Income-level classification of American citizens

Giulia Tortoioli

277590

Mathematics in Machine Learning

M.Sc. in Data Science and Engineering, Politecnico di Torino

Contents

Abstract

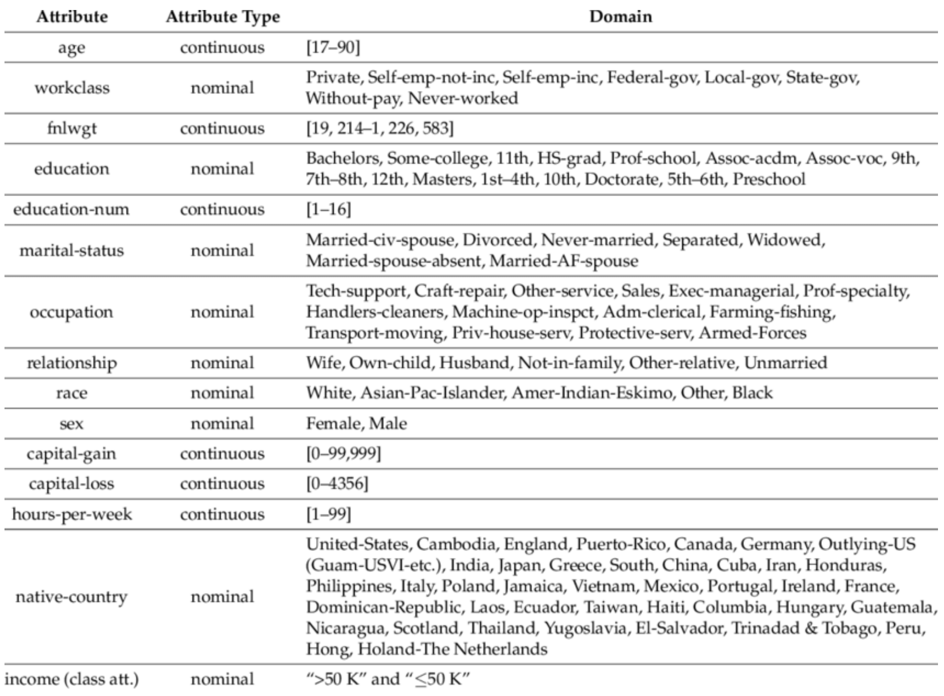
In this project some of the topics introduced in ”Mathematics in Machine Learning” course at Politecnico di Torino, M.Sc. in Data Science and Engineering, will be analysed. The task is to create a binary classification model to predict the income level (either low or high) of American adult individuals. In particular, different classifiers will be illustrated and evaluated on the classification task and there will be also a focus on ensemble techniques applied to decision trees. The first chapter will explain all the pre-processing steps applied on the data, the second chapter will introduce the different classification models used for the task and the last one will contain the results.

1 Data exploration and preprocessing

1.1 Dataset description

Adult dataset (also known as ”Census income” dataset) [1] contains over 30000 instances reporting demographic data of adult individuals. The target variable is the income level which can take two different values: $<= 50K$ or $> 50K$ (these values refer to the annual salary). The goal is to predict whether the annual income exceeds 50K or not basing on the provided data.

The dataset presents 14 attributes which describe the social, educational and economical condition of each individual. A detailed description of the dataset can be found in Figure 1.



Attribute	Attribute Type	Domain
age	continuous	[17–90]
workclass	nominal	Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked
fnlwgt	continuous	[19, 214–1, 226, 583]
education	nominal	Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th–8th, 12th, Masters, 1st–4th, 10th, Doctorate, 5th–6th, Preschool
education-num	continuous	[1–16]
marital-status	nominal	Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse
occupation	nominal	Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces
relationship	nominal	Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried
race	nominal	White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black
sex	nominal	Female, Male
capital-gain	continuous	[0–99,999]
capital-loss	continuous	[0–4356]
hours-per-week	continuous	[1–99]
native-country	nominal	United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US (Guam-USVI-etc.), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinadad & Tobago, Peru, Hong, Holand-The Netherlands
income (class att.)	nominal	“>50 K” and “$\leq$50 K”

Figure 1: Description of the *Census income* dataset.

The dataset contains both numerical and categorical attributes and presents some missing values which will be eliminated in the pre-processing phase. There is also a quite evident class imbalance, the samples belonging to the *low-income* class are around three times more represented than the ones for the *high-income* class (see Figure 2. The imbalance can be easily explained by assuming that an average salary is lower than the threshold value 50K.

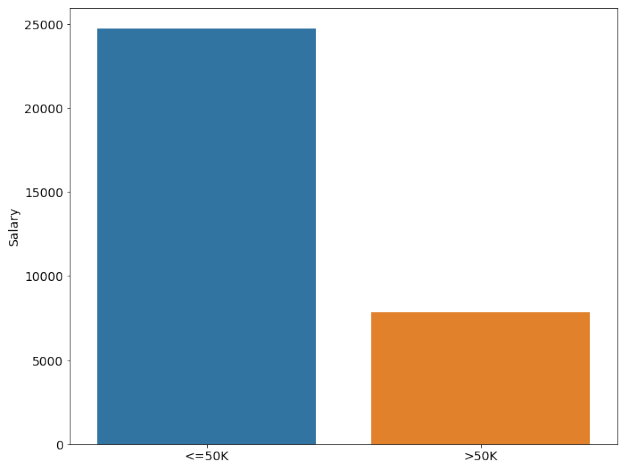


Figure 2: Imbalance between the two classes in the dataset.

1.2 Dataset cleaning

The dataset contains some missing values detected as '?' for the attributes *workclass*, *occupation* and *native-country*. The rows containing the missing values are deleted from the dataset for a total number of 2399 instances.

