Copenhagen Business School
HANDELSHØJSKOLEN

Case Study: Default detection for Credit Card Approval using Machine Learning Models

Quantitative Risk Management - An Application of Machine Learning (CCMVV1450E)

Written Exam

Student ID number: 158370

Examiner: Björn Press

Submission Details

Date of submission: 28-01-2024

Number of characters: 21.137

Number of normal pages**:** 13

** There is a second pdf document attached. It contains the code, detailed results of each of the models and fine tuning that was discussed. **

Abstract

This paper employs and assesses four different machine learning and deep learning models, Logistic Regression, Random Forest, XGBoost and Keras Neural Network, in order to establish which of the models is performing the best in the case of predicting which of the credit card users will default. The conclusion reached based on a comparison of recall of the risk users' class and closely examining the confusion matrix before and after the tuning process, is that XGBoost is the best model with the lowest recall to false positives tradeoff. Moreover, there are possible biases and discrimination concerns discussed which are in line with the recommendations of the Danish National Bank working paper (2022).

Introduction

In the era of digital finance, credit scoring remains a crucial process for financial institutions around the world. It is essential in the decision-making framework, determining the creditworthiness of prospective borrowers. Previously, the process relied on statistical methods and historical data, however, the development of machine learning has created a shift in this field. The traditional approach to credit scoring involves linear statistical models that now can be considered outdated due to the challenges being posed by the complexity and volume of financial data. The dynamic nature of the vastly developing world and the sheer volume of data sources create an issue for the traditionally used models which lack the complexity and possibility to capture the underlying patterns and non-linear relationships between variables. These shortcomings have created a gap and need for more sophisticated predictive models capable of adapting to the elaborateness of financial datasets. Therefore, this research paper examines the application of supervised machine learning models for credit scoring, a domain where precision and accuracy have substantial economic repercussions.

However, for these models to perform with high accuracy, they require extensive datasets, potentially comprising countless individual data points. This requirement raises critical concerns regarding privacy and data protection, particularly within the jurisdiction of the European GDPR. The GDPR mandates controls to ensure that individuals' privacy is not compromised in the pursuit of analytical insights. Compliance with such regulations necessitates a careful balance between data utility and privacy, often compelling the adoption of techniques such as data anonymisation before utilising sensitive information for model training.

Additionally, there exists the challenge of inherent bias in the data, a reflection of historical and societal inequalities. Such biases may perpetuate disparities through the decisions influenced by these models. As an example, gender bias is visible in financial datasets, as a result of long-standing societal norms. Around the globe, women have been historically underrepresented in credit markets, a factor that negligently may steer algorithmic learning in a way that could disadvantage female applicants. This is a systemic issue and should not only be acknowledged but counteracted.

Research Question

How accurately can different machine learning models predict the risk of default and what could be the implications of bias in the data itself on the outcome?

Data

Dataset Description & EDA

The dataset (Credit Card Approval - With Target, 2020), named "Credit Card Approval with Target," is derived from Kaggle, and features 19 key variables such as applicant gender, number of children, and education level. It encompasses 537,668 entries, ensuring a substantial sample for analysis. The dataset's completeness, devoid of missing values, strengthens its analytical validity.

Demographic analysis shows a gender distribution of 62% female and 38% male applicants, an aspect that may influence credit approval trends. Lifestyle indicators reveal that 57% of applicants own a vehicle and 65% possess real estate, primarily apartments or houses (90% as illustrated in Figure 1). These statistics may reflect the financial priorities or stability of the applicants.

The majority of the dataset's

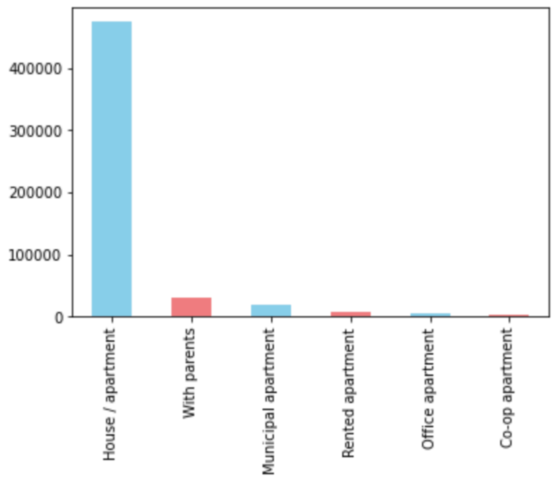


Figure 1: Type of housing

applicants are married (over 70%), with 64% having no children. Educational background shows that 67% have secondary or special secondary education, while 29% hold lower secondary education. These factors contribute to understanding the socio-economic background of the applicants. In terms of communication tools, the data indicates limited use: 72% lack a work phone, 70% a personal phone, and 90% do not use email. This could provide insights into the accessibility or preferences of the applicants.

The 'Status' variable is crucial for assessing creditworthiness. About 42% of applicants settled their debts within a month, 37% defaulted for 1-29 days, and nearly 20% had no loans in the assessed month. A 0.36% are high-risk, suggesting an imbalance in financial stability within the dataset. This imbalance is critical for model accuracy in predicting credit approval.

The histogram of Total Income displays a right-skewed distribution, indicating that a larger number of applicants have lower total incomes, with fewer individuals earning higher amounts. A correlation plot has also been created in order to see if there are any underlying relationships between the data. Age and Employment Stability: The positive correlation observed between days since birth and days of employment variables suggests a direct relationship between the age of applicants and their employment tenure. This relationship may have significant implications for credit approval processes, as it can be indicative of an applicant's financial stability and career progression. The correlation heat map indicates a general absence of multicollinearity among independent variables, as evidenced by the predominantly low correlation coefficients. This condition supports the use of these variables in predictive modelling, allowing for more accurate estimations of the individual effect of each predictor on the credit approval decision. The 'TARGET' variable, representing creditworthiness, demonstrates weak correlations with the predictor variables. This weak predictive power across the board underscores the intricate nature of credit approval and emphasises the potential necessity for complex algorithms to predict outcomes accurately. Interestingly, asset ownership (vehicles and real estate) does not show strong correlations

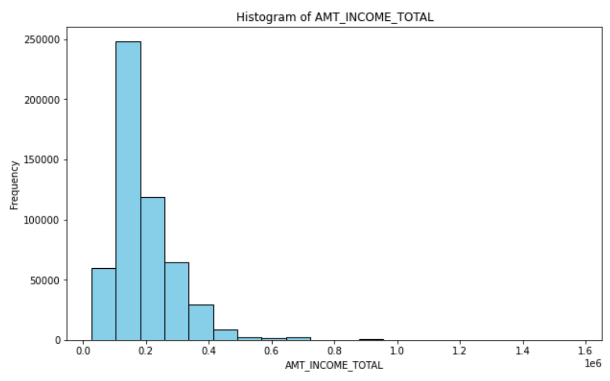


Figure 2: AMT Income Histogram

with other variables. While not primary indicators of creditworthiness on their own, these factors may contribute additional context when considered alongside other variables, painting a fuller picture of an applicant's financial health.

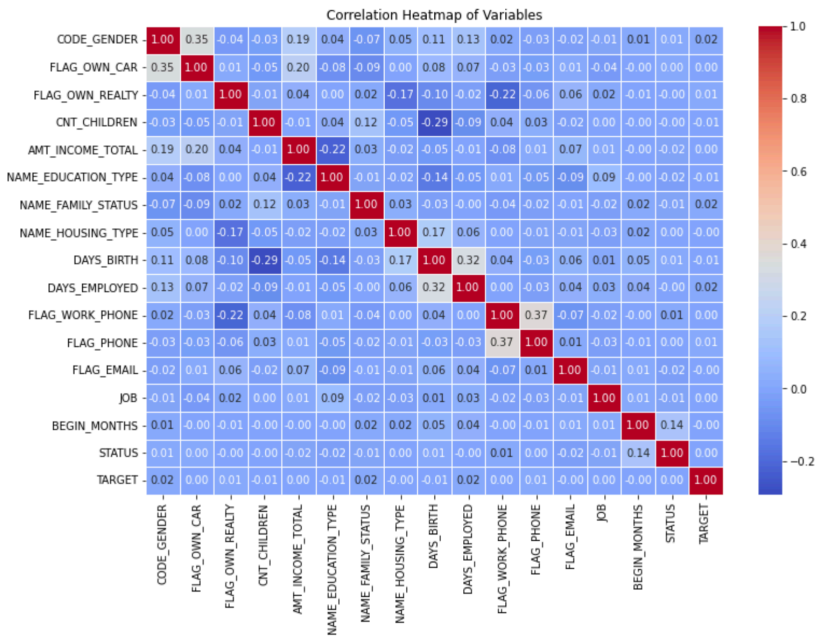


Figure 3: Correlation Heat Map

Preprocessing

For the analysis, there are several pre-processing steps taken. Firstly, the ‘ID’ and ‘FLAG_MOBIL’ columns are dropped as they do not generate any additional insights. Next, several categorical columns among others gender, ownership of a vehicle and job have been successfully encoded, transforming them into a numerical format for analysis. However, certain columns like Income, Days since birth and Target are excluded from this encoding process, as they already contain numerical or binary data. Moreover, to counteract the class imbalance of the target variable a combination of SMOTE and RandomUnderSampling was performed, which yielded that there are 234368 observations of the non-risk users and 187495 of the risk users. That generates a widely more balanced training set, as both methods were only applied to the training set. Splitting the data was performed on a global level to ensure that each of the models are performing on the exact same data which enables the comparison to be as precise as possible. Firstly, the data was split into train and test sets in 70-30 relation. Next, there was an additional split 50-50 into validation and test sets.

Methodology

This research paper will incorporate the use of four supervised machine learning and deep learning algorithms. Logistic regression, selected for its minimal complexity, will serve as the baseline model. It will be trained to deliver predictions on the test set without the enhancements from hyperparameter tuning. Next, the study will apply Random Forest, Neural Network, and XGBoost models. Each will be initially assessed on a validation set using a variety of parameter settings. Moreover, for the Random Forest and XGBoost, a grid search is implemented. Due to computational resource limitations, the Neural Network will not undergo this extensive parameter optimisation.

Logistic Regression

A Logistic Regression is being used to establish a base model for this dataset. This model is a statistical model used for binary classification. In the context of this paper, it predicts whether the applicant of the credit card will default or not. This model operates based on the logistic function of a linear combination of input features, while the output of it is a value between 0 and 1. The output of this function is a probability of default.

