Miguel GUZMÁN
June 12th, 2023

Data Science Challenge: Binary Classification

Document Overview:

The intention of this document is to provide a brief summary of the intuition behind the solution for the Binary Classification challenge. It is divided into sections that resemble the ones proposed by the Jupyter Notebook referenced in the Appendix. For a more robust data analysis, visualizations and step-by-step debrief, please refer to the Jupyter Notebook annotations.

1. Environment Setup

The experimentation is done in a Jupyter notebook automatized to load the raw data from a public GitHub repository. Consequently, in order to reproduce the experiments discussed in further sections it is not necessary to manually load the csv files into the notebook's path as long as it is run in a cloud computing service such as Google Colab (recommended way) or in a local computer with GitHub credentials and connection to the internet. Manual load is still a viable option.

2. Dataset Description

The raw data is split into 2 csv files named *Task1_1* and *Task1_2*. Both of them contain an 'ID' column that can be used to link the instances. *Task1_2* dataset contains the dependent variable *y* under the column name 'Type'. Excluding the ID column and the dependent variable, both datasets give a sum of 7 numerical features and 10 categorical features.

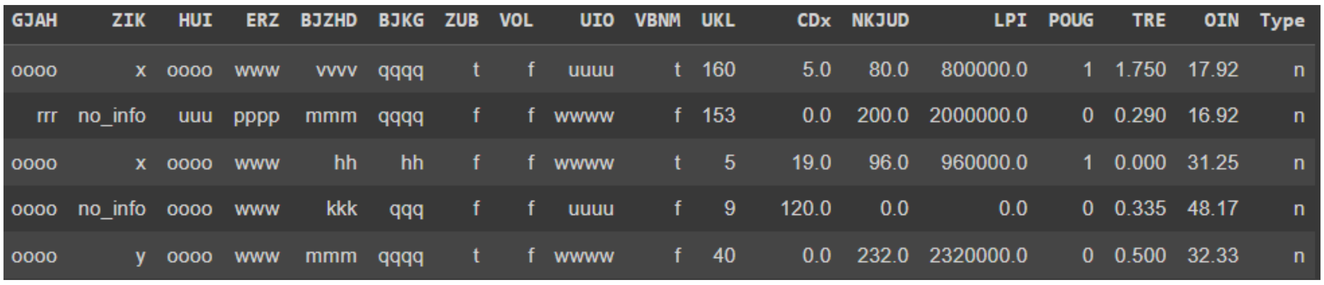
3. Data Preprocessing

3.1 Duplicated Instances:

The number of duplicated rows in DataFrame 1 is 370 out of 4070 rows, whereas the number of duplicated rows in DataFrame 2 is also 370 out of 4070 rows. All duplicated instances were removed keeping always the first appearance of an instance with duplicates.

3.2 Merge Datasets

Both raw datasets (having been excluded their respective duplicated rows) contain the same number of instances and moreover, the total count of unique values of the ID column equals the total number of rows for both datasets. Given this, a table inner join is performed to merge both datasets confidently without excluding any instance of data.



GJAH	ZIK	HUI	ERZ	BJZHD	BJKG	ZUB	VOL	UIO	VBNM	UKL	CDx	NKJUD	LPI	POUG	TRE	OIN	Type
oooo	x	oooo	www	vvvv	qqqq	t	f	uuuu	t	160	5.0	80.0	800000.0	1	1.750	17.92	n
rrr	no_info	uuu	pppp	mmm	qqqq	f	f	wwww	f	153	0.0	200.0	2000000.0	0	0.290	16.92	n
oooo	x	oooo	www	hh	hh	f	f	wwww	t	5	19.0	96.0	960000.0	1	0.000	31.25	n
oooo	no_info	oooo	www	kkk	qqq	f	f	uuuu	f	9	120.0	0.0	0.0	0	0.335	48.17	n
oooo	y	oooo	www	mmm	qqqq	t	f	wwww	f	40	0.0	232.0	2320000.0	0	0.500	32.33	n

Figure 1. Merged and Ordered Dataframe before feature transformation phase.

