Bank Marketing

Fan Jia Jiawei Li Strahinja Trenkic Qiqi Zhou

Contents

Introduction

Understanding the Project

A retail bank has a customer call centre, through which the bank communicates with potential new clients and offers term deposits . Term deposits are defined as

> "A term deposit is a fixed-term investment that includes the deposit of money into an account at a financial institution. Term deposit investments usually carry short-term maturities ranging from one month to a few years and will have varying levels of required minimum deposits."

It is obvious that such an instrument would generate revenue for the bank, hence the bank records the outcomes of these phone calls along side other data related to the person being called, the economic benchmarks at the time of the call and certain parameters of previous contact with the given person. The motivation behind the research is clear, by analysing previous phone calls the bank would like to improve it's results in two dimensions:

1. The efficiency dimension, or in other words how to reduce the number of phone calls the bank is performing and therefor reduce the costs associated with telemarketing
2. The effectiveness dimension, or in other words how to potentially improve the result and get more clients or at least the same number to deposit their money with our bank

In order to fully understand the objectives at hand we must also appreciate the economic and historic context from which we draw our data. It is the year 2008 and the world is about to be plunged in the deepest economic recession since the 1930s. After the credit default of The Lehman Brothers the financial world was shook to its core, and governments as well as central banks started unprecedented levels of quantitative easing in order to save jobs and revitalize the economy.

As can be seen on the above picture this caused a huge spike in the famous TED spread one of the leading indicators for credit risk, since the difference between

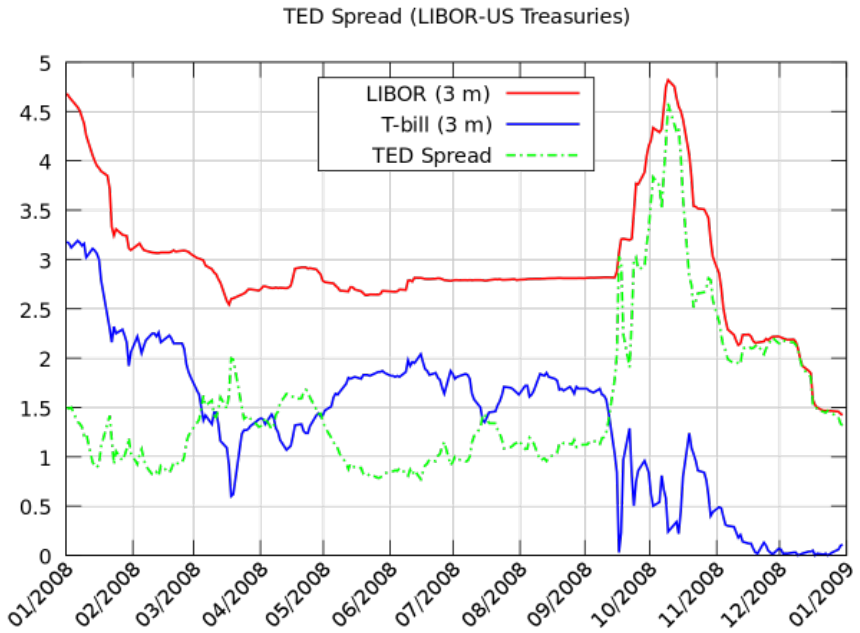


Figure 1: Spread

a riskless investment (T-Bill) and the risk banks transfer when loaning to each other. It takes no further economic knowledge to conclude that such an event will have major impact not only on the result of the telemarketing campaign but the representability and timelessness of the data connected through it.

At this point in time it would be a good idea to raise a few questions that we plan on answering through this paper.

1. *How did the economic crisis affect consumer behaviour and how did it manifest itself in the data ?*
2. *How does one's education, marital status, job… affect their economic choices ?*
3. *Do people better respond to frequent calls, or first contact ?*
4. *Do they prefer being called on the mobile phone or landline ?*

And ultimately can we develop a data driven, machine learning approach to tackle this problem and answer the outlined questions?

Overview of Features

At our disposal we had 20 input and 1 output feature to work with. The input features were categorized and given to us as follows:

A. Bank client data:

`age`: age (numeric)

`job`: type of job (categorical: 'admin.', 'blue-collar', 'entrepreneur', 'housemaid', 'management', 'retired', 'self-employed', 'services', 'student', 'technician', 'unemployed', 'unknown')

`marital`: marital status (categorical: 'divorced', 'married', 'single', 'unknown'; note: 'divorced' means divorced or widowed)

`education`: (categorical: 'basic.4y', 'basic.6y', 'basic.9y', 'high.school', 'illiterate', 'professional.course', 'university.degree', 'unknown')

`default`: has credit in default? (categorical: 'no', 'yes', 'unknown')

`housing`: has housing loan? (categorical: 'no', 'yes', 'unknown')

`loan`: has personal loan? (categorical: 'no', 'yes', 'unknown')

B. Related with the last contact of the current campaign:

`contact`: contact communication type (categorical: 'cellular', 'telephone')

`month`: last contact month of year (categorical: 'jan', 'feb', 'mar', ..., 'nov', 'dec')

`day_of_week`: last contact day of the week (categorical: 'mon', 'tue', 'wed', 'thu', 'fri')

`duration`: last contact duration, in seconds (numeric). Important note: this attribute highly affects the output target (e.g., if duration=0 then y='no'). Yet, the duration is not known before a call is performed. Also, after the end of the call y is obviously known. Thus, this input should only be included for benchmark purposes and should be discarded if the intention is to have a realistic predictive model.

C. Other attributes:

`campaign`: number of contacts performed during this campaign and for this client (numeric, includes last contact)

`pdays`: number of days that passed by after the client was last contacted from a previous campaign (numeric; 999 means client was not previously contacted)

`previous`: number of contacts performed before this campaign and for this client (numeric)

`poutcome`: outcome of the previous marketing campaign (categorical: 'failure', 'nonexistent', 'success')

D. Social and economic context attributes

`emp.var.rate`: employment variation rate, quarterly indicator (numeric)

`cons.price.idx`: consumer price index, monthly indicator (numeric)

`cons.conf.idx`: consumer confidence index, monthly indicator (numeric)

`euribor3m`: euribor 3 month rate, daily indicator (numeric)

`nr.employed`: number of employees, quarterly indicator (numeric)

Output variable (desired target):

y: has the client subscribed a term deposit? (binary: 'yes', 'no')

The obvious challenge is the large number of categorical and or Boolean features, how to approach them and handle them appropriately.

Exploratory Data Analysis

Exploratory Data Analysis is a process to explore the dataset with no assumptions or hypotheses using non-graphical and graphical, univariate and multivariate methods. The objective is to gain intuitive insights, discover distribution characteristics, and find out missing values in the dataset.

Explorations of the Target Variable Y

The first thing we investigate is the target variable Y, which is a Y/N binary variable measuring the campaign outcome, representing whether a client has subscribed to a long-term deposit or not during our campaign.

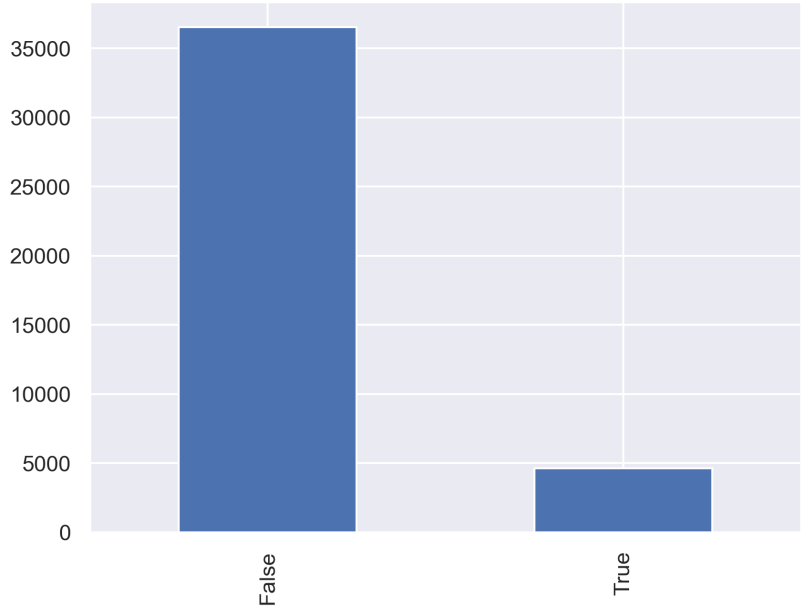


Figure 2: Distribution of camaign outcome Y.

There are more than 40,000 observations in our dataset, and only 11.3% of them have positive "YES" outcomes, which means that we have a significantly unbalanced dataset. Since our data was collected during the 2008 financial crisis, we pay particular attention to this time factor and visualize the positive Y values across that specific time frame.

In the graph above, the thin orange line indicates the outbreak of the financial crisis. We can see a massive surge in positive campaign outcomes afterward,

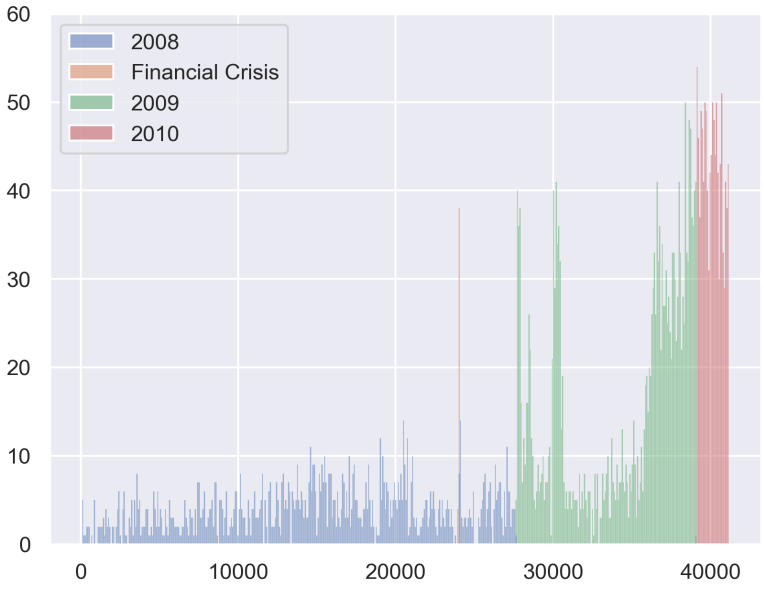


Figure 3: Uneven distribution of positive outcome during the financial crisis.

meaning people were actively taking advantage of certain economic factors, such as lower interest rates. We can also see a steady growth of the positive outcome rate since July 2009 from the graph below.

Highly relevant to the crisis are the five economic indicators in our dataset, displayed as above, which show significant predicting power in almost all of our models. In 2008, all of them went down first, but the consumer confidence index (green line) was the leading recovery factor, followed by the consumer price index (orange line). The recovery of the positive outcome rate of our campaign started at the same time. Furthermore, the drop of interest rates captured by the Euribor 3-month rate (red line) significantly correlated with the campaign success.

Missing Values

We use the `info()` function to get an overview of our data and find out many missing values, especially for features like Pdays and Poutcome, in which 90% of the rows have missing values, as shown in the bar chart below. These missing values require a lot of special attention during the feature engineering process.

Feature Explorations

Age

For client profile features, first, we plot the distribution of age relative to Y, as shown below.30-50-year-old people are the majority in our dataset. People in

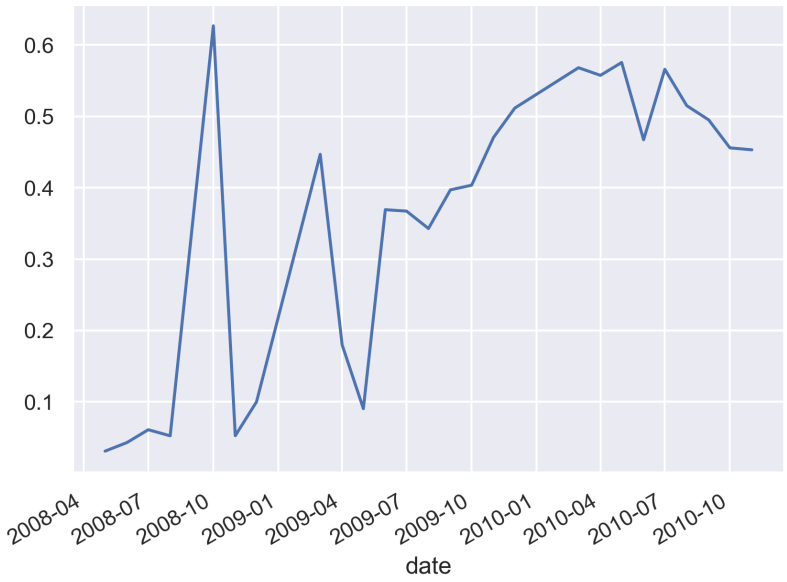


Figure 4: Positive outcome rates by month.

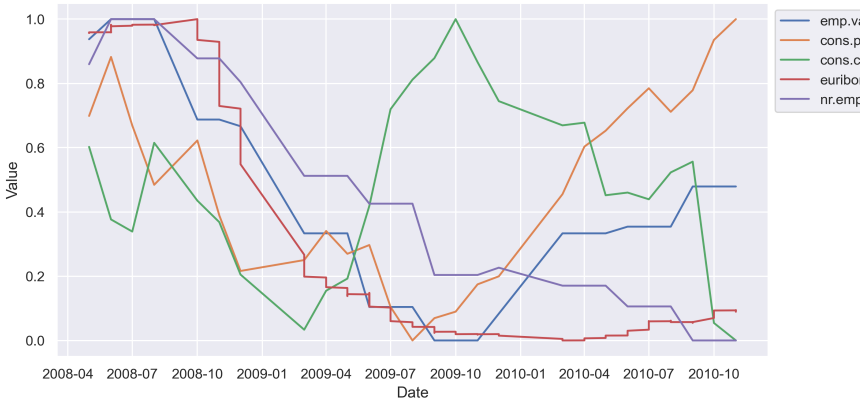


Figure 5: Five economic indicators.

