Bank Marketing

Fan Jia Jiawei Li Strahinja Trenkic Qiqi Zhou

Contents

Introduction

Motivation

A retail bank has a customer call centre as one of its units, through which the bank communicates with potential new clients and offers term deposits. Term deposits are defined as a fixed-term investment that includes the deposit of money into an account at a financial institution.

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 indicators and certain parameters of previous contact with the given person. The motivation behind the project is clear, by analysing previous phone calls the bank would like to improve its telemarketing 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 therefore 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.

We also need to understand the economic and historic context behind the dataset. In 2008, the world was plunged in the deepest economic recession since the 1930s. After the bankcruptcy of The Lehman Brothers, the financial market was shook to its core and governments 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 it represents the difference between a riskless investment (T-Bill) and the risk banks transfer when loaning to each other. The wider the spread, the more default risk is priced into the borrowing market. It is trival to conclude that such an increase in risk will have major impact not only on the telemarketing campaign but also the representability and timelessness of the data gathered through it.

Can we develop a data driven approach to help the bank increase its success rate of telemarketing while incorporating the economic context? There are several questions that could be interesting:

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, etc. affect their economic choices?
3. Do people prefer being called on the mobile phone or landline?
4. Does a predictive model exist that can predict a telemarkting outcome using client and economic data?

Dataset Description

The bank marketing dataset was collectedand ordered by date ranging from May 2008 to November 2010. The data was ordered by date even though its year has to be infered manually. The 20 input features were categorized and described as follows:

A. Bank client data:

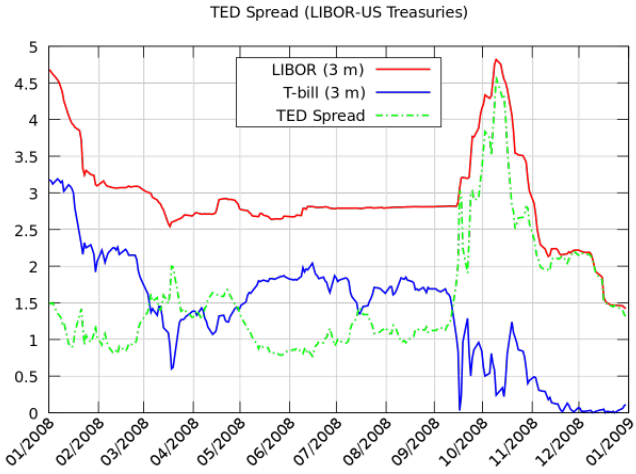


Figure 1: TED Spread

`age`: age

`job`: type of job

`marital`: marital status

`education`: education level

`default`: has credit in default?)

`housing`: has housing loan?)

`loan`: has personal loan?)

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

`contact`: contact communication type

`month`: last contact month of year

`day_of_week`: last contact day of the week

`duration`: last contact duration, in seconds

C. Other attributes:

`campaign`: number of contacts performed during this campaign and for this client

`pdays`: number of days that passed by after the client was last contacted from a previous campaign

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

`poutcome`: outcome of the previous marketing campaign

D. Social and economic context attributes:

`emp.var.rate`: employment variation rate, quarterly indicator

`cons.price.idx`: consumer price index, monthly indicator

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

`euribor3m`: euribor 3 month rate, daily indicator

`nr.employed`: number of employees, quarterly indicator

Output variable:

`y`: has the client subscribed a term deposit?

It is important to note that `duration` attribute highly affects the output targetd. 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.

Exploratory Data Analysis

Exploratory Data Analysis is a process of exploring the dataset with no assumptions or hypotheses made, 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.

Target Variable

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

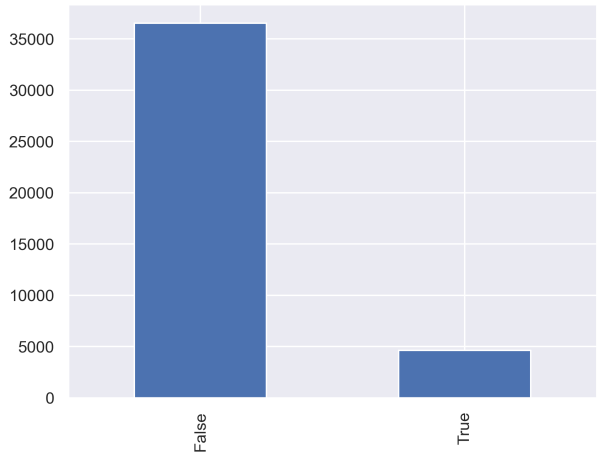


Figure 2: Distribution of camaign outcome.

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 afterwards, 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

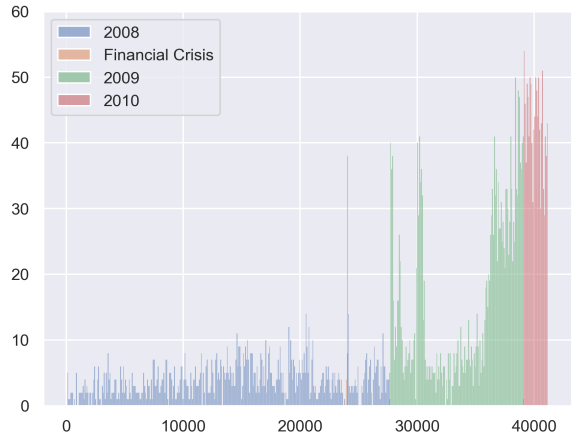


Figure 3: Uneven distribution of positive outcome during the financial crisis.

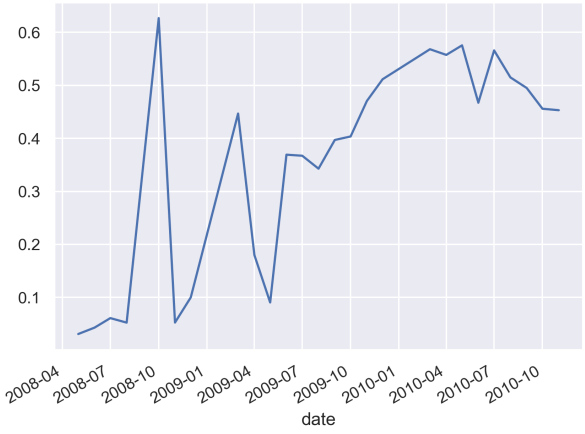


Figure 4: Positive outcome rates by month.

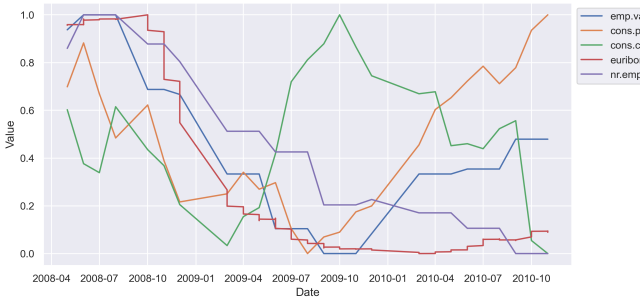


Figure 5: Five economic indicators.

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.

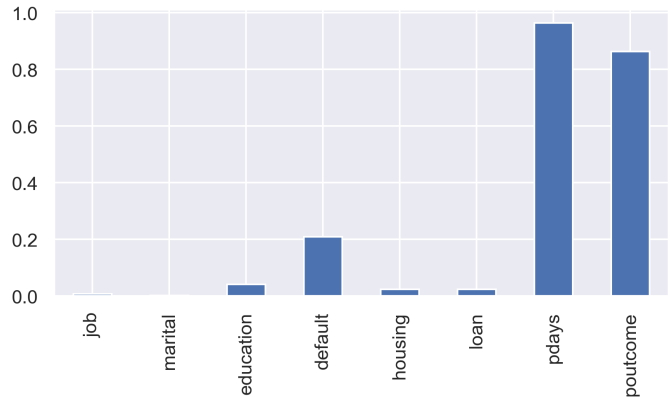


Figure 6: Percentage of missing values.

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 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.

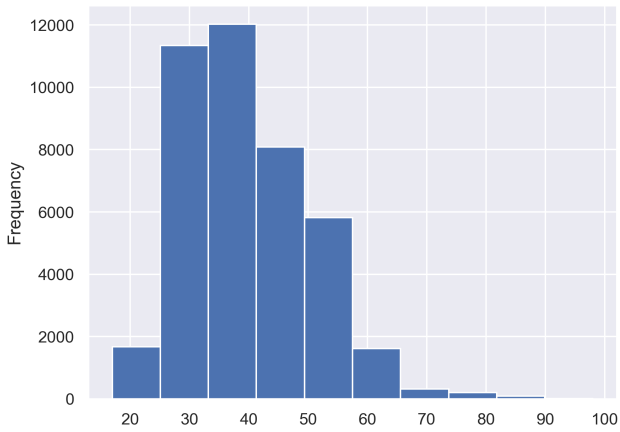


Figure 7: Age distribution histogram.

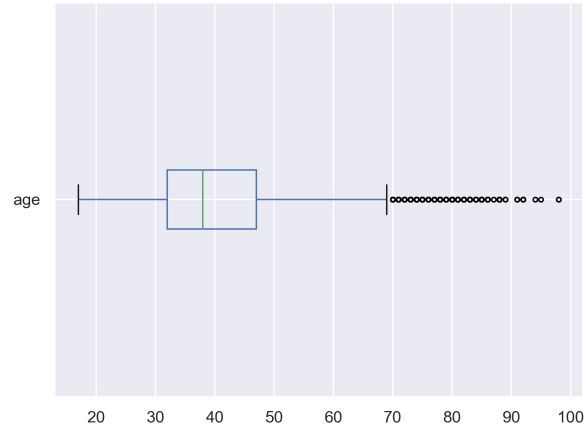


Figure 8: Age distribution box-plot.