Given the merged dataset, it is important to point out that the dependent variable labels are highly imbalanced, so that will be taken into consideration in the next sections.

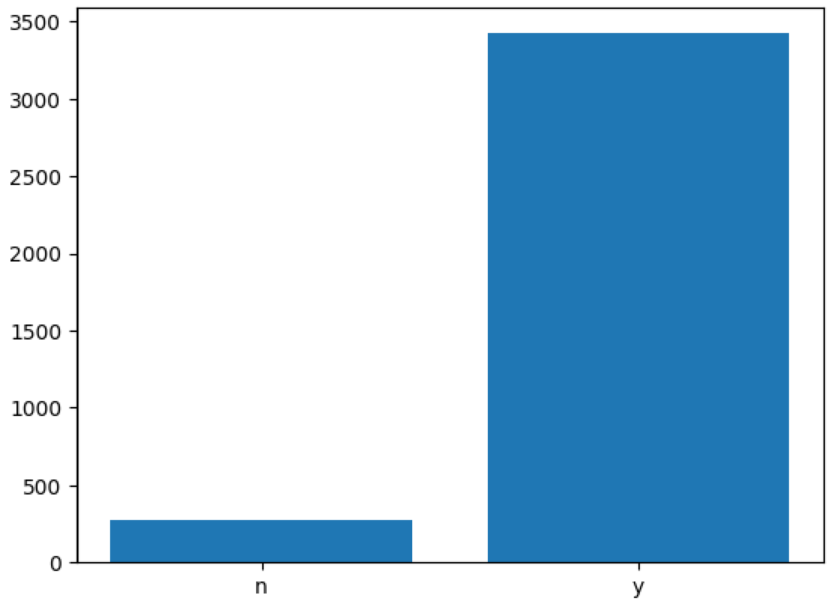


Figure 2. Label distribution of the dependent variable *y* (Column = ‘Type’). The distribution is quite imbalanced.

3.3 Split for Cross-validation

The Train-Test split follows a stratified approach. A stratified train-test split is a technique used in machine learning to split a dataset into training and testing sets while maintaining the same class distribution or proportion of classes in both sets. This is particularly useful when dealing with **imbalanced datasets**, which is the case for this challenge.

The stratified train-test split ensures that each class is represented in both the training and testing sets in roughly the same proportion as the original dataset. This helps to ensure that the model is trained and evaluated on representative samples from each class, which can lead to more reliable performance evaluation.

3.4 Missing Values:

The way of handling missing values depends on the type of variable (numerical / categorical) and the percentage of missing values per feature.

Most of the features containing missing values have them in a relatively low percentage (not exceeding 2.71%), so these missing values are imputed. However, there is a problematic exception for the ZIK feature that has around 58% of missing values. This is a HUGE percentage that requires a more specific approach. For this, all the missing values are labeled with a 'no-info' tag so there is an annotation to know when the value is missing that will be automatically One-hot encoded in the encoding section.

For the imputed features, the strategies are the following:

- **Numerical features**: Imputation based on the **mean** value of the training set.
- **Categorical features**: Imputation based on the **mode** value of the training set.

3.5 Encode Categorical Values:

Two different encoding approaches are applied to the categorical variables depending on their type:

- **Binary Categorical Variables**: Label Encoding $\in\{0,1\}$
- **Non-binary Categorical Variables**: One-Hot Encoding

3.6 Feature Scaling:

Standardization is chosen as a feature scaling method for all numerical variables.

$$x_{\text{stand}} = \frac{x - \text{mean}(x)}{\text{standard deviation } (x)}$$

Feature scaling is important for the next reasons:

Avoiding Bias: Many machine learning algorithms are sensitive to the scale of the features. When features have different scales, algorithms tend to give more weight to features with larger values, potentially leading to a biased model. By scaling the features, you ensure that each feature contributes equally to the learning process and prevent any undue influence based on scale.

Improved Numerical Stability: Feature scaling can improve the numerical stability of computations in machine learning. Large differences in feature scales can lead to numerical instabilities, such as overflow or underflow issues, during calculations. Scaling the features helps avoid such problems and ensures more stable computations.

