Hiring Classification Report

Martino Fusai

December 2023

Index

List of Tables

List of Figures

Disclaimer

Given the constraints imposed and the context of my studies, I used Chat GPT to untangle my code and save time on certain methods. However, it was only a tool to help me, and the thinking behind the code, as well as the majority of the code, was not generated by Chat GPT.

0 Introduction

In this report, I embarked on the task of predicting whether a candidate was hired or not. The journey began with a comprehensive exploration of the dataset, encompassing a thorough examination of data types, the detection of missing values, erroneous entries, and duplicates. Through data visualization, I gained insights into the underlying patterns and relationships within the dataset. Subsequently, I curated a strategic list of transformations that were applied to various columns, including one-hot encoding and mapping, to enhance the dataset's suitability for machine learning. Following the data preprocessing stage, I implemented the feature engineering techniques identified earlier, such as one-hot encoding, to further refine the dataset. Furthermore, I determined some key metrics to gauge the model's performance effectively. The report then delved into the experimentation phase, where I explored the capabilities of multiple machine learning models, including Random Forest, Extreme Gradient Boosting, and Support Vector Classifier with GridSearchCV for hyperparameter tuning. The goal was to identify the best-performing model that could accurately predict the hiring outcome of candidates based on the provided data.

1 Setup and Data Cleaning

First of all, I imported all the necessary packages and modules for my analysis. The dataset is centered around the prediction of candidate hiring outcomes, specifically with the `embauche` column as the target variable. It comprises 11 columns, each representing distinct attributes related to candidates and their applications (Table 1). Within this dataset, we can find information such as the application date (`date`), candidate age (`age`), highest educational diploma obtained starting from the bachelor (`diplome`), specialization of the diploma (`specialite`), salary expectations (`salaire`), immediate availability (`dispo`), gender (`sexe`), years of professional experience (`exp`), hair color (`cheveux`), grades received on an exercise (`note`), and the crucial hiring outcome (`embauche`), which is binary with `1` indicating successful hiring and `0` indicating unsuccessful hiring.

#	Column	Non-Null Count	Dtype
0	Unnamed: 0	20,000	int64
1	index	20,000	int64
2	date	19,909	object
3	cheveux	19,897	object
4	age	19,909	float64
5	exp	19,904	float64
6	salaire	19,905	float64
7	sexe	19,900	object
8	diplome	19,890	object
9	specialite	19,907	object
10	note	19,886	float64
11	dispo	19,894	object
12	embauche	20,000	int64

Table 1: `df.info()` output [4]

As we can see from the table above, many features have missing values. Therefore, I proceeded to drop all the rows with missing values, with the dataset resulting in having 19,109 non-null rows. A viable alternative option would have been to impute the missing values with a statistic of choice (for example, the median, or the mean in absence of outliers). Then, I also removed two columns from the dataframe, namely `Unnamed: 0` and `date`, to streamline the dataset, as they were redundant. Afterward, I set the `index` column as the new index for the dataframe, ensuring that it serves as the primary identifier for each row of data. This step helped me in maintaining data integrity and facilitates subsequent data operations. Finally, I checked for the presence of duplicate rows, meaning if there were candidates registered multiple times, but there were none.

2 Exploratory Data Analysis

In this phase, I conducted both univariate and bivariate data analyses to gain insights from the dataset.

2.1 Univariate Data Analysis

For univariate data analysis, I initially generated summary tables for both categorical and quantitative variables, and subsequently examined variable distributions using histograms (Figure 1).

