Machine Learning

2023-2024

Rayan Abdeli, Digangana Das

Professor: Fabrice Rossi

January 28, 2024

Contents

1 Introduction **3**

2 Data Preprocessing **3**

3 Machine Learning Models **5**

3.1 Logistic Regression 6

3.2 Random Forest 7

3.3 eXtreme Gradient Boosting 8

4 Conclusion **10**

5 Replication Process **10**

1 Introduction

In this machine learning project, we had to build a predictive model from a dataset which included 99985 French persons. We were given a learning set with 49993 individuals and a test set with 49992 individuals. Our task was to create the best classification model for two possible target values (B and G). For this project, we used Python as our programming language.

As a result, we iterated on the learning set to create the best model, which led to a predicted accuracy of 91.3% with XGBoost. Our main performance indicator for this binary classification problem was the confusion matrix.

In this project, a lot of information was provided. Therefore, we had to be careful with the manipulation of CSVs and make sure that we did not make any mistake while merging them. Therefore, we created a GitHub repository[1] to retrieve the data from there instead of running everything on our computers. It was also more efficient to collaborate and track our progress.

Also, we did not decide to run the code on our computers because the code would have taken too long to run. Therefore, we optimized our time by using Google Collab and Kaggle. These are free and help us run our code online. Moreover, Google Collab gave us access to 12.7 GB of RAM and Kaggle to 30 GB of RAM, which were helpful for some computations.

In the following sections, we will detail all the steps which led to these results. First, we will explain how we managed to preprocess the data, then we will see how we built our best model.

2 Data Preprocessing

The preprocessing was divided into three steps: merging the data to have our final dataset, imputing some missing values and creating a column transformer for our pipeline.

To process the data, we merged all the datasets that were provided and started to keep all the columns which were relevant to our models. For example, when a numerical value was

[1] https://github.com/RayNCode/code_collab/tree/main

available instead of a string value, we kept the first one to be more efficient and to reduce the dimensionality of our dataset. For example, this was the case for the Department code. We also had to drop columns which were adding noise to our models (like the ID of each person). Once we did this, we had to start handling some missing values. For example, if someone did not have a full-time job, they did not have any information regarding their emolument, even when they were unemployed or retired. Therefore, we set some missing values to 0. Moreover, for categorical variables, we created a "None" category, when the information was missing. We did this to keep this missing information as information in itself, and to capture their non-response.

After, we had to make sure that all the columns were in the right format (by setting categorical variables as such, and not as numerical variables) and then we were ready to start the second stage of our preprocessing.

In the second stage, we changed the labels of our target variable to 1 and 0. In general, this is the best solution because classifiers are better when they handle numerical values. Also, we decided to use a specific method to impute the missing values of people who were employed but did not have any information on emoluments or working hours. For this purpose, we used an iterative imputer, which has a Random Forest Regressor by design. Because we had a lot of information about the workers who did not have missing values, our model predicted these missing values. Also, we did not want to find the exact values while using this iterative imputer, we simply wanted to approximate them. Therefore, we did not fine-tune this feature as much as our main model and kept it simple. By doing so, our models performed significantly better than those with mean or mode imputation techniques. However, this method was not easy to integrate into our workflow. We had some issues while integrating this step in the column transformer. It took too much processing power and it crashed our sessions multiple times. This is the reason why we use the iterative imputer outside of the column transformer while tuning our model as well as other small imputations (we needed to have maximum information so the regressor would perform better). This issue was however not encountered in the final prediction file and we were able to do it in the column transformer. Once we completed this step, we did not have any missing values and were ready to use the Column Transformer.

To process our categorical variables, we had to use One Hot Encoder for most of them. When they were sorted in a specific way (ascending order), we used the Ordinal Encoder.

The numerical variables did not need to be processed again. Sometimes, dimensionality was an issue, especially for models like the Logistic Regression, which performed better with just a few variables than the whole dataset merged (without specifying hyperparameters), but as we will see later more complex models were more promising than Logistic Regression. Also, we always split our dataset using 80% of the learning set to train our model (40,000 people) and 20% (10,000 people) to validate our results. We stratified this sample based on the target values due to class imbalance, with 61% representing 'B' and 39% representing 'G'.

Before tuning hyperparameters, we tried to analyze which model led to the best results. We only used a simple reference model with cross-validation. We compared the performance of several models and realised that ensemble models had significantly higher results than linear models. Therefore, we chose to use Logistic Regression, Random Forest - to stick with the guidelines of this project - and XGBoost, to see if another alternative was possible. According to the table hereunder, we already see that the fitting time (in seconds) of the Random Forest model is 24 times slower than XGBoost, while Logistic Regression and XGBoost take roughly the same time to fit our training data.