Effective Distance Measures: Many machine learning algorithms rely on distance measures, such as Euclidean distance or cosine similarity, to make decisions. If the features are not scaled, features with larger scales can dominate the distance calculations, leading to biased results. Scaling the features ensures that the distance measures are based on the actual patterns and relationships in the data.

4. Classification Modeling

For creating and testing binary classification models both single predictor and ensemble method approaches are evaluated. Ensemble methods are expected to be better performant normally so they are used as a point of comparison for experimentation analysis.

The performance metrics are accuracy, precision, recall and F1-score. It is important to mention that since the predicted label distribution is quite imbalanced, precision should not be trusted as a significant performance metric.

4.1 Single Predictors

Experimentation is done with Decision Tree and SVM classifiers. These predictors were chosen since they have specifications that can help when dealing with an imbalanced predicted class.

- In **SVM** (Support Vector Machine), you can adjust the cost parameter (C) to penalize misclassifications of the minority class more.
- In **Decision Trees** you can set the `class_weight` parameter to give more weight to the minority class.

The best performant model metrics for the SVM classifier are shown in Table 1 and Figure 3, whereas the best performant model metrics for the Decision Tree classifier are shown in Table 2 and Figure 4.

	Precision	Recall	F1-Score	Support
Label: ‘n’	0.982	0.982	0.982	55
Label: ‘y’	0.999	0.999	0.999	685
Accuracy		0.997		740

Table 1. Metric scores for SVM classifier with the best performance. (Kernel=RBF, c=10, ɣ=0.1)

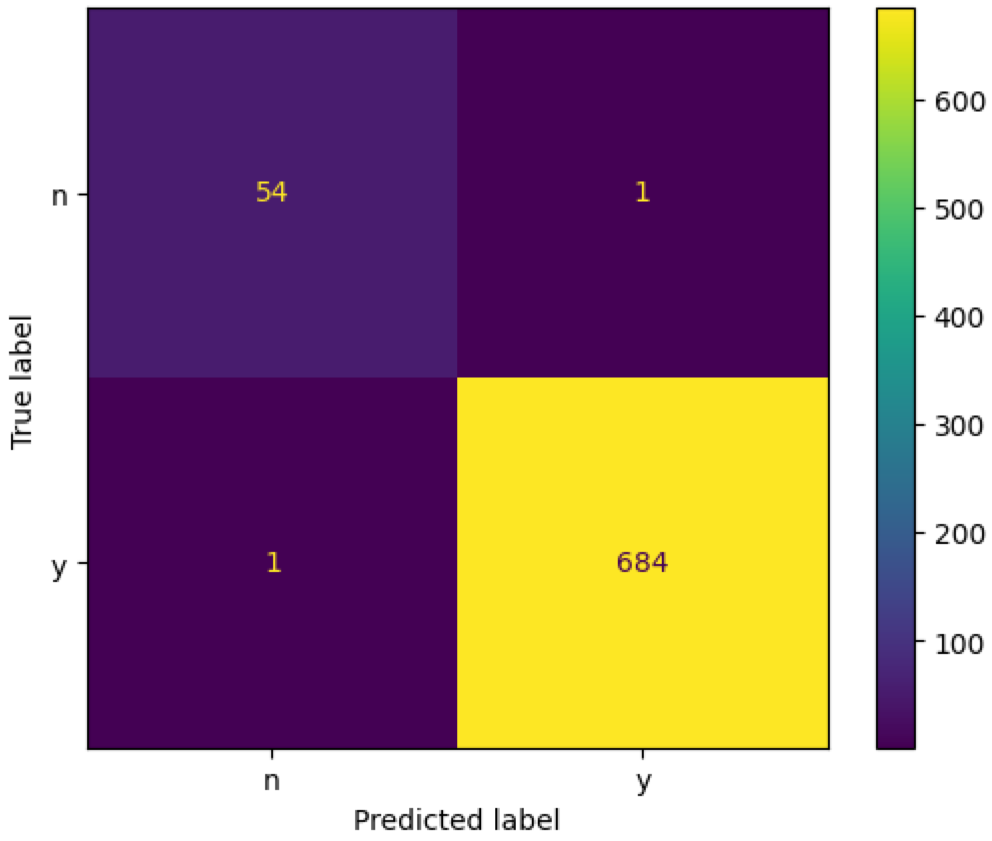


Figure 3. Confusion Matrix for SVM classifier with the best performance. (Kernel=RBF, c=10, ɣ=0.1)

