Apprendimento Statistico

Nenna Giulio

Data Analysis applied to a Marketing Dataset

1 Introduction

The main goal of this report is to apply different machine learning tecniques to a dataset that can be found on the UCI Machine Learning Repository. The data is related with direct marketing campaigns (phone calls) of a Portuguese banking institution. Each entry represents a phone call to a client of whom several features are provided.

The responce variable is binary (yes or no) and indicates whether a phone call successfully convinced the client to purchase a financial product the bank is tryng to sell (bank term deposit). Among the 20 provided features, 11 are categorical and the remaining are numerical. An extensive description of each feature will now follow.

1.1 Attribute information

Table 1 provides an extensive overview about each attribute featured in the dataset.

Important notes about some of the attributes are the following:

- Attributes from `Contact` to `duration` are related to the last contact of the current campaign
- The attribute `duration` affers to the call duration and it's not known until the call is completed. It highly affects the prediction (e.g. if `duration=0` then the outcome will always be *not successful*), hence it will be deleted since the model is ment to be deployed with real world data.

Name	Description	Type	Possible values
Age	Age of each client	numeric	-
Job	Type of job	categorical	Admn, blue-collar, entrepreneur...
Marital	Marital Status	categorical	Divorced, married, single, unknown
Education	level of education	categorical	basic.4y, basic.6y, basic.9y, high-school,...
Default	whether a client has credit in default	Categorical	Yes, No, Unknown
Housing	whether a client has an housing loan	Categorical	Yes, No, Unknown
Loan	whether a client has a personal loan	Categorical	Yes, No, Unknown
Contact	contact communication type	Categorical	cellular, telephone
Month	last contact month of the yeah	Categorical	jan, feb, mar...
Day of the week	last contact day of the week	Categorical	mon, tue, ...
Duration	Last contact duration in seconds (see notes)	Numeric	-
Campaign	number of contacts performed during this campaign for that client	Numeric	-
pdays	number of days that passed by after the client was last contacted from a previous campaign	numeric	-
previous	number of contacts performed before this campaign for this client	numeric	-
poutcome	outcome of the previous marketing campaign	Categorical	Failure, nonexistent, success
emp.var.rate	employment variation rate - quarterly indicator	Numeric	-
cons.price.idx	consumer price index - monthly indicator	Numeric	-
cons.conf.idx	consumer conficence index - monthly indicator	Numeric	-
euribor3m	euribor 3 month rate - daily indicator	Numeric	-
nr.employed	Number of employees - quarterly indicator	Numeric	-
Responce variable: y	Has the client subscribed to a term deposit	Binary	Yes, No

Table 1: Extensive description for each attribute in the dataframe

2 Data Exploration and Preprocessing

We first import the dataset and encode each categorical feature using *one-hot encoding*. This means that numerical features will be left unchanged while for each categorical feature the process is the following:

1. Determine all the distinct values of that feature (categories)

2. For each category generate a new binary column

3. Assign values to the binary columns according to categories featred in each line.

feature
category 1
category 2
category 3
category 2

category 1	category 2	category 3
1	0	0
0	1	0
0	0	1
0	1	0

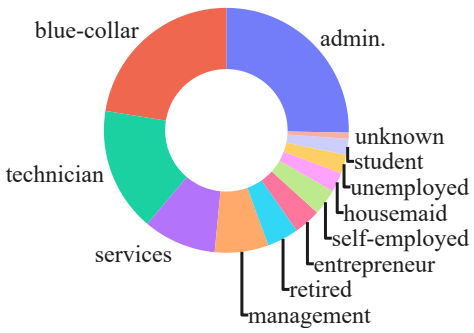
```
import pandas as pd

df = pd.read_csv('Data/bank/bank-additional-full.csv', sep=';')
display(df.head())
cat_col = df.dtypes=='O'
df_enc = pd.get_dummies(df.loc[:, cat_col], prefix=df.columns[cat_col])
df_enc = df_enc.join(df.loc[:, np.logical_not(cat_col)])

df_enc = df_enc.drop('y_no', axis=1) #deleting column since attribute "y" is
binary
df_enc = df_enc.drop('duration', axis = 1) #drop the duration column (see
attribute information)
```

Listing 1: Data encoding

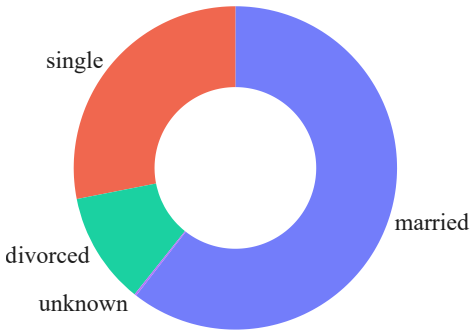
2.1 Data Exploration

In order to gain useful insights about some of the attributes featured in the dataset we can now perfom some *data exploration*. This step of the analysis is qualitative by nature and consists in plotting graphs and distributions relative to each feature in the dataset.

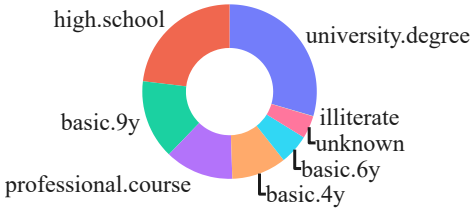
There are 11 categorical features and the remaining are numerical. Categorical data will be explored by means of *pie charts* while numerical data will be explored by means of a *scatterplot matrix*