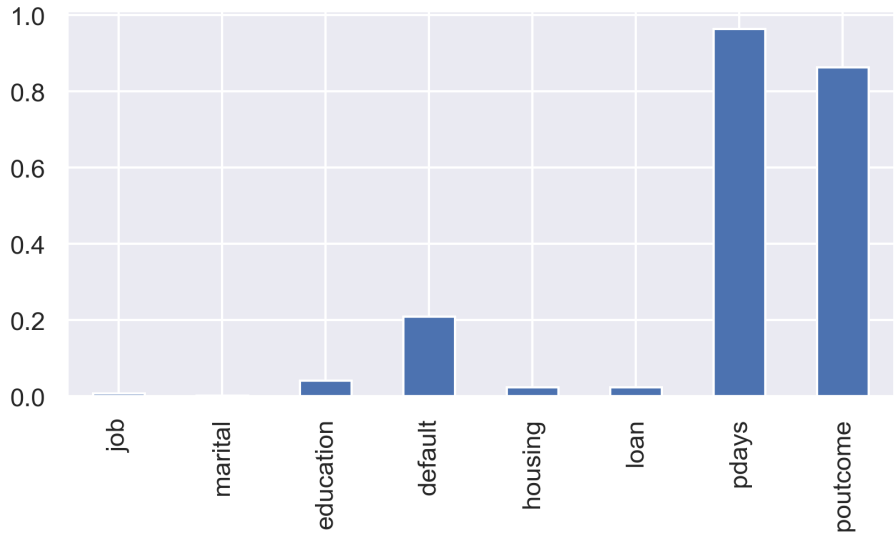
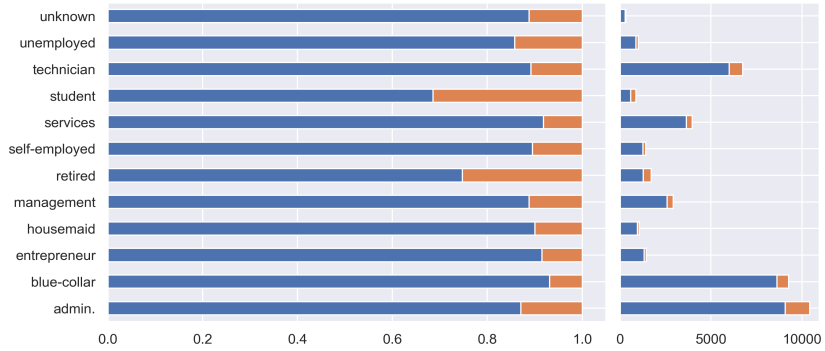


Figure 6: Percentage of missing values.

their 30s and people older than 60 are more likely to give positive responses to the deposit. There are not many outrageous outliers, therefore we keep all data for Age.

Job

This next graph shows the outcome Y percentage in each job group, with the orange color denoting positive outcomes. On the right is the instance count distribution of each group. There is a large percentage of technicians, blue-collar workers and admins. However, it is students and retired people that are most likely to say 'YES' to the long-term deposit.



Education

In terms of Education, although most people in our dataset have above-high-school education, the groups that are most likely to respond positively are the

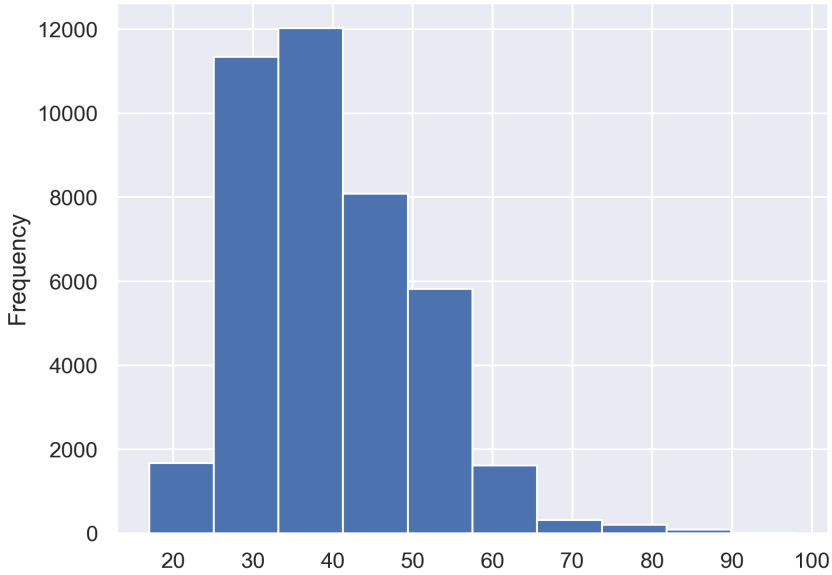


Figure 7: Age distribution histogram.

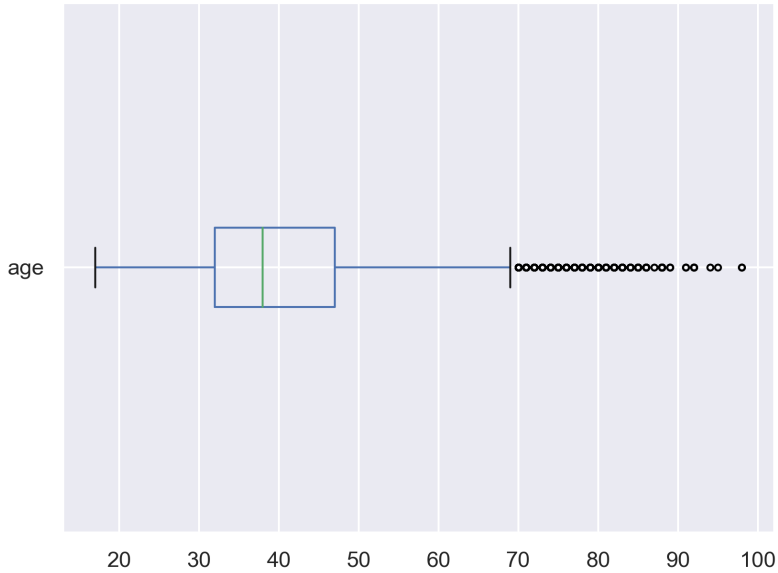


Figure 8: Age distribution box-plot.

least and the most educated.

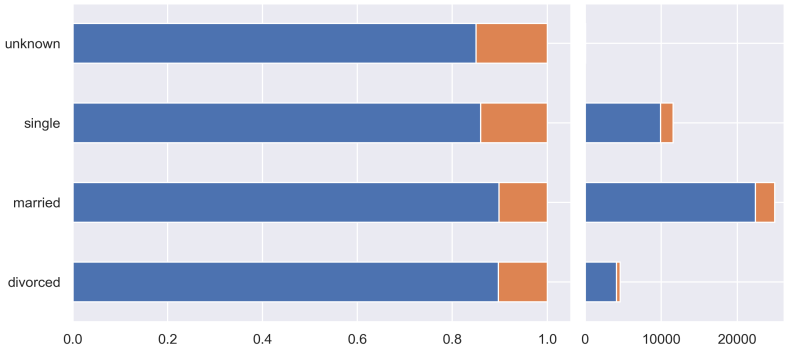


Figure 9: Outcome percentage and distribution by Education.

Default

Default is a peculiar feature that captures whether people previously had credit default, and it is a highly sensitive piece of information. As presented in the figure below, the majority of clients declare no default record, however, 8600 instances are "unknown". Given the private nature of this feature, we believe there are hidden stories open for interpretations behind the "unknown" values and they might influence people's financial decisions. Therefore, we decide to treat the unknown values as an individual category and let it speak for itself.

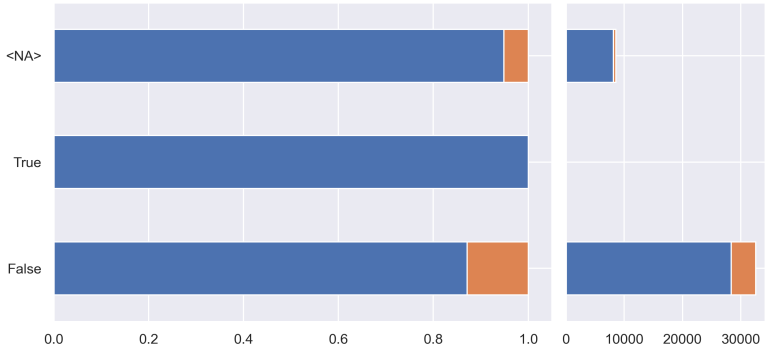


Figure 10: Outcome percentage and distribution by Default.

Contact

For features capturing characteristics of current campaign, Contact is an essential binary feature that represents the tool used to contact clients. And we see a robust positive outcome rate for cellular usage.

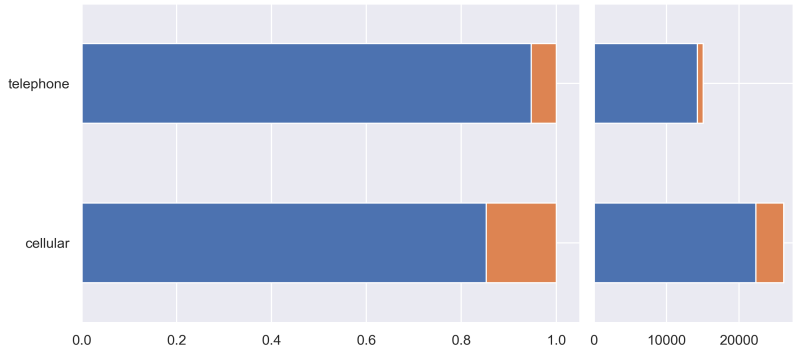


Figure 11: Outcome percentage and distribution by Contact.

Month

With regard to the months in which the last contact was made, the distribution concentrates in summer, however, these months have the lowest positive outcome rates. In months with fewer contacts, there seem to be higher success rates of securing a long-term deposit with the clients.

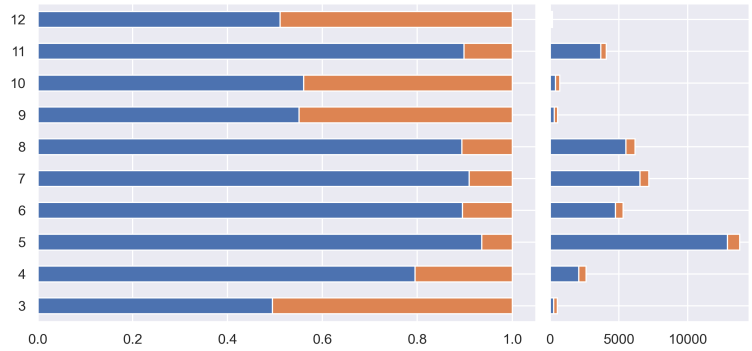
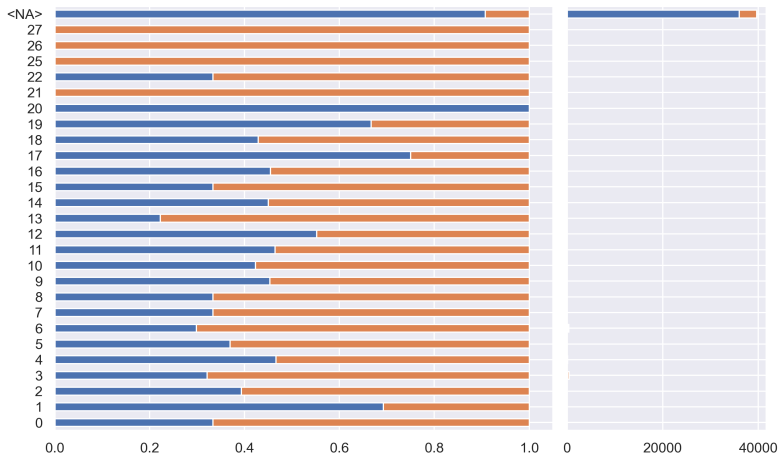


Figure 12: Outcome percentage and distribution by Month.

Pdays

Next is the most challenging feature, Pdays, which represents the number of days passed since the last contact with a client. It has almost 40,000 missing values. As demonstrated in the graph below, the "Na" category on the top dominates the entire distribution. Simultaneously, the other 1500 rows that do have values seem to show positive relationships with the outcome. We spent a significant amount of time and effort to deal with this feature, and this process will be discussed in the feature engineering section.



Previous

The Previous feature is also very challenging, which measures the number of contacts performed to a client before this campaign. It has 36,000 missing values, and for those observations with actual values, their relationships with the outcome are seemingly positive.

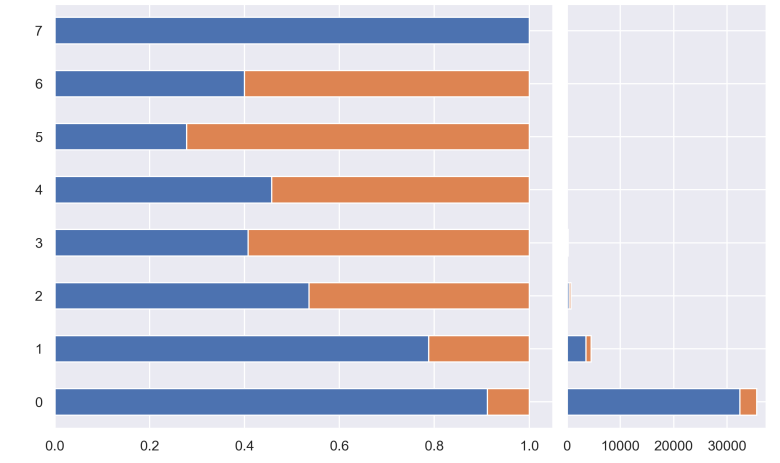


Figure 13: Outcome percentage and distribution by Previous.