	Precision	Recall	F1-Score	Support
Label: ‘n’	0.958	0.836	0.893	55
Label: ‘y’	0.987	0.997	0.992	685
Accuracy		0.985		740

Table 2. Metric scores for Decision Tree classifier with the best performance. (class-weight=balanced, criterion=gini)

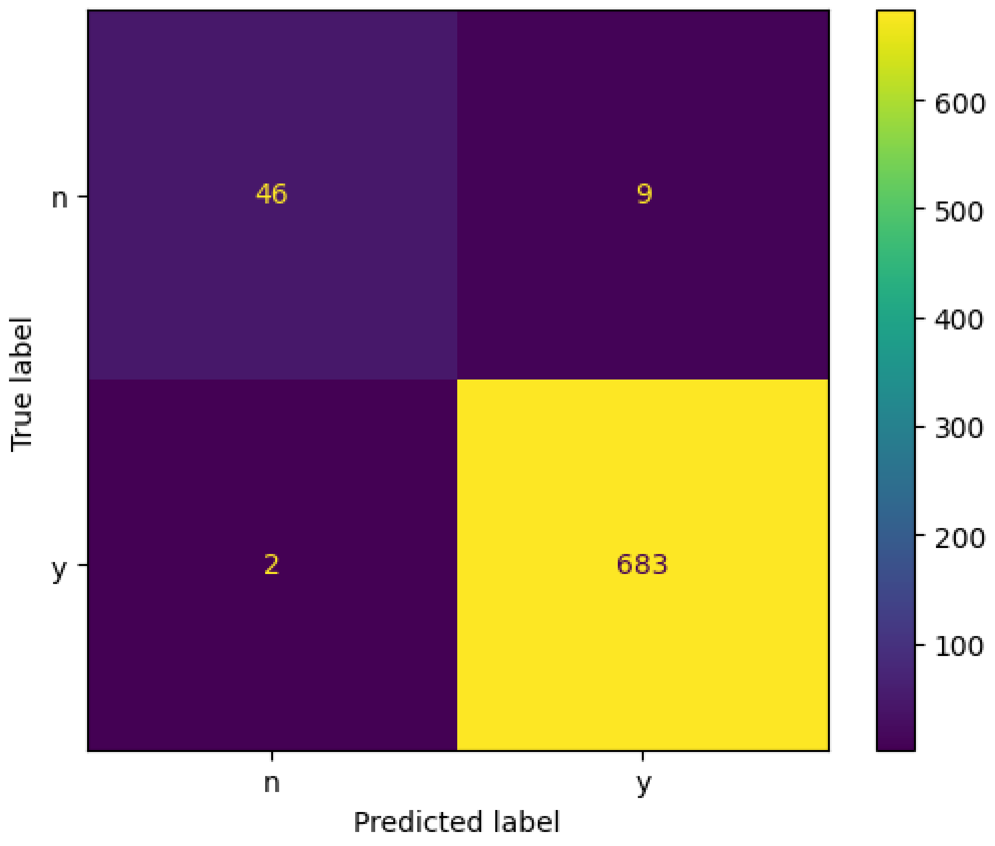


Figure 4. Confusion Matrix for Decision Tree classifier with the best performance.
(class-weight=balanced, criterion=gini)

SVM classifier clearly outperforms the Decision Tree classifier. The decision Tree classifier has a notable gap in terms of the recall score for label ‘n’.

4.2 Ensemble Methods

Since the performance of the hyperparameter-tuned SVM classifier performed excellent, we focused this section in experimentation aiming to improve the tree-based classifier performance given two powerful tree-based ensemble methods, Random Forest and XGBoost.