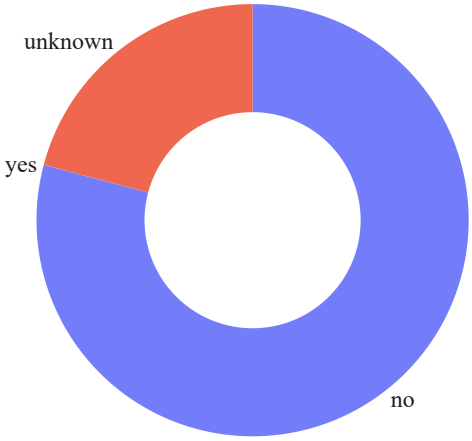
(a) Pieplot relative to `job`



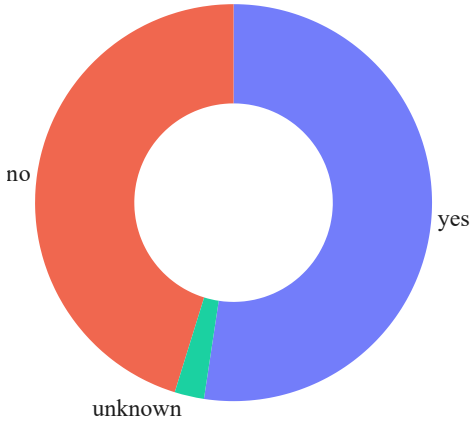
(b) Pieplot relative to `marital`



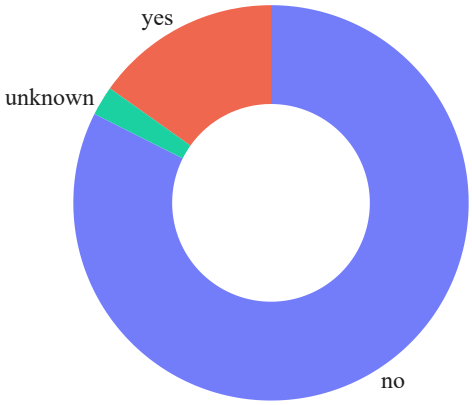
(c) Pieplot relative to `education`



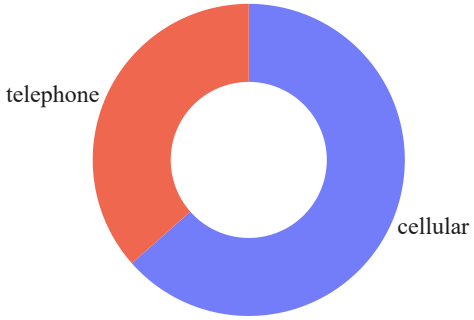
(d) Pieplot relative to `default`



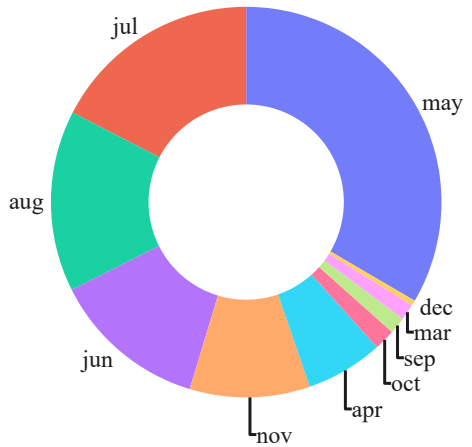
(e) Pieplot relative to `housing`



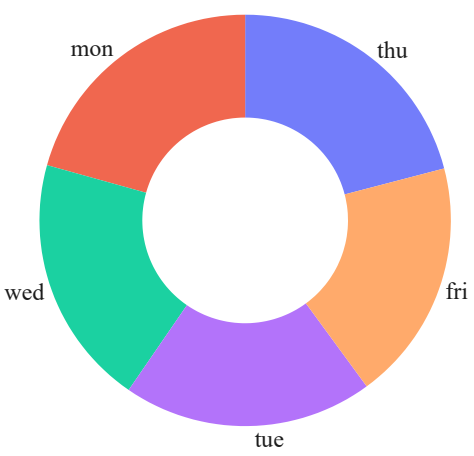
(f) Pieplot relative to `loan`



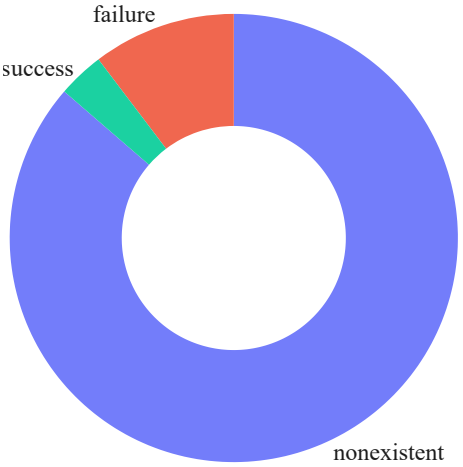
(g) Pieplot relative to `contact`



(h) Pieplot relative to `month`



(i) Pieplot relative to `day_of_the_week`



(j) Pieplot relative to `poutcome`

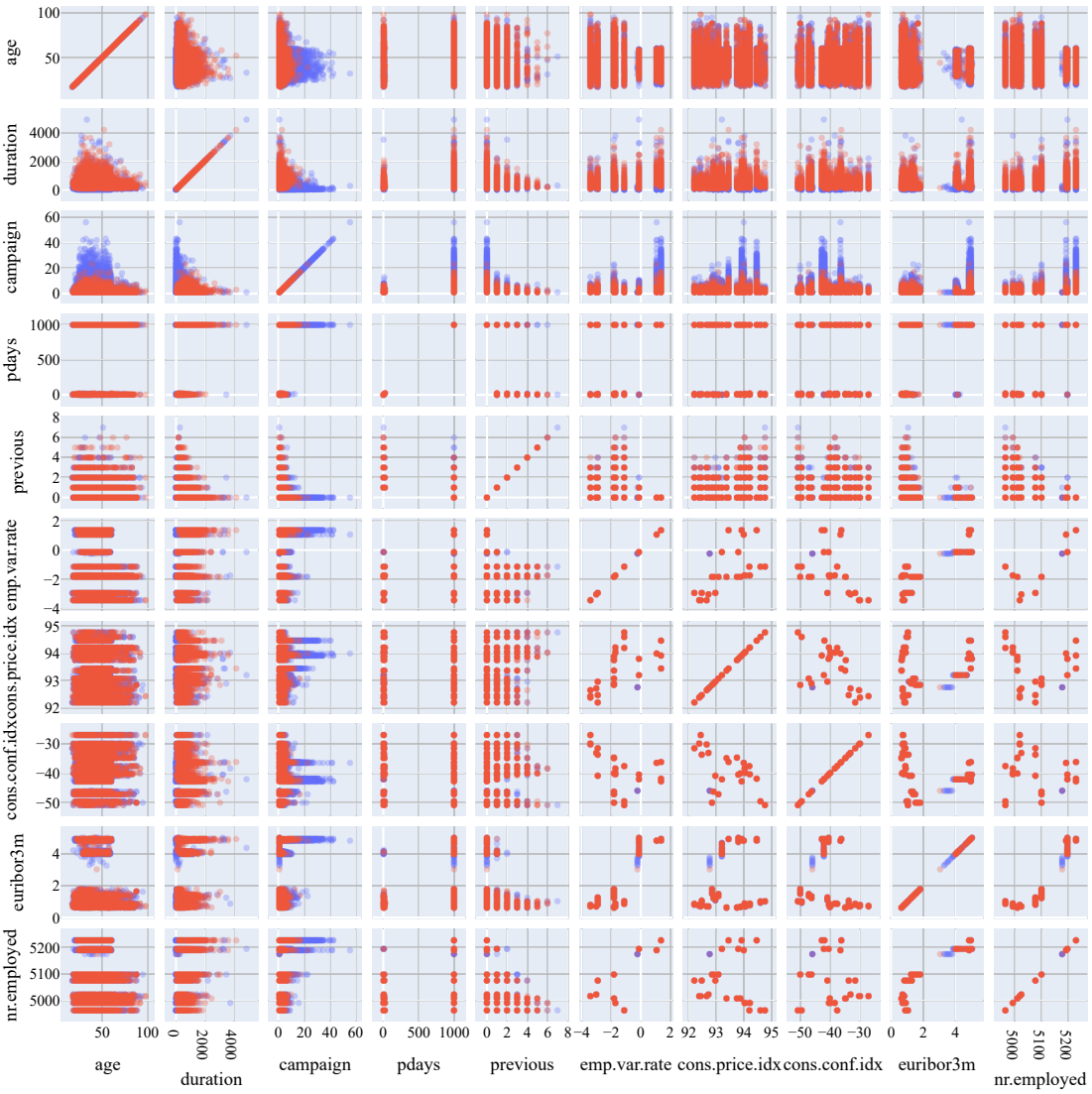


Figure 1: Scatterplot matrix of numerical features

From the scatterplot we don't see any particular correlation between numerical features. Yet, as for the coloring of the data, one can notice that the attribute `campaign` discriminates pretty well the outcome. As a matter of fact people that recieved a lot of calls from the call center never buy the product that the bank is trying to sell. This information alone could be of some value for reducing cost and better targeting calls.

2.2 Dataset splitting

After exploring data, the first thing to do is to perform a split of our dataset into training and test set.