Poutcome

This is the feature that reports the outcomes of the previous campaign. Over 35,000 observations have missing values, as presented below. However, when

combining this featuring with Pdays and Previous, there seem to be some contradictory stories.

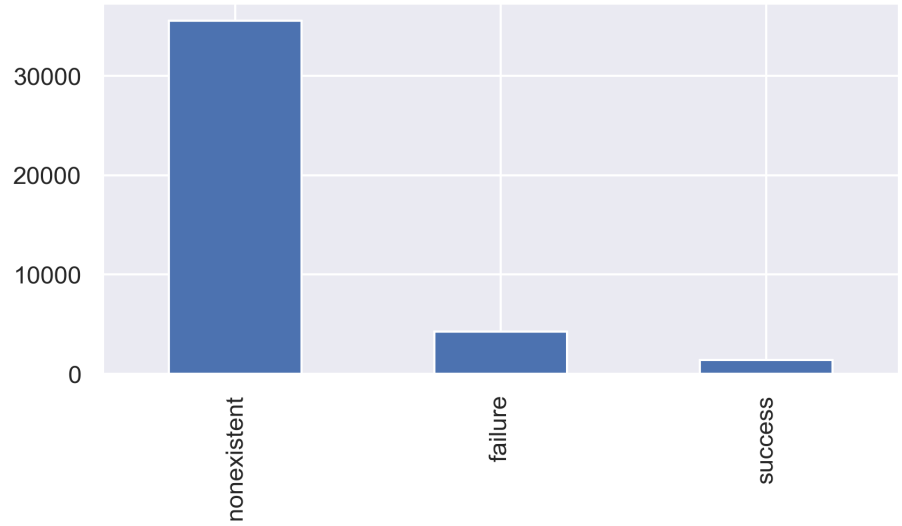


Figure 14: Poutcome distribution.

Missing values in Pdays mean that the clients were not previously contacted and therefore should not have values in Poutcome. But Poutcome has fewer missing values than Pdays dose, as seen in the second graph below. We print out the 4110 rows where clients have not been contacted but have Poutcome values and see how many times they have been contacted before. The results suggest that maybe these clients have been actually contacted, but it was more than 30 days ago, thus the contact date was not recorded.

Multivariate Explorations

Client Data

Furthermore, we explore some multivariate distributions of the positive outcome. First, we found both married and divorced retired people respond positively to our campaign, and single and divorced students are even more enthusiastic. It is also quite interesting that students, retired and illiterate people are more likely to say 'yes' to our long-term deposit. Additionally, divorced illiterate people respond to our campaign extremely well.

Key Numerical Features

We also made a scatterplot across important quantitative features, with the outcome variable denoted by two colors.

Correlation Heatmap

With this heatmap, we get a better look at the correlations among features. Four out of five economic indicators have strong correlations with each other.

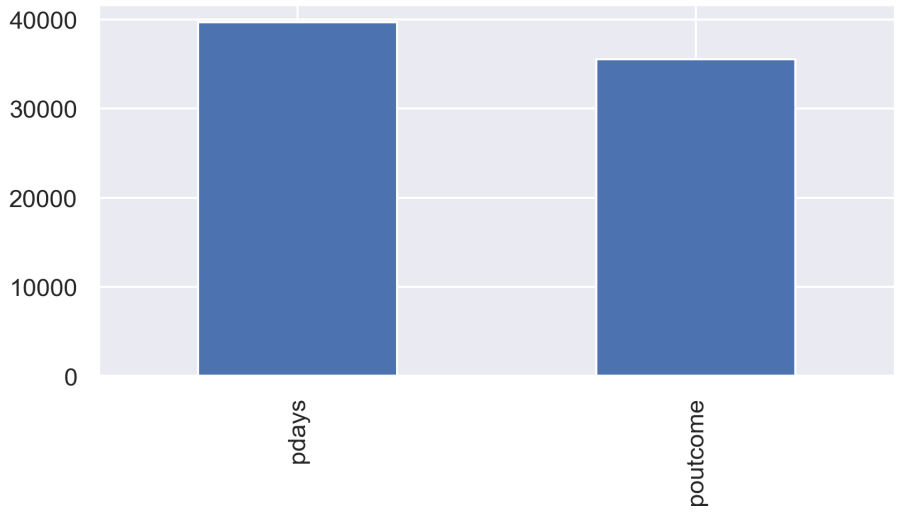


Figure 15: Pdays and Poutcome.

Figure 16: Positive Outcome Percentage by Job and Marital Status.

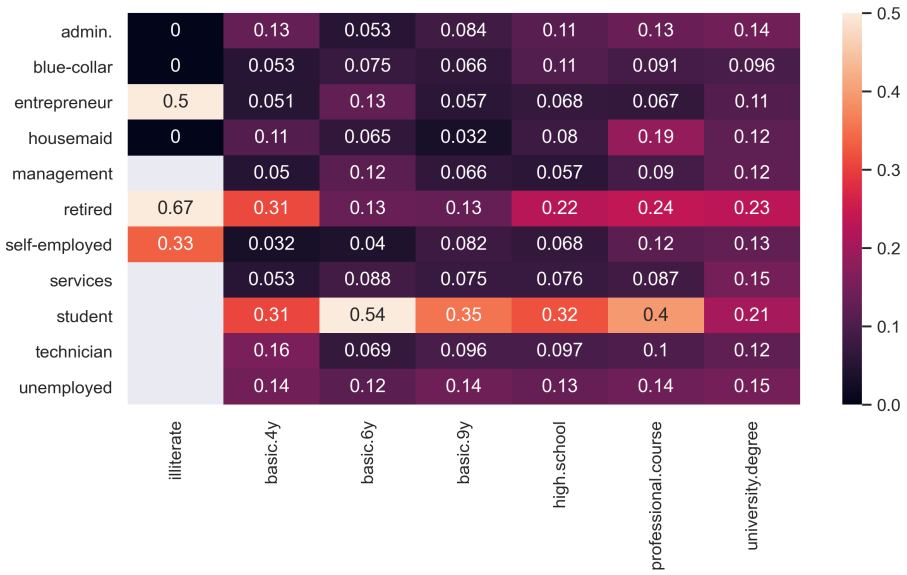


Figure 17: Positive Outcome Percentage by Job and Education.

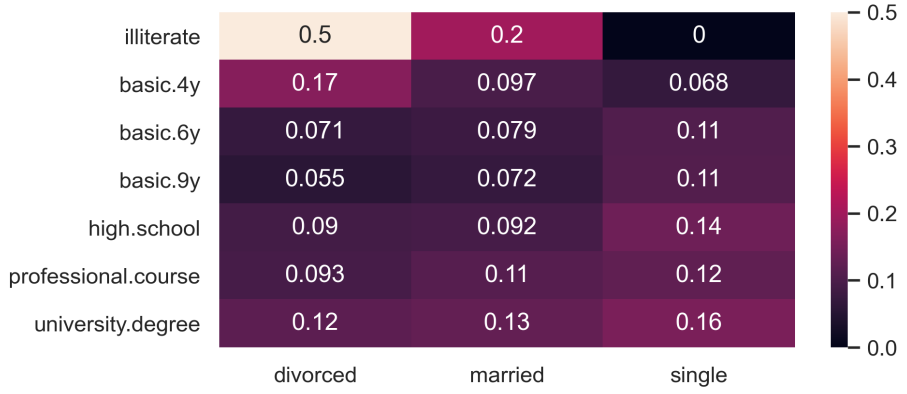


Figure 18: Positive Outcome Percentage by Education and Marital Status.

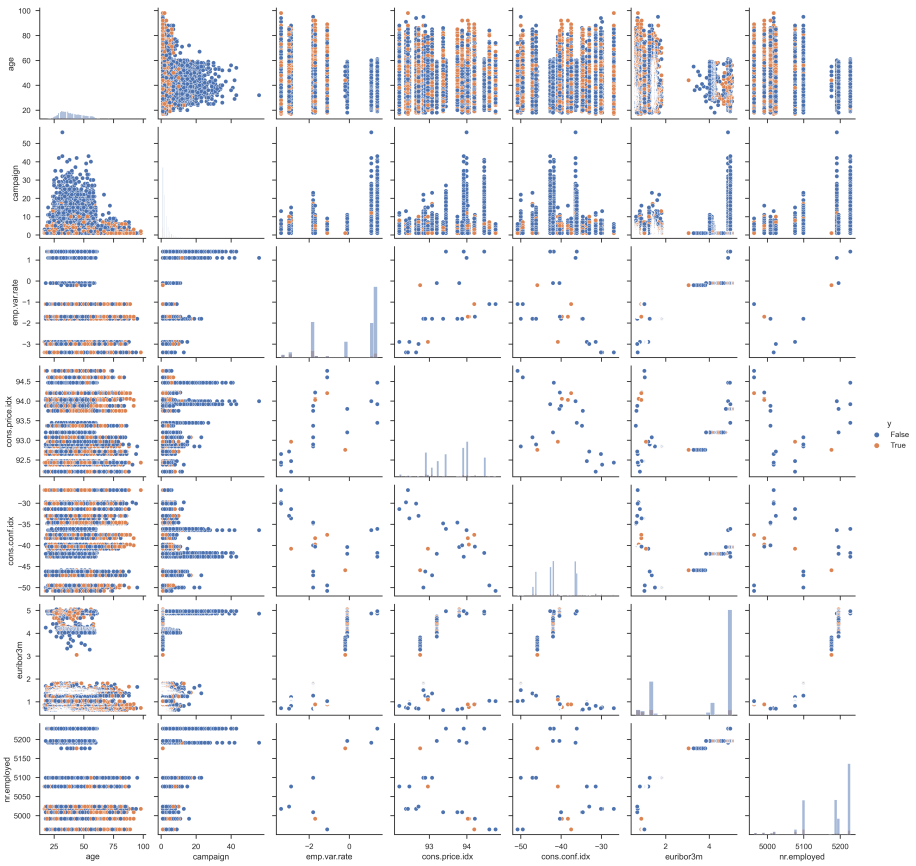


Figure 19: Scatterplots of key numerical features.

We are worried about collinearity and try many ways to deal with these features, such as deletion or transformation, but all efforts lead to relatively poor model results. Then we realize that they are probably very important features in our dataset, so we keep them for the moment. In addition, Some features show great correlations with the outcome, such as Previous and Poutcome. We try to use PCA on the entire dataset to avoid collinearity, but again, all efforts lead to poor model results. Therefore, we decide to keep all features and make changes if needed for specific models.

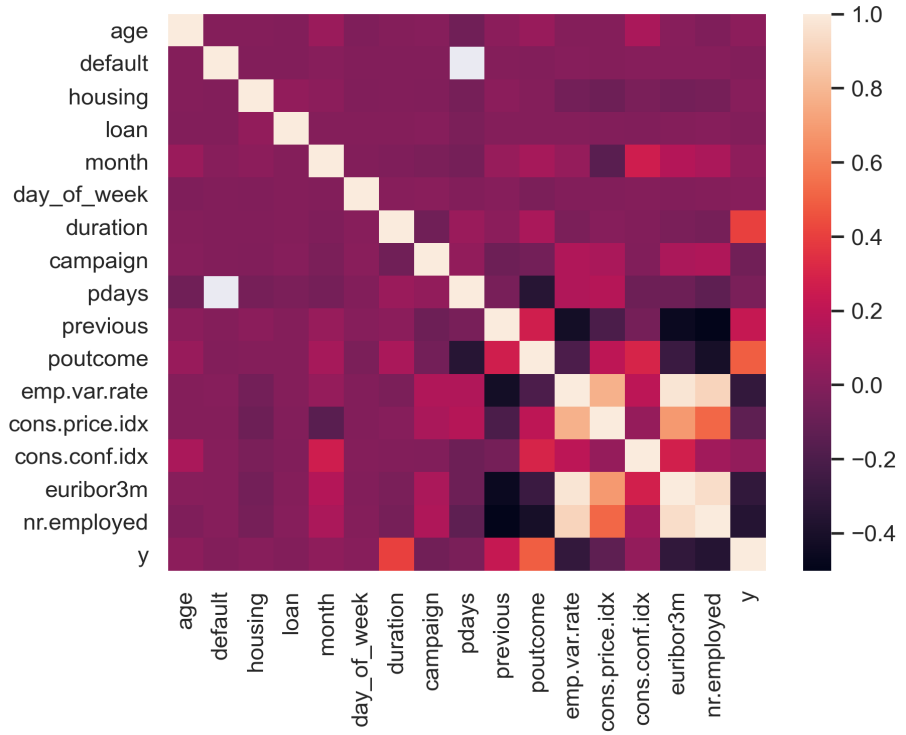


Figure 20: Correlation heatmap.

Early Feature Engineering Attempts

The final figure shows a memorable snapshot of our early feature engineering attempts, with each row being one version of feature engineering. We have tried many ways to handle the data and used the same basic model to compare the results. With this spreadsheet, we can compare each others' ideas and results before we finally decide to settle down with one version, which will be discussed next.