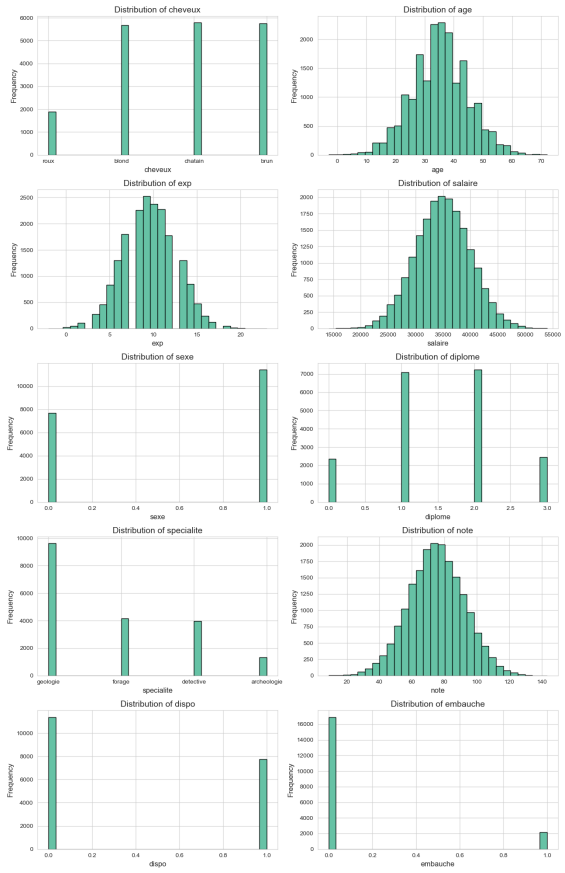

Figure 1: Variables distribution [16]

To delve deeper, I also created box plots for the numerical features (Figure 2). In particular, I observed that the target feature `"embauche"` exhibited significant imbalance, favoring class `0` (not hired). I dealt with this

class imbalance in the modeling section of my report. The other features appear to be well balanced and distributed, despite the presence of some outliers.

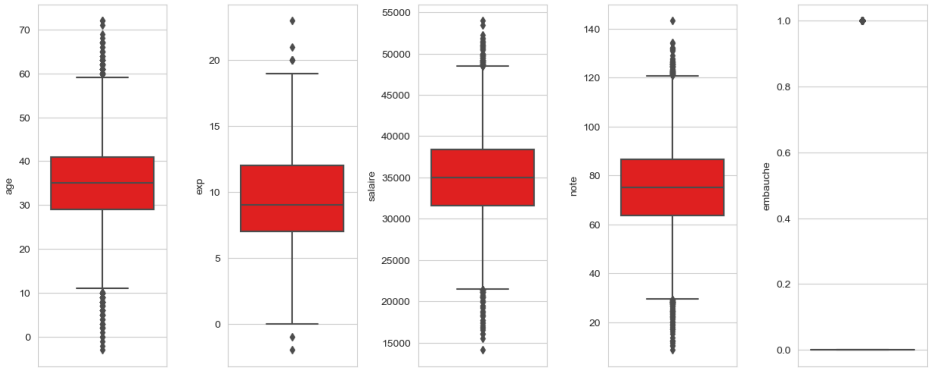

Figure 2: Quantitative variables boxplots [17]

To confirm the target variable's imbalance and visualize it in a better way, I proceeded to plot its distribution (Figure 3).

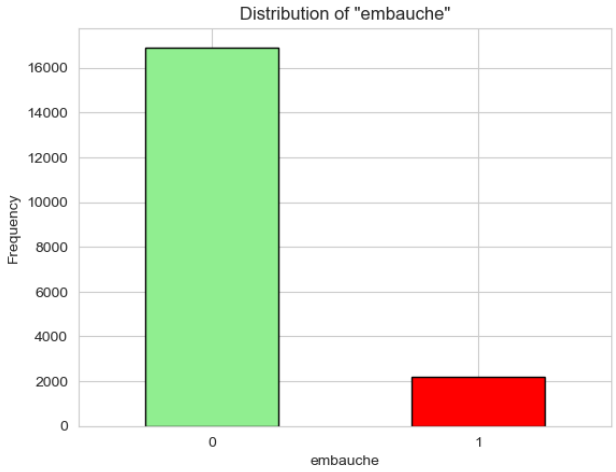

Figure 3: `"embauche"` distribution [18]

2.2 Bivariate Data Analysis

Subsequently, for bivariate data analysis, I constructed scatterplots between variables (Figure 4). This scatterplot matrix, also known as a pairs plot, is showing the relationship between the pairs of variables from our dataset.