```
df_train, df_test = train_test_split(df_enc, test_size=0.1, random_state=42)
```

Listing 2: Data splitting

As it is the case with some binary classification datasets, our data is heavly biased towards one class of outcome. In our case the most common outcome for each call is, reasonably, unsuccesful call: meaning that the client that has been called have **not** purchased the product that the bank is trying to sell. In order to quantify the class unbalance we can use a simple pie chart:

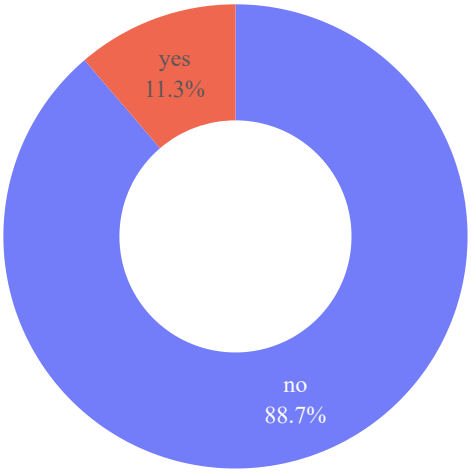
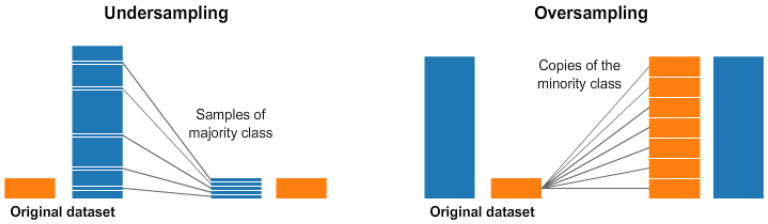


Figure 2: Responce variable class unbalance

As we can see $\sim 90\%$ of our data is biased towards the majority class (no). If we tried to feed the data as it is to any classification model the resulting classifier would seem to be "accurate" even if it really isn't.

As a simple example we could think of a classifier that classifies every data entry as "no" no matter of the features. This naive classifier, tested with our heavily biased data, would result to have $\sim 90\%$ accuracy, the same percentage of the majority class with respect to the whole dataset.

Various techniques can be used in order to solve this problem, among them there are **Oversampling** and **Undersampling**.