Data Preparation and Pipelines

After inspecting and exploring the dataset, this chapter focuses on data preparation and pipelines which build the foundation for the following chapters. Data

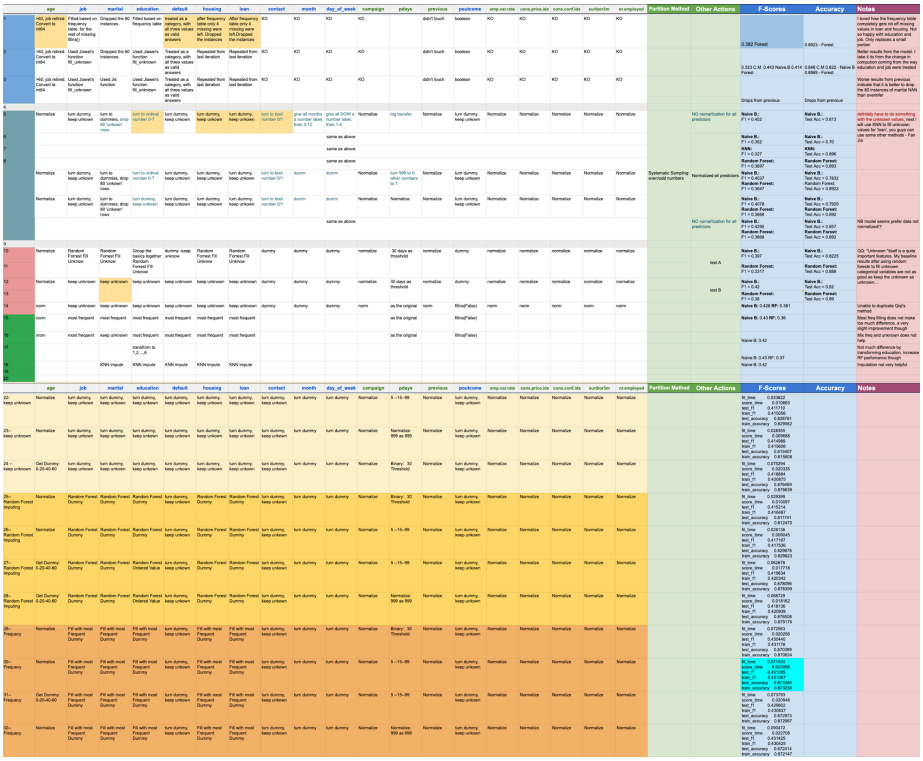


Figure 21: Early feature engineering attempts.

preparation or data transfromation is a process that transforms the data into proper types, shapes and sets. For example, categorical data are usually stored as strings in the original dataset and can not be fitted in `scikit-learn` models. To solve this issue, We need to utilise `pandas` or `scikit-learn` to transform such data into integers. It should be noted that simply transforming data in a `jupyter` notebook is not a best practice because such code is not reusable and readable. In our analytical practice, we wrap data preparation code into documented and modular `python` functions and `scikit-learn` pipelines. Such practice ensures that all collaborators can build and evaluate models from the same ground. The high level overview of the data preparation and pipeline is shown in the flow chart. The detailed code implementation will be provided in the appendix section.

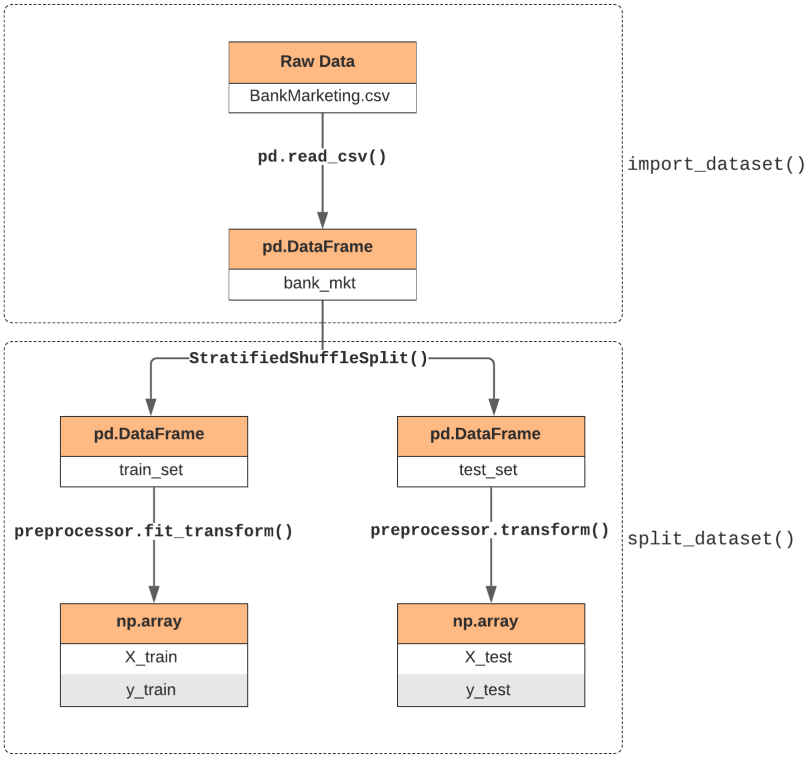


Figure 22: Data Life Cycle

Import Data

First, the `BankMarkteting.csv` file is imported using `read_csv()` from `pandas` and duplicated rows, missing values, categorical and boolean data are properly processed. The import process is abstracted into the function `import_dataset()`. A sample code is shown below:

```
import pandas as pd

# Import data from csv file
bank_mkt = pd.read_csv(
    "../data/BankMarketing.csv",
    na_values=["unknown", "nonexistent"],
    true_values=["yes", "success"],
    false_values=["no", "failure"],
)
# Treat pdays = 999 as missing values
bank_mkt["pdays"] = bank_mkt["pdays"].replace(999, pd.NA)
# Drop 12 duplicated rows
bank_mkt = bank_mkt.drop_duplicates().reset_index(drop=True)
# Convert types
    bank_mkt = bank_mkt.astype(
        dtype={
            "job": "category",
            "marital": "category",
            "education": "category",
            "y": "boolean",
        }
```

Partition Data

After importing the data, we need split the dataset into train set and test set. The models will be trained and tuned on the trainning set and test set will be used only for final validation purposes. However, simply sampling the dataset may lead to unrepresenatative partition given that our dataset is imbalanced and clients have different features. To solve this problem, `scikit-learn` provides a useful function `StratifiedShuffleSplit()` to select representative data into test set and train set, which is shown below as sample code:

```
from sklearn.model_selection import StratifiedShuffleSplit

train_test_split = StratifiedShuffleSplit(n_splits=1, test_size=0.2)
for train_index, test_index in train_test_split.split(
    bank_mkt.drop("y", axis=1), bank_mkt["y"]
):
    train_set = bank_mkt.iloc[train_index]
    test_set = bank_mkt.iloc[test_index]

X_train = train_set.drop(["duration", "y"], axis=1)
y_train = train_set["y"].astype("int").to_numpy()
X_test = test_set.drop(["duration", "y"], axis=1)
y_test = test_set["y"].astype("int").to_numpy()
```

Build Custom Preprocess Pipeline

After data partition, the train set and test set should be preprocessed into sutiable data type and shape for machine learning models. The train set and

test set should also be preprocessed seperately to avoid leaking test sample infomation into train set. If we scale data using both train set and test set, the scaling will ultimately be impacted by some test samples, which is not disired. To avoid the leakage, `scikit-learn` provides pipeline functionality that allows different treatments on train set and test set using `fit_transform()` and `transform()` as demostrated in the code below:

```
# preprocessor is a custom pipeline for preprocessing data
X_train = preprocessor.fit_transform(X_train, y_train)
X_test = preprocessor.transform(X_test)
```

Note that `preprocessor` is a custom preprocessing pipeline that we wrote for this specific project and can be named differently. A simplest preprocessor is a function that do some transformations in `pandas` as the following code:

```
from sklearn.preprocessing import FunctionTransformer

def encode_fn(X):
    """Encode categorical and boolean features into numeric values."""
    X = X.copy()
    X = X.apply(lambda x: x.cat.codes if pd.api.types.is_categorical_dtype(x) else (x.astype("Int64")
    X = X.astype("float")
    return X

encode_preprocessor = FunctionTransformer(encode_fn)
X_train = encode_preprocessor.fit_transform(X_train, y_train)
X_test = encode_preprocessor.transform(X_test)
```

Preprocessing pipeline can be extended according to our needs. For example, we might need one-hot encode categorical features and standardize numerical features after transforming categorical and boolean features into numeric values.

```
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import make_pipeline
cat_features = ["job",
                "marital",
                "education",
                "default",
                "housing",
                "loan",
                "poutcome"]

num_features =  ["age",
                 "campaign",
                 "pdays",
                 "previous",
                 "emp.var.rate",
                 "cons.price.idx",
                 "cons.conf.idx",
                 "euribor3m",
```

```
                "nr.employed"]

hot_scaler = ColumnTransformer([
    ("one_hot_encoder", OneHotEncoder(drop="first"), cat_features),
    ("scaler", StandardScaler(), num_features)
], remainder="passthrough")

hot_preprocessor = make_pipeline(FunctionTransformer(encode_fn), hot_scaler)
X_train = hot_preprocessor.fit_transform(X_train, y_train)
X_test = hot_preprocessor.transform(X_test)
```

From Pipeline To Workflow

In our project, the data partition and preprocessing is combined by the function `split_dataset()` which accepts `preprocessor` as a parameter. Its functionality will be further extended by the benchmarking function `benchmark()` which accepts `data`, `preprocessor`, `clf` as parameters and output model performance. Therefore, the ideal workflow will be:

1. Import data using `import_dataset()`;
2. Build a proper preprocessing pipeline `preprocessor`;
3. Build and tune an estimator `clf`;
4. Show model performance by calling `benchmark(data, preprocessor, clf)`.

Such workflow will be extensively reflected in the following chapters.

Feature Engineering

Impute Missing Values

Incorporate Dates

Drop Features

Sample Weights

Evaluation

From the Confusion Matrix, we can deride some key performance metrics. The false positive rate (FPR) measures the error rate of the negative outcomes:

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN}$$

The true negative rate (TNR) measures the accuracy rate for the negative outcomes:

$$TNR = \frac{TP}{N} = \frac{TN}{TN + FP} = 1 - FPR$$

The true positive rate (TPR) measures the accuracy rate for the positive outcomes:

$$TPR = \frac{TP}{P} = \frac{TP}{TP + FN}$$

Balanced accuracy is the average of true positive rate and true negative rate:

$$bACC = \frac{TPR + TNR}{2}$$

True positive rate is also known as recall (REC):

$$REC = TPR = \frac{TP}{TP + FN}$$

Precision (PRE) measures the accuracy of the predicted positive outcomes:

$$PRE = \frac{TP}{TP + FP}$$

To balance the up- and down-sides of optimizing PRE and REC, the harmonic mean of precision and recall is used:

$$F_1 = 2 \cdot \frac{PRE \times REC}{PRE + REC}$$

A receiver operating characteristic (ROC) is a graphical plot which illustrates the performance of a binary classifier system as its discrimination threshold is varied. It is created by plotting TPR against FPR at various threshold settings.

Average precision (AP) summarizes such a plot as the weighted mean of precisions achieved at each threshold, with the increase in recall from the previous threshold used as the weight. This implementation is not interpolated and is different from computing the area under the precision-recall curve with the trapezoidal rule, which uses linear interpolation and can be too optimistic.

$$AP = \sum_n (REC_n - REC_{n-1}) PRE_n$$

Logistic Regression

Logistic regression is commonly used to estimate the probability of an instance belonging to a particular class. If the probability is greater than 50%, the model will classify the instance to that class, otherwise, it will not. Therefore, Logistic regression is a binary classifier that can be applied to our dataset. Underlying the model is the logistic sigmoid function as shown below. This classifier can potentially perform very well on linearly separable classes. Although this might not be the case for our dataset, we still give it a try.

$$h(x) = 1/\left(1 + e^{-(theta_0 + theta_1 * x_1 + theta_2 * x_2 + theta_3 * x_3 + .. + theta_n * x_n)}\right)$$