Random Forest

Random Forest is an ensemble learning method, particularly effective for classification tasks. It operates by creating numerous decision trees during training and presents the mode of classification of individual trees. This model was chosen due to its ability to handle efficiently large datasets as well as its robustness. In the case of credit card default prediction, it is crucial to capture nonlinear relationships, patterns and anomalies among the variables in which this model distinguishes itself.

XGBoost

This model is a scalable and efficient application of Gradient Boosting, which is characterised by successive training combined with constant correction of the errors made by the previous models, commonly decision trees are used as the base learners. In the context of default prediction, this model's iterative learning enables it to handle large and complex data and also uncover the underlying patterns. Moreover, it provides a feature importance score, offering insights into which factors are most predictive of defaults.

Neural Network

The Neural Network that has been chosen for this is inherited from the Keras framework. It was chosen as it offers flexibility and recognition of complex, non-linear relationships in large datasets. In credit card default prediction, a neural network can learn from a multitude of features, including both numerical and categorical data, through its layered structure. However, the model requires careful tuning of parameters and a large volume of data for training to avoid overfitting and ensure generalisation to unseen data.

Results

When evaluating results there are multiple evaluation methods taken into account. The standard accuracy could not be the only method as the delicate case of default prediction requires the highest level of correct prediction of "risk" users. Thereafter, the accuracy does not solely represent the model's performance. Furthermore, the confusion matrix will be widely used to make judgments of the applicability of the model to the real-life scenario.

Initial Results

	Accuracy	Recall (class 1 - Risk)	Confusion Matrix
Logistic Regression	0.4737	0.74	[[75977 84738] [150 436]]
Random Forest	0.9888	0.73	[[79516 825] [82 227]]
XGBoost	0.9162	0.64	[[73693 6648] [112 197]]
Keras	0.9962	0.00	[[80341 0] [309 0]]

Table 1: Initial Results

As can be seen in Table 1 the accuracy is the highest (99,62%) for the Keras model, however, recall and the confusion matrix create a different conclusion. These results indicate that while the model is highly accurate overall, it fails entirely to identify any true positives for the risk users, as shown by the confusion matrix having zero in the true positive cell. It has no false positives but at the cost of not detecting the risk class at all. Logistic Regression shows moderate performance with an accuracy of 47,37% and a recall of 74%. The confusion matrix reveals a significant number of false positives (84738), which indicates a high number of instances where the model incorrectly predicted the risk users. Random Forest shows a high accuracy of 98,88% and a recall of 73%. The confusion matrix shows that while it has a much lower number of false positives compared to Logistic Regression, it still makes some errors in predicting the 'Risk' class, as seen by the 82 false negatives. XGBoost presents an accuracy of 0.9162 and a recall of 0.64. This model has fewer false positives (6648) than Logistic Regression but more than Random Forest, indicating a balance between the two in terms of precision and recall. However, the lower recall suggests it may not be as sensitive in detecting risky users as the other models.

Tuning

Logistic Regression

As previously mentioned, the logistic regression is a baseline model so it is only being trained and tested on the initial parameters also without resampling the Target variable. A weight class has been added to counteract a heavily imbalanced set. It is worth noting, that the data has been trained on the exact same data, however, the results are on the test set which is larger than the one for the other models, as it consists of both train and validation data of the other models.

Random Forest

The process of tuning for the Random Forest consists of numerous steps. Firstly, the model was run on different max_features parameters: log2 and sort in order to determine which one of them was performing better. As the results of accuracy and recall were the same for both of the max features the default was chosen - ‘sqrt’. Further, there were two different grid searches performed separately. It was done purely because of the computational constraints. It should be noted it is not the ideal way. The first grid search was performed to determine the min-sample-leaf and criterion parameters. It was determined that the best parameters from this grid search are entropy and the minimum samples on the leaf is four. The second Grid Search was deployed to determine the best number of estimators and the min-sample-split which are: 200 and 25 respectively. Moreover, it is worth noting that the grid searches are also performing cross-validation at the level of three to prevent overfitting. Lastly, the best fit is determined based on the recall parameter.

XGBoost

The tuning of the Boosting algorithm consists of one grid search which is deployed on max depth, number of estimators and the learning rate. Each of the parameters has three possibilities. This approach is possible due to the efficiency of the Boosting algorithm so it is not too computationally heavy. Therefore, 81 combinations of parameters are being checked at the same time. The final parameters which are deemed to produce the highest recall score are a learning rate of 0.1, a max depth of 10 and a number of estimators is 200. Moreover, cross validation of three is being applied as well.

Neural Network

The hyperparameter tuning of the Neural Networks due to computational constraints is only possible by running the models with different parameters, one at a time. The initial model has been run on 64 units with the 'relu' activation function. The learning rate of it was set at 0.001 level. There was no regularisation applied while the dropout rate was set to 0.5. Moreover, the model has been trained on 10 epochs while the batch size is 64. Lastly, it is worth noting that the model works based on the probability in the binary classification which was set to 0.5. That translates that each of the rows in the test set that the model is assessing has a probability assigned of how risky is the user, and if that probability is higher than half it would be classified as one. Therefore, based on that, there have been three other models run with different values of the parameters of the above mentioned. The model that has achieved the best recall, and is classifying the risk users correctly has the following parameters: units - 64, the activation function is 'relu', the learning rate of 0.001, strength of regularisation of 0.01, drop out rate of 0.3, 20 epochs and the batch size of 128.

Final Results

	Accuracy	Recall (class 1 - Risk)	Confusion Matrix
Logistic Regression	0.4737	0.74	[[75977 84738] [150 436]]
Random Forest	0.9874	0.75	[[79426 948] [68 209]]
XGBoost	0.9877	0.77	[[79441 933] [63 214]]
Keras	0.0034	1.00	[[0 80374] [0 277]]

Table 2: Final Results

The results for the logistic regression are the same as in the initial results. This model is somewhat effective at identifying customers who are likely to default. However, with the extremely high number of false positives (84738). The results of the random forest show a good recall score of 75% with 209 true positives and 948 false positives reflecting a balanced approach. This model is better at identifying true defaulters without affecting a large number of non-defaulters than the logistic regression. With the highest recall of 0.77 and 214 true positives, XGBoost slightly outperforms Random Forest in identifying defaulters. Compared to the Random Forest it flags almost the same amount of false positives, a number which is still manageable manually. The Neural Network's recall of 1.00 with 277 true positives and no false negatives indicates that this model identifies every actual defaulter. However, it has an extremely high number of false positives (80374), and the lack of predictions of the non-risk class makes this model highly ineffective, as all of the users are marked as risky users which does not give any insights into the financial institutions.

Overall, XGBoost is the model that presents the best tradeoff between the number of correctly identified risk users and users who are falsely assessed as ones. The number of incorrectly classified risk users is 63 in this model. Random Forest also remains a strong contender with similar but slightly (68 misclassified risky users) lower performance metrics compared to XGBoost.

Discussion

The comparison between the initial and final results shows an improvement of the Random Forest and XGBoost of the recall measure and also in the amount of the correctly identified risk users. Therefore, it can be concluded that the fine tuning process has performed well and the models are now better at capturing the defaulters. In the case of the Neural Network, the variability of the results is such extreme that a comparison should not be drawn. The initial model does not identify a single risky user while the final model only predicts the risky users. The Keras model's performance in the final results is impractical for real-world application, despite the perfect recall, due to the extremely high false positive rate. Its inconsistency between the two evaluations also raises concerns about its stability and reliability. In practical terms, Random Forest or XGBoost would be recommended for deployment in a credit card default prediction scenario, with a preference for XGBoost based on the slightly better performance metrics. However, before choosing the model it is crucial to assess and decide what is the cost of misclassification. The cost of a false negative (a defaulter being classified as a non-defaulter) can be high, as it directly translates to credit loss. However, the cost of a false positive (a non-defaulter being classified as a defaulter) can also be significant in terms of lost business and customer churn. Therefore, there is no simple answer to the question which model is performing the best, especially when considering RF and XGBoost. As these methods are supervised during training the misclassified instances can be acknowledged. However, when the models are applied in a business context a quarter of the risky users will not be classified which would lead to a financial loss. On the other hand, it is essential to asses what is the relative cost of one defaulter to the number of new customers that a financial institution can earn interest from due to the efficiency of the machine learning algorithms. Thus, there is no clear answer as to whether the presented models are performing well enough as that metric is subjectively linked to each organisation.

Possible Bias and Discrimination

The detection of all risky customers would be the optimal solution. However, the ethics of employing the models especially, in the extremely private and demographic related credit score predictions is crucial. The detection of risk through the models is highly relevant in the current state of research and innovation and also the volumes of data that are being produced each day. However, the governance related risk detection is not less important and should also be taken into account when building and employing quantitative models. The governance of the models and checking them for possible bias and discrimination. The usage of automation in areas like credit rating becomes legitimate when it significantly enhances decision-making accuracy and efficiency, as long as it's within ethical and regulatory frameworks. As Danmarks Nationalbank (2022) mentions the models should be supervised not only at the deployment point but through its life span. It is also recommended to establish quantifiable frameworks that would relate the abstract regulatory principles to the measures that they need to satisfy. Moreover, the relation of the marginal efficiency improvement to the increased black box nature is crucial when selecting the models that would be deployed. In the case of credit card default prediction, an area which is substantial for many people, the ability to explain the model's prediction becomes over-normatively important to understand the workings of the quantitative models. Data drift poses a significant challenge in the practical use of AI models when assessing credit scoring. As the financial data and consumer behaviours evolve, the model's initial training data may no longer represent current trends, which may lead to decreased accuracy over time. Constant monitoring and updating of the model with new data are crucial to maintain its relevance and accuracy. The possibility of recalibrating the model on newer datasets could enable to address the changes in the patterns of the data and keeping the models' robustness constant.

Limitations & Further Research

There are several points to address in terms of risks and limitations connected with this study. An enormous class imbalance of the Target variable was a reason for implementing both under and over sampling on the target variable. The combination of these methods was proven to yield the most accurate prediction scores. However, it should be addressed that oversampling the minority class with SMOTE is introducing noise to the dataset as the samples are synthetic, thus they do not represent real life. On the other hand, under sampling the majority class randomly deletes rows of data, which is not a desired outcome. However, due to the size of the class imbalance, it was necessary to resample the class and the combination of these two methods was deemed to be the most applicable. On top of that, it could be seen that the Random Forests tend to favour the majority class in the datasets.

