missing: 0 (0%)** | **Distinct: 2** | **Type: Nominal** | **Unique: 0 (0%)**

No.	Label	Count
1	TRUE	246
2	FALSE	306

Selected attribute

Name: A9 | **Missing: 6 (1%)** | **Distinct: 9** | **Type: Nominal** | **Unique: 0 (0%)**

No.	Label	Count
1	v	310
2	h	113
3	bb	53
4	ff	46
5	j	6
6	z	7
7	o	2
8	dd	6
9	n	3



Selected attribute
Name: A10 **Type: Numeric**
Missing: 0 (0%) **Distinct: 122** **Unique: 55 (10%)**

Statistic	Value
Minimum	0
Maximum	28.5
Mean	2.399
StdDev	3.551

Selected attribute
Name: A11 **Type: Nominal**
Missing: 0 (0%) **Distinct: 2** **Unique: 0 (0%)**

No.	Label	Count
1	TRUE	314
2	FALSE	238

Selected attribute
Name: A12 **Type: Numeric**
Missing: 0 (0%) **Distinct: 22** **Unique: 4 (1%)**

Statistic	Value
Minimum	0
Maximum	67
Mean	2.614
StdDev	5.161

Selected attribute
Name: A13 **Type: Nominal**
Missing: 0 (0%) **Distinct: 2** **Unique: 0 (0%)**

No.	Label	Count
1	FALSE	305
2	TRUE	247

Selected attribute
Name: A14 **Type: Numeric**
Missing: 10 (2%) **Distinct: 157** **Unique: 104 (19%)**

Statistic	Value
Minimum	0
Maximum	2000
Mean	186.928
StdDev	182.59

Selected attribute
Name: A15 **Type: Nominal**
Missing: 0 (0%) **Distinct: 3** **Unique: 0 (0%)**

No.	Label	Count
1	g	496
2	s	50
3	p	6

Selected attribute
Name: A16 **Type: Nominal**
Missing: 0 (0%) **Distinct: 2** **Unique: 0 (0%)**

No.	Label	Count
1	Success	255
2	Failure	297

Figure 3.2: Detailed Views of All Attributes

As above figures represent, there are some missing values in the dataset. And also columns that contain numerical data have values in different ranges. Therefore it can be concluded that data is heterogeneous and distributed in a wide range.

Next, data preprocessing procedure will be discussed.

Data Preprocessing

Handling Missing Values

This dataset has 552 records(rows) and comparatively the number of records in the dataset is smaller. Therefore dropping rows which have missing values might lead the machine learning model to erroneous results. Hence filling missing values with appropriate methods can always be helpful.

```
# summarize missing value count for each column
dataframe.isnull().sum()
```

```
A1      8
A2     10
A3      4
A4      4
A5      0
A6      6
A7      0
A8      0
A9      6
A10     0
A11     0
A12     0
A13     0
A14    10
A15     0
A16     0
dtype: int64
```

Figure 3.3: Missing Data Count for each Column

```
# summarize missing value count for each row
is_NaN = df.isnull()
row_has_NaN = is_NaN.any(axis=1)
rows_with_NaN = df[row_has_NaN]
print('Total number of missing rows: %d' % len(rows_with_NaN))
```

```
Total number of missing rows: 28
```

Figure 3.4: Number of Rows Contain Missing Values

As Figure 3.4 shows, there are 28 rows that contain missing data.

Firstly dropping those 28 rows method was attempted and the resulting outcome was around 86% of mean accuracy score.

Then correlation matrix was calculated and visualized to get a better view of relationship among features.

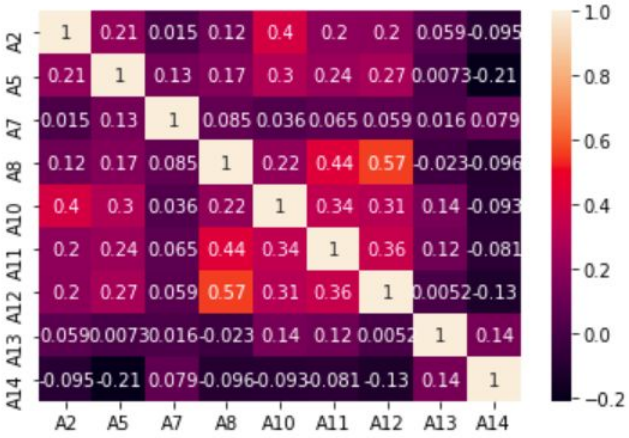


Figure 3.5: Visualized Correlation Matrix

Figure 3.5 shows that there is no strong correlation between any column in the given dataset. Therefore an algorithm like *linear regression* would not be useful to fill missing data.