`job`

This next graph shows the outcome 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.

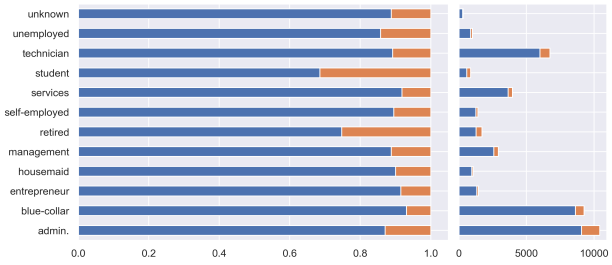


Figure 9: Outcome percentage and distribution by `job`.

`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 least and the most educated.

`default`

default is a peculiar feature that captures whether people previously had defaulted on credit, 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.

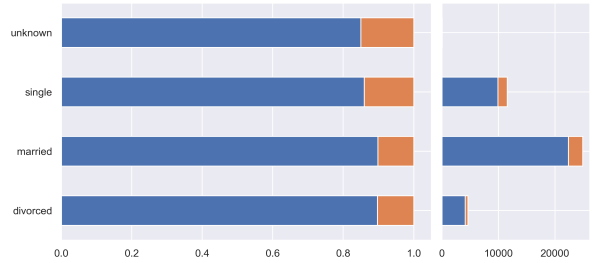


Figure 10: Outcome percentage and distribution by `education`.

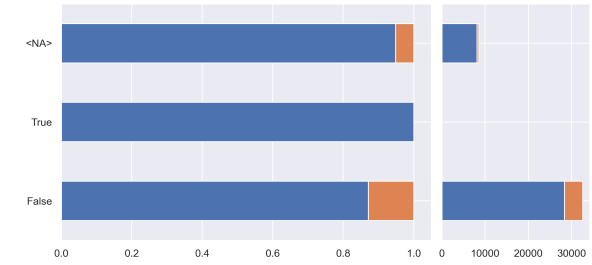


Figure 11: Outcome percentage and distribution by `default`.

`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.

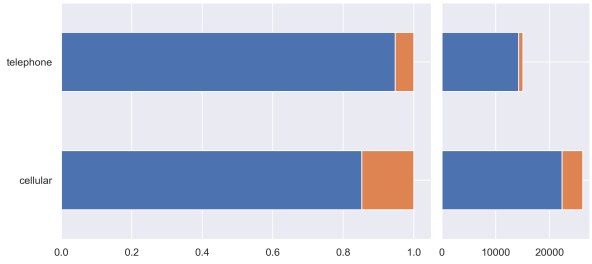


Figure 12: Outcome percentage and distribution by `contact`.

`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.

`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

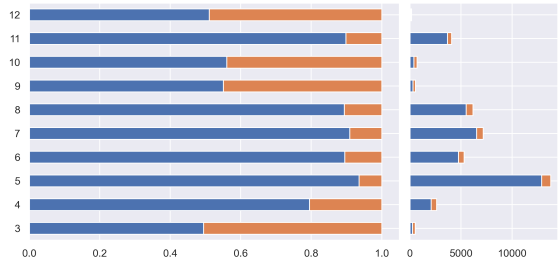


Figure 13: Outcome percentage and distribution by month.

significant amount of time and effort to deal with this feature, and this process will be discussed in the feature engineering section.

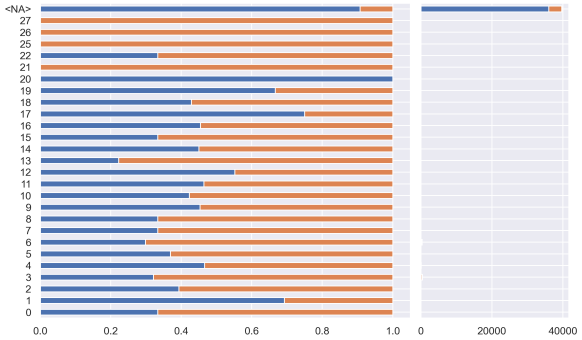


Figure 14: Outcome percentage and distribution by `pdays`.

`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.

`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 contradictions.

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 does, 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.

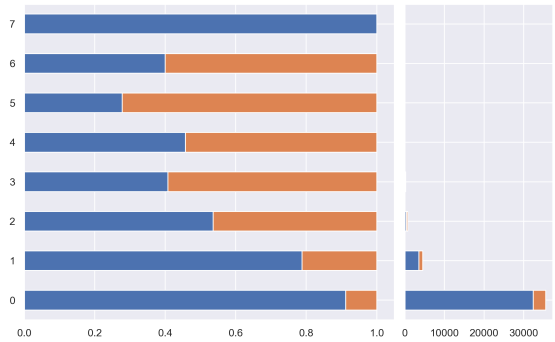


Figure 15: Outcome percentage and distribution by `previous`.

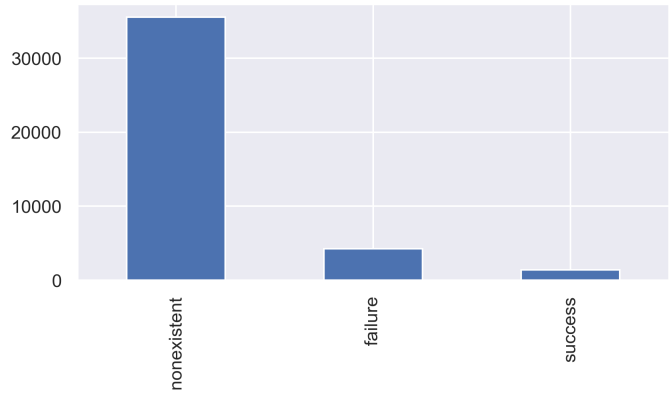


Figure 16: `poutcome` distribution.

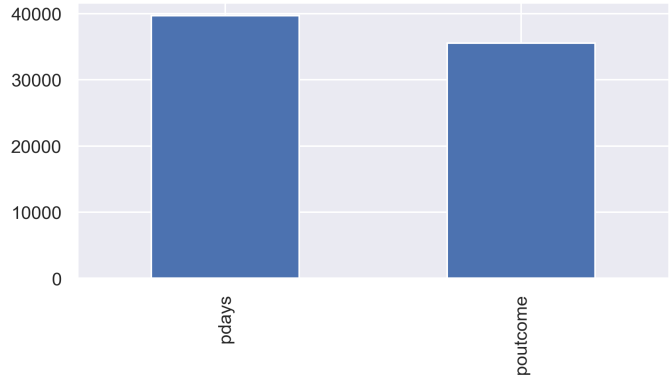


Figure 17: `pdays` and `poutcome`.

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.

Figure 18: Positive outcome percentage by `job` and `marital`.

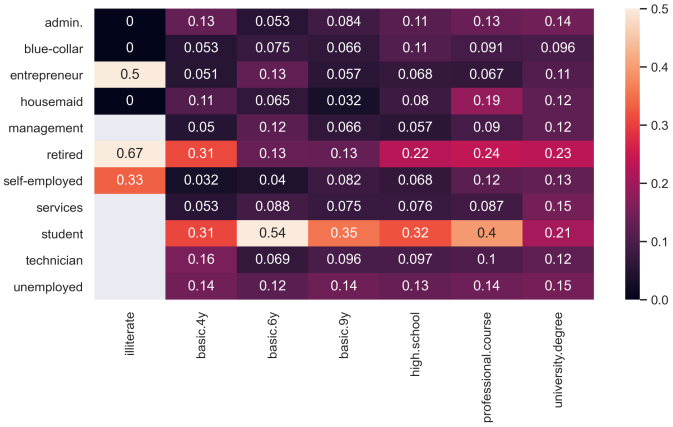


Figure 19: Positive outcome percentage by `job` and `education`.

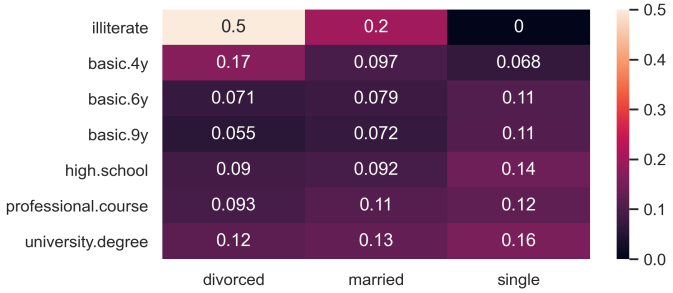


Figure 20: Positive outcome percentage by `education` and `marital`.

Key Numerical Features

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

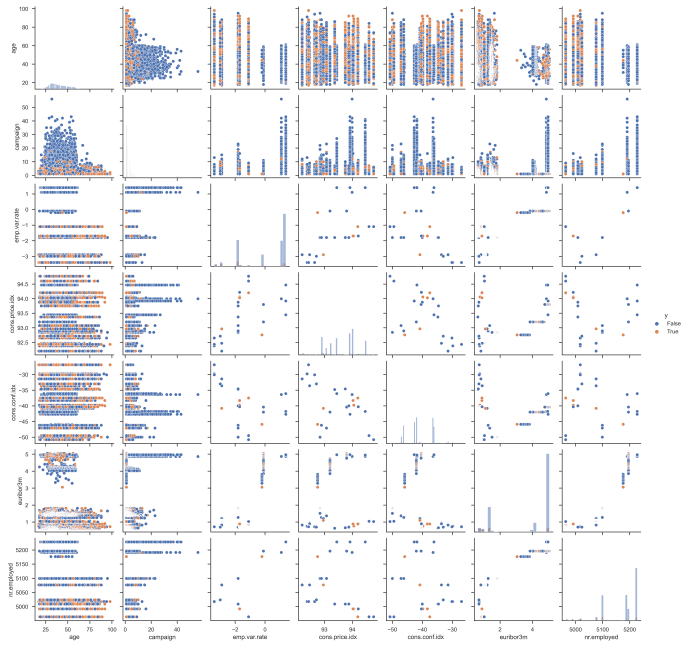


Figure 21: Scatterplots of key numerical features.