Another noteworthy limitation is the lack of computational resources which have led to not fully correct usage of the grid search in the case of the Random Forest, which may have yielded better results if the tool was used as one with interactions through all possible combinations. As further research one ought to zoom into the Neural Networks' highly disadvantageous results among all combinations. The changes in the typical hyper-parameters did not improve any of the classifications. Thus, further experimentation with the number of hidden layers within the networks or even replacing the Keras model with a different one should be looked into.

Conclusions

This paper researches the application of different machine learning and deep learning models in the context of predicting who will default on their credit card and, therefore, who should not be issued one. Moreover, the paper looks into the ethical considerations of a shift from humans making the acceptance decision to machines doing it instead. From the model section can be concluded that, after tuning, the Extreme Gradient Boost is the best model which has the best balance between detecting the defaulters and false positives. Random Forest also achieved quite a good balance between those two measures, with slightly lower recall value of the risk users and marginally more false positives. However, from the perspective of explainability mentioned by the Danish National Bank (2022), Random Forest would be better. Addressing potential biases in the data, especially gender bias, is crucial to prevent the perpetuation of societal inequalities through machine learning models. This also aligns with ethical standards and recommendations from authorities like the Danish National Bank. Moreover, the ethicality in using machine learning models forces to keep a balance between the implemented efficiency and the moral responsibility which implies the explainability of the decisions made by the algorithm. This research paper also acknowledges limitations such as class imbalance in various variables and the computational constraints, especially in the Neural Network and Random Forest, which might had an influence on the outcome of the model. In further research, it would be recommended to focus on the results of the Neural Network as the prospect of its functionality was not fully explored.

References

Credit card approval - with target. (2020, June 1). Kaggle. https://www.kaggle.com/datasets/laotse/credit-card-approval/data

Danmarks Nationalbank. (2022, April 1). *AI and machine learning in the financial sector: Five focus points*. Nationalbanken. Retrieved January 11, 2024, from https://www.nationalbanken.dk/en/news-and-knowledge/publications-and-speeches/archive-publications/2022/ai-and-machine-learning-in-the-financial-sector-five-focus-points

QRM-Code

January 26, 2024

```
[1]: import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     import seaborn as sns
     from sklearn.preprocessing import LabelEncoder, StandardScaler
     from sklearn.model_selection import train_test_split, GridSearchCV
     from sklearn.linear_model import LogisticRegression
     from sklearn.ensemble import RandomForestClassifier
     from sklearn.metrics import accuracy_score, classification_report,␣
      ↪confusion_matrix, precision_score, recall_score
     from imblearn.over_sampling import RandomOverSampler, SMOTE
     from keras.models import Sequential
     from keras.optimizers import Adam
     import category_encoders as ce
     from imblearn.pipeline import Pipeline
     from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Dense, Dropout
     from tensorflow.keras import regularizers
     from imblearn.under_sampling import RandomUnderSampler
     import xgboost as xgb
     from xgboost import XGBClassifier

     df = pd.read_csv('credit_card_approval.csv')
```

```
2024-01-26 01:36:47.238407: I tensorflow/core/platform/cpu_feature_guard.cc:182]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild
TensorFlow with the appropriate compiler flags.
```

```
[2]: df.head()
```

```
[2]:        ID CODE_GENDER FLAG_OWN_CAR FLAG_OWN_REALTY CNT_CHILDREN  \
     0  5065438           F            Y               N  2+ children
     1  5142753           F            N               N  No children
     2  5111146           M            Y               Y  No children
     3  5010310           F            Y               Y   1 children
     4  5010835           M            Y               Y  2+ children
```

```
   AMT_INCOME_TOTAL            NAME_EDUCATION_TYPE    NAME_FAMILY_STATUS  \
0          270000.0  Secondary / secondary special               Married
1           81000.0  Secondary / secondary special  Single / not married
2          270000.0               Higher education               Married
3          112500.0  Secondary / secondary special               Married
4          139500.0  Secondary / secondary special               Married

   NAME_HOUSING_TYPE  DAYS_BIRTH  DAYS_EMPLOYED  FLAG_MOBIL  FLAG_WORK_PHONE  \
0       With parents      -13258          -2300           1                0
1  House / apartment      -17876           -377           1                1
2  House / apartment      -19579          -1028           1                0
3  House / apartment      -15109          -1956           1                0
4  House / apartment      -17281          -5578           1                1

   FLAG_PHONE  FLAG_EMAIL                    JOB  BEGIN_MONTHS STATUS  TARGET
0           0           0               Managers            -6      C       0
1           1           0  Private service staff            -4      0       0
2           1           0               Laborers             0      C       0
3           0           0             Core staff            -3      0       0
4           0           0                Drivers           -29      0       0
```

[3]:
```python
print(df.shape)
print(df.isnull().sum())
print(df.info(show_counts=True, verbose=True))
df.describe()
```

```
(537667, 19)
ID                     0
CODE_GENDER            0
FLAG_OWN_CAR           0
FLAG_OWN_REALTY        0
CNT_CHILDREN           0
AMT_INCOME_TOTAL       0
NAME_EDUCATION_TYPE    0
NAME_FAMILY_STATUS     0
NAME_HOUSING_TYPE      0
DAYS_BIRTH             0
DAYS_EMPLOYED          0
FLAG_MOBIL             0
FLAG_WORK_PHONE        0
FLAG_PHONE             0
FLAG_EMAIL             0
JOB                    0
BEGIN_MONTHS           0
STATUS                 0
TARGET                 0
dtype: int64
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 537667 entries, 0 to 537666
Data columns (total 19 columns):
 #   Column              Non-Null Count   Dtype  
---  ------              --------------   -----  
 0   ID                  537667 non-null  int64  
 1   CODE_GENDER         537667 non-null  object 
 2   FLAG_OWN_CAR        537667 non-null  object 
 3   FLAG_OWN_REALTY     537667 non-null  object 
 4   CNT_CHILDREN        537667 non-null  object 
 5   AMT_INCOME_TOTAL    537667 non-null  float64
 6   NAME_EDUCATION_TYPE 537667 non-null  object 
 7   NAME_FAMILY_STATUS  537667 non-null  object 
 8   NAME_HOUSING_TYPE   537667 non-null  object 
 9   DAYS_BIRTH          537667 non-null  int64  
 10  DAYS_EMPLOYED       537667 non-null  int64  
 11  FLAG_MOBIL          537667 non-null  int64  
 12  FLAG_WORK_PHONE     537667 non-null  int64  
 13  FLAG_PHONE          537667 non-null  int64  
 14  FLAG_EMAIL          537667 non-null  int64  
 15  JOB                 537667 non-null  object 
 16  BEGIN_MONTHS        537667 non-null  int64  
 17  STATUS              537667 non-null  object 
 18  TARGET              537667 non-null  int64  
dtypes: float64(1), int64(9), object(9)
memory usage: 77.9+ MB
None
```

[3]:
```
                 ID  AMT_INCOME_TOTAL     DAYS_BIRTH  DAYS_EMPLOYED  \
count  5.376670e+05      5.376670e+05  537667.000000  537667.000000   
mean   5.079231e+06      1.971171e+05  -15010.958999   -2762.029935   
std    4.200200e+04      1.041390e+05    3416.418092    2393.919456   
min    5.008806e+06      2.700000e+04  -24611.000000  -15713.000000   
25%    5.044925e+06      1.350000e+05  -17594.000000   -3661.000000   
50%    5.079091e+06      1.800000e+05  -14785.000000   -2147.000000   
75%    5.115755e+06      2.295000e+05  -12239.000000   -1050.000000   
max    5.150487e+06      1.575000e+06   -7489.000000     -17.000000   

       FLAG_MOBIL  FLAG_WORK_PHONE     FLAG_PHONE     FLAG_EMAIL  \
count    537667.0    537667.000000  537667.000000  537667.000000   
mean          1.0         0.281615       0.298893       0.100730   
std           0.0         0.449787       0.457773       0.300971   
min           1.0         0.000000       0.000000       0.000000   
25%           1.0         0.000000       0.000000       0.000000   
50%           1.0         0.000000       0.000000       0.000000   
75%           1.0         1.000000       1.000000       0.000000   
max           1.0         1.000000       1.000000       1.000000   
```

```
        BEGIN_MONTHS         TARGET
count  537667.000000  537667.000000
mean      -19.305241       0.003649
std        14.037827       0.060298
min       -60.000000       0.000000
25%       -29.000000       0.000000
50%       -17.000000       0.000000
75%        -8.000000       0.000000
max         0.000000       1.000000
```

[4]:
```
df.drop(df.columns[0], axis=1, inplace=True)
print(df.shape)
df.head()
```

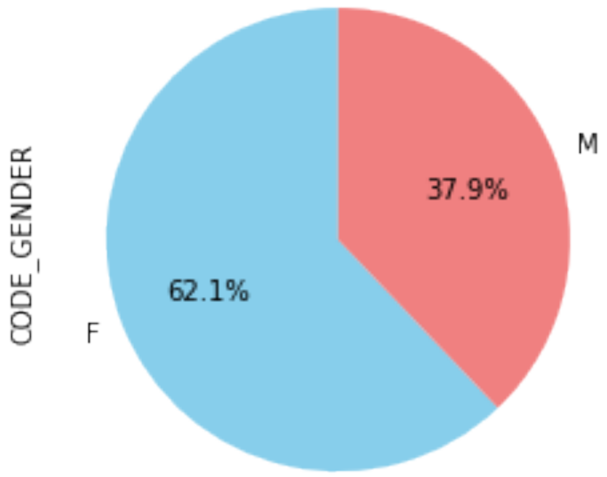
```
(537667, 18)
```

[4]:
```
  CODE_GENDER FLAG_OWN_CAR FLAG_OWN_REALTY CNT_CHILDREN  AMT_INCOME_TOTAL  \
0           F            Y               N  2+ children          270000.0
1           F            N               N  No children           81000.0
2           M            Y               Y  No children          270000.0
3           F            Y               Y   1 children          112500.0
4           M            Y               Y  2+ children          139500.0

             NAME_EDUCATION_TYPE    NAME_FAMILY_STATUS   NAME_HOUSING_TYPE  \
0  Secondary / secondary special               Married        With parents
1  Secondary / secondary special  Single / not married   House / apartment
2               Higher education               Married   House / apartment
3  Secondary / secondary special               Married   House / apartment
4  Secondary / secondary special               Married   House / apartment

   DAYS_BIRTH  DAYS_EMPLOYED  FLAG_MOBIL  FLAG_WORK_PHONE  FLAG_PHONE  \
0      -13258          -2300           1                0           0
1      -17876           -377           1                1           1
2      -19579          -1028           1                0           1
3      -15109          -1956           1                0           0
4      -17281          -5578           1                1           0

   FLAG_EMAIL                    JOB  BEGIN_MONTHS STATUS  TARGET
0           0               Managers            -6      C       0
1           0  Private service staff            -4      0       0
2           0               Laborers             0      C       0
3           0             Core staff            -3      0       0
4           0                Drivers           -29      0       0
```