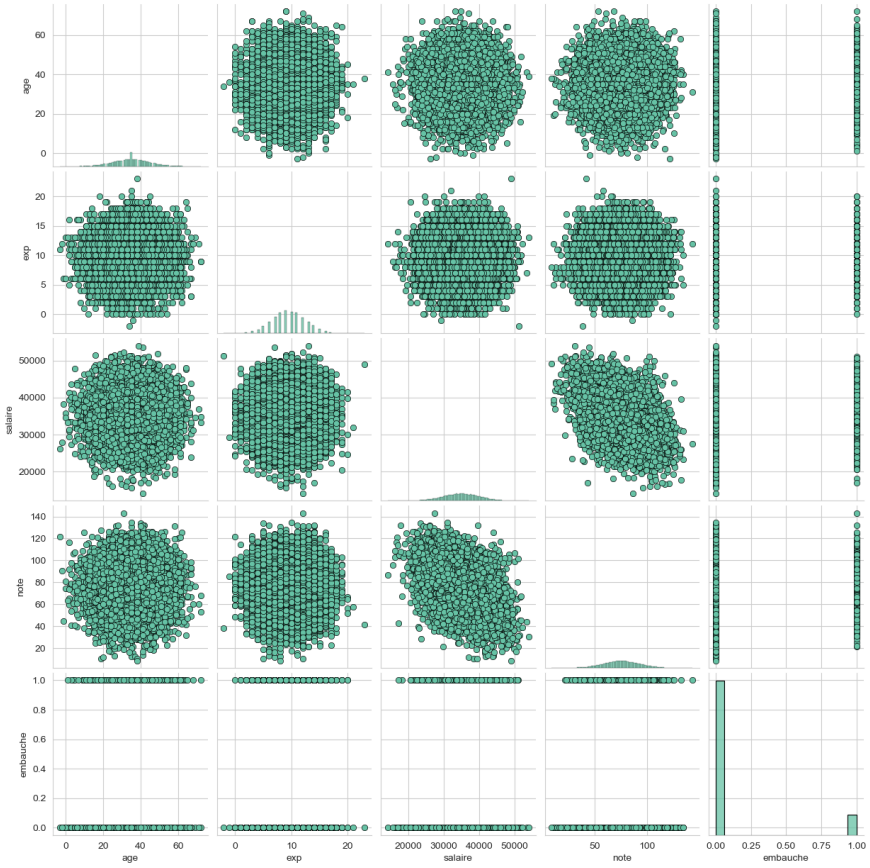

Figure 4: Scatterplot matrix [19]

Typically, we might expect salaries to increase with age as individuals advance in their careers. However, the scatterplot seems to show a wide dispersion of salaries for each age group, which might indicate that factors other than age are influencing salary. Similarly, experience is often expected to correlate with salary. The plot likely shows a positive correlation, suggesting that individuals with more experience tend to have higher salaries. When the `embauche` variable is plotted against age, experience, salary, and note, the scatterplots show how these variables are distributed with respect to the binary outcomes. For continuous variables like age, experience, and salary, the plots are vertical lines, since the binary outcome has only two values.

Then, I computed the correlation matrix (Table 2) and generated the associated heatmap (Figure 5).

	age	exp	salaire	note	embauche
age	1.000000	-0.002343	0.000033	-0.003818	-0.013319
exp	-0.002343	1.000000	0.010216	-0.011031	0.007371
salaire	0.000033	0.010216	1.000000	-0.447945	0.003479
note	-0.003818	-0.011031	-0.447945	1.000000	0.003307
embauche	-0.013319	0.007371	0.003479	0.003307	1.000000

Table 2: Correlation matrix [20]

The only relevant correlation to discuss here is between salary and grades (`note`). The correlation coefficient of approximately -0.448 indicates a moderate negative correlation. This suggests that there might be a tendency for individuals with higher salaries to have lower grades on the exercise. This is an interesting finding and would warrant further investigation to understand the underlying reasons for this relationship.