These two ensemble models have also built-in options to handle class imbalance:

- In **Random Forest** you can set the `class_weight` parameter to give more weight to the minority class.
- In **XGBoost** you can set the `scale_pos_weight` parameter to also give more weight to the minority class.

The best performant model metrics for the Random Forest classifier are shown in Table 3 and Figure 5, whereas the best performant model metrics for the XGBoost classifier are shown in Table 4 and Figure 6.

	Precision	Recall	F1-Score	Support
Label: ‘n’	0.979	0.836	0.902	55
Label: ‘y’	0.987	0.999	0.993	685
Accuracy	0.986			740

Table 3. Metric scores for Random Forest classifier with the best performance. (class-weight=None, estimators=90, criterion=entropy)

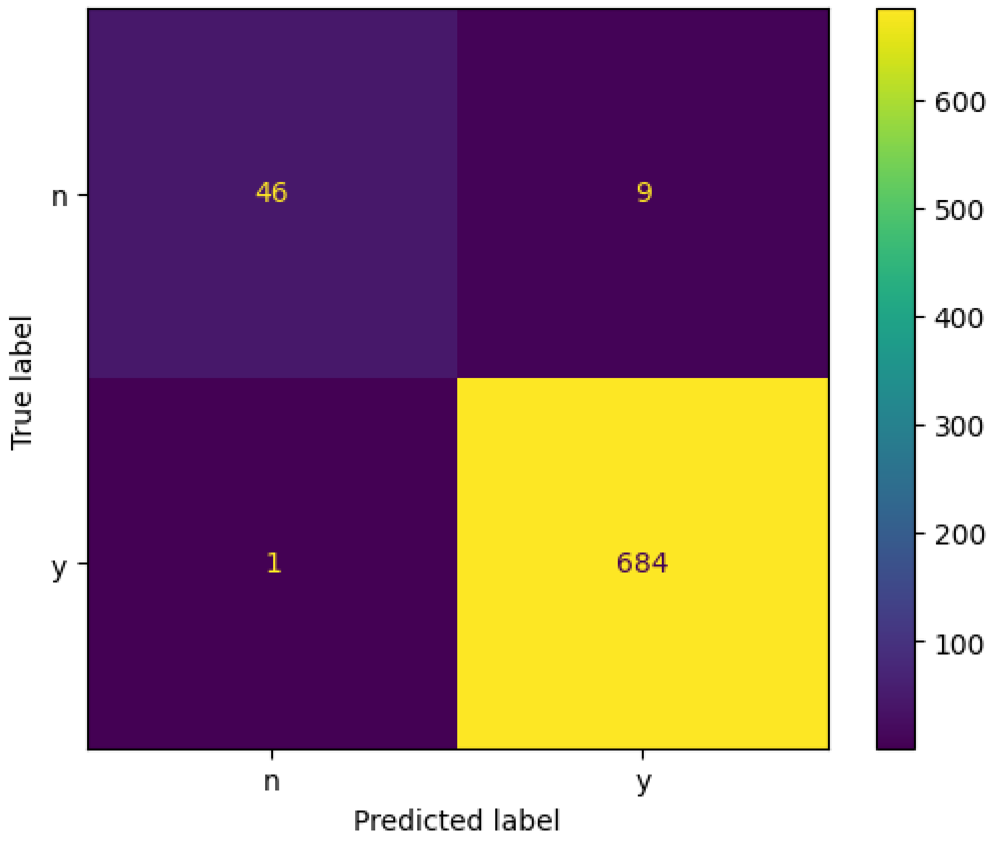


Figure 5. Confusion Matrix for Random Forest classifier with the best performance. (class-weight=None, estimators=90, criterion=entropy)

	Precision	Recall	F1-Score	Support
Label: ‘n’	0.979	0.836	0.902	55
Label: ‘y’	0.987	0.999	0.993	685
Accuracy	0.986			740

Table 4. Metric scores for XGBoost classifier with the best performance. (max-depth=10. estimators=50, scale_pos_weight= # negative instances / # positive instances)