Then the next step is to fill missing values using the imputation method. Scikit-learn library provides two main methods to impute missing data.

- Simple Imputer
- Iterative Imputer

Although the iterative imputer is said to be more accurate than the simple imputer, it caused difficulties and errors as the dataset contains categorical data. Therefore, to impute missing values Simple Imputer was used.

```
from sklearn.impute import SimpleImputer

# fill missing values with mean column values
imp_mean = SimpleImputer(missing_values=np.nan, strategy='mean')

values = X[num_ix].values
X[num_ix] = imp_mean.fit_transform(values)
```

```
# fill categorial missing values with most frequent
imp_freq = SimpleImputer(missing_values=np.nan, strategy='most_frequent')

values = X[cat_ix].values
X[cat_ix] = imp_freq.fit_transform(values)
```

Figure 3.6: Fill Missing Values using Simple Imputer

As the figure shows, numerical values were imputed using strategy, *mean* and the categorical values using strategy, *most frequent.*

After imputing missing data, the next procedure was to label encode and normalize data.

Label Encoding and Normalization

As discussed earlier, there are categorical features in the dataset. Some machine learning algorithms do not work well with this kind of data. Therefore, before going to run any model, those categorical text data should be converted into numerical data. For that conversion, Label encoder class which is provided by scikit-learn library can be used. The problem when using label encoder is that some of the machine learning models may tend to treat nominal values as ordinal values even though there is no ordered relation between categorical values. This is when *One Hot Encoding (OHE)* method comes to the picture.

Although one hot encoding method is considered to be more accurate on nominal data encoding, for this project accuracy of the predicted result was reduced when OHE was used. As we do not know what really the values represent in the dataset, it can be assumed that categorical data is ordinal rather than nominal because the accuracy was higher when label encoder was used.

As the next step of the data-preprocessing stage, all numerical values were normalized using the Min-Max Scaler provided by scikit-learn.

```
from sklearn.preprocessing import MinMaxScaler, LabelEncoder
# creating instance of labelencoder
labelencoder = LabelEncoder()

# label encode the target variable to have the classes 0 and 1
y_train = labelencoder.fit_transform(y)

for i in range (len(cat_ix)):
    # Assigning numerical values for categorial values
    X[cat_ix[i]] = LabelEncoder().fit_transform(X[cat_ix[i]])

min_max_scaler = MinMaxScaler()
X[num_ix] = min_max_scaler.fit_transform(X[num_ix])
X_train = X
```

	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15
0	1	0.240627	1	0	0.000000	12	0.00000	1	7	0.043860	1	0.014925	0	0.1010	0
1	0	0.668408	1	0	0.159286	10	0.00560	1	3	0.106667	1	0.089552	0	0.0215	0
2	0	0.143362	1	0	0.017857	10	0.00824	0	3	0.052632	1	0.000000	0	0.1400	0
3	1	0.194530	1	0	0.055000	12	0.00003	1	7	0.131579	1	0.074627	1	0.0500	0
4	1	0.151045	1	0	0.401786	1	0.01208	1	7	0.087719	1	0.253731	0	0.1000	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
547	1	0.368777	1	0	0.058036	1	0.04700	1	7	0.052632	1	0.149254	0	0.0930	0
548	1	0.367394	1	0	0.214286	9	0.01097	1	7	0.045263	1	0.074627	1	0.0540	0
549	1	0.253534	1	0	0.029643	13	0.03290	1	7	0.046842	1	0.119403	1	0.1515	0
550	1	0.396896	1	0	0.001429	4	0.00000	1	7	0.001404	0	0.014925	0	0.2800	2
551	1	0.512139	1	0	0.151786	9	0.00000	0	7	0.004386	1	0.000000	1	0.1125	0

552 rows × 15 columns

Figure 3.7: Train Dataset after Preprocessing

Feature Selection

Feature selection is an important step in data preprocessing. This can be useful to reduce the complexity of the machine learning model and to enable it to train faster and reduce overfitting problems.