[5]:
```
print(df['CODE_GENDER'].unique())
df['CODE_GENDER'].value_counts().plot.pie(autopct='%1.1f%%', colors=['skyblue',␣
 ↪'lightcoral'], startangle=90)
```

```
['F' 'M']
```

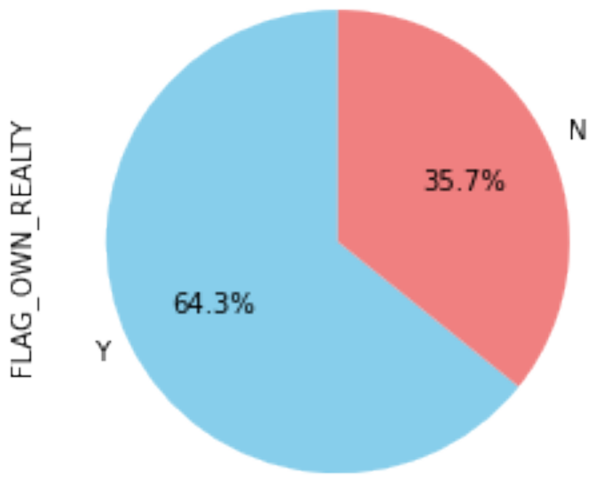
```
[5]: <AxesSubplot:ylabel='CODE_GENDER'>
```



```
[6]: print(df['FLAG_OWN_REALTY'].unique())
     df['FLAG_OWN_REALTY'].value_counts().plot.pie(autopct='%1.1f%%',␣
      ↪colors=['skyblue', 'lightcoral'], startangle=90)
```

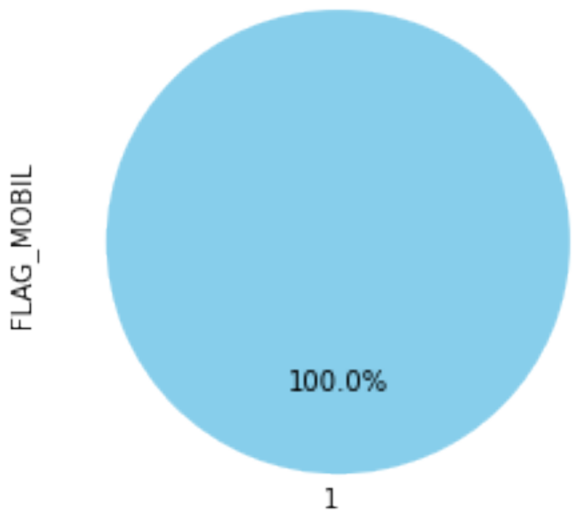
```
['N' 'Y']
```

```
[6]: <AxesSubplot:ylabel='FLAG_OWN_REALTY'>
```



```
[7]: print(df['FLAG_MOBIL'].unique())
     df['FLAG_MOBIL'].value_counts().plot.pie(autopct='%1.1f%%', colors=['skyblue',␣
      ↪'lightcoral'], startangle=90)
     df = df.drop("FLAG_MOBIL", axis=1)
     print(df.shape)

     #dropped cause no additional knowledge
```

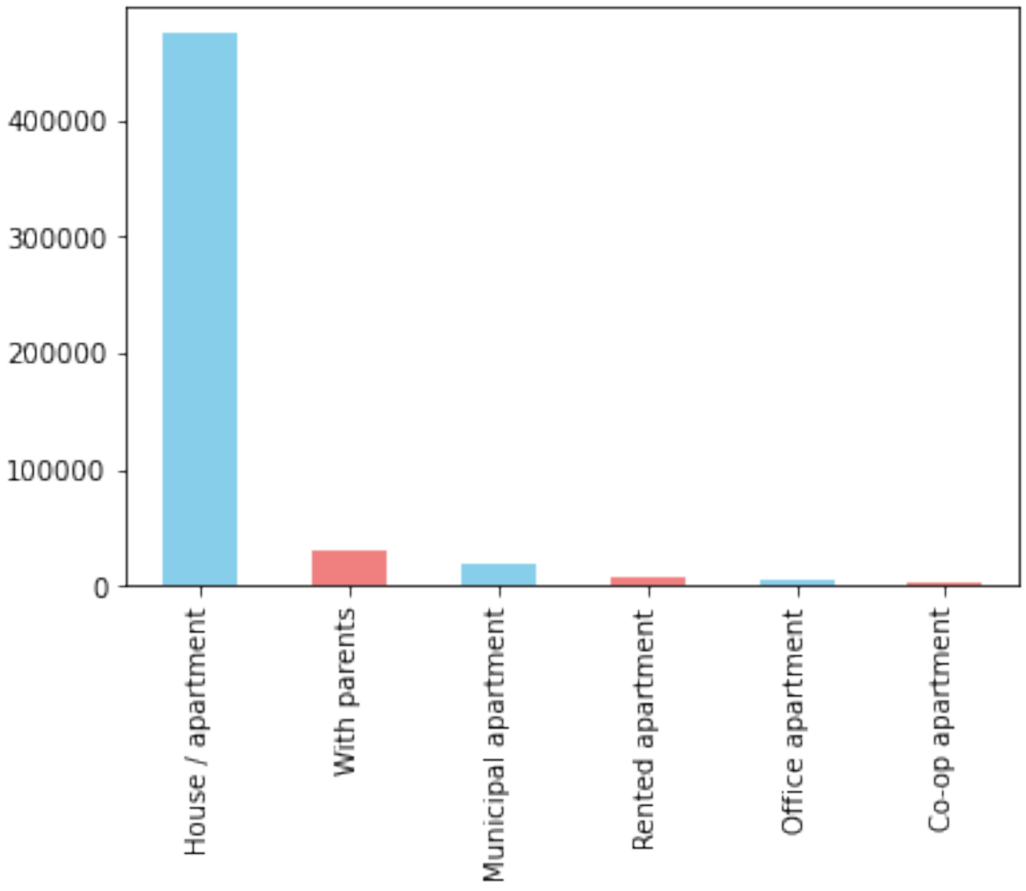
```
[1]
(537667, 17)
```



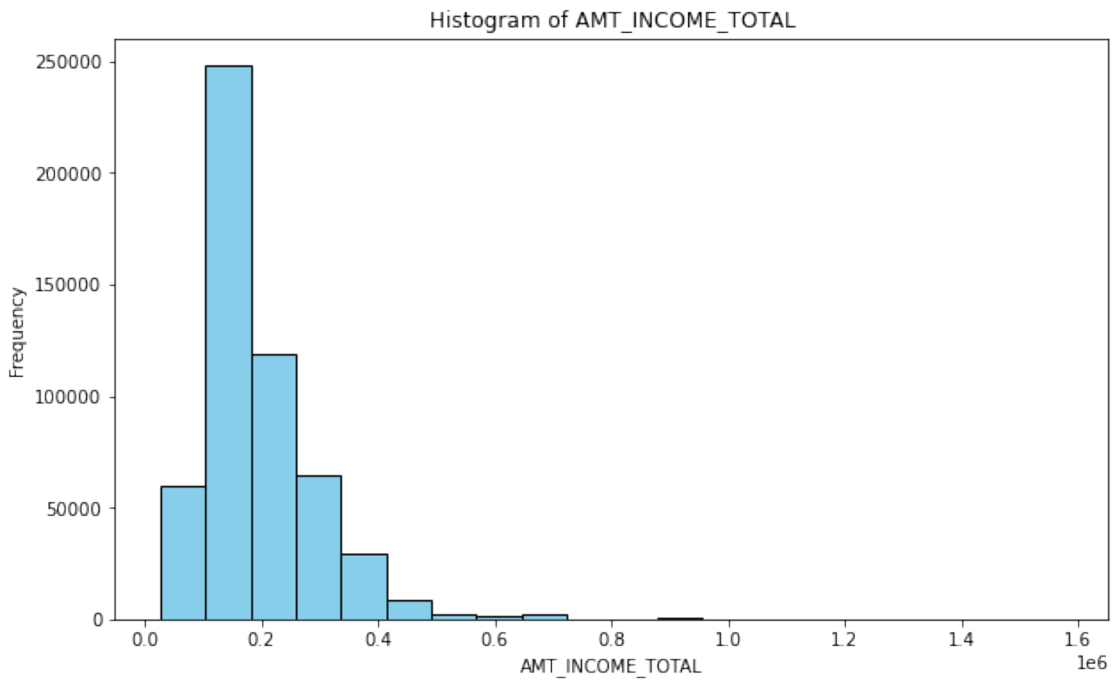
```
[8]: print(df['NAME_HOUSING_TYPE'].unique())
     df['NAME_HOUSING_TYPE'].value_counts().plot(kind='bar', color=['skyblue',␣
      ↪'lightcoral'])
```

```
['With parents' 'House / apartment' 'Rented apartment'
 'Municipal apartment' 'Co-op apartment' 'Office apartment']
```

```
[8]: <AxesSubplot:>
```



[9]:
```python
plt.figure(figsize=(10, 6))
plt.hist(df['AMT_INCOME_TOTAL'], bins=20, color='skyblue', edgecolor='black')
plt.title('Histogram of AMT_INCOME_TOTAL')
plt.xlabel('AMT_INCOME_TOTAL')
plt.ylabel('Frequency')
plt.show()
```