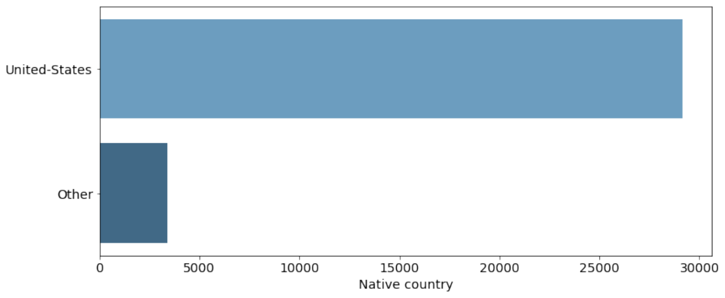


Figure 3: Proportion of the value "United-States" over the others for the attribute *native-country*.

Some attributes present imbalance in the values and this can be seen especially in the attribute *native-country*, where the value "United-States" is shared by more than 90% of the instances (see Figure 3). This attribute is considered to be not very informative, it could instead lead mistakes since the countries could have different economical conditions and standards but they are not well represented. For this reason and considering the large number of samples only the data of individual born in United States are considered for this task.

1.3 Categorical features analysis

Visualization and analysis of the features is divided between numerical and non-numerical features. For the non-numerical features one way for identifying the most significant attributes is to visualize the number of observations for each categorical feature divided for the two classes. In Figure 4 the "countplot" for the categorical attributes *workclass* and *education* are represented.

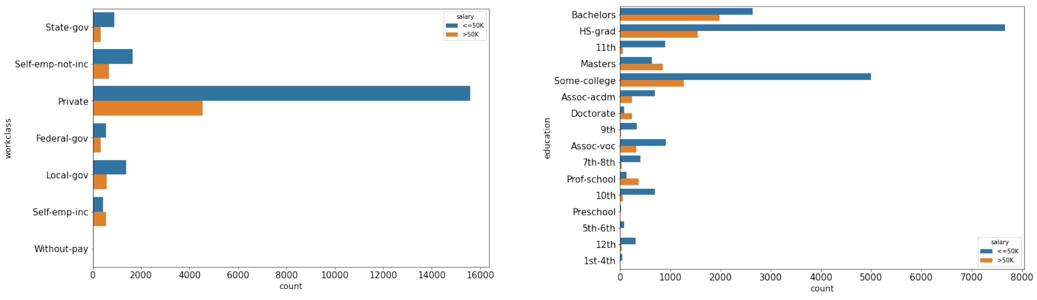


Figure 4: Countplots for *workclass* and *education*.

The graph on the left shows that self-employed individuals have on average a income above 50K unlike the other categories. The countplot representing the income for different levels of education reveals that the salary is high for the majority of the individuals who possess a master's degree.

For the classification task each categorical variable is converted into dummy variables ($n-1$ variables, where n is the number of values that the attribute can take) and the resulting dataset has 63 columns for the attributes.

1.4 Numerical features: kernel density estimation and correlation

When considering numerical features, **kernel density estimation** (KDE) can be used to visualize which ones could be the most important in the classification phase. The KDE technique is a way to estimate the probability density function of a random variable through the empirical data distribution. For each feature and class label separately, KDE technique was applied in order to visually inspect which were the most "characterizing" features by inspecting the distribution of *low-income* versus *high-income* records of the same feature. In this case the KDE plots for the attributes *age* and *education-num* are represented in Figure 5 and Figure 6, together with the box plots representing the attributes distribution. These features were found to be significant for the classification task, for this reason the visualization of their distribution is chosen.

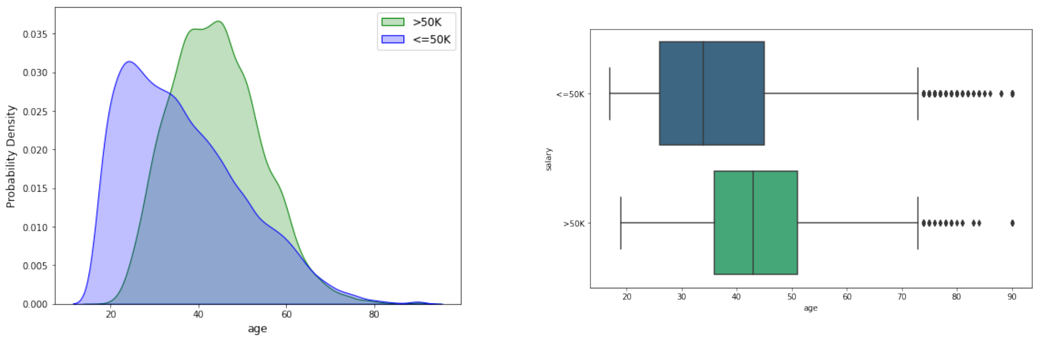


Figure 5: KDE and boxplot for *age*.

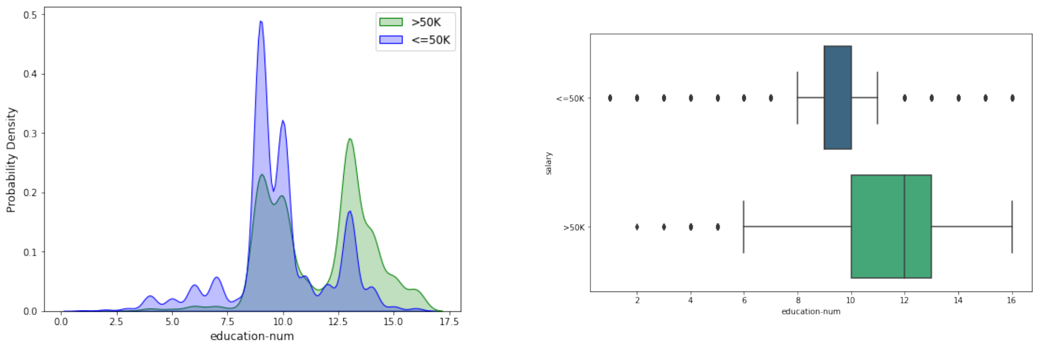


Figure 6: KDE and boxplot for *education-num*.