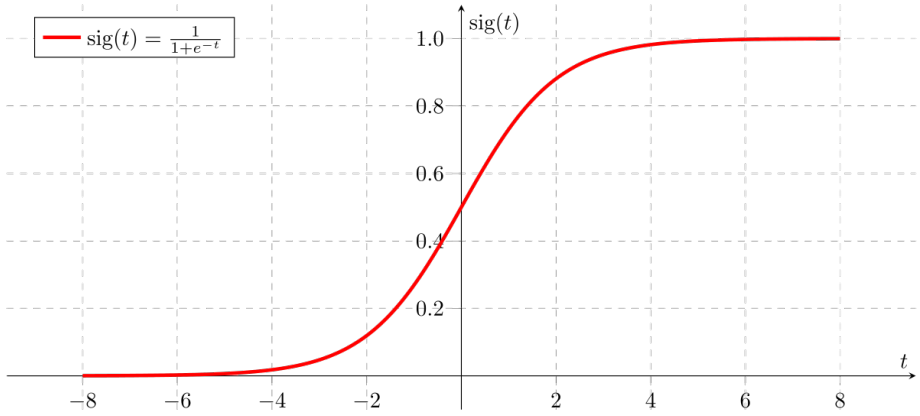


Figure 23: Logistic sigmoid function

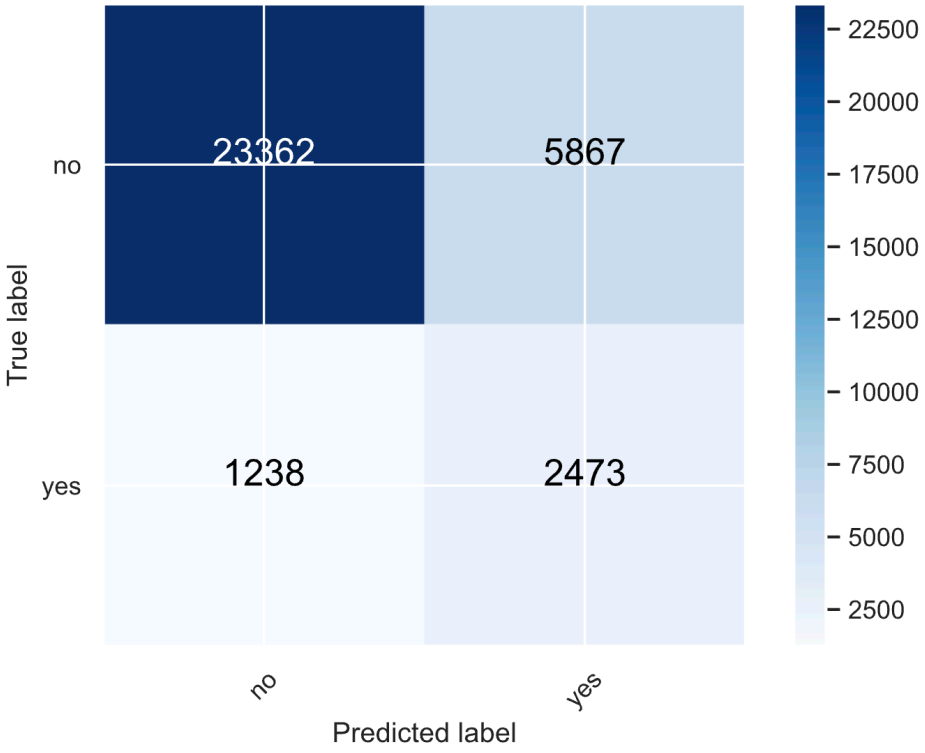
Fitting and Testing

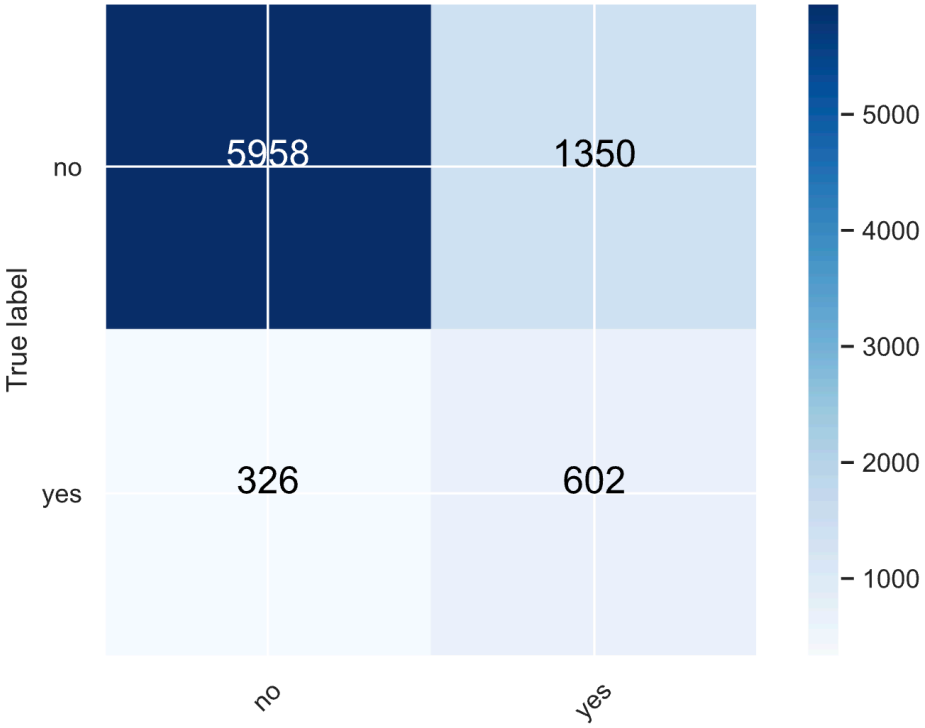
First, we import the Logistic Regression from SKlearn, and set two important parameters: 1. "class weight equals balanced", which is necessary to handle our imbalanced dataset; 2. The maximum number of iterations taken for the solvers to converge. The result shows a 79.65% accuracy score, a 44.2% average precision score and a ROC value of 0.783 for the test set. The confusion matrices and performance measures are presented below.

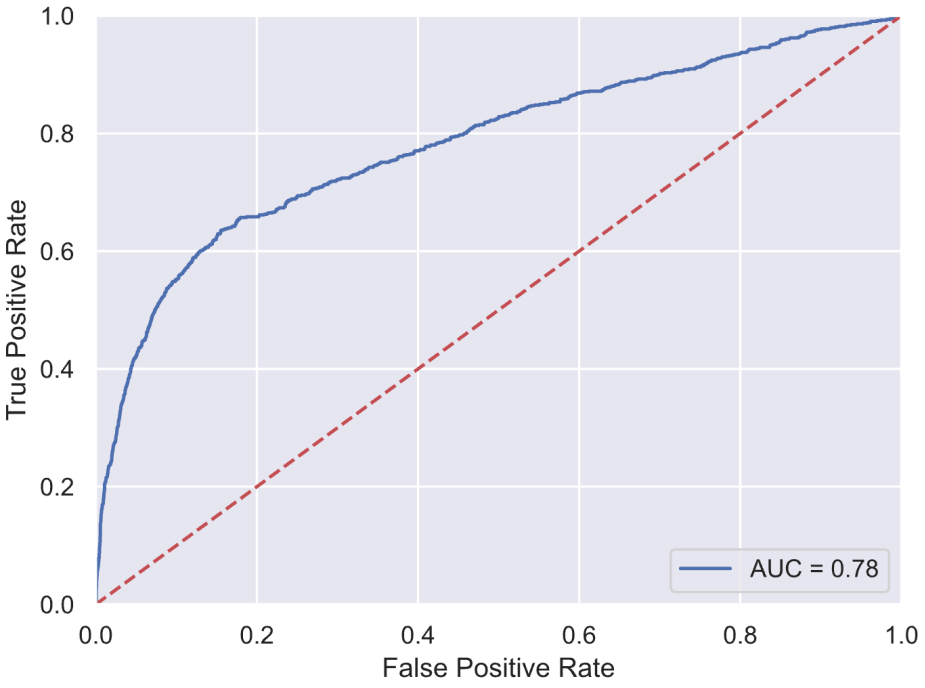
```
lrmodel = LogisticRegression(class_weight='balanced',max_iter=10000)
lrmodel.fit(X_train, y_train)
y_train_pred = lrmodel.predict(X_train)
# model measures for training data
cmtr = confusion_matrix(y_train, y_train_pred)
acctr = accuracy_score(y_train, y_train_pred)
aps_train = average_precision_score(y_train, y_train_pred)
# fit test set
lrmodel.fit(X_test, y_test)
y_test_pred = lrmodel.predict(X_test)
# model measures for testing data
cmte = confusion_matrix(y_test, y_test_pred)
accte = accuracy_score(y_test, y_test_pred)
aps_test = average_precision_score(y_test, y_test_pred)

print('Accuracy Score:',acctr)
print('Accuracy Score:',accte)
```

	Train	Validate	Test
TNR	0.798401	0.800376	0.802545
TPR	0.668912	0.659030	0.658405
bACC	0.733656	0.729703	0.730475
ROC	0.786022	0.783308	0.782608
REC	0.668912	0.659030	0.658405
PRE	0.296418	0.295290	0.297468
AP	0.440210	0.445930	0.441797




5958
1350
326
602
no
yes
True label
no
yes
Predicted label
5000
4000
3000
2000
1000


True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
AUC = 0.78

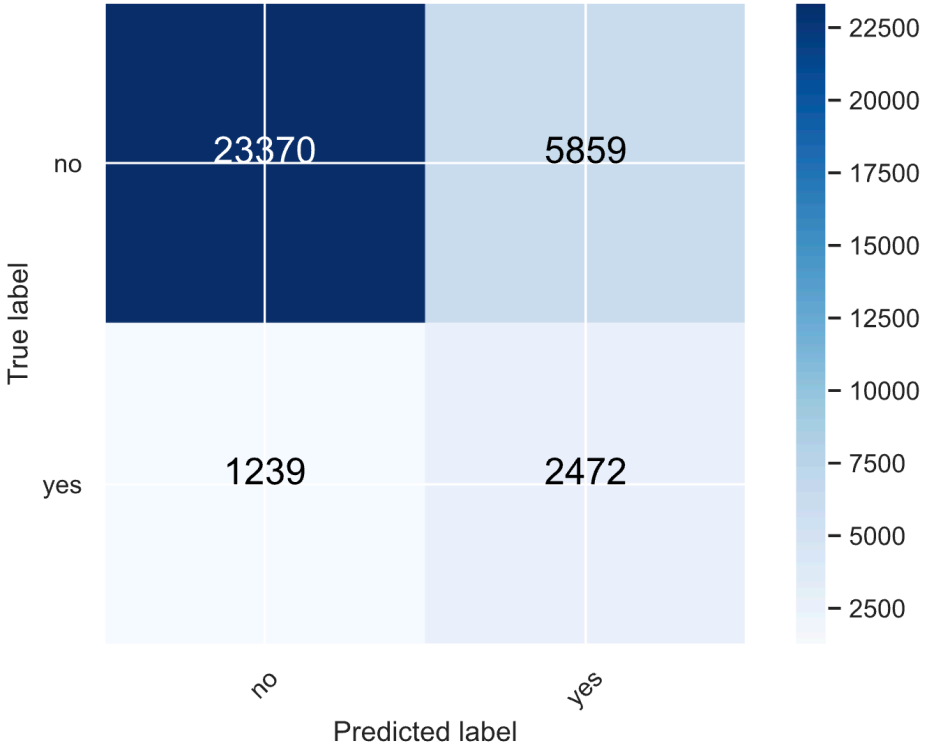
Grid Search

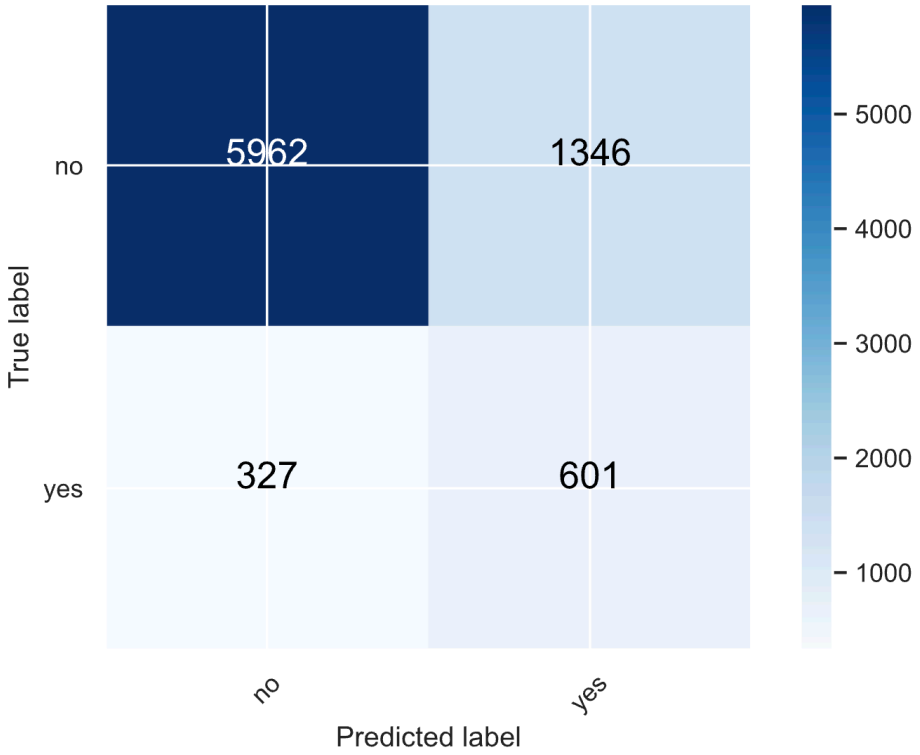
Next, we jumped right into Grid Search to find the optimal parameters for the model. For the first Grid Search, we picked two parameters: the penalty L2 and its inverse parameter C. The L1, L2 regularisation parameters are used to avoid overfitting of data due to either collinearity or high-dimensionality. They both shrink the estimates of the regression coefficients towards zero. When two predictors are highly correlated, L1 will pick one of the two predictors, and in contrast, L2 will keep both of them and jointly shrink the coefficients together a little bit. Parameter C is the inverse of the regularization strength, with smaller values leading to stronger regularization.

```
#### Try the 1st GridSearch param_grid combination:
lrmodel = LogisticRegression(class_weight='balanced', max_iter=10000)
#### Grid Search
param_grid = {'penalty': ['l2'],
              'C':[0.001,.009,0.01,0.05,0.09,5,10,25,50,100]}
GS_lrmodel_1 = GridSearchCV(lrmodel, param_grid, scoring='average_precision', n_jobs=-1)
GS_lrmodel_1.fit(X_train, y_train)
lrmodel_gs1 = lrmodel.set_params(**GS_lrmodel_1.best_params_)
#### use calibrated model on train set
lrmodel_gs1.fit(X_train, y_train)
y_train_pred = lrmodel_gs1.predict(X_train)
y_train_score = lrmodel_gs1.decision_function(X_train)
cmtr_gs1 = confusion_matrix(y_train, y_train_pred)
acctr_gs1 = accuracy_score(y_train, y_train_pred)
aps_train_gs1 = average_precision_score(y_train, y_train_score)
#### test the model
lrmodel_gs1.fit(X_test, y_test)
y_test_pred = lrmodel_gs1.predict(X_test)
y_test_score = lrmodel_gs1.decision_function(X_test)
cmte_gs1 = confusion_matrix(y_test, y_test_pred)
accte_gs1 = accuracy_score(y_test, y_test_pred)
aps_test_gs1 = average_precision_score(y_test, y_test_score)
print('Confusion Matrix:\n',cmtr_gs1,'\nAccuracy Score:\n',acctr_gs1, '\nAPS:\n',aps_train_gs1)
print('Confusion Matrix:\n',cmte_gs1,'\nAccuracy Score:\n',accte_gs1, '\nAPS:\n',aps_test_gs1)
print('best parameters:',GS_lrmodel_1.best_params_)
```

The results show a slight improvement compared to the initial model, with a 78.69% accuracy score, a 44.18% average precision score and an ROC value of 0.783 for the test set. Additionally, this Grid Search finds {'C': 10, 'penalty': 'l2'} as the best parameter combination. The confusion matrices and performance measures are presented below.