[10]:
```
#Encoding catogerical varaibles and mapping out the encoding
def encode_categorical_columns(df):
    label_encoder = LabelEncoder()
    encoding_mapping = {}

    for column_name in df.columns:
        if df[column_name].dtype == 'object':
            try:
                df[column_name] = label_encoder.fit_transform(df[column_name])
                encoding_mapping[column_name] = dict(zip(label_encoder.
 ↪classes_, label_encoder.transform(label_encoder.classes_)))
                print(f"Column '{column_name}' successfully encoded.")
            except Exception as e:
                print(f"Error encoding column '{column_name}': {e}. Skipping.")
        else:
            print(f"Column '{column_name}' is not of type 'object'. Skipping.")

    return df, encoding_mapping
df_encoded, encoding_mapping = encode_categorical_columns(df)

for column, mapping in encoding_mapping.items():
    print(f"\nEncoding Mapping for Column '{column}':")
    print(pd.DataFrame(list(mapping.items()), columns=['Original Values',␣
 ↪'Encoded Values']))
```

```
Column 'CODE_GENDER' successfully encoded.
Column 'FLAG_OWN_CAR' successfully encoded.
Column 'FLAG_OWN_REALTY' successfully encoded.
Column 'CNT_CHILDREN' successfully encoded.
Column 'AMT_INCOME_TOTAL' is not of type 'object'. Skipping.
Column 'NAME_EDUCATION_TYPE' successfully encoded.
Column 'NAME_FAMILY_STATUS' successfully encoded.
Column 'NAME_HOUSING_TYPE' successfully encoded.
Column 'DAYS_BIRTH' is not of type 'object'. Skipping.
Column 'DAYS_EMPLOYED' is not of type 'object'. Skipping.
Column 'FLAG_WORK_PHONE' is not of type 'object'. Skipping.
Column 'FLAG_PHONE' is not of type 'object'. Skipping.
Column 'FLAG_EMAIL' is not of type 'object'. Skipping.
Column 'JOB' successfully encoded.
Column 'BEGIN_MONTHS' is not of type 'object'. Skipping.
Column 'STATUS' successfully encoded.
Column 'TARGET' is not of type 'object'. Skipping.

Encoding Mapping for Column 'CODE_GENDER':
  Original Values  Encoded Values
0               F               0
1               M               1

Encoding Mapping for Column 'FLAG_OWN_CAR':
  Original Values  Encoded Values
0               N               0
1               Y               1

Encoding Mapping for Column 'FLAG_OWN_REALTY':
  Original Values  Encoded Values
0               N               0
1               Y               1

Encoding Mapping for Column 'CNT_CHILDREN':
  Original Values  Encoded Values
0      1 children               0
1     2+ children               1
2     No children               2

Encoding Mapping for Column 'NAME_EDUCATION_TYPE':
                 Original Values  Encoded Values
0                Academic degree               0
1               Higher education               1
2              Incomplete higher               2
3                Lower secondary               3
4  Secondary / secondary special               4

Encoding Mapping for Column 'NAME_FAMILY_STATUS':
        Original Values  Encoded Values
0        Civil marriage               0
1               Married               1
2             Separated               2
3  Single / not married               3
4                 Widow               4

Encoding Mapping for Column 'NAME_HOUSING_TYPE':
       Original Values  Encoded Values
0      Co-op apartment               0
1    House / apartment               1
2  Municipal apartment               2
3     Office apartment               3
4     Rented apartment               4
5         With parents               5

Encoding Mapping for Column 'JOB':
          Original Values  Encoded Values
0             Accountants               0
1          Cleaning staff               1
2           Cooking staff               2
3              Core staff               3
4                 Drivers               4
5                HR staff               5
6   High skill tech staff               6
7                IT staff               7
8                Laborers               8
9      Low-skill Laborers               9
10               Managers              10
11         Medicine staff              11
12   Private service staff             12
13          Realty agents              13
14            Sales staff              14
15            Secretaries              15
16         Security staff              16
17    Waiters/barmen staff             17

Encoding Mapping for Column 'STATUS':
  Original Values  Encoded Values
0               0               0
1               1               1
2               2               2
3               3               3
4               4               4
5               5               5
6               C               6
7               X               7
```

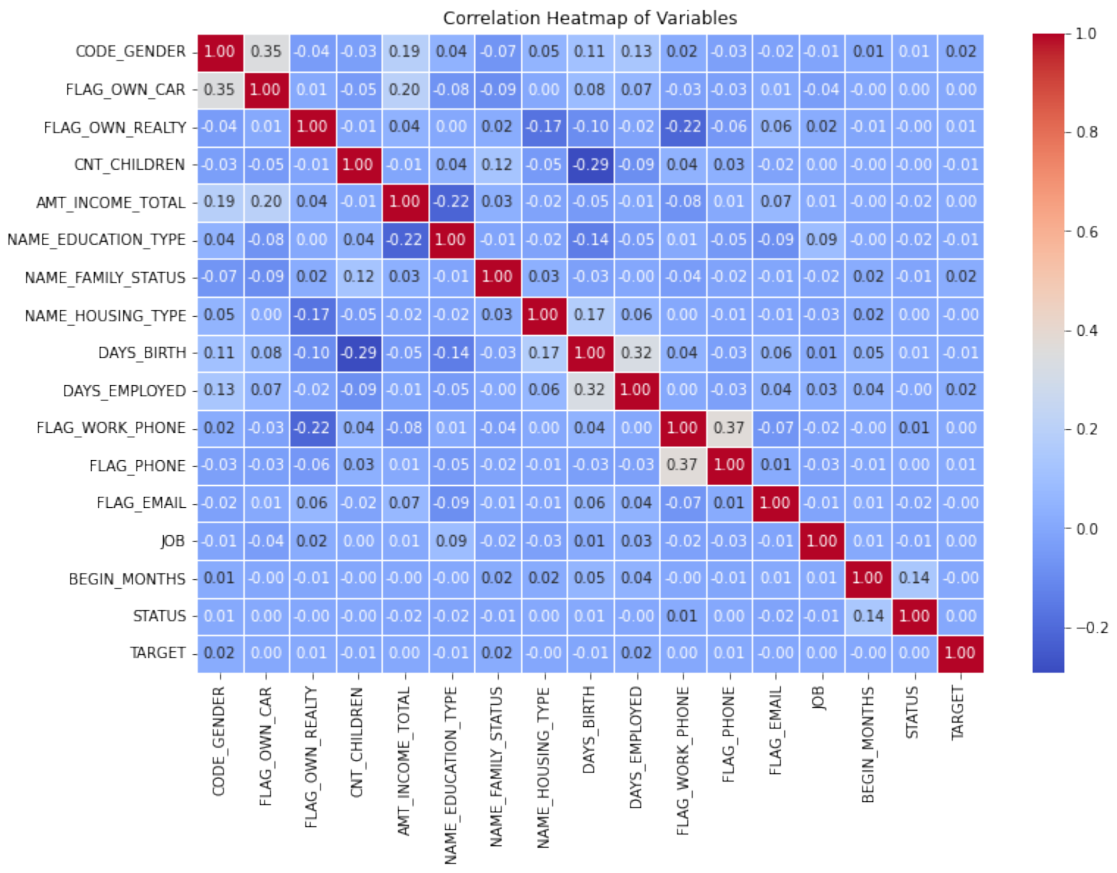
```
[11]: #Function for checking the distribution of the dataset
def check_class_distribution_percentages(df):
    total_rows = df.shape[0]
    for column in df.columns:
        print(f"Column: {column}")
#binary
        if df[column].nunique() == 2:
            class_distribution = df[column].value_counts(normalize=True) * 100
            print(class_distribution)
        else:
#non binary
            unique_values_count = df[column].nunique()
            if unique_values_count <= 10:
                class_distribution = df[column].value_counts(normalize=True) *␣
↪100
                print(class_distribution)
            else:
                print(f"Too many unique values ({unique_values_count}) for␣
↪detailed output.")

check_class_distribution_percentages(df_encoded)
```

```
Column: CODE_GENDER
0    62.088988
1    37.911012
Name: CODE_GENDER, dtype: float64
Column: FLAG_OWN_CAR
0    56.95105
1    43.04895
Name: FLAG_OWN_CAR, dtype: float64
Column: FLAG_OWN_REALTY
1    64.253711
0    35.746289
Name: FLAG_OWN_REALTY, dtype: float64
Column: CNT_CHILDREN
2    63.822217
0    23.749830
1    12.427953
Name: CNT_CHILDREN, dtype: float64
Column: AMT_INCOME_TOTAL
Too many unique values (195) for detailed output.
Column: NAME_EDUCATION_TYPE
4    66.642922
1    28.599486
2     3.829508
3     0.847365
0     0.080719
Name: NAME_EDUCATION_TYPE, dtype: float64
```

```
Column: NAME_FAMILY_STATUS
1    71.420229
3    12.264841
0     8.198941
2     5.838930
4     2.277060
Name: NAME_FAMILY_STATUS, dtype: float64
Column: NAME_HOUSING_TYPE
1    88.191576
5     5.458955
2     3.352075
4     1.592249
3     0.773527
0     0.631618
Name: NAME_HOUSING_TYPE, dtype: float64
Column: DAYS_BIRTH
Too many unique values (5206) for detailed output.
Column: DAYS_EMPLOYED
Too many unique values (3299) for detailed output.
Column: FLAG_WORK_PHONE
0    71.838517
1    28.161483
Name: FLAG_WORK_PHONE, dtype: float64
Column: FLAG_PHONE
0    70.110682
1    29.889318
Name: FLAG_PHONE, dtype: float64
Column: FLAG_EMAIL
0    89.927037
1    10.072963
Name: FLAG_EMAIL, dtype: float64
Column: JOB
Too many unique values (18) for detailed output.
Column: BEGIN_MONTHS
Too many unique values (61) for detailed output.
Column: STATUS
6    42.067860
0    37.370715
7    19.001910
1     1.194606
5     0.202170
2     0.100806
3     0.033664
4     0.028270
Name: STATUS, dtype: float64
Column: TARGET
0    99.63509
1     0.36491
```

```
Name: TARGET, dtype: float64
```

[12]:
```
##Heat Map plot
plt.figure(figsize=(12,8))
sns.heatmap(df_encoded.corr(), annot=True, cmap='coolwarm', fmt='.2f',␣
↪linewidths=0.5)
plt.title('Correlation Heatmap of Variables')
plt.show()
```



[13]:
```
# split into train and test (there will be a further split into validation)
def split_data(df_encoded, test_size=0.3, random_state=None):
    X = df_encoded.drop(columns=['TARGET', 'STATUS']) ## Status is delated␣
↪cause it basically gives the answer espacially for the already defaulted
    y = df_encoded['TARGET']
    X_train, X_test, y_train, y_test = train_test_split(X, y,␣
↪test_size=test_size, random_state=42)
    return X_train, X_test, y_train, y_test

X_train, X_test, y_train, y_test = split_data(df_encoded)
```