Model	**Fitting time**	**Scoring time**	**Accuracy**	**Precision**	**Recall**	**F1_score**	**AUC_ROC**
XGBoost	3.120254	0.335858	0.864930	0.859768	0.853224	0.864311	0.944298
Random Forest	73.720035	1.151724	0.858379	0.852202	0.847265	0.857881	0.935914
Gradient Boosting	12.608882	0.298119	0.829275	0.825040	0.809950	0.827246	0.911467
Adaboost	3.990189	0.539284	0.823849	0.818121	0.805488	0.822061	0.910438
Logistic Regression	8.954012	0.349314	0.800370	0.790873	0.783690	0.799124	0.881027

Table 1: Comparison of Model Performance

3 Machine Learning Models

We built 3 machine learning models to make predictions on the test dataset. We used the 5-fold cross-validation technique to check the robustness of every model. We used a randomised grid search technique for our models to tune the hyperparameters for the most efficient results since all our models have large hyperparameter spaces. Also, when we fit our data

3.1 Logistic Regression

The linear model used to make predictions on our dataset is logistic regression since we had a binary classification problem. The evaluation metrics of the model gave us an accuracy of 88.41% on the test dataset which already indicated very good prediction prowess for the logistic regression, with an 87/81% macro precision rate.

The grid of hyperparameters used to tune the logistic regression were -

- 'logreg__ penalty': ['l2'], ('l1' penalty is not compatible with 'newton-cg')
- 'logreg__ C': [0.001, 0.01, 0.1, 1, 10, 100],
- 'logreg__ solver': ['newton-cg'], (only 'newton-cg' is directly compatible with multinomial)
- 'logreg__ max_ iter': [100, 200, 300],
- 'logreg__ class_ weight': [None, 'balanced'],
- 'logreg__ fit_ intercept': [True, False],
- 'logreg__ intercept_ scaling': [1, 2, 5],
- 'logreg__ dual': [False], (dual should be False for 'newton-cg')
- 'logreg__ multi_ class': ['multinomial']

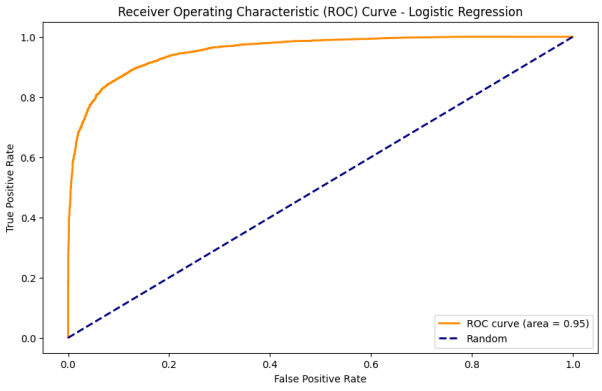


Figure 1: ROC Curve for the Logistic Regression Model

To quantify the model's ability to distinguish between the positive and negative classes across different probability thresholds, we plot the Receiver Operating Characteristic

Curve for the Logistic Regression Model (Figure 1). We see that the area under the ROC curve is 0.95, which indicates its excellent performance.

3.2 Random Forest

The next model that we built to compare with the performance of our logistic regression was a Random Forest. Interestingly, the accuracy of the Random Forest was slightly lower than the logistic regression at 86.22% and a macro precision level of 85.59%.

The grid of hyperparameters used to tune the random forest were -

- 'random_ forest__ n_ estimators': [50, 100, 200],
- 'random_forest__ criterion': ['gini', 'entropy'],
- 'random_forest__ max_ depth': [None, 10, 20, 30],
- 'random_forest__ min_samples_ split': [2, 5, 10],
- 'random_forest__ min_ samples_ leaf': [1, 2, 4],
- 'random_forest__ max_ features': ['auto', 'sqrt', 'log2', None],
- 'random_forest__ max_ leaf_ nodes': [None, 10, 20, 30],
- 'random_forest__ min_ impurity_ decrease': [0.0, 0.1, 0.2],
- 'random_forest__ bootstrap': [True], (we enabled bootstrap for our model and removed the false feature of the hyperparameter)
- 'random_ forest__ oob_ score': [True, False],
- 'random_ forest__ n_ jobs': [None, -1, 1, 2]

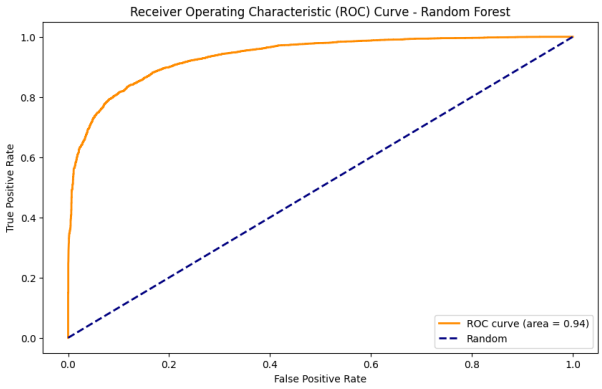


Figure 2: ROC Curve for the Random Forest Model

The area under the ROC Curve for the Random Forest is lower than the logistic regression model by 0.01 at 0.94. This follows the model's accuracy and precision scores. These statistics further confirm that our logistic regression model performs better.