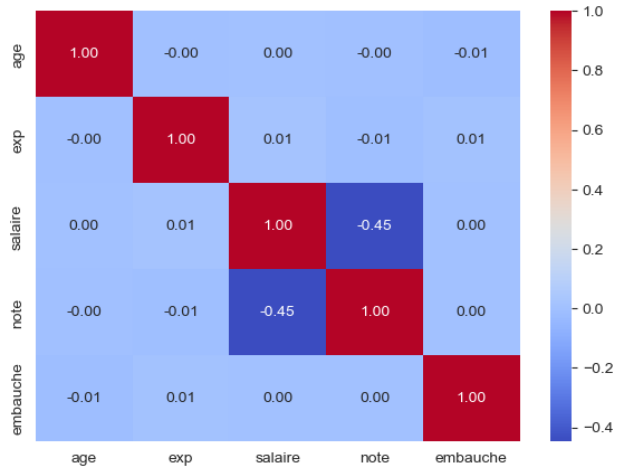

Figure 5: Heatmap [21]

Then, along with the correlation coefficient, I also calculated the p-value to determine the statistical significance of the results. The most relevant result from the analysis was the confirmation of the statistically significant negative correlation between salary and grades. All other correlations were not statistically significant, meaning there was no evidence of a linear relationship between those pairs of variables in my data set.

3 Preprocessing and Modeling

In this section, I employed a pipeline approach, integrating `LogisticRegression`, `RandomForest`, `SVC`, and `eXtremeGradientBoosting` models with a preprocessor. Furthermore, I fine-tuned these models using `GridSearchCV` for hyperparameter tuning, with the exception of `SVC` as it's computationally too expensive.

3.1 Enconding and Preprocessor

I identified the variables `specialite` and `cheveux` as candidates for one-hot encoding, while I manually label encoded other variables, such as `sexe`, `diplome` and `dispo`. This encoding choice was based on considerations of either evident cardinality (`diplome`) or just two unique values per variable (the other two). I opted not to use `LabelEncoder()` because, in general, it is more suitable for encoding the target variable, not the categorical predictor variables, as it could inadvertently introduce ordinal relationships that don't exist in the data.

Subsequently, I separated the target variable by assigning it to `y`, and I divided the dataset into three sets: a training set (70%), a validation set (10%), and a test set (20%). Afterward, I checked the sizes of each set to be sure of having applied a correct splitting (Table 3).

Train Size	Validation Size	Test Size
13,376	1,911	3,822

Table 3: Sets sizes [27]

Then, I used a preprocessor called `ColumnTransformer`, where I applied one-hot encoding specifically to the categorical columns mentioned before, and defined in the `categorical_ohe` variable. For all other columns, I set the preprocessor to `passthrough`, meaning they were left unchanged.

3.2 LogisticRegression, RandomForest and XGBoost Pipelines

After the preprocessing setup, I created three separate machine learning pipelines using `ImbPipeline`, which is a variant of the standard scikit-learn pipeline designed to handle imbalanced datasets. Each pipeline is designed to preprocess data (one-hot encode the chosen variables), apply the Synthetic Minority Oversampling Technique (`SMOTE`) for addressing class imbalance, and then use a different classifier for predictions (`LogisticRegression`, `RandomForest`, and `eXtremeGradientBoosting`). For LogisticRegression I included a `StandardScaler` to reach model convergence. After setting up the pipelines for each model, I then focused on optimizing the models using grid search, with the exception of `SupportVectorClassifier`, for which I refrained from applying the same hyperparameter-tuning due to its significant computational inefficiency, and therefore I limited to apply the model on the training data.

3.2.1 LogisticRegression GridSearchCV

First, I defined a parameter grid named `param_grid_lr` for the logistic regression classifier. This grid includes various values for the regularization strength (`C`), with a range spanning from 0.001 to 10, to see how different levels of regularization affect the model. I also chose to test two different optimization algorithms, `lbfgs` and `saga`, which are solvers used in logistic regression. Additionally, I explored varying the maximum number of iterations the solver runs (`max_iter`), with values like 200, 300, 1000, and 2000, to determine the optimal number for model convergence. Next, I set up a grid search using `GridSearchCV`, passing in the logistic regression pipeline (`pipeline_lr`), our parameter grid (`param_grid_lr`), and specifying a 5-fold cross-validation (`cv=5`). The scoring method selected was `roc_auc`, focusing on the area under the receiver

operating characteristic curve, which is a robust metric for evaluating the performance of binary classifiers. Finally, I executed the grid search by fitting it to your training data (`X_train` and `y_train`) (Figure 6). This process involved training multiple versions of the logistic regression model, each with different combinations of hyperparameters from the grid, and evaluating their performance to find the best possible combination based on the ROC AUC score (that will be used as main metric for each model).