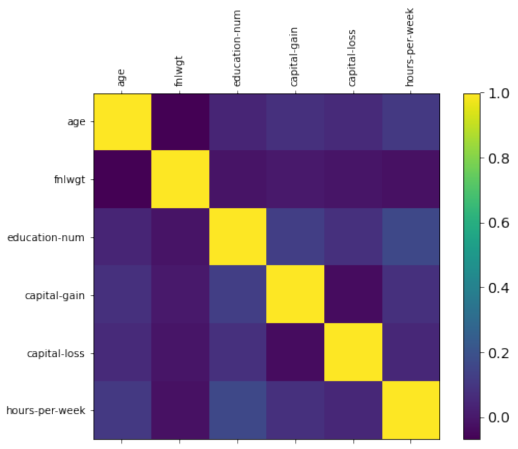


Figure 7: Heatmap showing pairwise Pearson correlation coefficient among couples of numerical attributes.

Correlation is any statistical relationship, whether causal or not, between two random variables. Essentially, it is the measure of how two or more variables are related to one another. **Pearson correlation coefficient** or linear correlation between two statistical variable X and Y, is the covariance of the two variables divided by the product of their standard deviations σ_X and σ_Y, the formula (for a population) is shown below:

$$\rho_{X,Y} = \frac{Cov(X,Y)}{\sigma_X \sigma_Y}$$

Pairwise Person coefficient was calculated in order to inspect correlation among the numerical features, the corresponding heatmap is shown in Figure 7.

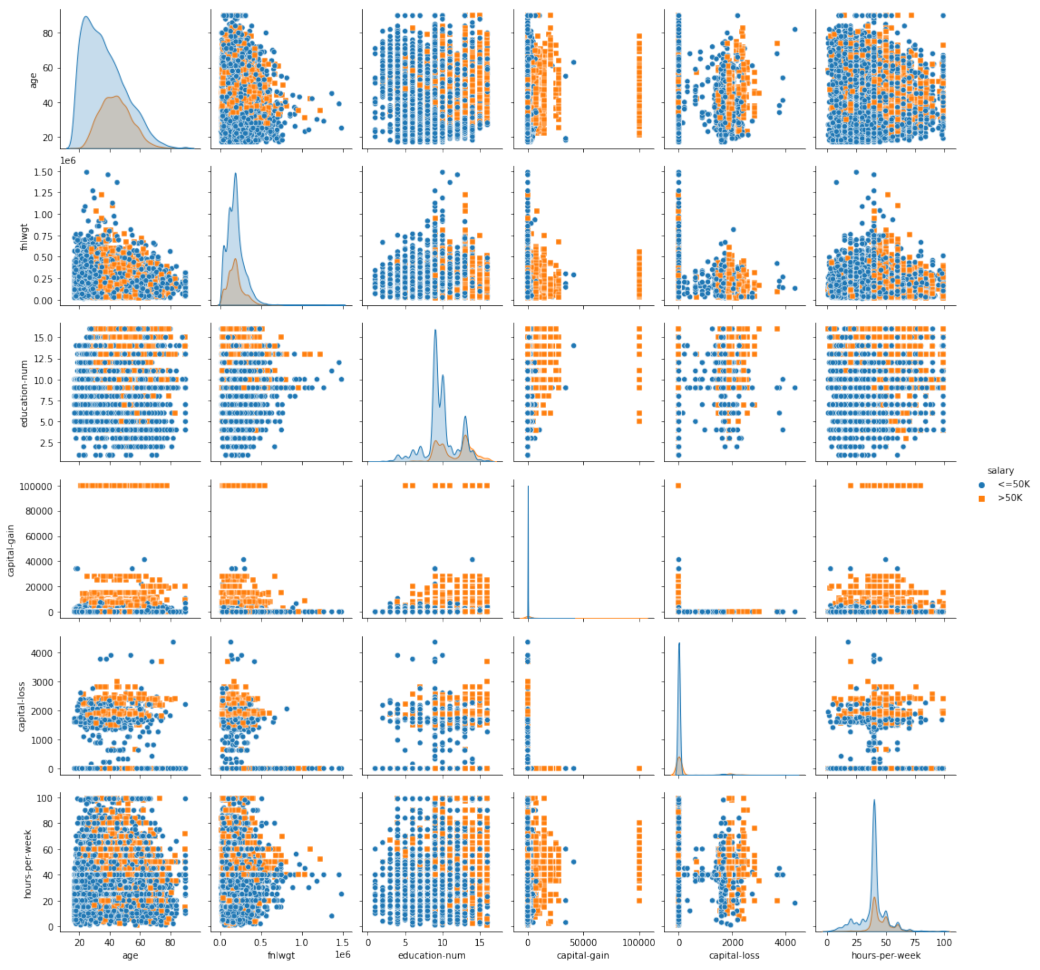


Figure 8: Scatter plots of the correlation between the numerical attributes.

The heatmap shows that the features are not significantly correlated (most of the correlation values are between 0 and 0.2), for this reason none of them can be dropped.

For the sake of visualization, the scatter plots of the numeric variables are shown in Figure 8 in order to see the relationship between the pairwise attributes. Inspecting the scatter plot of the attribute *capital-gain*, it can be noticed that this attribute is not correlated to the other and that it presents very high values for some of the instances, all belonging to the *high-income* class. This will be taken into consideration when removing outliers from the training set.