[14]:
```python
# metrics to call for all models
def evaluations(y_test, y_pred):

    accuracy = accuracy_score(y_test, y_pred)
    classification_rep = classification_report(y_test, y_pred)
    confusion_mat = confusion_matrix(y_test, y_pred)
    precision = precision_score(y_test, y_pred)
    recall = recall_score(y_test, y_pred)

    metrics_dict = {
        'accuracy': accuracy,
        'classification_report': classification_rep,
        'confusion_matrix': confusion_mat,
        'precision': precision,
        'recall': recall
    }

    return metrics_dict

def print_metrics(metrics_dict):
    print(f"Accuracy: {metrics_dict['accuracy']:.4f}")
    print("\nClassification Report:")
    print(metrics_dict['classification_report'])
    print("\nConfusion Matrix:")
    print(metrics_dict['confusion_matrix'])
    print(f"Precision: {metrics_dict['precision']:.4f}")
    print(f"Recall: {metrics_dict['recall']:.4f}")
```

[15]:
```python
# Under and over sampling
over = SMOTE(sampling_strategy=0.5)
under = RandomUnderSampler(sampling_strategy=0.8)
steps = [('o', over), ('u', under)]
pipeline = Pipeline(steps=steps)
X_resampled, y_resampled = pipeline.fit_resample(X_train, y_train)

# Checking class distribution after sampling
num_observations_resampled = X_resampled.shape[0]
print(f'Number of observations after resampling: {num_observations_resampled}')

y_resampled_distribution = pd.Series(y_resampled).value_counts()
print("Class distribution after resampling:")
print(y_resampled_distribution)
```

```
Number of observations after resampling: 421863
Class distribution after resampling:
0    234368
1    187495
Name: TARGET, dtype: int64
```

[16]:
```
# Function for under-sampling the target variable to balance the training␣
↪dataset
#def undersample_with_random(X_train, y_train, random_state=42):
#    undersampler = RandomUnderSampler(random_state=random_state,␣
↪sampling_strategy='auto')
#    X_train_undersampled, y_train_undersampled = undersampler.
↪fit_resample(X_train, y_train)
#   return X_train_undersampled, y_train_undersampled

#X_train_undersampled, y_train_undersampled = undersample_with_random(X_train,␣
↪y_train)

#num_observations_undersampled = X_train_undersampled.shape[0]
#print(f'Number of observations in the undersampled data:␣
↪{num_observations_undersampled}')

#y_train_distribution_undersampled = pd.Series(y_train_undersampled).
↪value_counts()
#print("Class Distribution in y_train after under-sampling:")
#print(y_train_distribution_undersampled)

#check_class_distribution_percentages function call can be used here if needed
#check_class_distribution_percentages(X_train_undersampled)
```

[17]:
```
##Oversampling the Target variable for a more balanced training dataset
#def oversample_with_smote(X_train, y_train, random_state=42):
#    smote = SMOTE(random_state=random_state, sampling_strategy='auto')
#    X_train_oversampled, y_train_oversampled = smote.fit_resample(X_train,␣
↪y_train)
#    return X_train_oversampled, y_train_oversampled
#X_train_resampled, y_train_resampled = oversample_with_smote(X_train, y_train)

#num_observations = X_train_resampled.shape[0]
#print(f'Number of observations in the oversampled data: {num_observations}')

#y_train_distribution = pd.Series(y_train_resampled).value_counts()
#print("Class Distribution in y_train:")
#print(y_train_distribution)

#check_class_distribution_percentages(X_train_resampled)
```

[18]:
```
##Logistic Regression - no
###class weight included to make it more balanced

model_lro = LogisticRegression(class_weight='balanced', random_state=42)
model_lro.fit(X_train, y_train)
```

```
y_pred_lro = model_lro.predict(X_test)
metrics_dict_lro = evaluations(y_test, y_pred_lro)
print_metrics(metrics_dict_lro)
```

```
Accuracy: 0.4737

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.47      0.64    160715
           1       0.01      0.74      0.01       586

    accuracy                           0.47    161301
   macro avg       0.50      0.61      0.33    161301
weighted avg       0.99      0.47      0.64    161301


Confusion Matrix:
[[75977 84738]
 [  150   436]]
Precision: 0.0051
Recall: 0.7440
```

[19]:
```
# spliting the test into test and val so i can run few combinations
def split_test_set(X_test, y_test, validation_size=0.5, random_state=None):
    X_validation, X_test, y_validation, y_test = train_test_split(
        X_test, y_test, test_size=validation_size, random_state=random_state
    )
    return X_validation, X_test, y_validation, y_test

X_val, X_test, y_val, y_test = split_test_set(X_test, y_test, validation_size=0.
 ↪5, random_state=42)
```

[20]:
```
##Nural Network Model
def create_regularized_nn_model(units=64, activation='relu', learning_rate=0.
 ↪001, l2_strength=0.01, dropout_rate=0.5):
    model = Sequential()
    model.add(Dense(units=units, activation=activation,␣
 ↪kernel_regularizer=regularizers.l2(l2_strength), input_dim=X_resampled.
 ↪shape[1]))
    model.add(Dropout(dropout_rate))
    model.add(Dense(units=1, activation='sigmoid'))
    model.compile(optimizer=Adam(learning_rate=learning_rate),␣
 ↪loss='binary_crossentropy', metrics=['accuracy'])
    return model
```

[40]:
```
# Initial model
units = 64
activation = 'relu'
learning_rate = 0.001
l2_strength = None
dropout_rate = 0.5
epochs = 10
batch_size = 64

nn_model = create_regularized_nn_model(units=units, activation=activation,␣
 ↪learning_rate=learning_rate, l2_strength=l2_strength,␣
 ↪dropout_rate=dropout_rate)
nn_model.fit(X_resampled, y_resampled, epochs=epochs, batch_size=batch_size,␣
 ↪verbose=1)
y_pred_nn = (nn_model.predict(X_val) > 0.5).astype('int32')
metrics_dict_nn = evaluations(y_val, y_pred_nn)
print_metrics(metrics_dict_nn)
```

```
Epoch 1/10
6592/6592 [==============================] - 7s 988us/step - loss: 387.9081 -
accuracy: 0.5234
Epoch 2/10
6592/6592 [==============================] - 7s 989us/step - loss: 0.7071 -
accuracy: 0.5712
Epoch 3/10
6592/6592 [==============================] - 7s 1ms/step - loss: 0.6936 -
accuracy: 0.5556
Epoch 4/10
6592/6592 [==============================] - 6s 967us/step - loss: 0.6878 -
accuracy: 0.5556
Epoch 5/10
6592/6592 [==============================] - 6s 964us/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 6/10
6592/6592 [==============================] - 6s 969us/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 7/10
6592/6592 [==============================] - 6s 980us/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 8/10
6592/6592 [==============================] - 7s 995us/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 9/10
6592/6592 [==============================] - 7s 1ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 10/10
6592/6592 [==============================] - 7s 986us/step - loss: 0.6870 -
accuracy: 0.5556
```

```
2521/2521 [==============================] - 2s 613us/step
Accuracy: 0.9962

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     80341
           1       0.00      0.00      0.00       309

    accuracy                           1.00     80650
   macro avg       0.50      0.50      0.50     80650
weighted avg       0.99      1.00      0.99     80650


Confusion Matrix:
[[80341     0]
 [  309     0]]
Precision: 0.0000
Recall: 0.0000
```

```
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision is ill-defined and being set to 0.0 due to no predicted samples. Use
`zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

[21]:
```
# relu and low learning rate
units = 64
activation = 'relu'
learning_rate = 0.001
l2_strength = 0.01
dropout_rate = 0.3
epochs = 20
```

```
batch_size = 128

nn_model = create_regularized_nn_model(units=units, activation=activation,␣
 ↪learning_rate=learning_rate, l2_strength=l2_strength,␣
 ↪dropout_rate=dropout_rate)
nn_model.fit(X_resampled, y_resampled, epochs=epochs, batch_size=batch_size,␣
 ↪verbose=1)
y_pred_nn = (nn_model.predict(X_val) > 0.35).astype('int32')
metrics_dict_nn = evaluations(y_val, y_pred_nn)
print_metrics(metrics_dict_nn)
```

```
Epoch 1/20
3296/3296 [==============================] - 5s 1ms/step - loss: 491.5466 -
accuracy: 0.5255
Epoch 2/20
3296/3296 [==============================] - 5s 2ms/step - loss: 1.3983 -
accuracy: 0.5504
Epoch 3/20
3296/3296 [==============================] - 5s 1ms/step - loss: 0.7097 -
accuracy: 0.6052
Epoch 4/20
3296/3296 [==============================] - 4s 1ms/step - loss: 0.7014 -
accuracy: 0.5832
Epoch 5/20
3296/3296 [==============================] - 4s 1ms/step - loss: 0.7115 -
accuracy: 0.5812
Epoch 6/20
3296/3296 [==============================] - 5s 2ms/step - loss: 0.7200 -
accuracy: 0.5558
Epoch 7/20
3296/3296 [==============================] - 4s 1ms/step - loss: 0.7104 -
accuracy: 0.5556
Epoch 8/20
3296/3296 [==============================] - 5s 1ms/step - loss: 0.6876 -
accuracy: 0.5556
Epoch 9/20
3296/3296 [==============================] - 5s 2ms/step - loss: 0.6871 -
accuracy: 0.5556
Epoch 10/20
3296/3296 [==============================] - 7s 2ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 11/20
3296/3296 [==============================] - 8s 2ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 12/20
3296/3296 [==============================] - 11s 3ms/step - loss: 0.6870 -
accuracy: 0.5556
```

```
Epoch 13/20
3296/3296 [==============================] - 9s 3ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 14/20
3296/3296 [==============================] - 8s 2ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 15/20
3296/3296 [==============================] - 7s 2ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 16/20
3296/3296 [==============================] - 7s 2ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 17/20
3296/3296 [==============================] - 6s 2ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 18/20
3296/3296 [==============================] - 7s 2ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 19/20
3296/3296 [==============================] - 5s 2ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 20/20
3296/3296 [==============================] - 5s 1ms/step - loss: 0.6870 -
accuracy: 0.5556
2521/2521 [==============================] - 2s 851us/step
Accuracy: 0.0038

Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00     80341
           1       0.00      1.00      0.01       309

    accuracy                           0.00     80650
   macro avg       0.00      0.50      0.00     80650
weighted avg       0.00      0.00      0.00     80650


Confusion Matrix:
[[    0 80341]
 [    0   309]]
Precision: 0.0038
Recall: 1.0000
```