	Train	Validate	Test
TNR	0.800539	0.808758	0.803229
TPR	0.663523	0.669811	0.658405
bACC	0.732031	0.739285	0.730817
ROC	0.785491	0.786075	0.782550
REC	0.663523	0.669811	0.658405
PRE	0.296955	0.307740	0.298194
AP	0.434798	0.459581	0.441740





For the second Grid Search, we used the L1 penalty and Elasticnet penalty, which combines L1 and L2 penalties and will give a result in between. We also used the solver "Saga", which supports the non-smooth penalty L1 and is often used to handle the potential multinomial loss in the regression.

```
# Try the 2nd GridSearch param_grid combination:
lrmodel_gs = LogisticRegression(class_weight='balanced',max_iter=10000)
# Grid Search
param_grid = {"C":[0.001,.009,0.01,0.05,0.09,5,10,25,50,100],
              "penalty":["l1","elasticnet"],
              "solver": ["saga"]}
GS_lrmodel_2 = GridSearchCV(lrmodel_gs, param_grid, scoring='average_precision', n_jobs=-1)
GS_lrmodel_2.fit(X_train, y_train)
lrmodel_gs2 = lrmodel_gs.set_params(**GS_lrmodel_2.best_params_)
# use calibrated model on train set
lrmodel_gs2.fit(X_train, y_train)
y_train_pred = lrmodel_gs2.predict(X_train)
y_train_score = lrmodel_gs1.decision_function(X_train)
cmtr_gs2 = confusion_matrix(y_train, y_train_pred)
acctr_gs2 = accuracy_score(y_train, y_train_pred)
aps_train_gs2 = average_precision_score(y_train, y_train_pred)
# test the model
lrmodel_gs2.fit(X_test, y_test)
y_test_pred = lrmodel_gs2.predict(X_test)
y_test_score = lrmodel_gs1.decision_function(X_test)
cmte_gs2 = confusion_matrix(y_test, y_test_pred)
```

```
accte_gs2 = accuracy_score(y_test, y_test_pred)
aps_test_gs2 = average_precision_score(y_test, y_test_score)
print('Confusion Matrix:\n',cmtr_gs2,'\nAccuracy Score:\n',acctr_gs1, '\nAPS:\n',aps_train_gs1)
print('Confusion Matrix:\n',cmte_gs2,'\nAccuracy Score:\n',accte_gs2, '\nAPS:\n',aps_test_gs2)
print('best parameters:',GS_lrmodel_2.best_params_)
```

The results from the second Grid Search are almost identical to that of the first Grid Search. However, in the second case, {'C': 0.05, 'penalty': 'l1', 'solver': 'saga'} has been identified as the best parameter combination.

	animal
0	elk
1	pig
2	dog
3	quetzal

	Train	Validate	Test
TNR	0.793568	0.789942	0.798166
TPR	0.674301	0.648248	0.660560
bACC	0.733935	0.719095	0.729363
ROC	0.788165	0.770922	0.782781
REC	0.674301	0.648248	0.660560
PRE	0.293162	0.281451	0.293582
AP	0.447296	0.416163	0.441021

Figure 24: Performance measurement.

Statistical Summary

Finally, we used the `sm.Logit(y, X)` and `summary()` functions to summarise the performance of the Logistic Regression using raw data. Some features showed

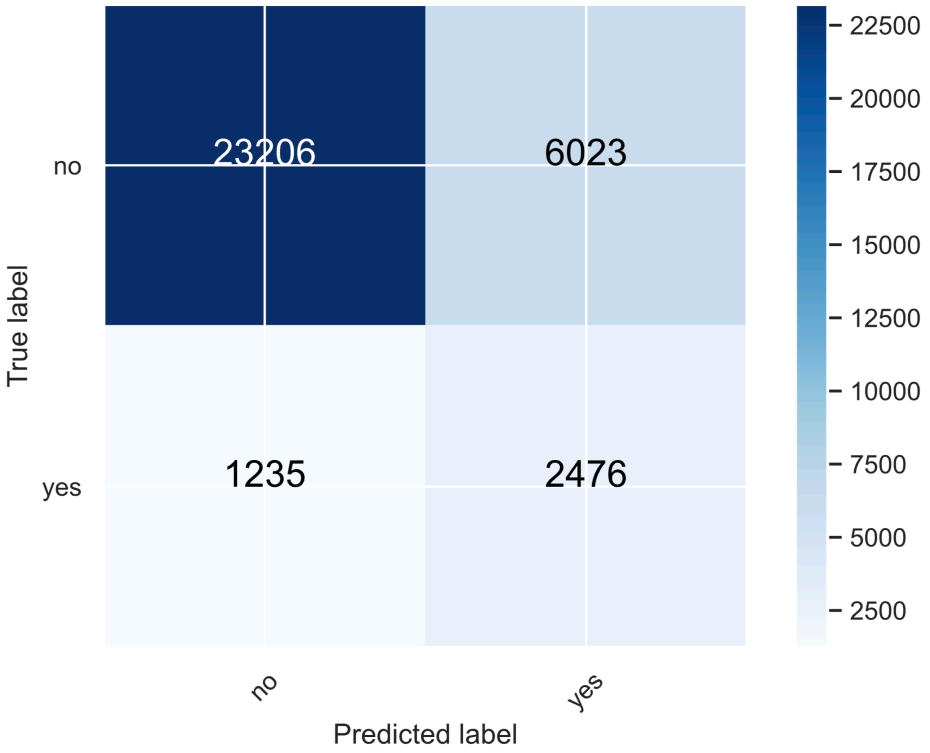


Figure 25: Confusion matrix for Train set.

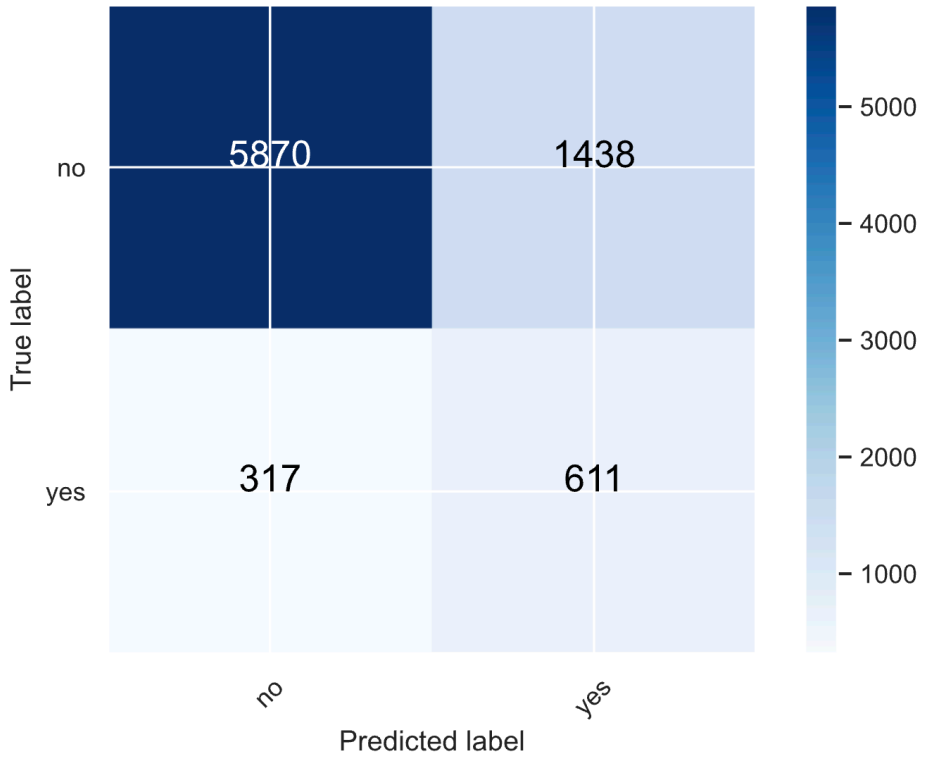


Figure 26: Confusion matrix for Test set.

very promising predictive power, such as the economic indicators, Marital and Education. However, Logistic Regression only achieved a 20% Pseudo R-square value with our dataset. Therefore, Logistic Regression is not an ideal model to handle our dataset.

Logit Regression Results

Dep. Variable:	y	No. Observations:	41176
Model:	Logit	Df Residuals:	41142
Method:	MLE	Df Model:	33
Date:	Sun, 06 Dec 2020	Pseudo R-squ.:	0.2015
Time:	21:17:56	Log-Likelihood:	-11574.
converged:	False	LL-Null:	-14496.
Covariance Type:	nonrobust	LLR p-value:	0.000

	coef	std err	z	P>|z|	[0.025	0.975]
x1	-0.2317	0.066	-3.507	0.000	-0.361	-0.102
x2	-0.1243	0.106	-1.173	0.241	-0.332	0.083
x3	-0.0808	0.126	-0.639	0.523	-0.328	0.167
x4	-0.0922	0.074	-1.243	0.214	-0.238	0.053
x5	0.3351	0.092	3.631	0.000	0.154	0.516
x6	-0.0770	0.100	-0.766	0.443	-0.274	0.120
x7	-0.1837	0.073	-2.501	0.012	-0.328	-0.040
x8	0.2809	0.096	2.928	0.003	0.093	0.469
x9	-0.0133	0.061	-0.219	0.827	-0.133	0.106
x10	-0.0062	0.110	-0.057	0.955	-0.221	0.209
x11	0.0278	0.059	0.471	0.638	-0.088	0.144
x12	0.1051	0.067	1.567	0.117	-0.026	0.237
x13	-1.0673	0.167	-6.401	0.000	-1.394	-0.740
x14	-0.9793	0.172	-5.708	0.000	-1.316	-0.643
x15	-1.0910	0.158	-6.895	0.000	-1.401	-0.781
x16	-1.0184	0.150	-6.800	0.000	-1.312	-0.725
x17	-0.9735	0.158	-6.151	0.000	-1.284	-0.663
x18	-0.8945	0.148	-6.030	0.000	-1.185	-0.604
x19	-0.0124	0.018	-0.703	0.482	-0.047	0.022
x20	-0.1209	0.026	-4.706	0.000	-0.171	-0.071
x21	-0.4997	0.026	-19.230	0.000	-0.551	-0.449
x22	-0.0074	0.025	-0.298	0.766	-0.056	0.042
x23	-1.3035	0.099	-13.154	0.000	-1.498	-1.109
x24	0.6383	0.057	11.289	0.000	0.527	0.749
x25	0.1881	0.026	7.319	0.000	0.138	0.238
x26	0.4878	0.163	2.992	0.003	0.168	0.807
x27	-0.2901	0.109	-2.652	0.008	-0.505	-0.076
x28	-18.7032	3.06e+04	-0.001	1.000	-6.01e+04	6e+04
x29	-0.0287	0.035	-0.811	0.417	-0.098	0.041
x30	0.0009	0.002	0.459	0.646	-0.003	0.005
x31	-1.0180	0.060	-16.887	0.000	-1.136	-0.900
x32	-0.0530	0.011	-4.630	0.000	-0.075	-0.031
x33	0.0432	0.012	3.507	0.000	0.019	0.067
x34	1.0794	0.055	19.771	0.000	0.972	1.186

Figure 27: Statistical Summary.

SVM

The purpose of Support Vector Machine is to find a hyperplane that distinctly classifies the data points in an N-dimensional space. Hyperplane is the decision boundary that classifies the data points and support vecotrs are the data points that are closer to the hyperplane which influence the position and orientation

of the hyperplane.

The worth of the classifier is how well it classifies the unseen data points and therefore our objective is to find a plane with the maximum margin i.e the distance of an observaiton from the hyperplane. The margin brings in extra confidence that the further data points will be classified correctly.

Theory and Hyperparameter

Maximum Margin

The understanding of SVM is derived from the loss function of Logistic Regression with l2 regularization:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}[y^{(i)}(-log(p^{(i)})) + (1-y^{(i)})(-log(1-p^{i}))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_2^{(j)}$$

where

$$p^{(i)} = \sigma(\vec{\theta}^T \cdot \vec{x}^{(i)}) = 1/(1+e^{-\vec{\theta}^T} \cdot \vec{x}^{(i)})$$

In the realm of loss function of Logistic Regression, the individual loss contribution to the overall function is $-log(p^{(i)})$ if $y^{(i)} = 1$ and $-log(1-p^{(i)})$ if $y^{(i)} = 0$.

By replacing the individual loss contribution to $max(0, 1 - \vec{\theta}^T \cdot \vec{x}^{(i)})$ and $max(0, 1 + \vec{\theta}^T \cdot \vec{x}^{(i)})$ for $y^{(i)} = 1$ and $y^{(i)} = 0$ respectively, SVM penalizes the margin violation more than logistic regression by requiring a prediction bigger than 1 for y =1 and a prediction smaller than -1 if y = 0.