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

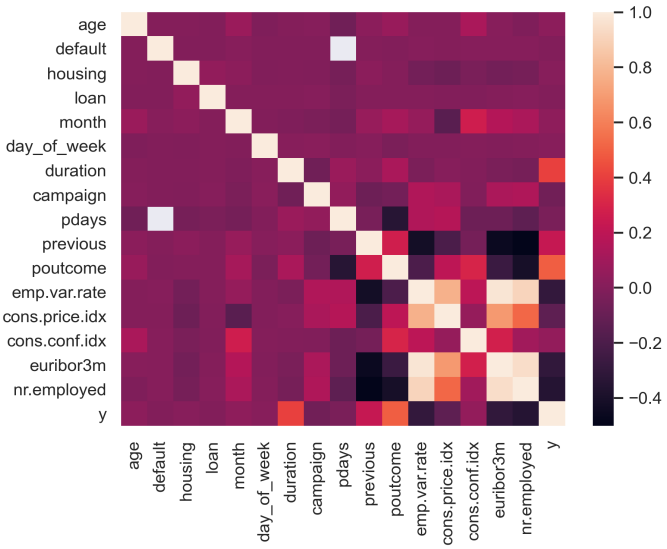


Figure 22: Correlation heatmap.

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 preparation or data transformation 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 the data preparation procesedures 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 code implementation will be provided in the appendix section.

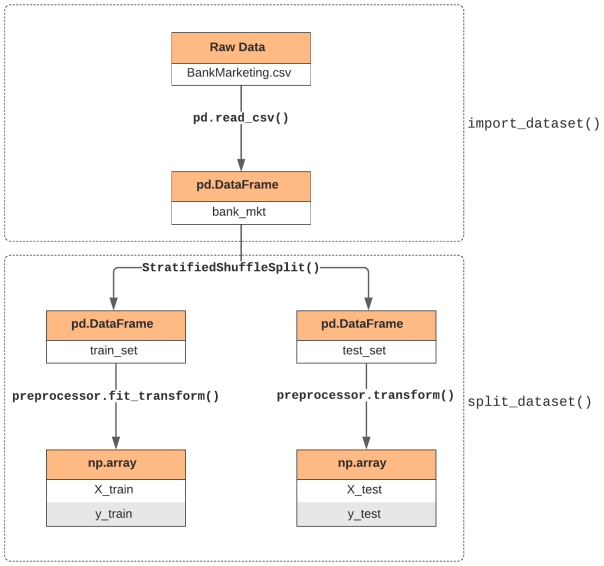


Figure 23: Data Life Cycle

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 to split the dataset into the train set and test set. `scikit-learn` models will then be trained and tuned on the train set. We only use the test set for final validation purposes. However, simply sampling the dataset may lead to unrepresentative 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 the test set and train set, which is shown below as a 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 suitable data type and shape for machine learning models. The train set and test set should also be preprocessed separately to avoid leaking test sample information into the train set. If we scale data using both the train set and test set, the scaling will ultimately be impacted by some test samples, which is not desired. 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 demonstrated 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. The simplest preprocessor is a function that does 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") if pd.api.types.is_bool_dtype(x) else x))
    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 to 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)
```

Workflows with Pipelines

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

1. Import data using `import_dataset()`;
2. Build a preprocessing pipeline `preprocessor`;
3. Build an estimator `clf`;
4. Tune the estimator by optimising model's performance on the train and validation set using `benchmark(data, preprocessor, clf)` or `scikit-learn`'s grid search functions;
5. Output the model's final performance on the test set using `benchmark(data, preprocessor, clf)`.

Such a workflow will be extensively reflected in the following chapters.

Feature Engineering

Feature Engineering is a practice of using domain knowledge to incorporate more features and improve machine learning models. Given that the dataset has relatively few features and rows, dimensionality reduction will not be applied. We instead focus on improving feature quality and generating new features, such as incorporating dates. As a result, our feature engineering approach includes missing value imputation, feature generation, feature transformations and adjusting sample weights.

Incorporate Dates

The most important feature in the dataset is the date of telemarketing which gives us the economic context. However, the dataset only includes `month` and `day_of_week` while `year` is missing. With the help of the dataset description, we find that the dataset is ordered by date and the telemarketing was conducted from May 2008 to November 2010. Even without this description, we can still reverse engineer the year information using the economic index, such as `CPI`. By manually inspecting `month`, we can infer each row's `year`.

```
bank_mkt = import_dataset("../data/BankMarketing.csv")
bank_mkt.loc[bank_mkt.index < 27682, "year"] = 2008
bank_mkt.loc[(27682 ⩽ bank_mkt.index) & (bank_mkt.index<39118), "year"] = 2009
bank_mkt.loc[39118 ⩽ bank_mkt.index, "year"] = 2010
bank_mkt["year"] = bank_mkt["year"].astype("int")
```

With `year` and `month`, we can approximate each marketing call's date as the start of the month.

```
bank_mkt["date"] = pd.to_datetime(bank_mkt[["month", "year"]].assign(day=1))
```

Simply feeding `date` into the models neither incorporates much information nor improve the model performance. Setting focal points instead is a much better way to treat date information. As discussed in chapter 2, there is a surge in success rate after the financial crisis. Therefore, we use the date when Lehman Brothers filed bankruptcy as the focal point and create a new feature `days` which is the days before or after Lehman Brothers bankruptcy.

```
bank_mkt["lehman"] = pd.to_datetime("2008-09-15")
bank_mkt["days"] = bank_mkt["date"] -  bank_mkt["lehman"]
bank_mkt["days"] = bank_mkt["days"].dt.days
```

Impute Missing Values

There are several strategies to handle missing values. The simplest way is to impute missing value as a different category, such as `-1`, depends on the context. For categorical data, `-1` is used. For `pdays`, both `999` and `-999` are used. Clients who have been contacted but do not have `pdays` record should be encoded as `999`, while clients who have not been contacted should be encoded as `-999`.

```
# Clients who have been contacted but do not have pdays record should be encoded as 999
bank_mkt.loc[bank_mkt["pdays"].isna() & bank_mkt["poutcome"].notna(), "pdays"] = 999
```

```
# Clients who have not been contacted should be encoded as -999
bank_mkt["pdays"] = bank_mkt["pdays"].fillna(-999)
# Fill other missing values as -1
bank_mkt = bank_mkt.fillna(-1)
```

It is also possible to impute missing values as the most frequent value using `SimpleImputer`.

```
from sklearn.impute import SimpleImputer

freq_features = ["job", "marital", "education", "default", "housing", "loan"]

freq_imputer = ColumnTransformer([
    ("freq_imputer", SimpleImputer(missing_values=-1, strategy="most_frequent"), freq_features)
], remainder="passthrough")

freq_encoder = make_pipeline(cat_encoder, freq_imputer)
X_train = freq_encoder.fit_transform(X_train, y_train)
X_test = freq_encoder.transform(X_test), axis=1)
```

Another imputation method worth mentioning is iterative imputation which attempts to estimate missing values. However, this approach may bring overfitting to the models.

```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer

ite_features = ["age", "job", "marital", "education", "default", "housing", "loan", "contact", "campaign", "m

ite_imputer = ColumnTransformer([
    ("ite_imputer",
     make_pipeline(
         IterativeImputer(max_iter=100, missing_values=-1, initial_strategy="most_frequent", random_state=42)
         FunctionTransformer(np.round)
     ),
     ite_features),
], remainder="passthrough")

ite_encoder = make_pipeline(cat_encoder, ite_imputer)
X_train = ite_encoder.fit_transform(X_train, y_train)
X_test = ite_encoder.transform(X_test), axis=1
```

Drop Demographic Features

To our surprise, the biggest improvement is achieved by dropping demographic features. Several reasons contribute to this result. First, the financial crisis and the debt crisis in Portugal may have lowered people's expectation of their future income and altered their investment choice. Thus, people reacted differently to the term deposit after the crisis and demographic data does not matter anymore. Second, we do not have enough data in 2009 and 2010 to learn this abnormal shift of attitude and therefore demographic data becomes noise.