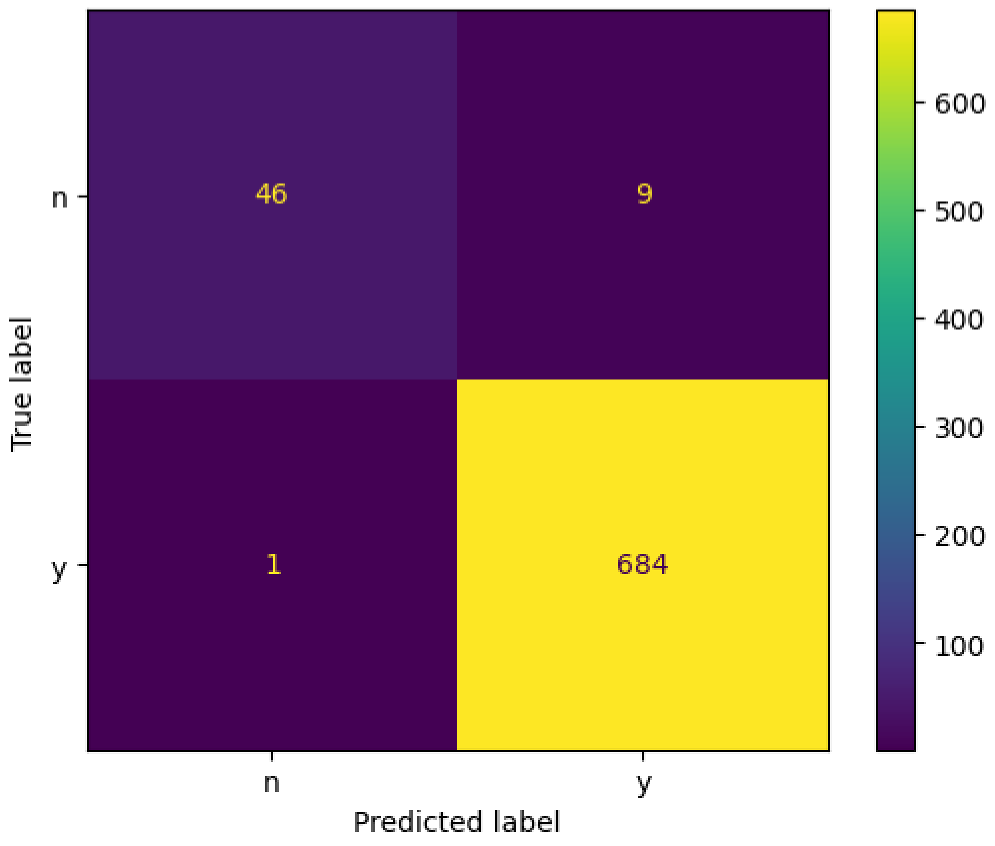


Figure 6. Confusion Matrix for XGBoost classifier with the best performance. (max-depth=10. estimators=50, scale_pos_weight= # negative instances / # positive instances)

Both ensemble methods have similar outputs and improve the single decision tree performance to a minimum degree but are still far from the excellent metrics obtained by the SVM classifier.

5. Conclusion

A simple SVM classifier can be extremely well performant and a fast computing solution for a binary classification task with imbalanced classes when chosen the right kernel followed by a proper tuning of c and γ to handle the bias-variance tradeoff. Decision Trees are also decent at handling the class imbalance with weight-balancing. However, even though ensemble methods such as Random Forest and XGBoost were applied to the base tree predictor they did not seem to significantly increase the overall tree-based performance and were still far from the best SVM metrics.

Further work may include more advanced imbalanced classes handling techniques, such as undersampling, oversampling (SMOTE) or generation of synthetic samples and also testing more models, such as logistic regression, ensemble methods with SVM predictors as base learners and neural network classifiers along with more robust hyperparameter tuning processes.

Appendix

The code and datasets to reproduce the experiments presented in this document can be found in the following link: https://github.com/mikeguzman1294/BinaryClassification

There, the notebook file named ‘BinaryClassification.ipynb’ implements all the steps of the data pipeline previously discussed.