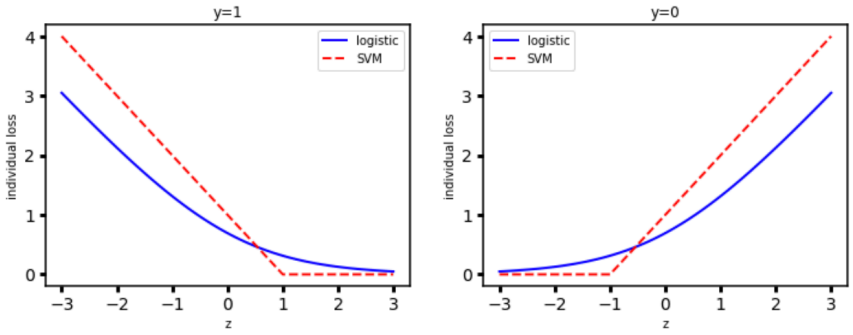


Figure 28: Screenshot 2020-12-04 at 14.17.12

Regularizaiton and Trade-off (C parameter)

The regularization term plays the role of widening the distance between the two margins and tells SVM how much we want to avoid the wrong misclassification. A hyperplane with maximal margin might be extremely sensitve to a change in the data points and may lead to overfitting problems.

To achieve the balance of a greater robustness and better classification of the model, we may consider it worthwhile to misclassify a few training data points to do a better job in separating the furture data points.

Hyperparameter C in SVM allows us to dictate the tradeoff between having a wide margin and correctly classiying the training data points. In other words, a large value for C will shrink the margin distance of hyperplane while a small value for C will aim for a larger-margin separator even if it misclassifies some data points.

Gamma

Gamma controls how far the influence of a single obeservation on the decision boundary. The high Gamma indicates only the points closer to the plausible hyperplane are considered and vice versa.

Kernel

For linearly separable and almost linearly separable data, SVM works well. For data that is not linearly separable, we can project the data to a space where it is linearly separable. What Kernel Trick does is utilizing the existing features and applying some transformations to create new features and calculates the nonlinear decision boundary in higher dimension by using these features.

Linear SVM

```
linear_svm = LinearSVC(dual=False, class_weight="balanced", random_state=42)

param_distributions = {"loss": ["squared_hinge", "hinge"],
                       "C": loguniform(1e0, 1e3)}

random_search = RandomizedSearchCV(linear_svm,
                                   param_distributions,
                                   scoring="average_precision",
                                   cv=5,
                                   n_jobs=-1,
                                   n_iter=100)

grid_fit = random_search.fit(X_train, y_train)
grid_results = random_search.cv_results_
grid_best_params = random_search.best_params_
grid_best_score = random_search.best_score_

print(f"best parameters found: {grid_best_params}, with mean test score: {grid_best_score}")

# Output
# best parameters found: {'C': 1.1361548657024574, 'loss': 'squared_hinge'}, with mean test score: 0.

param_grid = [
    {"C": [5,2,1]}
    ]
```

```
grid_search = GridSearchCV(linear_svm,
                           param_grid,
                           scoring="average_precision",
                           return_train_score=True,
                           cv=5,
                           n_jobs=-1)

grid_fit = grid_search.fit(X_train, y_train)
grid_results = grid_search.cv_results_
grid_best_params = grid_search.best_params_
grid_best_score = grid_search.best_score_

print(f"best parameters found: {grid_best_params}, with mean test score: {grid_best_score}")


linear_svm = LinearSVC(loss="squared_hinge", C=1, dual=False, class_weight="balanced", random_state=4

# Output
# best parameters found: {'C': 1}, with mean test score: 0.43356240611668306
```

	Train	Validate	Test
TNR	0.795108	0.791310	0.795703
TPR	0.664534	0.691375	0.659483
bACC	0.729821	0.741342	0.727593
ROC	0.785562	0.786010	0.781580
REC	0.664534	0.691375	0.659483
PRE	0.291691	0.296018	0.290736
AP	0.435728	0.432823	0.437258

Figure 29: Result

Non-Linear SVM

We use pipeline to ensure that in the cross validation set, the kernel function is only applied to training fold which is exactly the same fold used for fitting the model. We also do a comparison between SGDClassifier and Linear SVC and the latter one gave us slightly better AP rate.

```
rbf_sgd_clf = Pipeline([
    ("rbf", RBFSampler(random_state=42)),
    ("svm", SGDClassifier(class_weight="balanced"))
])

param_distributions = {
    "rbf__gamma": loguniform(1e-6, 1e-3),
    "svm__alpha": loguniform(1e-10, 1e-6)}

random_search = RandomizedSearchCV(rbf_sgd_clf,
                                   param_distributions,
                                   scoring="average_precision",
                                   cv=5,
                                   n_jobs=-1,
                                   n_iter=10)

grid_fit = random_search.fit(X_train, y_train)
grid_results = random_search.cv_results_
grid_best_params = random_search.best_params_
grid_best_score = random_search.best_score_

print(f"best parameters found: {grid_best_params}, with mean test score: {grid_best_score}")

# Output
# best parameters found: {'rbf__gamma': 0.0007087711398938291, 'svm__alpha': 1.2269339879156183e-07},

param_grid = {
    "rbf__gamma": [0.0008, 0.0001, 0.001],
    "svm__alpha": [1e-7, 1e-6, 1e-5]}

grid_search = GridSearchCV(rbf_sgd_clf,
                           param_grid,
                           scoring="average_precision",
                           cv=5,
                           n_jobs=-1)

grid_fit = grid_search.fit(X_train, y_train)
grid_results = grid_search.cv_results_
grid_best_params = grid_search.best_params_
grid_best_score = grid_search.best_score_

print(f"best parameters found: {grid_best_params}, with mean test score: {grid_best_score}")
```

```
# Output
# best parameters found: {'rbf__gamma': 0.0008, 'svm__alpha': 1e-06}, with mean test score: 0.4403394

rbf_sgd_tuned = rbf_sgd_clf.set_params(rbf__gamma=0.0009, svm__alpha=1e-6)
benchmark(bank_mkt, hot_transformer, rbf_sgd_tuned)
```

	Train	Validate	Test
TNR	0.792798	0.797639	0.644499
TPR	0.678680	0.681941	0.752155
bACC	0.735739	0.739790	0.698327
ROC	0.791381	0.789337	0.786777
REC	0.678680	0.681941	0.752155
PRE	0.293732	0.299586	0.211772
AP	0.436139	0.444426	0.437136

Figure 30: Result

```
rbf_clf = Pipeline([
    ("rbf", RBFSampler(random_state=42)),
    ("svm", LinearSVC(loss="squared_hinge", dual=False, class_weight="balanced", max_iter=1000))
])

param_distributions = {
    "rbf__gamma": loguniform(1e-6, 1e-3),
    "svm__C": loguniform(1e-1, 1e1)}

random_search = RandomizedSearchCV(rbf_clf,
                                   param_distributions,
                                   scoring="average_precision",
                                   cv=5,
```

```
                                    n_jobs=-1,
                                    n_iter=10)

grid_fit = random_search.fit(X_train, y_train)
grid_results = random_search.cv_results_
grid_best_params = random_search.best_params_
grid_best_score = random_search.best_score_

print(f"best parameters found: {grid_best_params}, with mean test score: {grid_best_score}")

# Output
# best parameters found: {'rbf__gamma': 0.00026560333125098774, 'svm__C': 6.5900965177317055}, with m

param_grid = {
    "rbf__gamma": [0.0001, 0.001, 0.01],
    "svm__C": [1, 10, 20]}

grid_search = GridSearchCV(rbf_clf,
                           param_grid,
                           scoring="average_precision",
                           cv=5,
                           n_jobs=-1)

grid_fit = grid_search.fit(X_train, y_train)
grid_results = grid_search.cv_results_
grid_best_params = grid_search.best_params_
grid_best_score = grid_search.best_score_

print(f"best parameters found: {grid_best_params}, with mean test score: {grid_best_score}")

# Output
# best parameters found: {'rbf__gamma': 0.001, 'svm__C': 10}, with mean test score: 0.439864774178839

rbf_tuned = rbf_clf.set_params(rbf__gamma=0.0009, svm__C=1)
```

References

(Hastie, Tibshirani and Friedman, 2009)

(Chen, 2019)

(Patel, 2017)

('Machine Learning 4 Support Vector Machine', no date)

Neural Network

Neural Network is a "brain structure" with the following components:

1. An input layer
2. An arbitrary amount of hidden layer

	Train	Validate	Test
TNR	0.788906	0.787889	0.794745
TPR	0.677669	0.676550	0.668103
bACC	0.733288	0.732220	0.731424
ROC	0.787619	0.782527	0.784626
REC	0.677669	0.676550	0.668103
PRE	0.289580	0.288175	0.292453
AP	0.437404	0.453640	0.440392

Figure 31: Result

3. An output layer
4. A set of weights and biases between each layer
5. An activation function for each hiddeb layer

Training the neural network model is to find the right values for the weights and biases and it involves multiple iterations of exposing the training dataset to the network. Each iteration of the training process consists of these 2 steps:

```
Forward Propagation: the input data is fed in the forward direction. Each hidden layer accepts the input d
```

```
Backpropagation: fine tuning the weights of a neural net based on error rate obtained in the previous iter
```

Hyperparameters

Learning rate

Learning rate controls how much we are adjusting the weights of our network with respect to the loss gradient.

A small learning rate will only make slight change to the weights each update and therefore requires more training epochs and travels along the downward slope slowly, whereas a large learning rate leads to rapid changes to the weight. However, a too large learning rate may cause the model to converge too quickly to a local minima or overshoot the optimal solution.

Hidden layer sizes

This hyperparameter defines how many hidden layers and how many neurons on each layer we want to have when we are building the achitechture of the neural network.

We want to keep the neural network archtechture as simple as possible so that it can be trained fast and well generalized and meanwhile, we also need it to classify the input data well, which may require a relatively complex architecture.

L2 Regularization

The regularization term will drive down the weights of the matrix and decorrelate the neural network, which in turn decreases the effects of the activation function and prevent overfitting by fitting a less complex model to the data.

Activation Function

Activation functions are mathematical equations attached to each neuron in the network and they determine the output of the learning model, the accuracy and the computational efficiency of training the model.

With the use of Non-linear activation function, we are able to create complex mapping between the network's inputs and outputs which are essential for learning and modeling complex data.

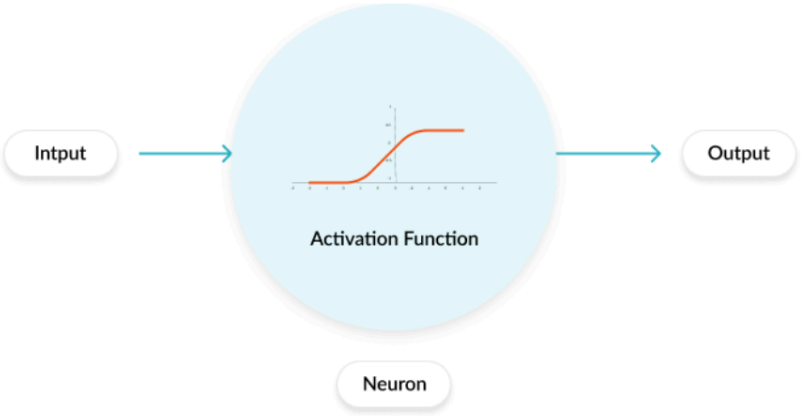


Figure 32: Activation Function

GridSearch

```
mlp=MLPClassifier(random_state=42,max_iter=1000)

param_grid ={
    'solver':['lbfgs', 'sgd', 'adam'],
    'learning_rate':["constant","invscaling","adaptive"],
    'hidden_layer_sizes':[(100,),(200,),(20,5,),(10,5,),(100,50,25,)],
    'alpha':[0.0,0.001,0.01],
    'activation' :["logistic","relu","tanh"] }

grid_search = GridSearchCV(estimator=mlp,
                           param_grid=param_grid,
                           scoring = "average_precision",
                           return_train_score=True,
                           cv = 5,
                           n_jobs=-1)
grid_fit = grid_search.fit(X_train, y_train)
grid_results = grid_search.cv_results_
grid_best_params = grid_search.best_params_
grid_best_score = grid_search.best_score_

mlp_trained=MLPClassifier(solver ="lbfgs",
                           random_state=42,
                           max_iter=1000,
                           activation = 'relu',
                           alpha = 0.01,
                           hidden_layer_sizes = (100,),
                           learning_rate = 'constant')
```

```
nn_best = benchmark(bank_mkt, hot_transformer, mlp_trained)
```

	Train	Validate	Test
TNR	0.984219	0.959117	0.969896
TPR	0.470192	0.295148	0.310345
bACC	0.727206	0.627133	0.640120
ROC	0.900857	0.730865	0.746097
REC	0.470192	0.295148	0.310345
PRE	0.790935	0.478166	0.566929
AP	0.688217	0.354089	0.396211

Figure 33: Result

Reflections

In machine learning, data can be roughly divided into four categories: Image, Sequence , Graph and Tabular data. The first three types of data have obvious patterns, such as the spatial locality of images and graphs, the contextual relationship and timing dependence of sequences, and so on. However,in tabular data, each feature represents an attribute, such as gender, price, etc. There is generally no obvious and common pattern between features.

Neural networks are more suitable for the first three types of data, that is, data with obvious patterns. Because we can design the corresponding network structure according to the data pattern, so as to select features more efficiently. For example, the CNN (Convolutional Neural Network) is designed for images, and the RNN (Recurrent Neural Network) is designed for sequence data.

For tabular data where there is no obvious pattern, an inefficient fully connected network may not work as well as the more traditional machine learning models such as Gradient Boosting Tree.

In the case of tabular data, feature engineering might be more important because

it integrates the prior knowledge into the data and makes the model understand the data better.

References

(Loy, 2020)

('7 Types of Activation Functions in Neural Networks: How to Choose?', no date)

(Brownlee, 2019)

Decision Tree and Its Ensembles

Decison Tree

A decision tree is a flowchart-like structure in which each internal node represents a "test" on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes). The paths from root to leaf represent classification rules.