1.5 Data transformation: splitting normalization and outliers removal

The target set is mapped to 0-1 values: 0 for *low-income* and 1 for *high-income*. Data are shuffled and split with stratification (in order to preserve the percentage of samples for each class) in training and testing set with a 7:3 ratio. The train set end up having 14356 low-income records and 4896 high-income ones.

Since data are differently scaled there was need for **standardization**. Numeric data dare standardized by removing the mean of the training samples of that feature column and dividing by the standard deviation. Standardization is a common requirement for many machine learning estimators. They might behave badly if the individual features do not look like standard normally distributed data: Gaussian with zero mean and unit variance. The standard score of a sample x is calculated as follows

$$z = \frac{x - u}{s}$$

where u is the mean of the training samples and s is the standard deviation.

Another operation performed on the numeric variables of the training set is an **outlier detection** with boxplots and **deletion**. Boxplots are a standardized way of displaying the distribution of data based on the minimum $Q_1 - 1.5IQR$, first quartile Q_1, median, third quartile Q_3, and the maximum $Q_3 + 1.5IQR$. Boxplot visualization helps to detect outliers, they were initially filtered out using interquartile range method (IQR), that is a measure of statistical dispersion, being equal to the difference between $75th$ and $25th$ percentiles. Outliers were defined as observations that fall below the minimum or above the maximum, where $IQR = Q_3 - Q_1$.

IQR method for outliers deletion has some issues if applied for the attribute *capital-gain* since it filtered out a large quantity of points belonging to *high-income* class ending up with a situation of enhanced strong unbalancing between classes. This would have worsened classification performances and pointed out that most of the outliers for this attribute could be useful for the prediction. Thus, the outliers removal was not applied for the *capital-gain* attribute. Only the training data were filtered out in order to not tamper with the unknown data distribution of the targets.

After outliers removals, data were re-normalized. In Figure 9 the boxplot of the numerical attributes after outliers removal and standardization is shown.

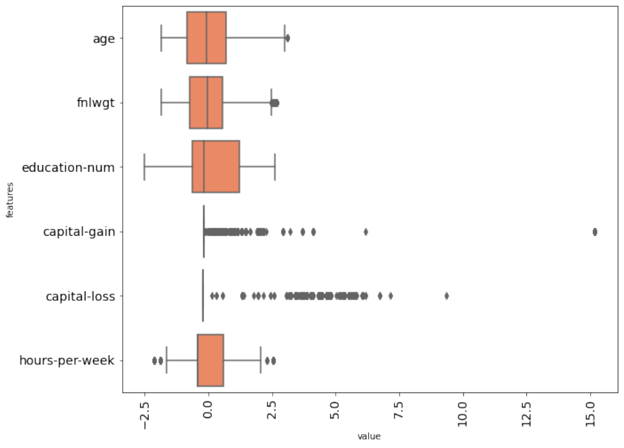


Figure 9: Boxplots of standardized attributes.

1.6 Dimensionality reduction: PCA and SVD

Principal component analysis linearly transforms data by simultaneously mapping them into a new space whose dimensionality is smaller and trying to keep most of the information of the original data. Dimensionality reduction can be done for several purposes: enhancing interpretability of data, finding meaningful structures of the data or for illustration purposes.

What PCA does is finding the ”directions” of the data that explain most of the information present in them. Starting from our data matrix X of m samples and d dimensions (our attributes), PCA does the **eigendecomposition** of the covariance (between attributes) matrix $X^T X$ (d, d), where eigenvectors and eigenvalues show the direction and the magnitude of the spread of data respectively. We then obtain the matrix W of eigenvectors its columns representing the *principal components* are called *loadings*. The loadings are ordered depending on eigenvalues (and they are all uncorrelated), in the sense that the first column of W is associated with the highest eigen-

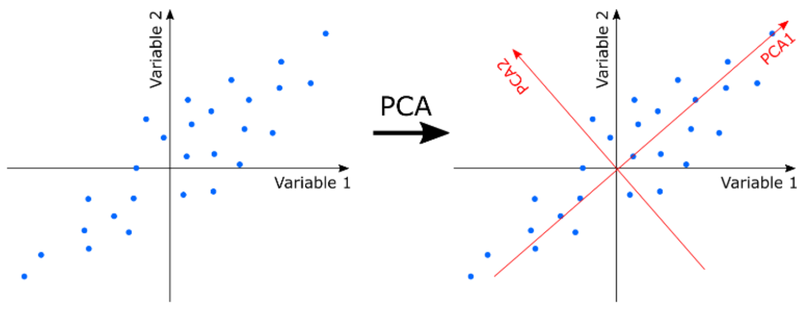


Figure 10: Principal components directions PCA1 and PCA2 in a 2D case

value and will explain more variance than the second, the third will explain more variance than the third and so on. Thus, we can choose the number r of components we want to keep by truncating W and taking the first r loadings. The matrix T of the new transformed data (or *scores*) will be the product between the original data matrix X and the truncated matrix W_r,

$$T_r = XW_r$$

of size (m, r), where r is the final number of dimensions we want to keep. It turns out that the computation of the covariance matrix and of its eigendecomposition is not necessary the most computationally efficient, here comes **Singular Value Decomposition**. We can decompose the original data matrix

$$X = U\Sigma V^T$$

where U and V^T respectively represent the left and right singular vectors and Σ is the diagonal matrix that has the singular values σ_i on its diagonal, where $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_m$. Since V is identical to the eigenvectors matrix W, it turns out that the truncated matrix of the scores can be obtained as

$$T_r = U_r\Sigma_r$$

Finding the number of components r is crucial. One can either decide to arbitrary decide a number of dimensions to take or to find that r that explains a certain amount of the variability of the data by putting a percentage threshold.