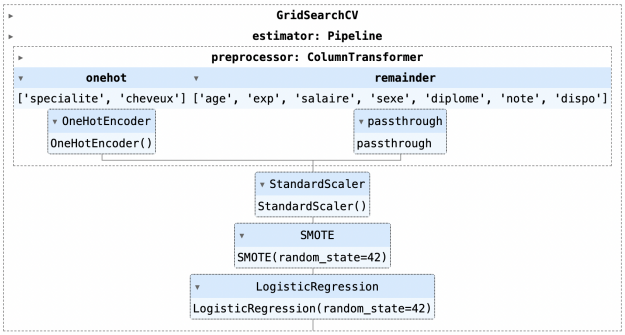

Figure 6: `LogisticRegression` pipeline [30]

Best Parameters	'classifier__C': 0.01, 'classifier__max_iter': 200, 'classifier__solver': 'lbfgs'
Best Score	0.5669094821722391

Table 4: Best `LinearRegression` parameters and score [31]

3.2.2 RandomForest GridSearchCV

In the same way, I focused on fine-tuning the Random Forest classifier within the pipeline I had previously set up. To do this, I created a grid where I specified a range of hyperparameters for the Random Forest model. I included different numbers of trees (`n_estimators`) to see how many trees would give the best results. I also varied the `max_depth` of the trees to determine the optimal depth, ranging from no limit to specific values. Additionally, I considered different values for `min_samples_split` and `min_samples_leaf` to understand the best configuration for splitting the nodes and the minimum number of samples in the leaf nodes. After defining this parameter grid, I used the `GridSearchCV` tool just as before to conduct an exhaustive search over these hyperparameters, and executed the grid search by fitting it with the training data (Figure 7).

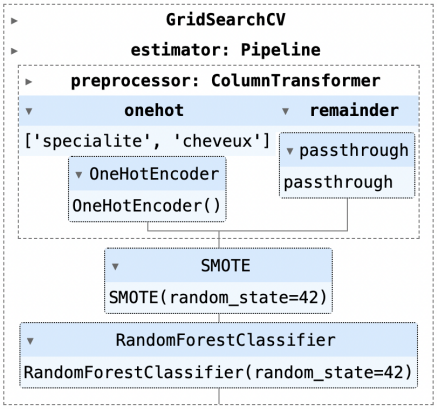

Figure 7: `RandomForest` pipeline [32]

Best Parameters	{'classifier__max_depth': 20, 'classifier__min_samples_leaf': 4, 'classifier__min_samples_split': 2, 'classifier__n_estimators': 300}
Best Score	0.8810104475457633

Table 5: Best `RandomForest` parameters and score [33]

3.2.3 XGB GridSearchCV

As we did before with the other models, I applied a very similar process of hyperparameter tuning for the `XGBoost` classifier using `GridSearchCV`. The parameter grid, `param_grid_xgb`, was carefully chosen to explore a range of values for various `XGBoost` hyperparameters. These included the `learning_rate`, which controls the step size at each iteration while moving towards a minimum of a loss function, and `max_depth`, which sets the depth of the trees. I also adjusted the `min_child_weight`, a parameter that decides the minimum sum of weights of all observations required in a child, and subsample, which denotes the fraction of samples to be used for each tree. Varying the `n_estimators` allowed me to test different numbers of trees in the ensemble, and `colsample_bytree` determined the fraction of features to be used for each tree. These parameters together define the structure and behavior of the `XGBoost` model. By setting up the `GridSearchCV` with the `XGBoost` pipeline and this parameter grid, and specifying a 5-fold cross-validation and the ROC AUC score as the evaluation metric, I was able to systematically explore various combinations of these parameters. Running `grid_search_xgb.fit(X_train, y_train)` trained the `XGBoost` models on the training data across different parameter combinations, identifying the set that achieved the best ROC AUC score (Figure 8).