```
drop_features = ["age",
                 "job",
                 "marital",
                 "education",
                 "housing",
                 "loan",
                 "default",
                 "duration",
```

```
                "y"]
bank_mkt = bank_mkt.drop(drop_features, axis=1)
```

Performance Evaluation

As discussed in chapter 3, we split data into different sets and tune hyperparameters by comparing model performance on these sets. How do we define the performance of machine learning models? A plain and simple approach is using accuracy scores:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Number of Total Predictions}}$$

However, the accuracy score may be too simplistic. First, it does not reflect the unbalanced outcome of our dataset. If the model predicts every outcome as negative while the majority of the outcomes is negative, the model can still achieve a very high accuracy score. Second, the accuracy score does not distinct false positive and false negative errors which may cost the business differently. In the bank marketing context, missing a potential customer (false negative) costs more than phoning an uninterested buyer (false positive). These problems can be solved by using a confusion matrix and utilising more metrics for model evaluation.

Confusion Matrix

The confusion matrix is a contingency table that outputs the counts of the true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions. We can use an analogy of fishing to gain more intuitions: The sea consists of fish and rubbish. A fisherman throws the net into the sea and hopes to capture as many fish and as little rubbish as possible. After the fisherman retrieves the net, the fish in the net (the wanted capture) is a true positive outcome, the rubbish in the net (the unwanted capture) is a false positive outcome, the fish in the sea (the wanted leftover) is a false negative outcome, and the rubbish in the sea (the unwanted leftover) is the true negative outcome.

The confusion matrix can be computed by calling `confusion_matrix` in `scikit-learn` as follows:

```
from sklearn.model_selection import cross_val_predict
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix

bank_mkt = import_dataset("../data/BankMarketing.csv")
preprocessor = FunctionTransformer(dftransform)
X_train, y_train, *other_sets = split_dataset(bank_mkt, preprocessor)
clf = KNeighborsClassifier(n_neighbors=10)
y_pred = cross_val_predict(clf, X_train, y_train, cv=5, n_jobs=-1)
conf_mat = confusion_matrix(y_train, y_pred)
f, ax = plt.subplots()
conf_ax = sns.heatmap(
    conf_mat, ax=ax, annot=True, fmt="", cmap=plt.cm.Blues, cbar=False
)
conf_ax.set_xlabel("Predicted")
conf_ax.set_ylabel("True")
```

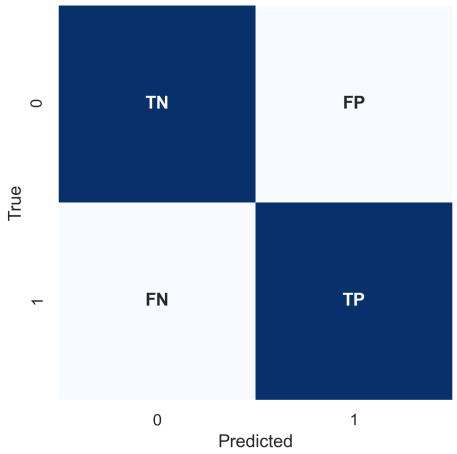


Figure 24: Confusion Matrix

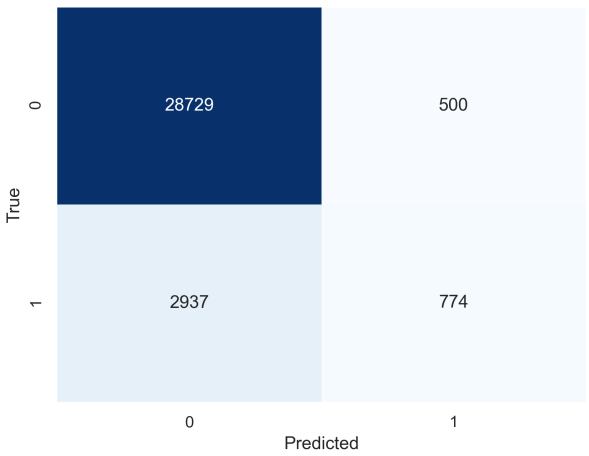


Figure 25: The confusion matrix of a KNN classifier.

Metrics From Confusion Matrix

From the Confusion Matrix, we can derive some key performance metrics such as precision and recall.

Precision (PRE) measures the accuracy of the predicted positive outcomes. As in the fisherman analogy, precision is the fish in the net divided by the total number of objects (fish and rubbish) in the net.

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

Recall (REC) measures the accuracy of the positive samples, also known as sensitivity or the true positive rate (TPR). Using the fishing analogy, recall is the fish in the net divided by the fish population in total.

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

Precision and recall are usually trade-offs. A fisherman can narrow his net to capture more fish and therefore increase recall. However, it comes at the cost of capturing more rubbish at the same time and decrease precision. To balance these two metrics, the harmonic mean of precision and recall (F1) is used.

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

The trade-off logic also applies to the true negative rate (TNR) and the true positive rate (TPR). The true negative rate measures the accuracy of the negative samples and the true positive rate measures the accuracy of the negative samples. The false positive rate is simply one minus the true positive rate.

$$\begin{aligned} FPR &= \frac{FP}{N} = \frac{FP}{FP + TN} \\ TNR &= \frac{TN}{N} = \frac{TN}{TN + FP} = 1 - FPR \\ TPR &= \frac{TP}{P} = \frac{TP}{TP + FN} \end{aligned}$$

Balanced accuracy (bACC) is the average of true positive rate and true negative rate.

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

A performance metric table of classifiers on bank marketing dataset is shown below.

	Constant Prediction	Random Prediction	K-Nearest Neighbors	Linear SVM	Decision Tree	Logistic Regression
FPR	1	0.498957	0.0171063	0.00526874	0.143419	0.25875
TNR	0	0.501043	0.982894	0.994731	0.856581	0.74125
TPR	1	0.495284	0.208569	0.0382646	0.613851	0.698733
bACC	0.5	0.498164	0.595731	0.516498	0.735216	0.719992
REC	1	0.495284	0.208569	0.0382646	0.613851	0.698733
PRE	0.112659	0.111923	0.607535	0.47973	0.352087	0.255317
F1	0.202505	0.182586	0.310532	0.070876	0.4475	0.373981

Metrics From Decision Function

As hinted in the precision-recall trade-off, fisherman can narrow or loss his net. Bank may be happy to capture more potential customers by phoning more clients. A classifier can also adjust its threshold and therefore achieves different precision and recall results. For example, a logistic regression classifier uses the following decision function to distinct the label 0 and 1. When the result of the decision function is 0, the probability for each label is 0.5.

$$\text{Decision Function} = b_0 + b_1 x_1 + ... + b_k x_k$$

The following code plots the logistic regression's precision and recall against the threshold.

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import precision_recall_curve

bank_mkt = import_dataset("../data/BankMarketing.csv")
preprocessor = FunctionTransformer(dftransform)
X_train, y_train, *other_sets = split_dataset(bank_mkt, preprocessor)
clf = LogisticRegression(class_weight="balanced")
y_score = cross_val_predict(clf, X_train, y_train, cv=5, method="decision_function")
f, ax = plt.subplots()
precisions, recalls, thresholds = precision_recall_curve(y_train, y_score)
pre_rec_df = pd.DataFrame(
    {"Precision": precisions[:-1], "Recall": recalls[:-1]}, index=thresholds
)
pre_rec_ax = pre_rec_df.plot.line(ax=ax, ylim=(-0.05, 1.05))
threshold = 0
pre_rec_ax.plot((threshold, threshold), (-2, 2), linestyle="--", linewidth=1)
```

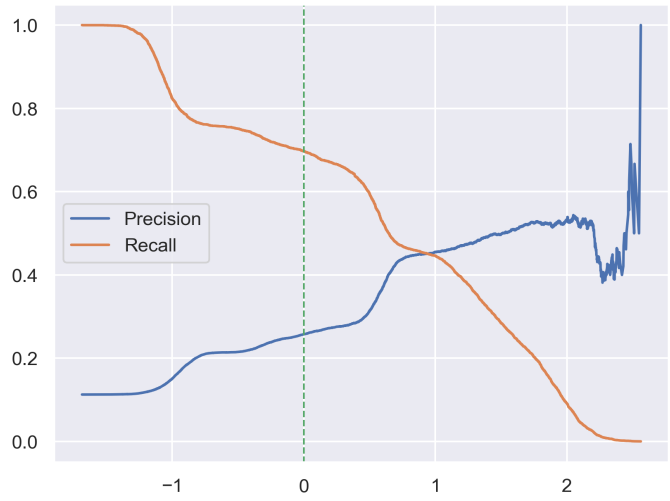


Figure 26: Precision-recall curve against the threshold.

We can also plot precision and recall against each other as shown in the following graph. For a marketing campaign, the bank wants to capture as many potential clients as possible given certain budget constraints. Therefore the precision-recall curve should be pushed as far as possible. However, such a mechanism is not reflected by the metrics derived from confusion matrix and F1 can be biased towards models with equal precision and recall.

To solve this problem, we introduce receiver operating characteristic (ROC) and average precision (AP) as alternative metrics that incorporate model thresholds. Average precision (AP) summarises

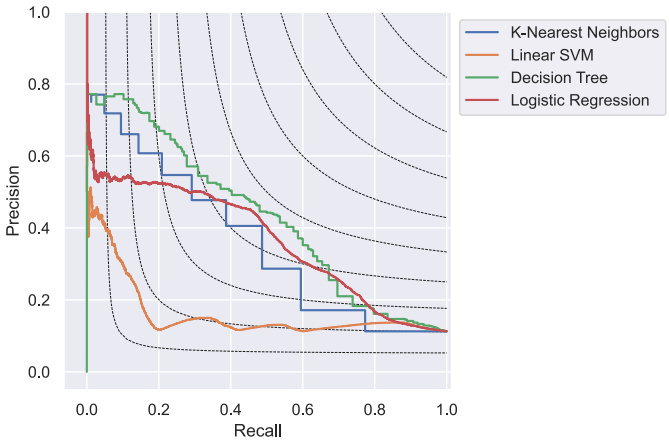


Figure 27: Precision-recall curve with F1 contour.

the area under the precision-recall curve as the weighted mean of precisions achieved at each threshold, with the increase in recall from the previous threshold used as the weight.

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

A receiver operating characteristic (ROC) adopts the same logic by plotting TPR against FPR at various threshold settings and calculating the area under the curve, plotted as follows.