The plot in Figure 11 shows the number of components on the x-axis and the *cumulative variance* and *the variance explained* on the y-axis. As the number of components grows, the cumulative variance (equal to the cumulative normalized sum of the the singular values) of X grows until reaching a plateau while the variance explained by each of the component decrease

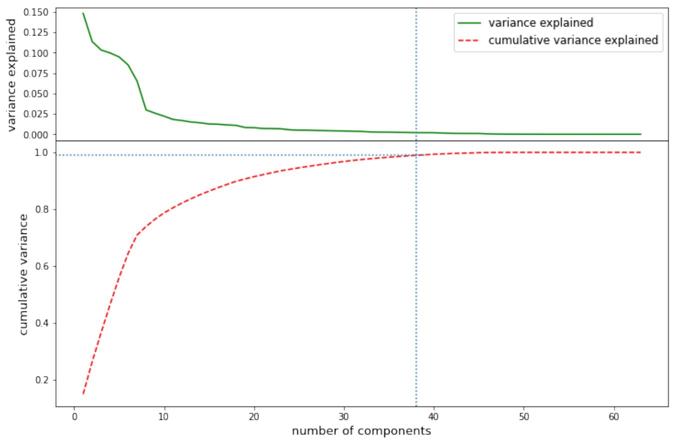


Figure 11: Cumulative variance and variance explained vs number of components. 38 components explain around 99% of the variance.

since they are ordered (the first component explain most of the variance). In this case we need more than 10 components for explaining around 80% of the variability, and a reduction to 38 (from the original 63) components explains 99% of the variance. Since we want to keep a large amount of information, this reduction will be used in the classification task.

1.7 Oversampling: SMOTE technique

SMOTE is an oversampling technique where the synthetic samples are generated for the minority class. This algorithm helps to overcome the overfitting problem posed by random oversampling. It focuses on the feature space to generate new instances with the help of interpolation between the positive instances that lie together.

The number n of samples to generate is chosen at the start and in this case the proportion 1:1 for the two classes is set. The synthetic training records for the *high-income* class are generated by randomly selecting one or more of the k-nearest neighbors for each example in the minority class. The sampling rate of the neighbors is chosen according to the imbalance proportion.

Each example is generated by using the following formula:

$$x' = x + rand(0,1) * |x - x_k|$$

After the oversampling process, the data is reconstructed. In Figure 12 the training data before and after the oversampling are represented. After the oversampling the density of the samples belonging to the *high-income* class is increased, but the distribution of the data is preserved.

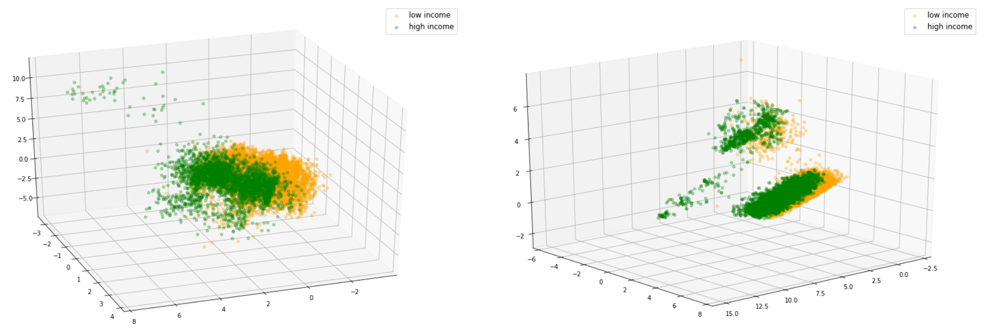


Figure 12: 3D scatter plot after PCA dimensionality reduction. In the right the original data, in the left the data after performing SMOTE oversampling.

This technique leads to good results in the classification phase, but the training time is high due to the dimension of the reconstructed dataset. For this reason another technique for tackling the imbalance in the dataset was chosen in the final model.

2 Classification task

For the classification task different techniques and algorithms are analysed and compared. At first, the ensemble techniques using **Decision Tree** classifiers will be explained and compared in the performances on the classification task (definitions from [2]). Then, other two classifiers will be shown and evaluated on the dataset: **Support Vector Machines** and **K-Nearest Neighbors** models.

In order to tackle the imbalance in the training data, the parameter *class_weight* is set to "balanced" for all the classifiers. By using this parameter the model automatically assigns the class weights inversely proportional to their respective frequencies.

2.1 Ensemble techniques on Decision Trees

Ensemble techniques are thought to tackle the *bias-variance trade-off* (Figure 13), introduced when learning through the *empirical risk minimization* (ERM, that is finding a predictor h that minimizes the empirical risk) restricted to a specific hypothesis class H, a set of predictors that reflects some prior knowledge about the task. It is the conflict in trying to simultaneously minimize the two following sources of error of an ERM algorithm over a class H:

- **Approximation error**, or *bias* of the algorithm towards choosing a hypothesis from H (it does not depend on the sample size), in other words the minimum risk achievable by a predictor in the hypothesis class.

- **Estimation error**, or *variance* of the algorithm that depends on the size or complexity of H. It results because the empirical risk (i.e. the training error) is only an estimate of the true risk, so the predictor minimizing the empirical risk is only an estimate of the predictor minimizing the true risk;

High bias leads to **underfitting**, high variance leads to **overfitting**. Enlarging the hypothesis class can decrease the approximation error but at the same time can increase the estimation error as a rich (i.e. complex) H might lead to overfitting. On the other hand, choosing H to be a very small set might increase the approximation error and lead to underfitting.

One way of resolving the trade-off is to use mixture models and ensemble learning such as Bagging, Boosting or Random Forest.