Figure 8: `XGB` pipeline []

Best Parameters	{'classifier__colsample_bytree': 1.0, 'classifier__learning_rate': 0.01, 'classifier__max_depth': 9, 'classifier__min_child_weight': 3, 'classifier__n_estimators': 300, 'classifier__subsample': 0.5}
Best Score	0.887406481501888

Table 6: Best `XGB` parameters and score []

4 Models Evaluation

In this section, I will assess the performance of the models I applied.

4.1 ROC AUC Evaluation on Validation Set Prediction

I proceeded to evaluate their performance on the validation set. This involved predicting the probabilities for the positive class (`1`) using our models. For each model, I extracted the probabilities corresponding to the positive class from the output of the `predict_proba` method. With these probabilities, I calculated the ROC AUC scores for each model, that is a performance measurement for the classification problems at various threshold settings. It represents the probability that a randomly chosen positive instance is ranked higher than a randomly chosen negative one. A higher ROC AUC score indicates better model performance. To identify the best model on the validation set, I compared the ROC AUC scores of all four models (Table 7). I used a series of conditional statements to determine which model had the highest score, thus performing the best on the validation set. The ROC AUC scores indicate that the `XGBoost` model performed better

Model	ROC AUC
XGBoost	0.8607
RandomForest	0.8606
LogisticRegression	0.5639
SVC	0.4799

Table 7: Ranked ROC AUC scores []

than the others on the validation set, with a score of 0.8607. The `RandomForest` model was a close second at 0.8606. `LogisticRegression` and `SVC`, on the other hand, lagged behind with scores of 0.5639 and 0.4799, respectively.

4.2 ROC AUC Evaluation on Test Set Prediction

Consequently, I proceeded to calculate the ROC AUC score of `XGB` on the test set. However, besides evaluating only the best model, I decided to create an ensemble model to see if combining the strengths of each individual model could lead to better performance. First, I calculated the total ROC AUC score by summing up the scores of all our four models. This total score helped me in determining the relative contribution of each model to the ensemble. To achieve this, I calculated the weight of each model's ROC AUC score relative to the total. These proportions represented the weights for `LogisticRegression`, `RandomForest`, `XGBoost`, and `SVC`, which I then used to calculate a weighted average of their predicted probabilities on the test set. I obtained these probabilities by running the `predict_proba` method for each model on the test data, focusing only on the probabilities for the positive class. Then, I combined these probabilities into a single prediction for each instance in the test set. This combination was done by taking a weighted average, where each model's probabilities were multiplied by its respective weight and summed up. Finally, I evaluated the performance of this ensemble approach by calculating the ROC AUC score for these combined predictions against the actual labels of the test set.

Model	ROC AUC
XGBoost	0.8998
Ensemble	0.8974

Table 8: ROC AUC scores on test set []

I also proceeded to plot the ROC Curves scores of all the models, too visualize their performance graphically (Figure 9).