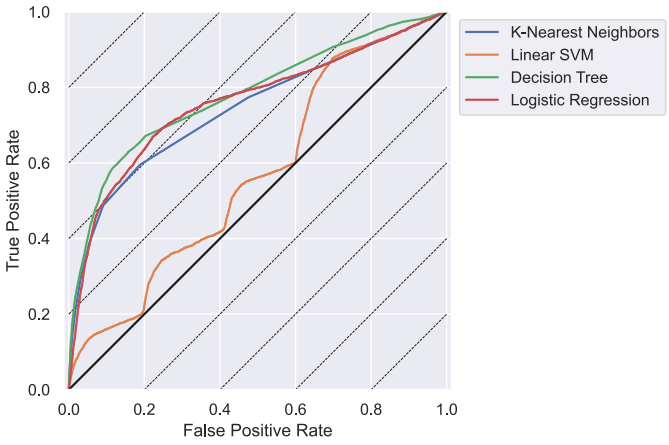


Figure 28: ROC curve with bACC contour.

Performance Evaluation in Practice

In practice, we utilise multiple metrics to evaluate and optimise our models. When metrics show conflicting results, we prioritise AP for two reasons. First, AP has a range between the minority class percentage and 1. It gives a more straightforward picture of prediction improvements in an imbalanced dataset. Second, AP puts more weight on positive outcomes. In our case, losing a potential subscriber costs the bank more than phoning an uninterested buyer. AP matches this reality more when we evaluate model performance.

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 gave it a try.

$$h(x) = \frac{1}{1 + e^{-\theta_0+\theta_1 x_1+\theta_2 x_2+\theta_3 x_3+\ldots+\theta_n x_n}}$$

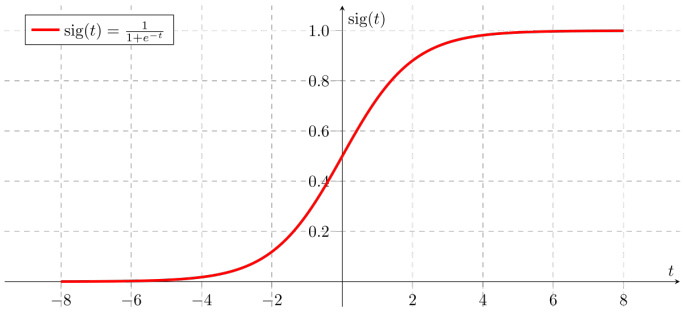


Figure 29: Logistic Sigmoid Function

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 AUC 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

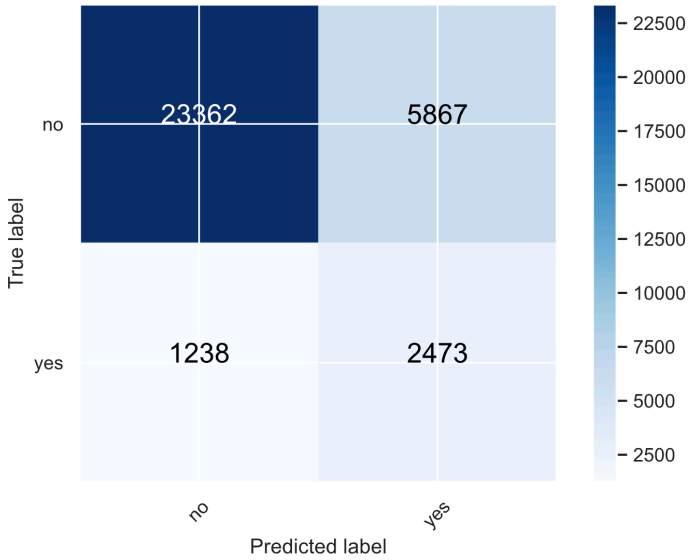


Figure 30: Confusion matrix for Train set.

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'],
```

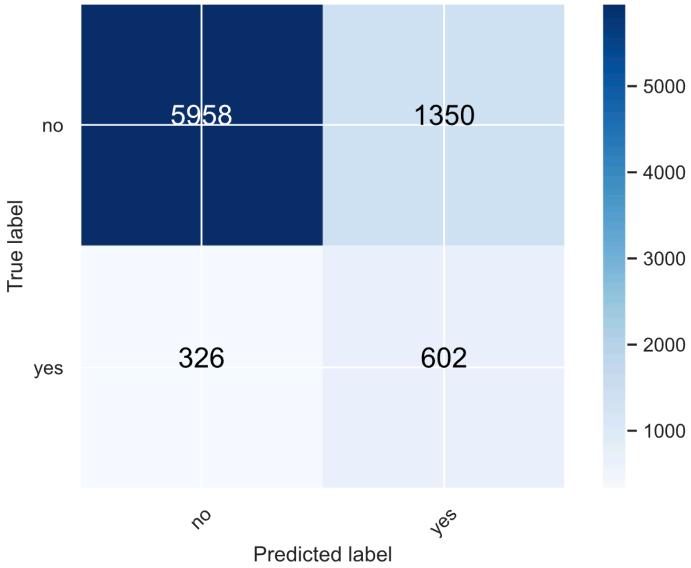


Figure 31: Confusion matrix for Test set.

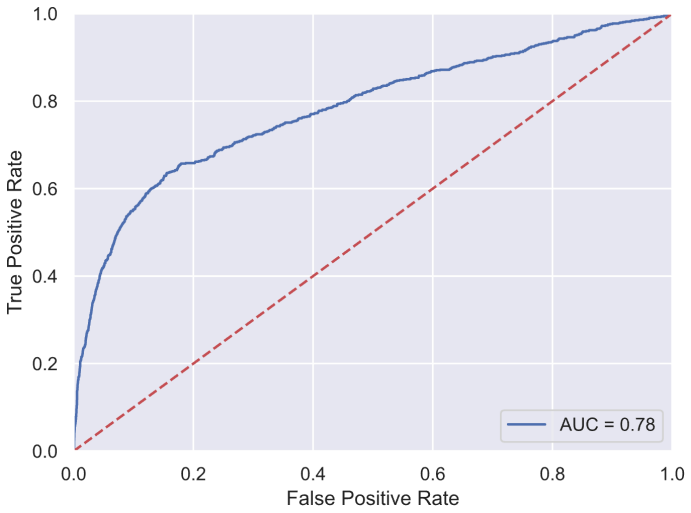


Figure 32: ROC

```
              '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)
```

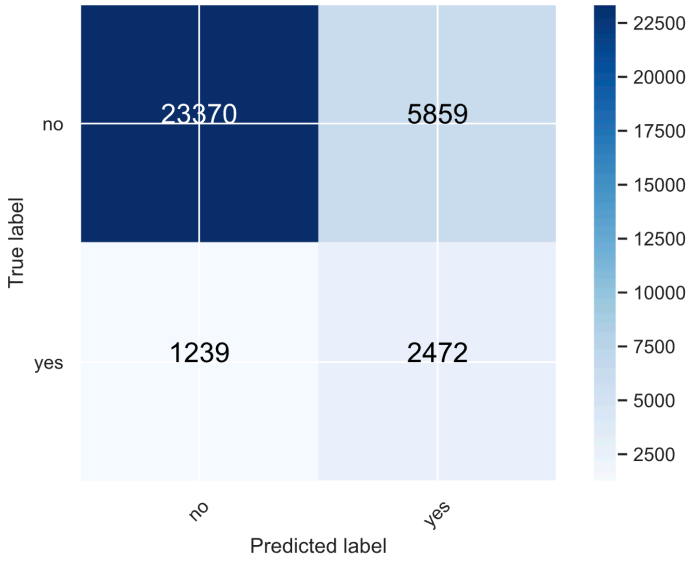


Figure 33: Confusion matrix for Train set.

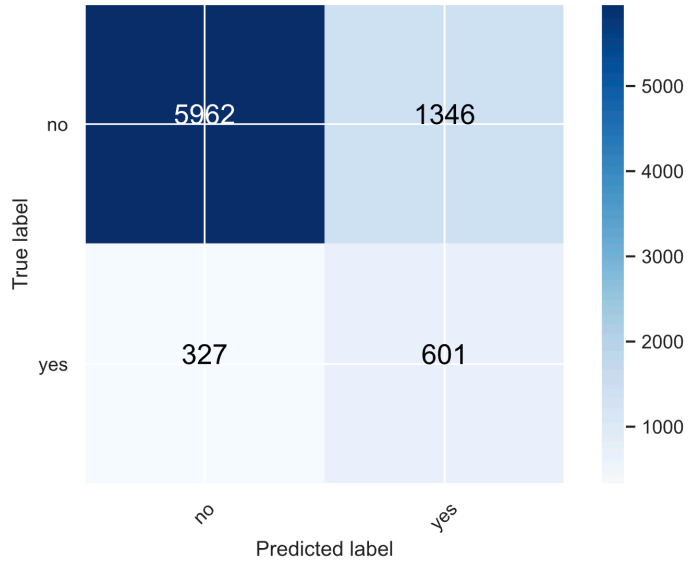


Figure 34: Confusion matrix for Test set.

```
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.

	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

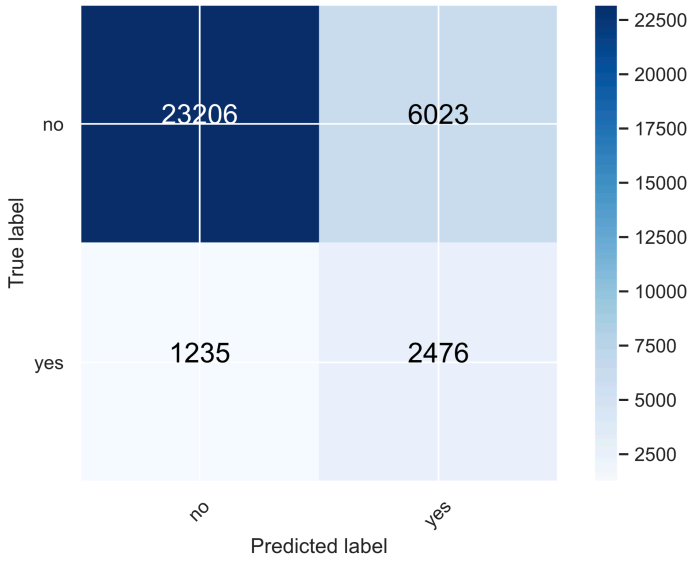


Figure 35: Confusion matrix for Train set.