Visual exapmple of Oversampling and Undersampling

As the name suggests, oversampling takes place when the minority class is used to generate new bootstrapped data to balance classess. On the other hand undersampling consists in randomly sampling from the majority class enough samples such that the two classes are balanced. In our case, since we have enough data, we can opt for undersampling in order to balance classes.

```
response = df_train['y_yes']
feature_matrix = df_train.loc[:, df_train.columns!='y_yes']

#rebalancing classes (y_yes=1 is the minority class)--------
ratio = df_train['y_yes'].sum()/df_train['y_yes'].size
print("Original training dataset has", df_train.index.size, "Observations. \n", "
    Among them", response.sum(), "are positive and", response.size-response.sum(),
    "are negative \n Performing dataset rebalancing by undersampling... \n\n" )
df_train_false = df_train.loc[response==0, :]
df_train_false_resampled = df_train_false.sample(frac=ratio)

df_train_true = df_train.loc[response==1,:]

df_train = pd.concat([df_train_true, df_train_false_resampled])
ratio = df_train['y_yes'].sum()/df_train['y_yes'].size

response = df_train['y_yes']
feature_matrix = df_train.loc[:, df_enc.columns!='y_yes']

print("Training Dataset rebalancing performed. Dataset has now", df_train.index.
    size, "Observations. \n", "Among them", response.sum(), "are positive and",
    response.size-response.sum(), "are negative \n\n" )

X_train = df_train.loc[:, df_train.columns!='y_yes']
X_test = df_test.loc[:, df_test.columns!='y_yes']
y_train = df_train['y_yes']
y_test = df_test['y_yes']
```

Listing 3: Data rebalancing

One can notice that only the training tast was subject to the rebalancing process. This is because real-world data will not be balanced and rebalancing test data would cause the same problems concearning test accuracy.

2.3 Scaling the dataset

The encoded dataset is characterized by features with very different scale/range of values. In this case, if we apply PCA or any kind of training model on this data, some important features can be

"covered" by other feature just for the unit measure disparity. This is why is a good practice to **rescale** the data. The most used scalers used are the **Min-max Scaler** and **Standard Scaler**.

The **Min-Max Scaler** trasnforms the data such that the values of each column are distributed between a minimum value m and a maximum value M. Let $X \in \mathbb{R}^{n\times p}$ be our feature matrix, with $X = [x_1, x_2, \dots, x_p]$. Let $X' \in \mathbb{R}^{n\times p}$ be our scaled matrix with $X' = [x'_1, x'_2, \dots, x'_p]$. If $M_i = \max\{x_i\}$, $m_i = \min\{x_i\}$, $i \in \{1, \dots, p\}$ then

$$x'_i = \frac{x_i - m_i}{M_i - m_i} \quad i \in \{1, \dots, p\} \tag{2.1}$$

where all operations are intended to be element-wise.

On the other hand the **Standard-Scaler** makes the values of each feature in the data to have zero-mean and unit-variance. In particular:

$$x'_i = \frac{x_i - \bar{x_i}}{s_i} \quad i \in \{1, \dots, p\} \tag{2.2}$$

where $\bar{x_i}$ is the *sample mean* and s_i is the *sample variance* of the i-th column

```
#Performing scaling using min-max scaler
scaler_mm = MinMaxScaler()
scaler_mm.fit(X_train)

X_train_scaled_mm = scaler_mm.transform(X_train)
X_test_scaled_mm = scaler_mm.transform(X_test)

#Performing scaling using standard scaler
scaler_std = StandardScaler()
scaler_std.fit(X_train)

X_train_scaled_std = scaler_std.transform(X_train)
X_test_scaled_std = scaler_std.transform(X_test)
```

Listing 4: Data scaling

As it is the case with most steps in our analysis pipeline, scaling is *fitted* on **training data**, meaning that all the values used for the scaling are computed on the training dataset. When performing a prediction, test data should also undergo the same rescaling process using however the values fitted with the training data. Not doing so would result in what in data analysis is called *data leakage*, meaning that information about test data *leaked* into the training process.

2.4 Performing PCA on the dataset

PCA stands for *principal component analysis* and is a technique used to reduce the dimensionality of the data (number of columns) while retaining as much information as possible.

Let $X \in \mathbb{R}^{n\times p}$ be our feature matrix and let

$$\Sigma = \frac{1}{N-1}\bar{X}^T\bar{X} \in \mathbb{R}^{p\times p} \tag{2.3}$$

Be the *Covariance matrix* of X, where $\bar{X}$ is the *centered* data matrix i.e. $\bar{X} = [x_1 - \bar{x_1}, \dots x_p - \bar{x_p}]$ (all the columns of X are centered w.r.t. their sample mean).
Since the covariance matrix Σ of our centered data is a symmetric semi-definite positive matrix, we can compute its **eigendecomposition**:

$$\Sigma = V\Lambda V^T \tag{2.4}$$

with:

$$V = [v_1, v_2, \dots, v_p] \in \mathbb{R}^{p\times p}, \qquad \Lambda = \begin{bmatrix} \lambda_1 & \dots & 0 \\ \vdots & \ddots & 0 \\ 0 & \dots & \lambda_n \end{bmatrix}, \qquad \lambda_1 \geq \lambda_2 \geq \dots \geq 0. \tag{2.5}$$

Then it can be shown that **the columns of V describe the directions of greater variance of data**, depending on the eigenvalues magnitude. This means that V is a new basis for $\mathbb{R}^p$ and can be seen as a rotation of the data. Let $x^{(i)} \in \mathbb{R}^p$, then $y^{(i)} = V^T x^{(i)}$ is the same vector but represented with basis V. By this token we could compute the rotation of each point in our original data X obtaining the matrix $Y = \bar{X}V \in \mathbb{R}^{n \times p}$. The resulting covariance matrix Σ_Y would be:

$$\Sigma_Y = \frac{1}{N-1} Y^T Y = V^T \bar{X}^T \bar{X} V = V^T \Sigma V = \Lambda \tag{2.6}$$

This means that data written w.r.t. the new basis V has a diagonal covariance matrix Λ and the variance of Y w.r.t. the axis v_i is λ_i. We call $v_1, \ldots, v_p$ **principal components** and λ_i is the **Explained variance** of the principal component v_i.