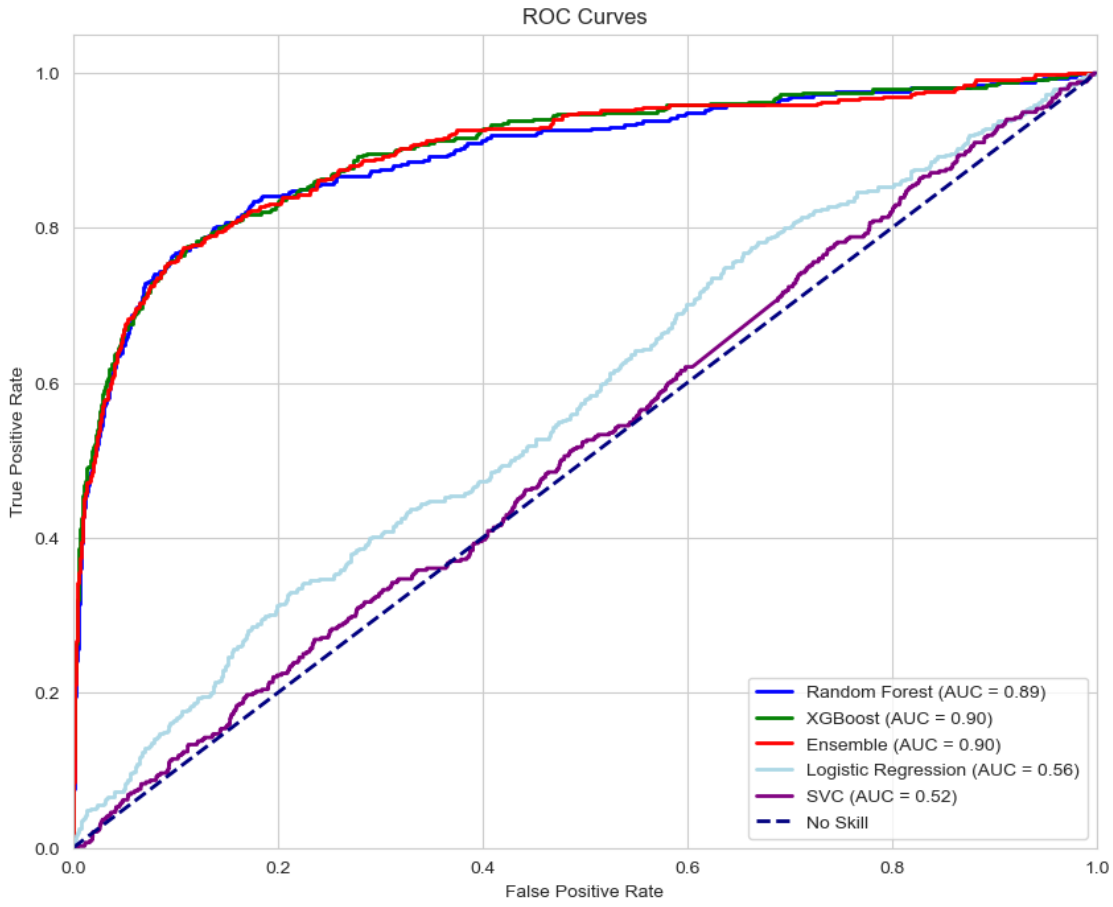

Figure 9: ROC curves []

The `RandomForest` and `XGBoost` models have very similar ROC curves and AUC values of 0.89 and 0.90 respectively, indicating they both perform quite well, with `XGBoost` having a slight edge. The Ensemble method, which combines predictions from all models, also has an AUC of 0.90, showing that it performs on par with the best individual model, `XGBoost`. In contrast, the `LogisticRegression` and `SVC` models have lower AUC values of 0.56 and 0.52 respectively, which are not far from the `No Skill` line (the dashed line representing an AUC of 0.5). This `No Skill` classifier represents a model that makes random predictions and serves as a baseline; any model that is close to this line has little to no discriminative ability between the positive and negative classes.

4.3 Classification Report Evaluation on Training and Test Set Prediction

Finally, I generated a classification report for each model, excluding the ensemble (since it was an average of the other ones), in particular in order to see if there is any over or underfitting in our models.

4.3.1 Logistic Regression

For the `LogisticRegression` model, the performance seems poor. The precision on the positive class is low, but recall is reasonably high, indicating the model is capturing a majority of the positive class but at the cost of a high false positive rate. There's no significant difference between the training and test set performance, so there is no clear evidence of overfitting or underfitting.

Training Set	Precision	Recall	F1-Score	Support
0	0.90	0.58	0.70	11844
1	0.13	0.51	0.21	1532
Accuracy			0.57	13376
Macro Avg	0.52	0.54	0.46	13376
Weighted Avg	0.81	0.57	0.65	13376
Test Set	Precision	Recall	F1-Score	Support
0	0.90	0.58	0.71	3387
1	0.13	0.49	0.21	435
Accuracy			0.57	3822
Macro Avg	0.52	0.54	0.46	3822
Weighted Avg	0.81	0.57	0.65	3822

Table 9: Classification report for `LogisticRegression` []

4.3.2 Random Forest

The `RandomForest` model shows good performance with high precision and recall on the training set, particularly for the negative class. The performance dips slightly on the test set, which is normal, but it doesn't indicate severe overfitting. The model maintains a balance between bias and variance.