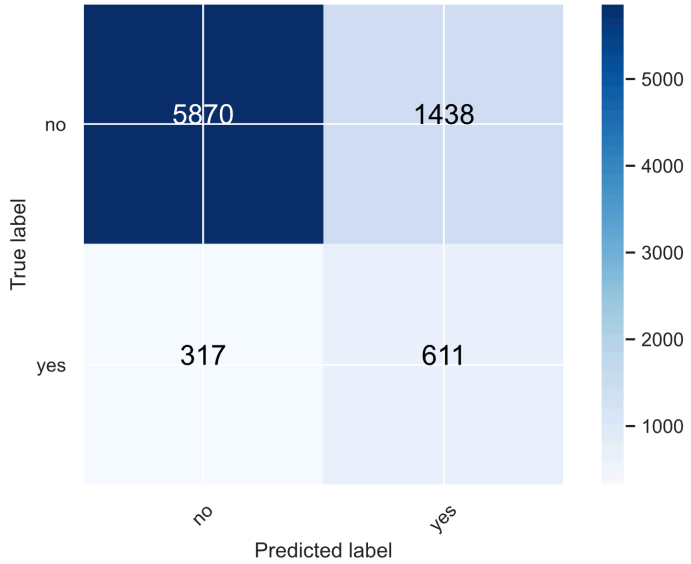


Figure 36: Confusion matrix for Test set.

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 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 37: Statistical Summary.

Support Vector Machine

The purpose of the Support Vector Machine is to find a hyperplane that distinctly classifies the data points in an N-dimensional space. The hyperplane is the decision boundary that classifies the data points and support vectors are the data points that are closer to the hyperplane which influences 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 observation from the hyperplane. The margin brings in extra confidence that the further data points will be classified correctly.

Hyperparameters

Maximum Margin

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

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

where

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

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

By replacing the individual loss contribution to $max(0, 1-\theta^T \cdot x^i)$ and $max(0, 1+\theta^T \cdot x^i)$ 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.

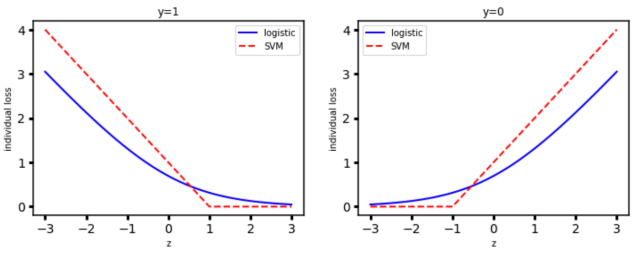


Figure 38: LG SVM Comparison

Regularizaiton

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 sensitive to a change in the data points and may lead to overfitting problems.

To achieve the balance of 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 future data points.

Hyperparameter C in SVM allows us to dictate the tradeoff between having a wide margin and correctly classifying 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 observation 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 it utilizes the existing features and applies 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: 0.43354326475418103

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=42)

# 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

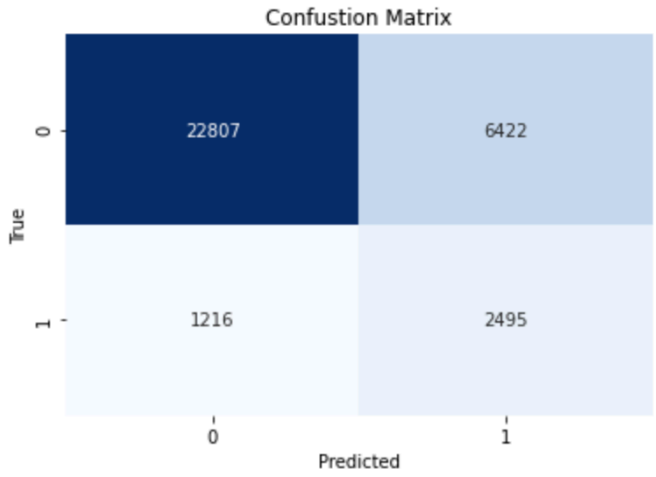


Figure 39: Linear_svm_train_conf

Non-Linear SVM

We use the 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.

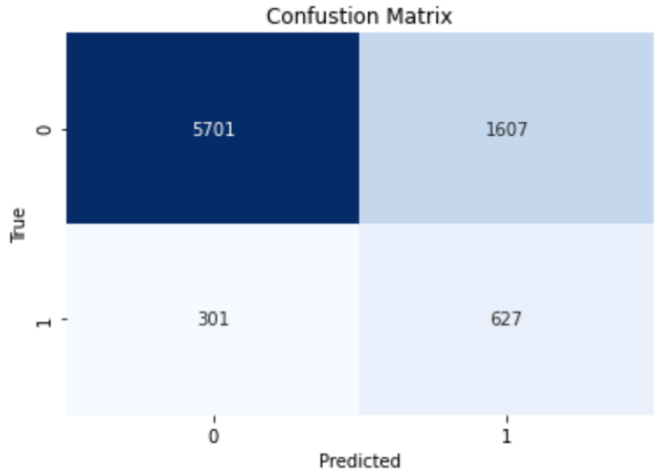


Figure 40: Linear_svm_test_conf

```
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},
# with mean test score: 0.4350168498949894

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: 0.4403394302575112

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

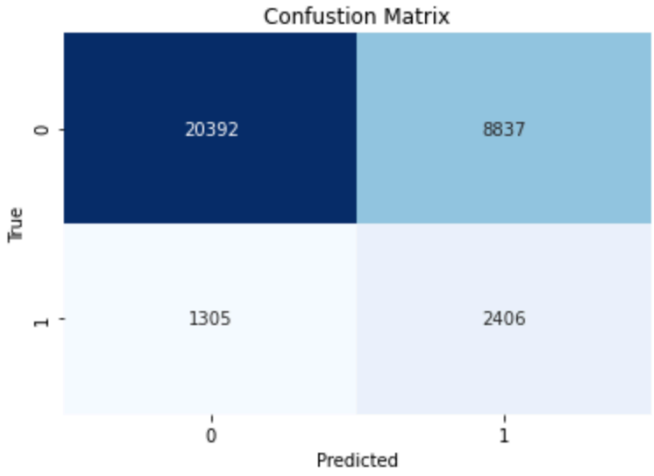


Figure 41: Non_linear_sgs_train_conf

```
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,
```

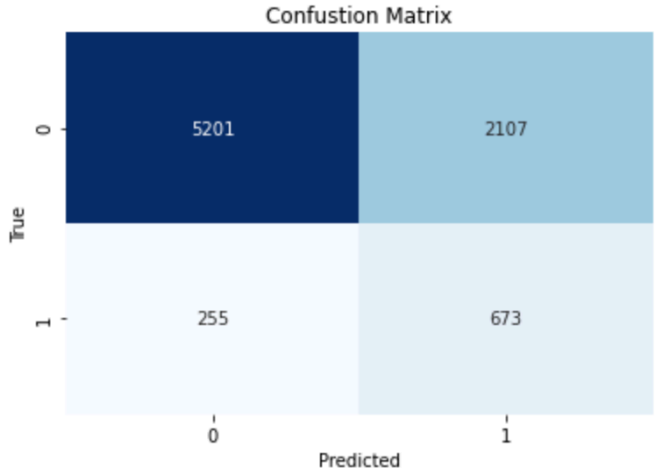


Figure 42: Non_linear_sgs_test_conf

```
                        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 mean test score: 0.4381080007088255

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: 0.43986477417883973

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

	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	0.688103
PRE	0.289580	0.288175	0.292453
AP	0.437404	0.453640	0.440392

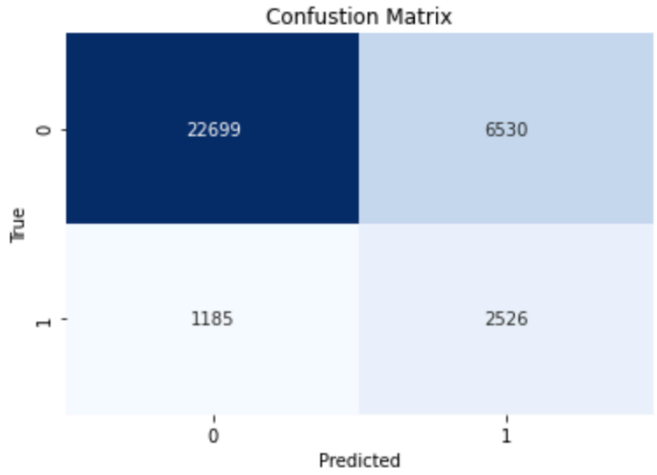


Figure 43: Non_linear_svc_train_conf

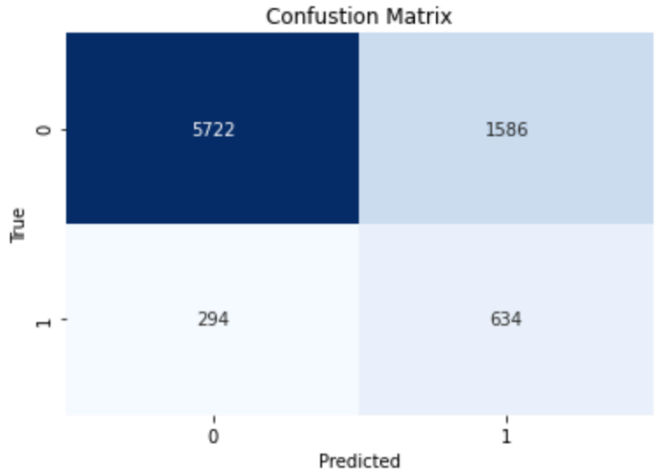


Figure 44: Non_linear_svc_test_conf

(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 layers;
3. An output layer;
4. A set of weights and biases between each layer;
5. An activation function for each hidden 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 Forward Propagation and Backpropagation. The Forward Propagation is a process where the input data is fed in the forward direction. Each hidden layer accepts the input data and processes it as per the activation function and passes the results to the successive layer. The Backpropagation is to fine tune the weights of a neural net based on error rate obtained in the previous iteration. Error is calculated between the expected outputs and the outputs forward propagated from the network. These errors are then propagated backwards from the output layer to the hidden layer, finding contributions to the errors and updating the weights.