Since eigenvalues are computed in descreasing order, the first components of the data written in the basis V are the one with the most explained variance. Hence, truncating the rotated data matrix to the first k columns (Y_k), causes a loss of explained variance that is quantified by the remaining $\lambda_{k+1} \ldots \lambda_p$ eigenvalues. The **percentage of explained variance** by the first k components is hence computed as:

$$\texttt{ev\_ratio} = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{p} \lambda_i}. \tag{2.7}$$

In figure 3 percentage of explained variance is plotted against the number of components, and we compare the performance of the *min-max scaler* against the *standard scaler* with respect to the explained variance.

```
#Performing pca using min max scaler

pca_mm = PCA()
pca_mm.fit(X_train_scaled_mm)
var_mm = pca_mm.explained_variance_ratio_

#performing pca using std scaler

pca_std = PCA()
pca_std.fit(X_train_scaled_std)
var_std = pca_std.explained_variance_ratio_

n_components=20

X_train_scaled_mm_pca = pca_mm.transform(X_train_scaled_mm)[:, :n_components]
X_test_scaled_mm_pca = pca_mm.transform(X_test_scaled_mm)[:, :n_components]
```

Listing 5: Data scaling

If the number of components to keep is fixed to `n_components = 20` then, as we can see in figure 3, data scaled with the min-max scaler retains more variance than the standard scaler. In particular with 20 principal components the percentage of explained variance computed on data scaled with the *min-max scaler* is $\sim 83\%$

3 Classification methods used on data

What will follow is a brief theoretical overview of all the classification models used on the data, with the objective of classifing if a call will be successful or not w.r.t. the features provided to the model.

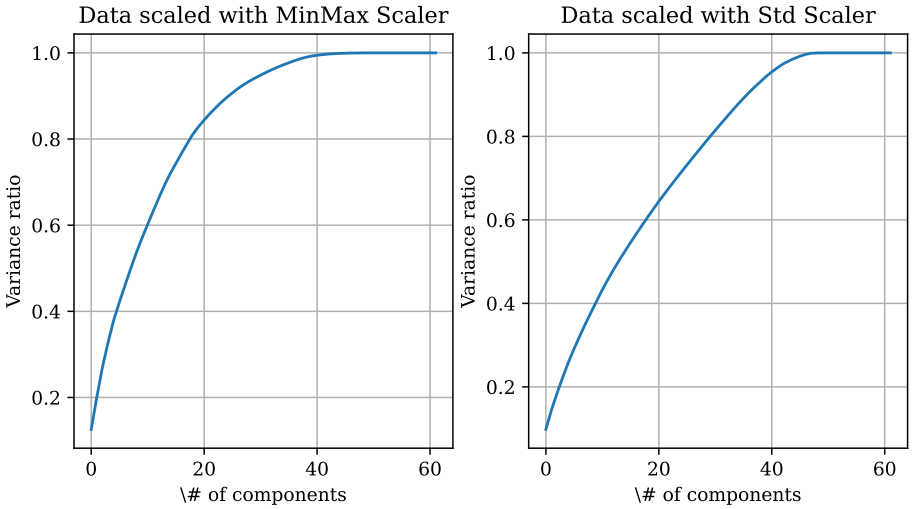


Figure 3: Percentage of explained variance comparison between scalers

3.1 From Decision Trees to Random Forests

Decision Trees

Let χ be our feature space (meaning the space in which features live in, e.g. $\chi = \mathbb{R}^p$). The main idea behind a decision tree is to partition the feature space such that data points will be classified according to the partition to which they belong.

This method is called decision *tree* because it can be seen as a binary tree that mimics the splitting process of the feature space where each node ν represents a (sub)split of the space and each leaf ω represents a subset $R_\omega \subseteq \chi$. Hence each leaf of the tree is associated to a classifier g^ω that (in the classification case) gives as output the class relative to that particular partition of the feature space

Let $\tau = \{x_i, y_i\}_{i=1,\dots,n}$ be our training set, then our classifier will look like:

$$g(x) = \sum_{\omega} g^\omega(x) \mathbb{1}\{x \in R_\omega\} \tag{3.1}$$

And our loss function can be written as:

$$l_\tau(g) = \frac{1}{n}\sum_{i=0}^{n} \text{Loss}(y_i, g(x_i)) = \sum_{\omega} \underbrace{\frac{1}{n}\sum_{i=1}^{n} \mathbb{1}\{x_i \in R_\omega\}\text{Loss}(y_i, g^\omega(x_i))}_{\text{Loss contribution of each leaf } \omega} \tag{3.2}$$

This means that **total loss can be seen as the sum of the loss on each region (leaf) of the feature space**. Hence, since minimizing loss finding the optimal partitioning would be an *np-hard* problem, we can formulate a *greedy* algorithm that tries to split recursively our feature space trying to minimize loss at each step.

Let ν be an internal node of a decision tree. For each node we can assign a region $R_\nu \subseteq \chi$ and a subset of the training training set containing points that live in that region $\sigma_\nu = \{(x, y) \in \tau : x \in R^\nu\}$. To each node is associated a splitting rule s that can define two subset of the training set:

$$\sigma_T = \{(x, y) \in \sigma \text{ s.t } s(x) = \texttt{True}\}$$
$$\sigma_F = \{(x, y) \in \sigma \text{ s.t } s(x) = \texttt{False}\}$$

Hence we can define the following algorithm:

Algorithm 1 Greedy Decision Tree

Imput: Tree node ν, $\sigma_\nu \subset \tau$
if The terminal condition is met(ν is a leaf) **then**
define $g^\nu = \arg\max_{z\in\{0,\dots c-1\}} = \frac{1}{|\sigma_\nu|} \sum_{(x_i,y_i)\in R_\nu} \mathbb{1}\{y_i = z\}$
end if
Search for the best splitting rule s
Generate σ_T and σ_F
Generate ν_T and ν_F
$\mathbb{T}_{\nu_T} \leftarrow$ `Greedy Decision Tree`(ν_T, σ_T) (Left subtree)
$\mathbb{T}_{\nu_F} \leftarrow$ `Greedy Decision Tree`(ν_F, σ_F) (Right subtree)
return $\mathbb{T}_\nu$ (subtree starting from ν)

But how the best splitting rule is defined? Given a node ν and the relative $\sigma_\nu \subseteq \tau$, since the loss is additive with respect to regions of χ, we can compute the contribution of ν to the loss:

$$\frac{1}{n}\sum_{i=1}^{n} \mathbb{1}\{x_i \in \sigma_\nu\}\text{Loss}(y_i, g^\nu(x_i)) \tag{3.3}$$

Alternatively we can compute the contribution of ν to the loss if splitted into σ_T and σ_F

$$\frac{1}{n}\sum_{i=1}^{n} \mathbb{1}\{x_i \in \sigma_T\}\text{Loss}(y_i, g^T(x_i)) \; + \; \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}\{x_i \in \sigma_F\}\text{Loss}(y_i, g^F(x_i)) \tag{3.4}$$

Where g^T and g^F computed in a *greedy way*, meaning that σ_T and σ_F are seen as leaves. Since 3.4 is always less or equal than 3.3, in trying to find the best splitting rule our objective is to minimize 3.4. Difference between the two quantities can be used in the terminal condition: if splitting the node ν does not produce a sensible reduction in loss then the algorithm can stop.

Since ours is a classification task, 3.4 can be written as:

$$\frac{1}{|\sigma_T|}\sum_{(x_i,y_i)\in\sigma_T} \mathbb{1}\{y_i \neq y_T^*\} \; + \; \frac{1}{|\sigma_F|}\sum_{(x_i,y_i)\in\sigma_F} \mathbb{1}\{y_i \neq y_F^*\} \tag{3.5}$$

Where y_T^* and y_F^* are the majority classes in σ_T and σ_F respectively. Morover, since:

$$\frac{1}{|\sigma_T|}\sum_{(x_i,y_i)\in\sigma_T} \mathbb{1}\{y_i \neq y_T^*\} = 1 - \frac{1}{|\sigma_T|}\sum_{(x_i,y_i)\in\sigma_T} \mathbb{1}\{y_i = y_T^*\} = 1 - p_z^{\sigma_T} \tag{3.6}$$

Where $p_z^{\sigma_T}$ is the ratio of the majoirty class in σ_T relative to the other classes and can be seen as an **Impurity index**. The minimization of the loss at each split can thus be seen as the minimization of an impority index. Hence various impurity indexes can be used such as Entropy impurity, and the Gini index.

Random Forests

Assume we could have an ensamble of training sets $\tau_1, \dots, \tau_B$ indipendent and identically distributed. Let g_{τ_b} the classification trees modeled on each training set. Then:

$$g_{\text{maj}}(x) = \underset{z\in\{0,\dots,c-1\}}{\text{argmax}} \sum_{b=1}^{B} \mathbb{1}\{g_{\tau_b}(x) = z\} \tag{3.7}$$

would be the result of a classification function that is the result of a "voting mechanism" where each g_{τ_b} contributes to the vote and the majority class would win. The more independent training sets we have, the more the resulting g_{maj} would be "wise". Obviously in a real world envoirment we

would never have such luxury of having i.i.d. training sets, hence we have to resort to **Bootstrapping**, meaning that from one training set τ we generate B training sets $\tau_1^*, \dots, \tau_B^*$ sampling from τ.

Depending on the dimensions of the original dataset, using bootstrapping makes us loose independency between training sets. Hence trees generated from each training set will be "correlated[1]" meaning that they would be similar since they are generated basically from the same data. In order to *decorrelate* trees generated from $\tau_1^*, \dots, \tau_B^*$ we use a simple trick: **at each split of the tree we only consider a subset of $m < p$ features instead of the full feature set.**

This simple trick will generate "randomized" trees that are "decorrelated" from each other. After generating $\{g_{\tau_b^*}\}_{b=1,\dots B}$ random trees we can consider the "majority classifier" as seen in 3.7: we have generated a **Random Forest.**

3.2 Boosting

The practice of **Boosting**, as the name implies, is an iterative method used to boost the performances of a *Weak learner* by generating other weak learners.

We call a learner *Weak* if it is a classification/regression model that can be trained very easily at the expence of accuracy (e.g. a decision tree with depth equal to 2 is a weak learner).

Let g_0 be a weak learner, then our objective is to find another weak learner $h_1 \in \mathcal{H}$ such that:

$$h_1 = \underset{h_1 \in \mathcal{H}}{\operatorname{argmin}} \frac{1}{n} \sum_{(x_i, y_i) \in \tau} \operatorname{loss}(g_0(x_i) + \gamma h_1(x_i), y_i) \tag{3.8}$$

Once we find h_1 we can define $g_1 = g_0 + h_1$ and by the same token we can find h_2 and so on. Hence after B boosting steps we obtain:

$$g_B(x) = g_0(x) + \sum_{i=1}^{B} \gamma h_i(x) \tag{3.9}$$

In a classification setting, where $g_0(x) \in \{-1, +1\}$ and $h_i(x) \in \{-1, +1\}, i = 1 \dots B$ then:

$$g_B(x) = \operatorname{sign}\left[g_0(x) + \sum_{i=1}^{B} \gamma_i h_i(x) \right] \tag{3.10}$$

To compute $g_B(x)$ we can use the **AdaBoost algorithm** that deploys an "adaptive" boosting algorithm by wheighting the loss on the points. This algorithm arises from defining $\operatorname{Loss}(y, \hat{y}) = e^{y\hat{y}}$.