Training Set	Precision	Recall	F1-Score	Support
0	0.95	0.99	0.97	11844
1	0.90	0.58	0.71	1532
Accuracy			0.94	13376
Macro Avg	0.92	0.79	0.84	13376
Weighted Avg	0.94	0.94	0.94	13376
Test Set	Precision	Recall	F1-Score	Support
0	0.94	0.98	0.96	3387
1	0.75	0.51	0.60	435
Accuracy			0.92	3822
Macro Avg	0.84	0.74	0.78	3822
Weighted Avg	0.92	0.92	0.92	3822

Table 10: Classification report for `RandomForest` []

4.3.3 Extreme Gradient Boosting

`XGBoost` also shows good performance, similar to `RandomForest`, with a slight drop in precision and recall for the positive class from the training set to the test set. The model is consistent without major signs of

overfitting or underfitting, and overall it's the best one.

Training Set	Precision	Recall	F1-Score	Support
0	0.94	0.98	0.96	11844
1	0.81	0.55	0.65	1532
Accuracy			0.93	13376
Macro Avg	0.87	0.76	0.81	13376
Weighted Avg	0.93	0.93	0.93	13376
Test Set	Precision	Recall	F1-Score	Support
0	0.94	0.97	0.96	3387
1	0.74	0.54	0.62	435
Accuracy			0.93	3822
Macro Avg	0.84	0.76	0.79	3822
Weighted Avg	0.92	0.93	0.92	3822

Table 11: Classification report for `XGBoost` []

4.3.4 Support Vector Classifier

The `SVC` model, as highlighted before, has a really poor performance. Despite a high recall, the precision is very low for both classes, especially on the training set, which suggests that the model is overfitting to the positive class. This is further indicated by the drastic drop in accuracy from the training to the test set. The model is highly biased toward predicting the positive class.

Training Set	Precision	Recall	F1-Score	Support
0	0.92	0.12	0.22	11844
1	0.12	0.92	0.21	1532
Accuracy			0.22	13376
Macro Avg	0.52	0.52	0.22	13376
Weighted Avg	0.83	0.22	0.22	13376
Test Set	Precision	Recall	F1-Score	Support
0	0.91	0.12	0.22	3387
1	0.12	0.90	0.21	435
Accuracy			0.21	3822
Macro Avg	0.51	0.51	0.21	3822
Weighted Avg	0.82	0.21	0.22	3822

Table 12: Classification report for `SVC` []

5 Conclusion

In conclusion, this report detailed my journey through the process of predicting hiring outcomes. I began with a detailed exploration of the dataset, addressing missing values and erroneous entries, and enhancing the data's readiness for modeling through various transformations, including one-hot encoding and mapping. I implemented feature engineering techniques to refine the dataset further.

The core of my analysis was the experimental phase, where I tested multiple machine learning models: Random Forest, Extreme Gradient Boosting, and Support Vector Classifier. Utilizing GridSearchCV, I tuned these models to identify which could most accurately predict hiring outcomes. The ROC AUC evaluations on the validation and test sets, complemented by classification reports, provided a rigorous assessment of each model's performance. The XGBoost model demonstrated superior performance on both the validation and test sets, followed closely by the Random Forest model. The ensemble approach, which combined the strengths of each individual model, also showed promising results. All these models revealed no or very little overfitting/underfitting.

Moving forward, I would recommend several steps that could potentially improve our model's performance. First, it would be valuable to experiment with additional feature engineering, including interaction terms and polynomial features, which might capture more complex relationships within the data. Moreover, exploring more advanced ensemble techniques like stacking or boosting may yield better predictive power by leveraging the diverse strengths of different models. Lastly, it's crucial to continue refining the data preprocessing and cleaning steps. This could involve gathering more data to reduce the potential bias and variance in the models or applying more sophisticated imputation techniques for missing values. These modifications, along with an ongoing cycle of testing and validation, are likely to yield better results and a more robust predictive model for hiring classifications.