3.3 eXtreme Gradient Boosting

We also built an eXtreme Gradient Boosting (henceforth XGBoost) and found that it was the best-performing model for our dataset with an accuracy of 91.29% and a macro precision score of 90.69%.

The grid of hyperparameters used to tune the XGBoost were -

- 'xgbclassifier__ subsample': 0.9000000000000001,
- 'xgbclassifier__ scale_ pos_ weight': 0.8300000000000001,
- 'xgbclassifier__ reg_ lambda': 0.08,
- 'xgbclassifier__ reg_ alpha': 0.8,
- 'xgbclassifier__ n_ estimators': 151,
- 'xgbclassifier__ min_ child_ weight': 3.279999999999994,
- 'xgbclassifier__ max_ depth': 5,
- 'xgbclassifier__ max_ delta_ step': 1.54,
- 'xgbclassifier__ learning_rate': 0.76,

- 'xgbclassifier__ gamma': 0.9980000000000002,
- 'xgbclassifier__ colsample_ bytree': 0.7100000000000001,
- 'xgbclassifier__ colsample_ bylevel': 0.9400000000000002

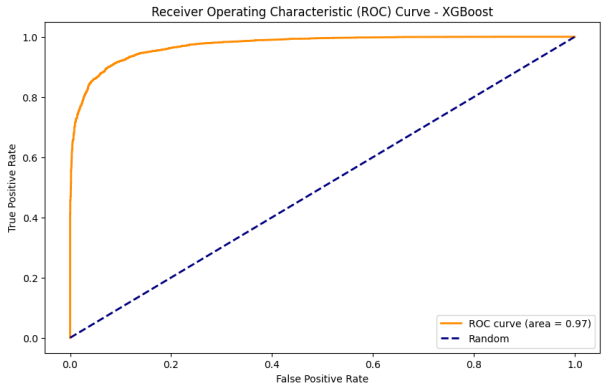
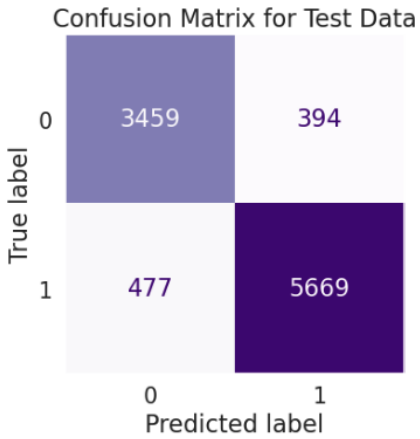


Figure 3: ROC Curve for the eXtreme Gradient Boosting Model

Figure 3 above plots the ROC Curve for the XGBoost model and indicates a high value of 0.97 under the ROC Curve further indicating its high True Positive Rate (Sensitivity) and a low False Positive Rate.

Figure 4 below shows our main results for XGBoost in the first stage of our work. These hyperparameters were used for the real test dataset.



XGBoost Performance Summary on Test Data

	XGBoost
Accuracy	91.29%
Macro Precision	90.69%
Macro Recall	91.01%
Macro F1-score	90.84%
Macro AUC	97.3%

Figure 4: Confusion Matrix and Performance Summary of the eXtreme Gradient Boosting Model

These results were obtained in 12 minutes by running the file "XGB_final_model.ipynb". However, it took a lot more time to have a great hyperparameter space to iterate on.

Moreover, by running our notebooks, it may be the case that you do not find the same hyperparameters, this is because we use Randomised GridSearch. It always outputs a different solution-.

4 Conclusion

As the various tests and predictive statistics above suggest, given our dataset, and the extra data to complement it to improve our predictions, the XGBoost model was the best fit. However, if we further wanted to improve performance, we could look at external data to better understand the status of those people who weren't engaged in any employed activity, i.e., they were either unemployed or retired, for our emolument variable. This aspect could be explored further.

5 Replication Process

Our code runs well in Google Collab, Jupyter Notebooks and Kaggle, but extra authorisation is required to run it on VSCode to retrieve the GitHub data. It must be noted that the predictions are not entirely reproducible every time we run our code, even though the XGBoost accuracy remains the same.

- 'BestModelWithoutGridSearch.ipynb': Comparison of models without doing any grid search (cross-validation only) and code to replicate Table 1.
- 'LogisticRegression.ipynb': Code to get the performance of a linear model.
- 'RandomForest_Complete.ipynb': Code to get the performance of a Random Forest model.
- 'XGB_final_model.ipynb': To see how we found our hyperparameters for our main model.
- 'XGB_Predictions.ipynb': Code to get the predictions.
- 'ROC_all.ipynb': Code to replicate the ROC Curves.