3.3 Logistic Regression

Let Y_i be the random variable associated with the responce y_i of the i-th data entry (successful/unsuccessful call). Assume Y_i has a *Bernoulli* distribution:

$$Y_i \sim \operatorname{Ber}(\pi_i) \tag{3.11}$$

Using the **Generalized Linear Models** framework, we assume that π_i is a function of a linear combination of the parameters:

$$\pi_i = \mathbb{E}[Y_i] = h(x_i^T \beta) = \frac{1}{1 - e^{-x_i^T \beta}}. \tag{3.12}$$

[1] Here the word *correlation* is slightly misused since we do not actually mean correlation in a statistical sense buy more in an heuristical sense.

As in linear regression, we can find a suitable β by maximizing the *log-likelihood* function of our data. After some computation we obtain:

$$-\frac{1}{n}\log[f(y|\beta, X)] = \frac{1}{n}\sum_{i=1}^{n}(1-y_i)x_i^T\beta + \log(1+e^{-x_i^T\beta}) \tag{3.13}$$

Which is a *convex minimization problem*, thus easy to solve. In solving this minimization we can also introduce a *regularization argument* using l_1, l_2 norm or both on β.
After computing the optimal $\hat{\beta}$, we can then deploy our classificator which is based on computing the probability of each outcome (y_i successful/unsuccessful)

$$g_\tau(x_i) = \underset{y\in\{0,1\}}{\operatorname{argmax}}\left[(h(x_i^T\hat{\beta}))^y(1-h(x_i^T\hat{\beta}))^{1-y}\right] \tag{3.14}$$

3.4 K-Nearest Neighbors Classification

K-Nearest Neighborst classification makes use of a distance metric in order to compute, as the name suggests, the k nearest neighbors of a given data point x. Thus, given x we could define the set of the K nearest neighbors of x according to the chosen metric.

$$\tau(x) = \{(x_{(1)}, y_{(1)}), \dots (x_{(K)}, y_{(K)})\}$$

Then the *K-nearest neighbors* classification rule classifies x according to the most frequently occurring class labels in $\tau(x)$.

4 Results of the analysis

Before presenting the results of our analysis, a brief overview of the methods used to evaluate our classification methods is needed.

Since we are dealing with an unbalanced classification problem, the usual "accuracy" score is not suited for actually asses the quality of our classification. As said in 2.2 a naive classifier that classifies every entry as "unsuccesful call" would have a $\sim 90\%$ accuracy due to the unbalanced nature of the dataset.

What we are really interested in is "how much our classifier outperforms randomply calling people" meaning that we are interested in a deeper analysis of our **confusion matrix**.
The class we focus on is the "successful call" class, meaning that we are intrested in accuracy indicators about calls to clients that successfully subscribed to a bank account. We can introduce two measure of accuracy about our class:

- the *precision* is the fraction of all calls classified as succssful that are actually successful:

$$\text{precision} = \frac{\text{tp}}{\text{tp}+\text{fp}} \tag{4.1}$$

- The *recall* is the fracion of all successfull calls that are correctly classified as such.

$$\text{recall} = \frac{\text{tp}}{\text{tp}+\text{fn}}$$

- The F_1 score is the harmonic mean of precision and recall.

Where *tp, fp* and *fn* are short for *true positive, false positive* and *false negative* respectively. The above mentioned naive classifier would score 0 for all three measures. We will focus on the F_1 score for all the tests.

4.1 Results in Random Forests

In experimenting with hyperparameters in Random Forest, the most impactful parameters are the one that affer to the termination criterion for each tree. In particular, the parameters `max_depth` and `min_sample_leaf` are the ones that were optimized for this analysis:

- `max_depth` controls the maximum depth of each tree in the forest

- `min_sample_leaf` controls the minimum number of samples in each leaf of the trees.

The following code was used to search for the best parameters:

```
    from sklearn.metrics import f1_score
    from sklearn.ensemble import RandomForestClassifier

    #Searching for the best max_depth parameter
    max_depth_params = np.arange(3, 20)
    max_depth_score = []
    for d in max_depth_params:
    clf = RandomForestClassifier(max_depth=d, n_jobs=-1, random_state=42)
    clf.fit(X_train_scaled_mm, y_train)
    score = f1_score(y_true=y_test, y_pred = clf.predict(X_test_scaled_mm))
    max_depth_score.append([d, score])
    scores_md = [s[1] for s in max_depth_score]
    best_max_depth = max_depth_score[np.argmax(scores_md)][0]

    #Searching for the min_sample_leaf parameter
    min_sample_leaf_params = np.arange(1, 20)
    min_sample_leaf_score = []

    for l in min_sample_leaf_params:
    clf = RandomForestClassifier(#max_depth=best_max_depth,
                                 min_samples_leaf=l,
                                 n_jobs=-1,
                                 random_state=42 )
    clf.fit(X_train_scaled_mm, y_train)
    score = f1_score(y_true=y_test, y_pred = clf.predict(X_test_scaled_mm))
    min_sample_leaf_score.append([l, score])
    scores_ml =  [s[1] for s in min_sample_leaf_score]
    best_min_sample_leaf = min_sample_leaf_score[np.argmax(scores_ml)][0]
```

Listing 6: Searching for the best parameters for a random forest classifier

And plotting the impact of each parameter of F_1 score:

As it is evident in figure 4 we can see that the `max_depth` parameter heavily influences the F_1 score. If we then train a model using `max_depth = 11` and `min_sample_leaf = 10`, we could compute the accuracy measures and the confusion matrix:

```
from sklearn.metrics import precision_recall_fscore_support
## Training the best random Forest
clf = RandomForestClassifier(max_depth=best_max_depth,
                             min_samples_leaf=best_min_sample_leaf,
                             random_state=42)
clf.fit(X_train_scaled_mm, y_train)

conf_matrix = confusion_matrix(y_test, clf.predict(X_test_scaled_mm))
rf_scores = precision_recall_fscore_support(y_test,
                                            clf.predict(X_test_scaled_mm))
```