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 a 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 minimum 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 architecture of the neural network.

We want to keep the neural network architecture 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

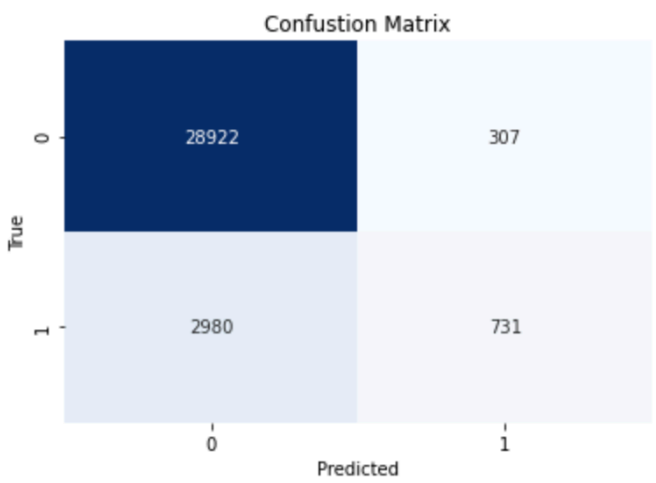
Activation functions are mathematical equations attached to each neuron in the network and they determine the output of the learning model, the accuracy and the computational efficiency of training the model.

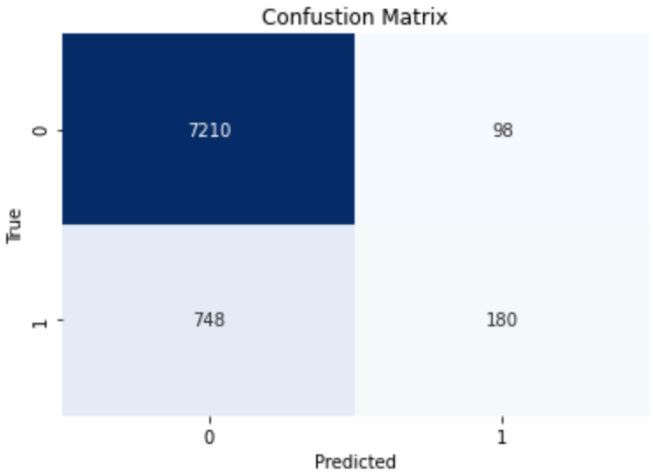
With the use of Non-linear activation function, we can create a complex mapping between the network's inputs and outputs which are essential for learning and modelling complex data.

Grid Search

```
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	0.310145
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





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.

(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:** 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:** The minimum number of samples required to split an internal node.
- **criterion:** 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:** The minimum number of samples required to be at a leaf node.
- **class_weight:** Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one.
- **random_state:** 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 first run it through our standard transformer, explained in the pipeline section of this text.

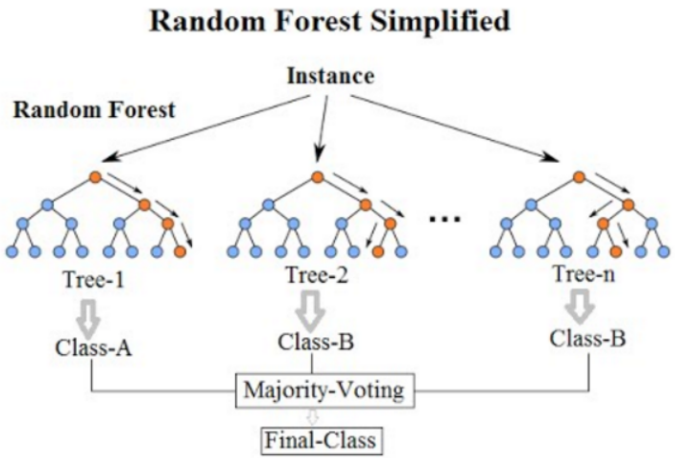


Figure 45: Random Forest

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

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

- **n_estimators:** 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="auto",
                            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=2)
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 provide with highest model oscillations. The hyperparameters that were selected for the RandomForestClassifier in the definition of variable RF are 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=6, n_estimators= 1750,random_state=42)
```

We decide that 1750 estimators in subsequent testing show 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_estimators=1500,
                                       max_leaf_nodes=1000)
benchmark(bank_mkt, tree_transformer, RF_validation)
```

	Train	Validate	Test
TNR	0.866655	0.864865	0.870963
TPR	0.627821	0.629380	0.618534
bACC	0.747238	0.747122	0.744749
ROC	0.813153	0.801666	0.801848
REC	0.627821	0.629380	0.618534
PRE	0.374147	0.371519	0.378378
AP	0.503536	0.449784	0.474725

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 = 0.0819466885063493
cons.price.idx = 0.06185600062913342
```

Figure 46: Table of Feature Importance

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.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.size=(15,10)
plt.show()
```

All in all the Random Forest lived up to its expectation. By adding just a small amount of bias

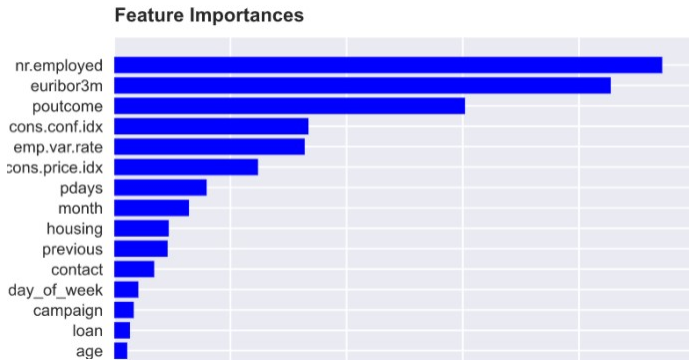


Figure 47: Feature importance

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:** 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:** Learning rate shrinks the contribution of each classifier by `learning_rate`. There is a trade-off between `learning_rate` and `n_estimators`.
- **n_estimators:** 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=5)
```

```
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(max_depth=1),
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_samples_split=2))

benchmark(bank_mkt, tree_transformer, AB_validation)
```

	Train	Validate	Test
TNR	0.985982	0.972650	0.979312
TPR	0.393488	0.262162	0.272630
bACC	0.689715	0.617406	0.625971
ROC	0.878366	0.743267	0.757640
REC	0.393448	0.262162	0.272630
PRE	0.780308	0.58023	0.628933
F1	0.523125	0.354662	0.380386
AP	0.632423	0.374018	0.423666

In a similar fashion we derive feature importance as well:

```
for name, importance in zip(columns, AB_validation.feature_importances_):
    print(name, "=", importance)
importances = AB_validation.feature_importances_
indices = np.argsort(importances)
plt.barh(range(len(indices)), importances[indices])
plt.yticks(range(len(indices)), [columns[i] for i in indices])
plt.xlabel("Relative Importance")
plt.size=(15,10)
plt.show()
```

```
nr.employed = 0.24512965121119204
euribor3m = 0.2190410622256826
poutcome = 0.1397408073745672
emp.var.rate = 0.08445578032649204 62
```

Figure 48: Feature importance

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 the paper Random Forests (Breiman, 1999), the author states the following conjecture: “AdaBoost is

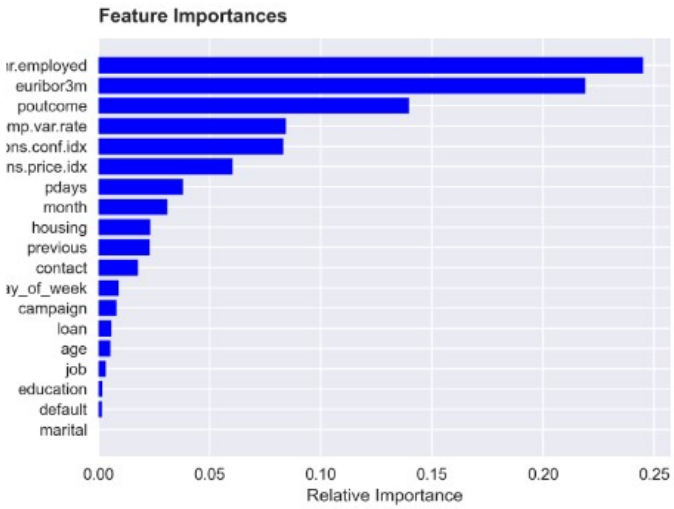


Figure 49: Feature importance

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.