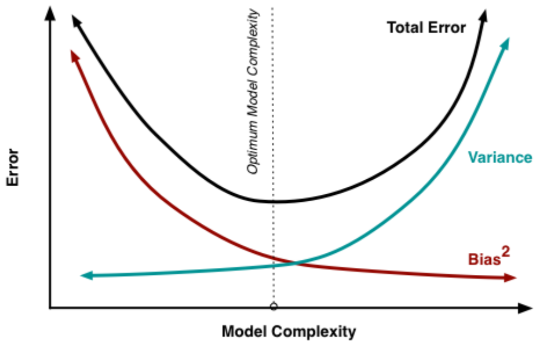


Figure 13: Bias-variance trade-off.

2.1.1 Concept of weak and strong learnability

A **weak learner** (or base model) is a learning algorithm lacking in complexity and easy to learn that, no matter what the distribution over the training data is, it will always perform better than chance when it tries to label data. In **strong learnability**, instead, the error is required to be arbitrarily small. More formally, under the PAC learning framework in a binary classification context, a learning algorithm is γ-weak learner for a class H if there exist a function m_H such that for every $\gamma \in (0, 1)$, for every distribution D over χ and for every labelling function $f : \chi \rightarrow \{\pm 1\}$, if when running the algorithm on $m > m_H$ i.i.d. examples generated by D, it returns a hypothesis h such that with probability of at least $1 - \gamma$, the true error

$$L_{D,f}(h) \leq 1/2 - \gamma$$

The lower the γ value, the weaker the learner, the more complex the ensemble technique should be to build an efficient strong learner. Most of the time, these basics models perform not so well by themselves either because they have a high bias or because they have too much variance to be robust. Thus, the idea of ensemble methods is to try reducing bias and/or variance of such weak learners by combining several of them together in order to create a strong learner (or ensemble model) that achieves better performances.

2.1.2 Decision Trees

Decision trees are predictors that assign a label associated with an instance by travelling from a root node to a leaf node in a tree. At each node of the root-to-leaf path, the successor child is chosen on the basis of a splitting (or **partitioning**) of the input space, which is segmented and stratified in several distinct and non-overlapping regions depending on the splits defined by the tree (example in Figure 14).

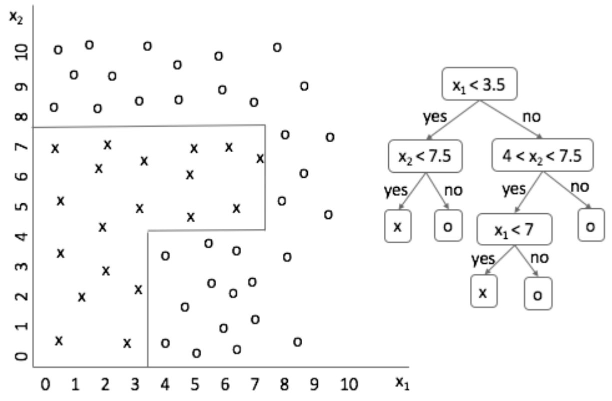


Figure 14: Partitioning space (in two dimensions) of a decision tree.

Each internal node represents an attribute of the dataset, each branch represents the different regions in which the attribute is split, and each leaf node represents a class label. Between all the possible ones, the splits are chosen based by exploiting some "gain" or "impurity measure", such as *Gini Index*, that guarantee the goodness of the split (attributes with homogeneous class distribution and lower degree of impurity are preferred). Gini index over K classes is defined as:

$$Gini = \sum_{i=1}^{K} p_{mk}(1 - p_{mk})$$

where p_{mk} represents the proportion of training observations in the mth region that are from the kth class. A small value indicates that a node contains predominantly observations from a single class.

When building a decision tree, one should consider a **trade-off** between fitting data well with a large and deep tree and avoiding overfitting. In fact, the larger and deeper a decision tree is, the better its accuracy on training data, but the lower its generalization capability. In other words, since each leaf represents a class, a tree with k leaves can shatter any set of k instances. If we do not limit in some way the number of leaves, letting the tree growing indefinitely, the VC dimension becomes infinite and the method would be prone to overfitting, out of PAC learning framework. Thus, some measures have to be taken such as setting maximum depth of the tree or pruning the tree.

2.1.3 Bagging trees and Random Forest

The term stands for *"Bootstrap aggregation"*, it is an ensemble technique that mainly focuses on **reducing the variability** of the simple model. It combines many weak models fitted on bootstrapped samples from the original dataset in an ensemble that has lower variance than the individual models.

Bootstrap is a re-sampling method that solves the question about which data to use for training several learners in ensemble classification. Given a standard training set D of size m, when building k sources for the weak learner (usually having the same size m), one simply sample with replacement from the original dataset. Each of the new generated set will have roughly 64% of the unique samples of D, the others being duplicates (not deleted). All these samples can be considered almost independent from each other, their approximation of the true distribution being more accurate the larger m is.

The base bagging models can be considered approximately independent and identically distributed (i.i.d.) as the bootstrap samples and, since different models are built independently, this process can be **parallelised**. The final prediction is taken by *majority voting* (for a classification task) with all contribution having the same importance. "Averaging" weak learners' outputs do not change the expected answer but reduce its variance just like averaging i.i.d. random variables preserve expected value but reduce variance. Thus, if we consider T weak learners, each one outputting a probability $p_i(x)$ for each class $i \in I$ (class set) for a query sample x, the ensemble prediction will be

$$f_{bag}(x) = \underset{i}{\operatorname{argmax}} \left\{ \frac{1}{T} \sum_{t=1}^{T} p_i^t(x) : i \in I \right\}$$

In order to reduce the variance introduced by the complexity of a tree (e.g. its depth), the *bagging* technique can help.