In Machine Learning this type of structure can be used to advise in classification and regression problems. Since our project's topic was a classification problem in its nature, we will focus on the decision Tree as a classifier in the remainder of the text.

Here is an outline of hyperparameters most commonly tuned for performance:

- **max_depth:** int, *default=None* ; The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- **min_samples_split:** int or float, *default=2* ; The minimum number of samples required to split an internal node.
- **criterion:** {"gini", "entropy"}, *default="gini"* ; The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Note: this parameter is tree-specific.
- **min_samples_leaf:** int or float, *default=1* ; The minimum number of samples required to be at a leaf node.
- **class_weight:** {"balanced", "balanced_subsample"}, *default=None* ; Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one.
- **random_state:** int or RandomState, *default=None*; Controls both the randomness of the bootstrapping of the samples used when building trees and the sampling of the features to consider when looking for the best split at each node.

It is important to note that Decision Trees can be combined to form Forests and can be used as primitive estimators in AdaBoost. This makes them redundant in terms of accuracy to these more advanced models that use them as a base. It is still however necessary to understand the individual tree estimator in order to

understand how these more advance structures work. The tree is also cheaper in terms of computing requirements and can be used as an initial estimator or for feature importance analysis.

To keep the scope of this paper reasonable, we will focus on elaborating on these more advanced models (Forest and AdaBoost) while keeping in mind that in their very foundation there is a tree, silently doing it's job.

Random Forest

The Random Forest represents a collection of decision trees. They a way of averaging multiple deep decision trees, trained on different parts of the same training set, with the goal of reducing the variance. This comes at the expense of a small increase in the bias and some loss of interpretability, but generally greatly boosts the performance in the final model.

In order to prepare the data for the Forest we initially ran it through our standard transformer, explained in the pipeline section of this text.

```
tree_transformer = FunctionTransformer(dftransform)
X_train, y_train, X_test, y_test, *other_sets = split_dataset(bank_mkt, tree_transformer)
```

We then proceeded onto hyperparameter tuning. Hyperparameter tuning of the random forest is basically hyperparameter tuning of the individual tree with one prominent differences. Namely:

- **n_estimators:** int, *default=100* ; which basically represents the number of trees in the forest.

In the case of our project:

```
RF = RandomForestClassifier(random_state=42,class_weight="balanced",criterion ="gini",max_features="a
                            min_samples_split= 2)


param_grid = {
    'max_depth':[6,8,10],
    'n_estimators':[1000,1500,1750,2000]
}
CV_RFmodel = GridSearchCV(estimator=RF,param_grid=param_grid,scoring="average_precision",n_jobs=-1,cv
CV_RFmodel.fit(X_train,y_train)
grid_results = CV_RFmodel.cv_results_
grid_best_params = CV_RFmodel.best_params_
grid_best_score = CV_RFmodel.best_score_
grid_best_estimator = CV_RFmodel.best_estimator_
print(f"best mean test score: {grid_best_score}, for {grid_best_estimator}")
```

It is worth noting that we also tried combinations of other hyperparameters. but for easier readability of the text decided to go here with the two hyperparameters that provided with highest model oscillations. The hyperparameters that were selected for the RandomForestClassifier in the definition of variable RF were the ones that showed in previous tunings to provide constant, superior results. The outputs given were

```
best mean test score: 0.454011896674566, for RandomForestClassifier(class_weight='balanced', max_depth=
```

We decided that 1750 estimators in subsequent testing showed signs of overfitting and went to produce our final performance matrix with 1500 estimators.

```
RF_validation = RandomForestClassifier(random_state=42,class_weight="balanced",max_depth=6,n_estimato
                                       max_leaf_nodes=1000)
benchmark(bank_mkt, tree_transformer, RF_validation)
```

As can be seen from the table RandomForestClassifier gives strong values in AUC ROC and proves in our dataset to be one of the best performing models.

One interesting use of the RandomForestClassifier is that it also returns feature importance metrics for individual features within the dataset. This can be a very useful method of the class since it allows a positive feedback loop between feature engineering, model testing and then returning to feature engineering for additional optimisations and manipulations on the most useful features. On the other hand features that show no importance in the classification can be dropped or merged.

```
columns = bank_mkt.drop(["duration", "y"], axis=1).columns.tolist()
rnd_clf = RandomForestClassifier(n_estimators=1750,max_depth=6, n_jobs=-
1, random_state=42)
rnd_clf.fit(X_train, y_train)
for name, importance in zip(columns, rnd_clf.feature_importances_):
    print(name, "=", importance)
```

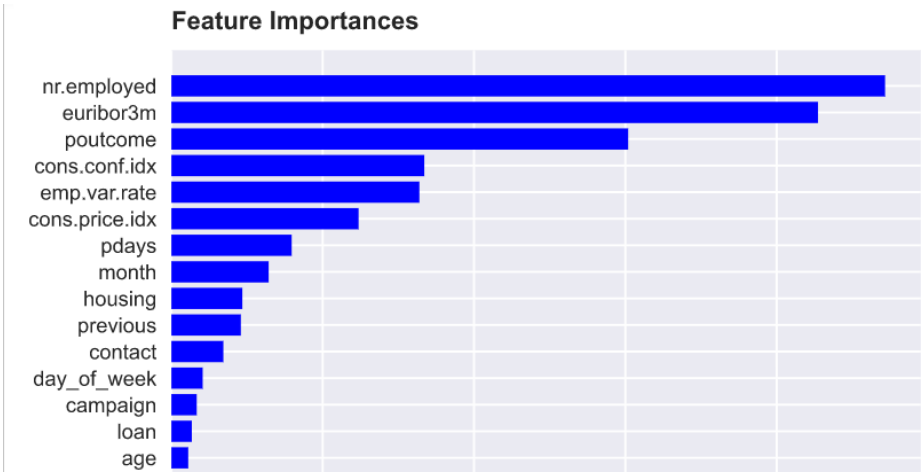
The features that constantly proved to be the most important for the model are:

```
nr.employed = 0.23581867160908354
euribor3m = 0.213662798284482
poutcome = 0.15093401974315632
cons.conf.idx = 0.08356367745032177
emp.var.rate = 0.081946685063493
cons.price.idx = 0.06185600062913342
```

This can also be plotted to give an informative picture about how features rank by importance:

```
importances = rnd_clf.feature_importances_
indices = np.argsort(importances)

plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='b', align='center')
plt.yticks(range(len(indices)), [columns[i] for i in indices])
plt.xlabel('Relative Importance')
plt.savefig("Feature importance.png")
plt.size=(15,10)
plt.show()
```



All in all the Random Forest lived up to its expectation. By adding just a small amount of bias it greatly improves the performance of Decision Trees. They are very robust and require little to none work in terms of encoding and feature manipulation.

AdaBoost

An AdaBoost classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.

The AdaBoost uses simple, primitive individual classifiers but in comparison with the Random Forest it gives them different, ever changing weights to their final decision. The individual primitive estimator is one of the hyperparameters

- **base_estimator:** object, *default=None* ; The base estimator from which the boosted ensemble is built. Support for sample weighting is required, as well as proper `classes_` and `n_classes_` attributes. If `None`, then the base estimator is `DecisionTreeClassifier(max_depth=1)`.

The process of tuning this parameter can at starts feel counterintuitive. Why wouldn't a tree of max_depth=2 return better overall results than this stump with only depth level of one. In this question lies the beauty and genius of this model. Because many individually primitive estimators with weighted, individually adjusted decisions will overall provide with a more effective and efficient model.

The other important hyperparameters to tune in the AdaBoost Classifier are

- **learning_rate:** float, *default=1* ; Learning rate shrinks the contribution of each classifier by `learning_rate`. There is a trade-off between `learning_rate` and `n_estimators`.
- **n_estimators:** int, *default=50* ; The maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early.

```
AB= AdaBoostClassifier(n_estimators=100,random_state=42,learning_rate=1.0)

param_grid = {
    'learning_rate':[0.8],
    'n_estimators':[800],
    'base_estimator':[DecisionTreeClassifier(max_depth=1),DecisionTreeClassifier(max_depth=4)]
}
CV_RFmodel = GridSearchCV(estimator=AB,param_grid=param_grid,scoring="average_precision",n_jobs=-1,cv
CV_RFmodel.fit(X_train,y_train)
grid_results = CV_RFmodel.cv_results_
grid_best_params = CV_RFmodel.best_params_
grid_best_score = CV_RFmodel.best_score_
grid_best_estimator = CV_RFmodel.best_estimator_
print(f"best mean test score: {grid_best_score}, for {grid_best_estimator}")
```

Which gave us the optimal model set up:

```
best mean test score: 0.448988423603947, for AdaBoostClassifier(base_estimator=DecisionTreeClassifier(m
                   learning_rate=0.8, n_estimators=800, random_state=42)
```

We proceeded to find the full performance matrix:

```
AB_validation = AdaBoostClassifier(n_estimators=800,learning_rate=0.8,random_state=42,
                                        base_estimator = DecisionTreeClassifier(max_depth=2,min_sample

benchmark(bank_mkt, tree_transformer, AB_validation)
```

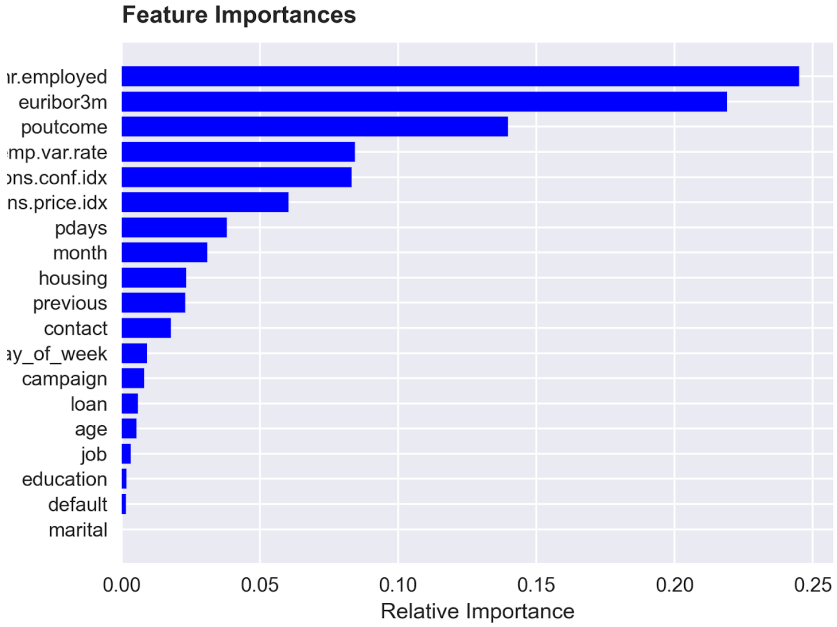
In a similar fashion we derive feature importance as well:

```
for name, importance in zip(columns, AB_validation.feature_importances_):
    print(name, "=", importance)
```

```
nr.employed = 0.2451296512111920 4
euribor3m = 0.2190410622256826
poutcome = 0.1397408073745672
emp.var.rate = 0.0844557803264920 4
```

```
importances = AB_validation.feature_importances_
indices = np.argsort(importances)

plt.title('Feature Importances')
plt.barh(range(len(indices)), importances[indices], color='b', align='center')
plt.yticks(range(len(indices)), [columns[i] for i in indices])
plt.xlabel('Relative Importance')
#plt.savefig("Feature importance.png")
plt.size=(15,10)
#plt.savefig("Feature importance_ada.png")
plt.show()
```



As we can see the AdaBoost gave strong results in the area underneath the ROC curve but was still behind the Random Forest for our dataset. The main advantages of Random forests over AdaBoost are that it is less affected by noise and it generalizes better in reducing variance because the generalization error reaches a limit with an increasing number of trees being grown (according to the Central Limit Theorem).

Feature importance for both AdaBoost and the Random Forest was strikingly similar. In section 7 of the paper *Random Forests* (Breiman, 1999), the author states the following conjecture: "AdaBoost is a Random Forest". This is an interesting claim , yet to be proven or disproven but AdaBoost with the base estimator of a tree stump can in certain datasets behave very much like a Random Forest of sorts. Proven or disproven it just confirms once more what we discussed in class, that Machine Learning is a trial and error process and data speaks its own language. There are no universal truths and *ad-hoc* solutions in this exciting field.

Conclusion

References

Appendix

'7 Types of Activation Functions in Neural Networks: How to Choose?' (no date) *MissingLink.ai*. Available at: https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/ (Accessed: 6 December 2020).

Brownlee, J. (2019) 'Understand the Impact of Learning Rate on Neural Network Performance', *Machine Learning Mastery*. Available at: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/ (Accessed: 6 December 2020).

Chen, L. (2019) 'Support Vector Machine — Simply Explained', *Medium*. Available at: https://towardsdatascience.com/support-vector-machine-simply-explained-fee28eba5496 (Accessed: 6 December 2020).

Hastie, T., Tibshirani, R. and Friedman, J. (2009) *The Elements of Statistical Learning*. New York, NY: Springer New York (Springer Series in Statistics). doi: 10.1007/978-0-387-84858-7.

Loy, J. (2020) 'How to build your own Neural Network from scratch in Python', *Medium*. Available at: https://towardsdatascience.com/how-to-build-your-own-neural-network-from-scratch-in-python-68998a08e4f6 (Accessed: 6 December 2020).

'Machine Learning 4 Support Vector Machine' (no date). Available at: https://kaggle.com/fengdanye/machine-learning-4-support-vector-machine (Accessed: 6 December 2020).

Patel, S. (2017) 'Chapter 2 : SVM (Support Vector Machine) — Theory', *Medium*. Available at: https://medium.com/machine-learning-101/chapter-2-svm-support-vector-machine-theory-f0812effc72 (Accessed: 6 December 2020).