XGBoost

Conclusion

References

'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).

Appendix 1: `import_dataset`

```
def import_dataset(filename):
    """
    Import the dataset from the path.

    Parameters
    ----------
        filename : str
            filename with path

    Returns
    -------
        data : DataFrame

    Examples
    --------
        bank_mkt = import_dataset("../data/BankMarketing.csv")
    """
    bank_mkt = pd.read_csv(filename,
                           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)
    # `month` will be encoded to the corresponding number, e.g. "mar" -> 3
    month_map = {"mar": 3,
                 "apr": 4,
                 "may": 5,
                 "jun": 6,
                 "jul": 7,
                 "aug": 8,
                 "sep": 9,
                 "oct": 10,
                 "nov": 11,
                 "dec": 12}
    bank_mkt["month"] = bank_mkt["month"].replace(month_map)
    # `day_of_week` will be encoded to the corresponding number, e.g. "wed" -> 3
    dow_map = {"mon": 1,
               "tue": 2,
               "wed": 3,
               "thu": 4,
               "fri": 5}
    bank_mkt["day_of_week"] = bank_mkt["day_of_week"].replace(dow_map)
    # Convert types, "Int64" is nullable integer data type in pandas
    bank_mkt = bank_mkt.astype(dtype={"age": "Int64",
```

```
                                    "job": "category",
                                    "marital": "category",
                                    "education": "category",
                                    "default": "boolean",
                                    "housing": "boolean",
                                    "loan": "boolean",
                                    "contact": "category",
                                    "month": "Int64",
                                    "day_of_week": "Int64",
                                    "duration": "Int64",
                                    "campaign": "Int64",
                                    "pdays": "Int64",
                                    "previous": "Int64",
                                    "poutcome": "boolean",
                                    "y": "boolean"})
    # Drop 12 duplicated rows
    bank_mkt = bank_mkt.drop_duplicates().reset_index(drop=True)
    # reorder ordinal categorical data
    bank_mkt["education"] = bank_mkt["education"].cat.reorder_categories(
        ["illiterate", "basic.4y", "basic.6y", "basic.9y", "high.school", "professional.course", "university.
    return bank_mkt
```

Appendix 2: `split_dataset`

```
def split_dataset(data, preprocessor=None, random_state=62):
    """
    Split dataset into train, test and validation sets using preprocessor.
    Because the random state of validation set is not specified, the validation set will be different each ti

    Parameters
    ----------
        data : DataFrame

        preprocessor : Pipeline

        random_state : int

    Returns
    -------
        datasets : tuple

    Examples
    --------
        from sklearn.preprocessing import OrdinalEncoder
        data = import_dataset("../data/BankMarketing.csv").interpolate(method="pad").loc[:, ["job", "educatio
        # To unpack all train, test, and validation sets
        X_train, y_train, X_test, y_test, X_ttrain, y_ttrain, X_validate, y_validate = split_dataset(data, Or
        # To unpack train and test sets.
        X_train, y_train, X_test, y_test, *other_sets = split_dataset(data, OrdinalEncoder())
        # To unpack test and validation set
        *other_sets, X_test, y_test, X_ttrain, y_ttrain, X_validate, y_validate = split_dataset(data, Ordinal
        # To unpack only train set.
        X_train, y_train, *other_sets = split_dataset(data, OneHotEncoder())
    """
    train_test_split = StratifiedShuffleSplit(
        n_splits=1, test_size=0.2, random_state=random_state)
    for train_index, test_index in train_test_split.split(data.drop("y", axis=1), data["y"]):
        train_set = data.iloc[train_index]
        test_set = data.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()

    train_validate_split = StratifiedShuffleSplit(n_splits=1, test_size=0.2)
    for ttrain_index, validate_index in train_validate_split.split(X_train, y_train):
        ttrain_set = train_set.iloc[ttrain_index]
```

```
    validate_set = train_set.iloc[validate_index]

X_ttrain = ttrain_set.drop(["duration", "y"], axis=1)
y_ttrain = ttrain_set["y"].astype("int").to_numpy()
X_validate = validate_set.drop(["duration", "y"], axis=1)
y_validate = validate_set["y"].astype("int").to_numpy()

if preprocessor =/= None:
    X_train = preprocessor.fit_transform(X_train, y_train)
    X_test = preprocessor.transform(X_test)
    X_ttrain = preprocessor.fit_transform(X_ttrain, y_ttrain)
    X_validate = preprocessor.transform(X_validate)

return (X_train, y_train, X_test, y_test, X_ttrain, y_ttrain, X_validate, y_validate)
```

Appendix 3: `benchmark`

```
def benchmark(data, preprocessor=None, clf=None):
    """
    Benchmark preprocessor and clf's performance on train, validation and test sets.
    All the data transformation should be handled by preprocessor and estimation should be handled by clf.

    Parameters
    ----------
        data : DataFrame

        preprocessor : Pipeline, default = None

        clf : estimator, default = None

    """
    X_train, y_train, X_test, y_test, X_ttrain, y_ttrain, X_validate, y_validate = split_dataset(
        data, preprocessor)
    X_sets = [X_ttrain, X_validate, X_test]
    y_sets = [y_ttrain, y_validate, y_test]

    metric_names = ["TNR", "TPR", "bACC", "ROC", "REC", "PRE", "AP"]
    set_names = ["Train", "Validate", "Test"]
    metric_df = pd.DataFrame(index=metric_names, columns=set_names)

    try:
        clf.fit(X_ttrain, y_ttrain, eval_set=(
            X_validate, y_validate), verbose=False)
    except (ValueError, TypeError):
        clf.fit(X_ttrain, y_ttrain)

    for name, X, y in zip(set_names, X_sets, y_sets):
        # Re-fit model on train set before test set evaluation except CatBoost
        if name == "Test" and not isinstance(clf, CatBoostClassifier):
            clf.fit(X_train, y_train)
        y_pred = clf.predict(X)

        try:
            y_score = clf.decision_function(X)
        except AttributeError:
            y_score = clf.predict_proba(X)[:, 1]

        metrics = [recall_score(y, y_pred, pos_label=0),
                   recall_score(y, y_pred),
                   balanced_accuracy_score(y, y_pred),
                   roc_auc_score(y, y_score),
```

```
                recall_score(y, y_pred),
                precision_score(y, y_pred),
                average_precision_score(y, y_score)]
    metric_df[name] = metrics

return metric_df
```

Appendix 4: `dftransform`

```
def dftransform(X,
                drop=None,
                cut=None,
                gen=None,
                cyclic=None,
                target=None,
                fillna=True,
                to_float=False):
    """
    Encode, transform, and generate categorical data in the dataframe.

    Parameters
    ----------
        X : DataFrame

        drop : list, default = None

        gen : list, default = None

        cut : list, default = None

        external : list, default = None

        cyclic : list, default = None

        fillna : boolean, default = True

        to_float : boolean, default = False

    Returns
    -------
        X : DataFrame

    Examples
    --------
    bank_mkt = import_dataset("../data/BankMarketing.csv")
    X = dftransform(bank_mkt)
    """
    X = X.copy()

    if gen =≠ None:
        if "year" in gen or "days" in gen:
            X.loc[X.index < 27682, "year"] = 2008
            X.loc[(27682 ⩽ X.index) & (X.index < 39118), "year"] = 2009
```

```
            X.loc[39118 <= X.index, "year"] = 2010
            X["year"] = X["year"].astype("int")
        if "days" in gen:
            X["date"] = pd.to_datetime(X[["month", "year"]].assign(day=1))
            X["lehman"] = pd.to_datetime("2008-09-15")
            X["days"] = X["date"] - X["lehman"]
            X["days"] = X["days"].dt.days
            X = X.drop(["lehman", "year", "date"], axis=1)
        if "has_previous" in gen:
            X["has_previous"] = X["previous"] > 0
        if "has_default" in gen:
            X["has_default"] = X["default"].notna()
        if "has_marital" in gen:
            X["has_marital"] = X["marital"].notna()

    if cut != None:
        if "pdays" in cut:
            # Cut pdays into categories
            X["pdays"] = pd.cut(X["pdays"], [0, 3, 5, 10, 15, 30, 1000], labels=[
                                3, 5, 10, 15, 30, 1000], include_lowest=True).astype("Int64")

    if cyclic != None:
        if "month" in cyclic:
            X['month_sin'] = np.sin(2 * np.pi * X["month"]/12)
            X['month_cos'] = np.cos(2 * np.pi * X["month"]/12)
            X = X.drop("month", axis=1)
        if "day_of_week" in cyclic:
            X['day_sin'] = np.sin(2 * np.pi * X["day_of_week"]/5)
            X['day_cos'] = np.cos(2 * np.pi * X["day_of_week"]/5)
            X = X.drop("day_of_week", axis=1)

    # Transform target encoded feature as str
    if target != None:
        X[target] = X[target].astype("str")

    # Other categorical features will be coded as its order in pandas categorical index
    X = X.apply(lambda x: x.cat.codes if pd.api.types.is_categorical_dtype(
        x) else (x.astype("Int64") if pd.api.types.is_bool_dtype(x) else x))

    if fillna:
        # Clients who have been contacted but do not have pdays record should be encoded as 999
        # Clients who have not been contacted should be encoded as -999
        X.loc[X["pdays"].isna() & X["poutcome"].notna(), "pdays"] = 999
        X["pdays"] = X["pdays"].fillna(-999)
        # Fill other missing values as -1
        X = X.fillna(-1)
    else:
        X = X.astype("float")

    if drop != None:
        # Drop features
        X = X.drop(drop, axis=1)

    if to_float:
        X = X.astype("float")
```

```
return X
```