Feature selection was done using Weka and scikit-learn attempting different methods to get the highest possible outcome. But, in fact, the feature selection step did not improve the accuracy of the model and caused the accuracy to decrease. Therefore, in this project all features were used to train the final model. Following figure shows the feature evaluation part using scikit-learn. SelectKBest, f_classif and recursive feature elimination (RFE) methods were used.

```
from numpy import set_printoptions
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif

# feature extraction
test = SelectKBest(score_func=f_classif, k=4)
fit = test.fit(X, y)
# summarize scores
set_printoptions(precision=3)
print(fit.scores_)
features = fit.transform(X)
# summarize selected features
print(features[0:5,:])

from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression

# feature extraction
model = LogisticRegression(solver='lbfgs')
rfe = RFE(model, 3)
fit = rfe.fit(X, y)
print("Num Features: %d" % fit.n_features_)
print("Selected Features: %s" % fit.support_)
print("Feature Ranking: %s" % fit.ranking_)
```

```
[5.074e-01 1.627e+01 2.207e+01 1.961e+01 2.545e+01 1.060e+01 1.438e+01
 1.553e+02 2.076e-01 6.478e+01 5.274e+02 1.012e+02 1.671e+00 4.399e+00
 7.328e+00]
[[1.    0.044 1.    0.015]
 [1.    0.107 1.    0.09 ]
 [0.    0.053 1.    0.   ]
 [1.    0.132 1.    0.075]
 [1.    0.088 1.    0.254]]
Num Features: 3
Selected Features: [False False  True False False False False False False  True  True False
 False False False]
Feature Ranking: [12  8  1 13  7  6  3  2 10  1  1  4  9  5 11]
```

Figure 3.8: Feature Selection Scores

Evaluate Models

Next the candidate models were evaluated using repeated stratified k-fold cross-validation. The k-fold cross-validation procedure provides a good general estimate of model performance that is not too optimistically biased.

```
>CART 0.797 (0.058)
>SVM 0.842 (0.041)
>BAG 0.862 (0.053)
>RF 0.882 (0.042)
>GBM 0.857 (0.041)
```

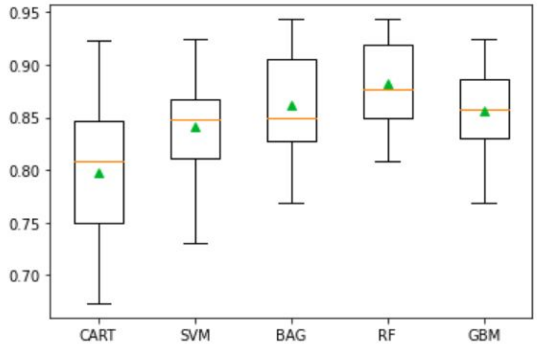


Figure 3.8: Results of the Evaluated Models

Figure 3.8 depicts the mean accuracy score of the models and the standard deviation of the scores.

Following models were evaluated in the process.

- Decision Tree (CART)
- Support Vector Machine (SVM)
- Bagged Decision Trees (BAG)
- Random Forest (RF)
- Gradient Boosting Machine (GBM)

As the figure shows, highest mean accuracy is achieved with Random Forest Model (88.2%) with a standard deviation of 0.042.

Therefore the *Random Forest model* was used as the final model to make predictions. In the final model parameters of the Random Forest model were set as follows.

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

Figure 3.9: Final Model with its Parameters

The final model was created with 100 decision trees in the forest and bootstrapped. Almost all the parameters were set to default values since they gave the best accuracy as scikit-learn library set its default values to be optimum.

"A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting." - scikit-learn.org

Therefore Random Forest classifier can be considered as the most suitable candidate for our machine learning model as it was proven to be the best out of other classification algorithms.

3.2. Results

Applying the previously mentioned methods, we were able to achieve an accuracy of 88.405% with the new test dataset.

4. Future Work

This project could be further developed to increase the accuracy of the results.

Since the given train dataset is not considerably large, instead of dropping the rows with the missing values we could impute data for the missing values using custom functions written for each and every attribute considering their relations.

5. Conclusion

In conclusion, in this imbalanced classification project, a given dataset was analyzed and a binary outcome was predicted. Imputing missing values with simple imputer, label encoding, normalizing data and finally random forest classifier was the approach that was used to address the problem. Using those methods, an accuracy of 88.405% was achieved at the end.

6. Bibliography

- *Scikit-learn.org. 2020. Scikit-Learn: Machine Learning In Python — Scikit-Learn 0.16.1 Documentation. [online] Available at: <https://scikit-learn.org/>.*
- *Géron, A., 2007. Hands-On Machine Learning With Scikit-Learn, Keras, And Tensorflow, 2nd Edition. 2nd ed. O'Reilly Media, Inc.*
- *Beaulieu, K. and Dalisay, D., 2020. Machine Learning Mastery. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/>.*