In addiction to bootstrap, that can enforce the different trees to be almost uncorrelated, a *feature bagging* approach can be introduced to enhance this constraint. More specifically, at each split in each tree, a random subset (of size s) of the d features is selected as candidates for the split, rather than selecting among all the features.

In fact, it may happen that one or few predictors are so strong that they will be selected by most of the trees, without exploiting all the other predictors and making some trees more correlated than others. This ensemble of trees with this kind double bagging technique applied to decision trees is named **Random Forest** (Figure 15), and usually a good number of features

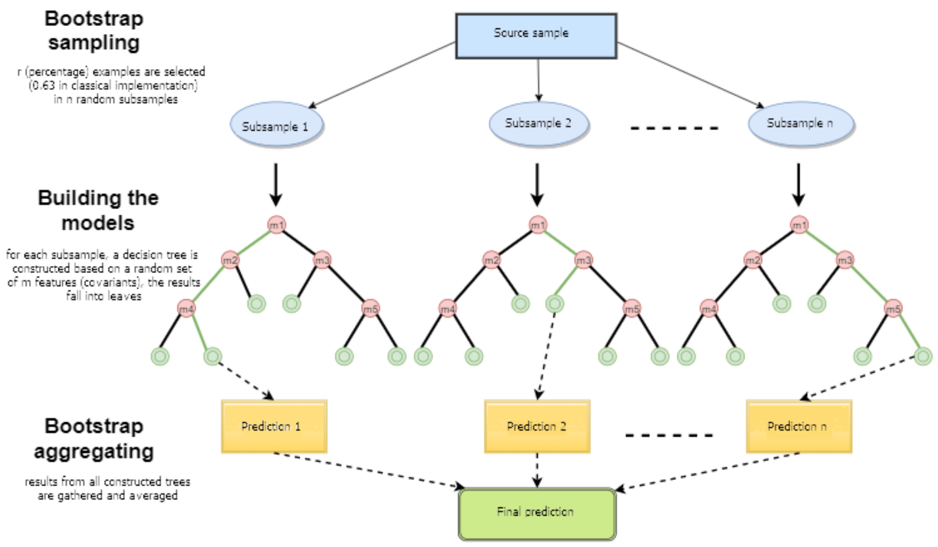

Bootstrap sampling
r (percentage) examples are selected (0.63 in classical implementation) in n random subsamples
Source sample
Subsample 1
Subsample 2
Subsample n
Building the models
for each subsample, a decision tree is constructed based on a random set of m features (covariants), the results fall into leaves
m1
m2
m3
m4
m5
Bootstrap aggregating
results from all constructed trees are gathered and averaged
Prediction 1
Prediction 2
Prediction n
Final prediction

Figure 15: Random forest of decision trees.

is $s = \sqrt{d}$.

2.1.4 Boosting and Adaboost

The boosting paradigm allows the learner to have smooth control over bias-variance trade-off. The idea of boosting is to fit models **sequentially** (iteratively) such that the training of model at a given step depends on all the models fitted at the previous steps. Boosting produces an ensemble model that is generally *less biased* than the weak learners that compose it. For the purpose of this project AdaBoost classifier is used.

Adaptive Boosting (Figure 16) is an iterative algorithm composed of a sequence of consecutive rounds. At each step t, the algorithm defines a distribution over the training samples $S = (x_1, y_1), ..., (x_m, y_m)$ called D^t. Each weak learner gets as input the distribution and it is required to return an hypothesis h_t, whose error it is at most $0.5 - \gamma$, as previously defined. The final strong hypothesis h_s will be a weighted combination of weak ones. The adaptive policy introduced by AdaBoost is to assign a weight w_t to this weak hypothesis, whose magnitude is inversely proportional to the error ϵ and indicates how much this weak learner should be taken into account into the ensemble model. Using this weight, the last step is to update the probability distribution in such a way that the training samples on which h_t makes mistakes will have a higher probability mass. Since this will be the distribution input for the next learner, it introduces an enforcement to focus on the problematic samples.

A single step of AdaBoost algorithm at time t is described below:

1. given training samples S and their distribution D^t, fit the best possible weak model and return a weak hypothesis h_t;
2. compute the weight w_t associated with the returned hypothesis h_t depending on its error ϵ;
3. update the strong learner by adding the new hypothesis h_t multiplied by its weight w_t to the weighted sum;
4. update the distribution D^{t+1} that expresses which observations we would like to focus on at the next iteration;

At step 1, the algorithm simply trains a weak learner on the original data, each sample has the same probability mass $D^1 = (\frac{1}{m}, ..., \frac{1}{m})$. For each successive iteration, the distribution D is updated in a way that incorrectly predicted samples have their probability mass increased, whereas samples that where predicted correctly have their probability mass decreased. In this sense, AdaBoost is sequential because, as iterations proceed, examples that are difficult to predict receive increasing influence and each subsequent learner is forced to concentrate on the examples that were mis-classified by the previous learner.

The main differences between bagging and boosting are the following:

- Bagging can be executed in parallel whereas boosting requires a sequential approach;
- Bagging considers all learners' contributions equal whereas boosting combines weighted contributions;
- Bagging mainly tackles the variability whereas boosting cares manly about bias (even if variance can also be reduced).

As previously stated, the *boosting* approach mainly aims at reducing the bias, not the variance. Moreover, it requires simpler (and thus possibly weaker) learners since it cannot work in parallel. That is the reason why, as we will see in the following section, boosting very deep trees do not bring to good performances.