```
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
```

```
  _warn_prf(average, modifier, msg_start, len(result))
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

[22]:
```
## NN with Sigmoid and higher learning rate
units = 64
activation = 'sigmoid'
learning_rate = 0.0001
l2_strength = 0.1
dropout_rate = 0.01
epochs = 20
batch_size = 64

nn_model = create_regularized_nn_model(units=units, activation=activation,␣
 ↪learning_rate=learning_rate, l2_strength=l2_strength,␣
 ↪dropout_rate=dropout_rate)
nn_model.fit(X_resampled, y_resampled, epochs=epochs, batch_size=batch_size,␣
 ↪verbose=1)
y_pred_nn2 = (nn_model.predict(X_val) > 0.49).astype('int32')
metrics_dict_nn2 = evaluations(y_val, y_pred_nn2)
print_metrics(metrics_dict_nn2)
```

```
Epoch 1/20
6592/6592 [==============================] - 9s 1ms/step - loss: 1.0051 -
accuracy: 0.5459
Epoch 2/20
6592/6592 [==============================] - 8s 1ms/step - loss: 0.6931 -
accuracy: 0.5497
Epoch 3/20
6592/6592 [==============================] - 11s 2ms/step - loss: 0.6902 -
accuracy: 0.5493
Epoch 4/20
6592/6592 [==============================] - 10s 2ms/step - loss: 0.6894 -
accuracy: 0.5500
Epoch 5/20
6592/6592 [==============================] - 11s 2ms/step - loss: 0.6893 -
accuracy: 0.5509
Epoch 6/20
6592/6592 [==============================] - 14s 2ms/step - loss: 0.6883 -
accuracy: 0.5528
```

```
Epoch 7/20
6592/6592 [==============================] - 11s 2ms/step - loss: 0.6883 -
accuracy: 0.5536
Epoch 8/20
6592/6592 [==============================] - 11s 2ms/step - loss: 0.6888 -
accuracy: 0.5510
Epoch 9/20
6592/6592 [==============================] - 11s 2ms/step - loss: 0.6879 -
accuracy: 0.5548
Epoch 10/20
6592/6592 [==============================] - 10s 2ms/step - loss: 0.6879 -
accuracy: 0.5533
Epoch 11/20
6592/6592 [==============================] - 9s 1ms/step - loss: 0.6881 -
accuracy: 0.5528
Epoch 12/20
6592/6592 [==============================] - 9s 1ms/step - loss: 0.6879 -
accuracy: 0.5529
Epoch 13/20
6592/6592 [==============================] - 12s 2ms/step - loss: 0.6877 -
accuracy: 0.5553
Epoch 14/20
6592/6592 [==============================] - 12s 2ms/step - loss: 0.6876 -
accuracy: 0.5547
Epoch 15/20
6592/6592 [==============================] - 11s 2ms/step - loss: 0.6877 -
accuracy: 0.5535
Epoch 16/20
6592/6592 [==============================] - 13s 2ms/step - loss: 0.6878 -
accuracy: 0.5540
Epoch 17/20
6592/6592 [==============================] - 11s 2ms/step - loss: 0.6875 -
accuracy: 0.5551
Epoch 18/20
6592/6592 [==============================] - 11s 2ms/step - loss: 0.6874 -
accuracy: 0.5553
Epoch 19/20
6592/6592 [==============================] - 11s 2ms/step - loss: 0.6876 -
accuracy: 0.5536
Epoch 20/20
6592/6592 [==============================] - 10s 2ms/step - loss: 0.6874 -
accuracy: 0.5550
2521/2521 [==============================] - 2s 941us/step
Accuracy: 0.9962

Classification Report:
              precision    recall  f1-score   support
```

```
           0       1.00      1.00      1.00     80341
           1       0.00      0.00      0.00       309

    accuracy                           1.00     80650
   macro avg       0.50      0.50      0.50     80650
weighted avg       0.99      1.00      0.99     80650


Confusion Matrix:
[[80341     0]
 [  309     0]]
Precision: 0.0000
Recall: 0.0000
```

```
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision is ill-defined and being set to 0.0 due to no predicted samples. Use
`zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

[23]:
```
## NN with Sigmoid
units = 64
activation = 'sigmoid'
learning_rate = 0.01
l2_strength = 0.2
dropout_rate = 0.2
epochs = 12
batch_size = 64

nn_model = create_regularized_nn_model(units=units, activation=activation,␣
 ↪learning_rate=learning_rate, l2_strength=l2_strength,␣
 ↪dropout_rate=dropout_rate)
```

```
nn_model.fit(X_resampled, y_resampled, epochs=epochs, batch_size=batch_size,␣
↪verbose=1)
y_pred_nn2 = (nn_model.predict(X_val) > 0.9).astype('int32')
metrics_dict_nn2 = evaluations(y_val, y_pred_nn2)
print_metrics(metrics_dict_nn2)
```

```
Epoch 1/12
6592/6592 [==============================] - 10s 1ms/step - loss: 0.7604 -
accuracy: 0.5384
Epoch 2/12
6592/6592 [==============================] - 10s 1ms/step - loss: 0.7577 -
accuracy: 0.5392
Epoch 3/12
6592/6592 [==============================] - 10s 2ms/step - loss: 0.7699 -
accuracy: 0.5387
Epoch 4/12
6592/6592 [==============================] - 11s 2ms/step - loss: 0.7517 -
accuracy: 0.5391
Epoch 5/12
6592/6592 [==============================] - 11s 2ms/step - loss: 0.7606 -
accuracy: 0.5385
Epoch 6/12
6592/6592 [==============================] - 11s 2ms/step - loss: 0.7550 -
accuracy: 0.5393
Epoch 7/12
6592/6592 [==============================] - 13s 2ms/step - loss: 0.7526 -
accuracy: 0.5390
Epoch 8/12
6592/6592 [==============================] - 12s 2ms/step - loss: 0.7552 -
accuracy: 0.5389
Epoch 9/12
6592/6592 [==============================] - 12s 2ms/step - loss: 0.7630 -
accuracy: 0.5384
Epoch 10/12
6592/6592 [==============================] - 14s 2ms/step - loss: 0.7479 -
accuracy: 0.5379
Epoch 11/12
6592/6592 [==============================] - 12s 2ms/step - loss: 0.7512 -
accuracy: 0.5392
Epoch 12/12
6592/6592 [==============================] - 12s 2ms/step - loss: 0.7545 -
accuracy: 0.5386
2521/2521 [==============================] - 3s 1ms/step
Accuracy: 0.9962

Classification Report:
              precision    recall  f1-score   support
```

```
           0       1.00      1.00      1.00     80341
           1       0.00      0.00      0.00       309

    accuracy                           1.00     80650
   macro avg       0.50      0.50      0.50     80650
weighted avg       0.99      1.00      0.99     80650


Confusion Matrix:
[[80341     0]
 [  309     0]]
Precision: 0.0000
Recall: 0.0000
```

```
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision is ill-defined and being set to 0.0 due to no predicted samples. Use
`zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

[24]:
```
# Initial XGBoost

xgb_model1 = XGBClassifier(objective='binary:logistic', eval_metric='logloss',␣
 ↪max_depth=3, learning_rate=0.1, n_estimators=100)
xgb_model1.fit(X_resampled, y_resampled)
y_pred_xgb1 = xgb_model1.predict(X_val)

metrics_dict_xgb1 = evaluations(y_val, y_pred_xgb1)
print_metrics(metrics_dict_xgb1)
```

```
Accuracy: 0.9162

Classification Report:
```

```
              precision    recall  f1-score   support

           0       1.00      0.92      0.96     80341
           1       0.03      0.64      0.06       309

    accuracy                           0.92     80650
   macro avg       0.51      0.78      0.51     80650
weighted avg       0.99      0.92      0.95     80650


Confusion Matrix:
[[73693  6648]
 [  112   197]]
Precision: 0.0288
Recall: 0.6375
```

[25]:
```
## XGBoost Grid Search
param_grid = {
    'max_depth': [3, 6, 10],
    'learning_rate': [0.01, 0.1, 0.3],
    'n_estimators': [100, 200, 300]
}

# basic model
xgb_model = XGBClassifier(objective='binary:logistic', eval_metric='logloss')

# grid search
grid_search_xgb = GridSearchCV(estimator=xgb_model, param_grid=param_grid,␣
 ↪scoring='recall', cv=3, verbose=1)
grid_search_xgb.fit(X_resampled, y_resampled)

print("Best Parameters:", grid_search_xgb.best_params_)
print("Best Score:", grid_search_xgb.best_score_)

#validation set
best_model_XGB = grid_search_xgb.best_estimator_
y_pred_best_xgb = best_model_XGB.predict(X_val)
metrics_dict_best_xgb = evaluations(y_val, y_pred_best_xgb)
print_metrics(metrics_dict_best_xgb)
```

```
Fitting 3 folds for each of 27 candidates, totalling 81 fits
Best Parameters: {'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 200}
Best Score: 0.9984906219506264
Accuracy: 0.9879

Classification Report:
              precision    recall  f1-score   support
```

```
           0       1.00      0.99      0.99     80341
           1       0.21      0.75      0.32       309

    accuracy                           0.99     80650
   macro avg       0.60      0.87      0.66     80650
weighted avg       1.00      0.99      0.99     80650


Confusion Matrix:
[[79442   899]
 [   76   233]]
Precision: 0.2058
Recall: 0.7540
```

[41]:
```
##Random Forest sampled

rf_no = RandomForestClassifier(random_state=42, n_jobs=-1)
rf_no.fit(X_resampled, y_resampled)
y_pred_rfno = rf_no.predict(X_val)
metrics_val_rfno = evaluations(y_val, y_pred_rfno)
print_metrics(metrics_val_rfno)
```

```
Accuracy: 0.9888

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.99      0.99     80341
           1       0.22      0.73      0.33       309

    accuracy                           0.99     80650
   macro avg       0.61      0.86      0.66     80650
weighted avg       1.00      0.99      0.99     80650


Confusion Matrix:
[[79516   825]
 [   82   227]]
Precision: 0.2158
Recall: 0.7346
```