Listing 7: Deploying the Random forest with the best parameters

Precision	Recall	F_1 score
0.34	0.65	0.45

True / Pred	Unsuccesful	Successful
Unsuccesful	3116	541
Successful	157	305

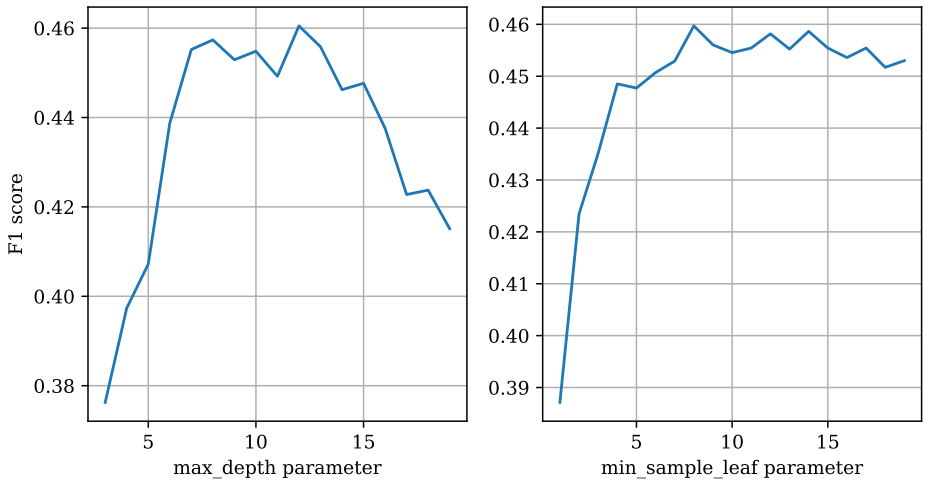


Figure 4: Comparing parameters impact on F_1 score on Random Forests

4.2 Results in Logistic Regression

Logistic regression in the binary case does not have much hyperparameters to experiment with. In solving the likelyhood maximization problem we can choose to use whether an l_1 or l_2 norm, but through experimentation this parameter does not affect results.
What instead affect the accuracy is whether we use the PCA transformed dataset as training data or not. Not only computation with PCA reduced data are faster but also more accurate.

```
from sklearn.linear_model import LogisticRegression

clf = LogisticRegression(random_state=42, solver='liblinear')
clf.fit(X_train_scaled_mm_pca, y_train)

conf_matrix = confusion_matrix(y_test, clf.predict(X_test_scaled_mm_pca))
rf_scores = precision_recall_fscore_support(y_test,
                                            clf.predict(X_test_scaled_mm_pca))
```

Listing 8: Deploying logistic regression

Deploying the above mentioned code yeld the following accuracy results:

Precision	Recall	F_1 score
0.28	0.70	0.40

True / Pred	Unsuccesful	Successful
Unsuccesful	2856	801
Successful	138	324

4.3 Results in K-Nearest Neighbors Classification

The most important hyperparameter to optimize for the K-NN classification is the number of neighbors K. The following code snippet is used to search for the best number of neighbors, according to the F_1 score:

```
from sklearn.neighbors import KNeighborsClassifier
neigh = np.arange(1, 16)
neigh_score= []

for n in neigh:
    clf = KNeighborsClassifier(n_neighbors = n)
    clf.fit(X_train_scaled_mm_pca, y_train)
    score = f1_score(y_true=y_test, y_pred = clf.predict(X_test_scaled_mm_pca))
    neigh_score.append([n, score])
scores_n = [n[1] for n in neigh_score]
best_neigh = neigh_score[np.argmax(scores_n)][0]
```

```
12
```

Listing 9: Optimizing for number of neighbors

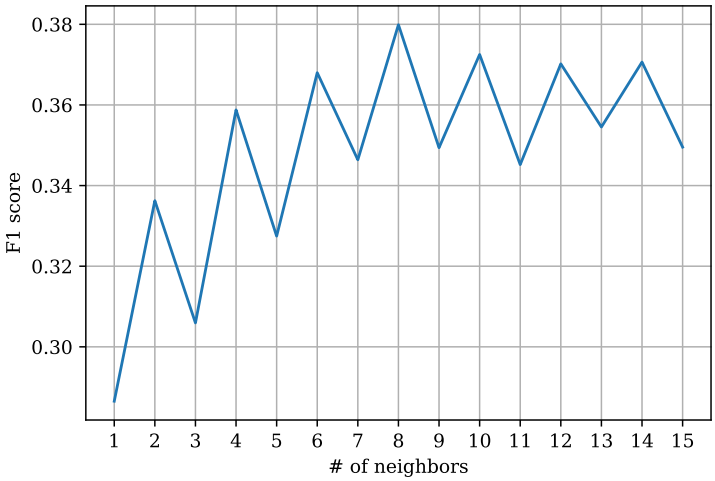


Figure 5: Comparing K impact on F_1 score in K-NN classification

As we can see in figure 5 we obtain the best score for $K = 8$ neighbors. Deploying the model with $K = 8$ yelds:

```
clf = KNeighborsClassifier(n_neighbors=best_neigh)
clf.fit(X_train_scaled_mm_pca, y_train)

conf_matrix = confusion_matrix(y_test, clf.predict(X_test_scaled_mm_pca))
kn_scores = precision_recall_fscore_support(y_test, clf.predict(
    X_test_scaled_mm_pca))
display(kn_scores)
display(conf_matrix)
```

Listing 10: Deploying K-NN

Precision	Recall	F_1 score
0.26	0.67	0.38

True / Pred	Unsuccesful	Successful
Unsuccesful	2797	860
Successful	152	310