[27]:
```
##Random Forest - max features log2

rf_1 = RandomForestClassifier(max_features = 'log2', random_state=42, n_jobs=-1)
rf_1.fit(X_resampled, y_resampled)
y_pred_rf1 = rf_1.predict(X_val)
metrics_val_rf1 = evaluations(y_val, y_pred_rf1)
print_metrics(metrics_val_rf1)
```

```
Accuracy: 0.9888

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.99      0.99     80341
           1       0.22      0.73      0.33       309

    accuracy                           0.99     80650
   macro avg       0.61      0.86      0.66     80650
weighted avg       1.00      0.99      0.99     80650


Confusion Matrix:
[[79516   825]
 [   82   227]]
Precision: 0.2158
Recall: 0.7346
```

[28]:
```
##Random Forest - max features sqrt

rf_2 = RandomForestClassifier(max_features = 'sqrt', random_state=42, n_jobs=-1)
rf_2.fit(X_resampled, y_resampled)
y_pred_rf2 = rf_2.predict(X_val)
metrics_val_rf2 = evaluations(y_val, y_pred_rf2)
print_metrics(metrics_val_rf2)
```

```
Accuracy: 0.9888

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.99      0.99     80341
           1       0.22      0.73      0.33       309

    accuracy                           0.99     80650
   macro avg       0.61      0.86      0.66     80650
weighted avg       1.00      0.99      0.99     80650


Confusion Matrix:
[[79516   825]
 [   82   227]]
Precision: 0.2158
Recall: 0.7346
```

[29]:
```
### the max features is producing the exact same results so I am going with␣
↪just the default one - sqrt
```

```
[31]: ##Random Forest Model with first Grid Search

param_grid = {
    'min_samples_leaf': [1, 2, 4],
    'criterion': ['gini', 'entropy']
}

##I have done check on min samples leaf and it came out as 1 and min samples␣
 ↪leaf and it came out 10 so now I am
## doing min samples split on 10,15,20 and adding a parameter grid search for␣
 ↪number of estimators
##the second one was on min sample split and n estimators from which both␣
 ↪highest values were decided so im going
## even higher
    ##Best Parameters: {'min_samples_split': 25, 'n_estimators': 300}}

scoring = {
    'accuracy': 'accuracy',
    'precision': 'precision',
    'recall': 'recall',
    'f1': 'f1'}

rf_gs1 = RandomForestClassifier(random_state=42, n_jobs=-1, min_samples_leaf=1)
grid_search_rf1 = GridSearchCV(rf_gs1, param_grid,cv=3, scoring=scoring,␣
 ↪n_jobs=-1, refit='recall')
grid_search_rf1.fit(X_resampled, y_resampled)

best_params_rf1 = grid_search_rf1.best_params_
best_rf1 = grid_search_rf1.best_estimator_

y_pred_val1 = best_rf1.predict(X_val)
metrics_val1 = evaluations(y_val, y_pred_val1)

print(f"Best Parameters: {best_params_rf1}")
print("Validation Metrics with Best Model:")
print_metrics(metrics_val1)

##the last grid search was with 300 n-estimators and 25 min samples split
```

```
Best Parameters: {'criterion': 'entropy', 'min_samples_leaf': 4}
Validation Metrics with Best Model:
Accuracy: 0.9878

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.99      0.99     80341
           1       0.20      0.72      0.31       309

    accuracy                           0.99     80650
   macro avg       0.60      0.86      0.65     80650
weighted avg       1.00      0.99      0.99     80650


Confusion Matrix:
[[79442   899]
 [   85   224]]
Precision: 0.1995
Recall: 0.7249
```

[32]:
```
##Random Forest Model with Grid Search

param_grid = {
    'n_estimators': [200,300,350],
    'min_samples_split': [20, 25, 35]}

scoring = {
    'accuracy': 'accuracy',
    'precision': 'precision',
    'recall': 'recall',
    'f1': 'f1'}

rf_gs2 = RandomForestClassifier(random_state=42, n_jobs=-1, min_samples_leaf=1)
grid_search_rf2 = GridSearchCV(rf_gs2, param_grid,cv=3, scoring=scoring,␣
 ↪n_jobs=-1, refit='recall')
grid_search_rf2.fit(X_resampled, y_resampled)

best_params_rf2 = grid_search_rf2.best_params_
best_rf2 = grid_search_rf2.best_estimator_

y_pred_val2 = best_rf2.predict(X_val)
metrics_val2 = evaluations(y_val, y_pred_val2)

print(f"Best Parameters: {best_params_rf2}")
print("Validation Metrics with Best Model:")
print_metrics(metrics_val2)

##the last grid search was with 300 n-estimators and 25 min samples split
```

```
Best Parameters: {'min_samples_split': 25, 'n_estimators': 200}
Validation Metrics with Best Model:
Accuracy: 0.9888

Classification Report:
              precision    recall  f1-score   support
```

```
           0       1.00      0.99      0.99     80341
           1       0.21      0.70      0.32       309

    accuracy                           0.99     80650
   macro avg       0.60      0.85      0.66     80650
weighted avg       1.00      0.99      0.99     80650


Confusion Matrix:
[[79530   811]
 [   92   217]]
Precision: 0.2111
Recall: 0.7023
```

[33]:
```
### FINAL MODELS
```

[34]:
```
## Neural Network

# relu and low learning rate
units = 64
activation = 'relu'
learning_rate = 0.001
l2_strength = 0.01
dropout_rate = 0.3
epochs = 20
batch_size = 128

nn_model = create_regularized_nn_model(units=units, activation=activation,␣
 ↪learning_rate=learning_rate, l2_strength=l2_strength,␣
 ↪dropout_rate=dropout_rate)
nn_model.fit(X_resampled, y_resampled, epochs=epochs, batch_size=batch_size,␣
 ↪verbose=1)
y_pred_nn = (nn_model.predict(X_test) > 0.35).astype('int32')
metrics_dict_nn = evaluations(y_test, y_pred_nn)
print_metrics(metrics_dict_nn)
```

```
Epoch 1/20
3296/3296 [==============================] - 6s 2ms/step - loss: 399.3007 -
accuracy: 0.5185
Epoch 2/20
3296/3296 [==============================] - 4s 1ms/step - loss: 2.3556 -
accuracy: 0.5364
Epoch 3/20
3296/3296 [==============================] - 4s 1ms/step - loss: 0.7167 -
accuracy: 0.5957
Epoch 4/20
3296/3296 [==============================] - 4s 1ms/step - loss: 0.6910 -
```

```
accuracy: 0.5955
Epoch 5/20
3296/3296 [==============================] - 4s 1ms/step - loss: 0.6860 -
accuracy: 0.5775
Epoch 6/20
3296/3296 [==============================] - 4s 1ms/step - loss: 0.6880 -
accuracy: 0.5557
Epoch 7/20
3296/3296 [==============================] - 3s 1ms/step - loss: 0.7184 -
accuracy: 0.5555
Epoch 8/20
3296/3296 [==============================] - 3s 1ms/step - loss: 0.7006 -
accuracy: 0.5556
Epoch 9/20
3296/3296 [==============================] - 4s 1ms/step - loss: 0.6871 -
accuracy: 0.5556
Epoch 10/20
3296/3296 [==============================] - 3s 1ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 11/20
3296/3296 [==============================] - 4s 1ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 12/20
3296/3296 [==============================] - 4s 1ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 13/20
3296/3296 [==============================] - 3s 1ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 14/20
3296/3296 [==============================] - 3s 1ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 15/20
3296/3296 [==============================] - 3s 1ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 16/20
3296/3296 [==============================] - 3s 1ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 17/20
3296/3296 [==============================] - 4s 1ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 18/20
3296/3296 [==============================] - 4s 1ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 19/20
3296/3296 [==============================] - 4s 1ms/step - loss: 0.6870 -
accuracy: 0.5556
Epoch 20/20
3296/3296 [==============================] - 3s 1ms/step - loss: 0.6870 -
```

```
accuracy: 0.5556
2521/2521 [==============================] - 2s 630us/step
Accuracy: 0.0034

Classification Report:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00     80374
           1       0.00      1.00      0.01       277

    accuracy                           0.00     80651
   macro avg       0.00      0.50      0.00     80651
weighted avg       0.00      0.00      0.00     80651


Confusion Matrix:
[[    0 80374]
 [    0   277]]
Precision: 0.0034
Recall: 1.0000
```

```
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/Users/oladurska/opt/anaconda3/lib/python3.9/site-
packages/sklearn/metrics/_classification.py:1248: UndefinedMetricWarning:
Precision and F-score are ill-defined and being set to 0.0 in labels with no
predicted samples. Use `zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

[35]:
```
## XGBoost

print("Best Parameters:", grid_search_xgb.best_params_)
print("Best Score:", grid_search_xgb.best_score_)
best_model_XGB = grid_search_xgb.best_estimator_
y_pred_best_xgb = best_model_XGB.predict(X_test)
metrics_dict_best_xgb = evaluations(y_test, y_pred_best_xgb)
print_metrics(metrics_dict_best_xgb)
```

```
Best Parameters: {'learning_rate': 0.1, 'max_depth': 10, 'n_estimators': 200}
Best Score: 0.9984906219506264
Accuracy: 0.9877
```

```
Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.99      0.99     80374
           1       0.19      0.77      0.30       277

    accuracy                           0.99     80651
   macro avg       0.59      0.88      0.65     80651
weighted avg       1.00      0.99      0.99     80651


Confusion Matrix:
[[79441   933]
 [   63   214]]
Precision: 0.1866
Recall: 0.7726
```

[36]:
```
## Random Forest

print(f"Best Parameters: {best_params_rf1}")
print(f"Best Parameters: {best_params_rf2}")
```

```
Best Parameters: {'criterion': 'entropy', 'min_samples_leaf': 4}
Best Parameters: {'min_samples_split': 25, 'n_estimators': 200}
```

[38]:
```
## Input the values for the final model
rf_final = RandomForestClassifier(max_features = 'sqrt', n_estimators = 200,␣
 ↪criterion = 'entropy', min_samples_split = 25, min_samples_leaf= 4,␣
 ↪random_state=42, n_jobs=-1)
rf_final.fit(X_resampled, y_resampled)
y_pred_rf_final = rf_final.predict(X_test)
metrics_val_rf_final = evaluations(y_test, y_pred_rf_final)
print_metrics(metrics_val_rf_final)
```

```
Accuracy: 0.9874

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.99      0.99     80374
           1       0.18      0.75      0.29       277

    accuracy                           0.99     80651
   macro avg       0.59      0.87      0.64     80651
weighted avg       1.00      0.99      0.99     80651


Confusion Matrix:
```

```
[[79426   948]
 [   68   209]]
Precision: 0.1806
Recall: 0.7545
```