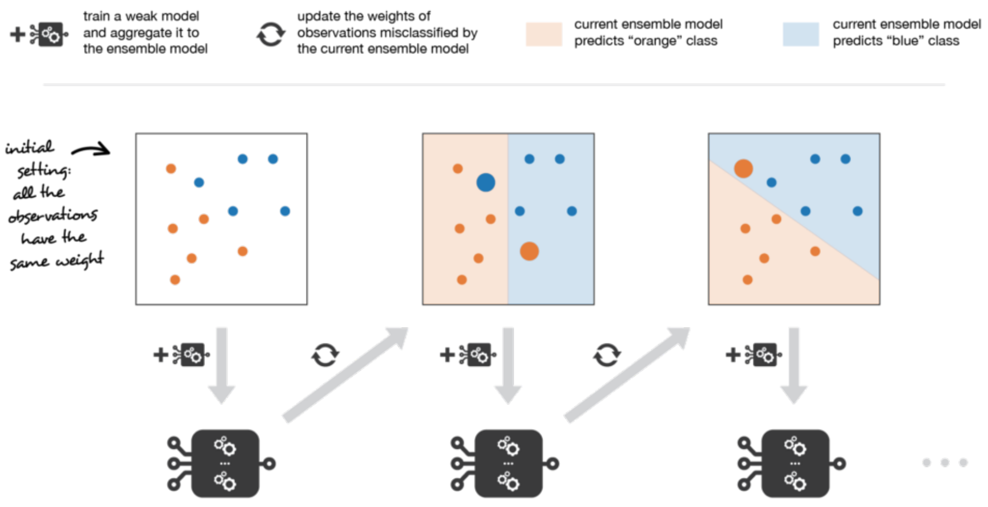

train a weak model and aggregate it to the ensemble model
update the weights of observations misclassified by the current ensemble model
current ensemble model predicts "orange" class
current ensemble model predicts "blue" class
initial setting: all the observations have the same weight

Figure 16: The Adaptive Boosting framework.

2.2 Support Vector Machines

SVM is a supervised learning algorithm that tries to find a separating hyperplane between two (or more) classes. Hard-margin SVM seeks the halfspace that separates the data perfectly with the largest margin, that is minimal distance between a point in the training set and the hyperplane. Soft-margin SVM, instead, does not assume separability of data and allow the constraints to be violated to some extent.

2.2.1 Hard margin vs soft margin

In a binary classification case, where labels $y_i \in \{\pm 1\}$, when training data $(\mathbf{x_1}, y_1), \dots, (\mathbf{x_m}, y_m)$ are linearly separable, in other words there exist a halfspace defined by $(\mathbf{w}, b)$ such that $\forall i \in |m|$, $y_i(\mathbf{w}\mathbf{x_i} + b) \geq 1$, we can select two parallel hyperplanes that separate the two classes of data, so that the distance between them is as large as possible (all points can be classified correctly).

The region bounded by these two hyperplanes is called the *margin*, and the maximum-margin hyperplane is the hyperplane that lies halfway between them. Geometrically, the distance between these two hyperplanes is $\frac{2}{||w||}$, so to maximize the distance between the planes we want to minimize $||w||$. The corresponding optimization problem is:

$$\underset{\mathbf{w},b}{\text{minimize}} \ \frac{1}{2}||\mathbf{w}||^2$$

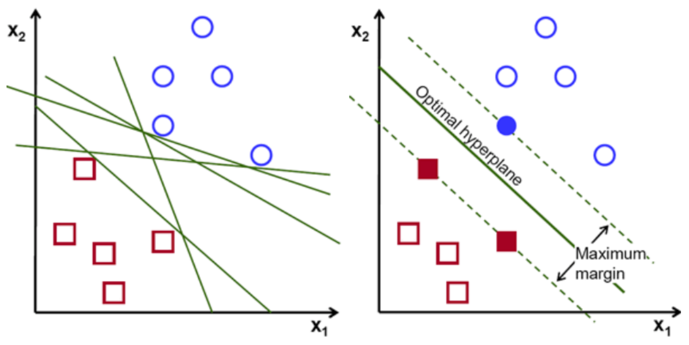


Figure 17: Maximum-margin hyperplane and margins for an SVM trained with samples from two classes.

$$s.t. \quad y_i(\mathbf{w}\mathbf{x_i} + b) \geq 1$$

and the max-margin hyperplane is completely determined by those vectors that lie nearest to it. These x_i are called *support vectors*.

Most of the times data are not linearly separable because of noise, thus we introduce non-negative slack variables $\xi_1, ..., \xi_m$ and rewrite the problem as follows:

$$\underset{\mathbf{w},b}{\text{minimize}} \ \frac{1}{2}||\mathbf{w}||^2 \quad + C\sum_{i=1}^{m}\xi_i$$

$$s.t. \quad y_i(\mathbf{w}\mathbf{x_i} + b) \geq 1 - \xi_i$$

$$\xi_i \geq 0$$

ξ_i measure how much the constraint is being violated. Soft-SVM objective function jointly minimizes the norm of $\mathbf{w}$ and the violations of the constraints that we are allowing. C is a regularization parameter of the classification algorithm which is proportional to the error that we allow because it multiplies the slack variables. Thus, C tells me how much the margin is soft (larger C) or hard (lower C).

2.2.2 Kernel trick

When softening the margin is not enough for making a proper classification, we would like to map training data into another feature space, usually of higher dimensions, thanks to a certain mapping Φ in order to make training data separable. This procedure ends up being theoretically appropriate but practically too computational demanding. Since the solution of the SVM optimization problem can be written as a function of inner products, we can use kernel functions,

$$K(\mathbf{x}, \mathbf{x}') = \langle \Phi(\mathbf{x}), \Phi(\mathbf{x}') \rangle$$

in order to make the algorithm non-linear. The only thing we should know becomes how to calculate inner products between the mapped instances in the higher feature space, or equivalently to calculate the kernel function without explicitly applying the transformation Φ and without using instances' representation in that space. There exist several types of kernel functions, the linear or the Gaussian (Radial Basis Function) are the most used.

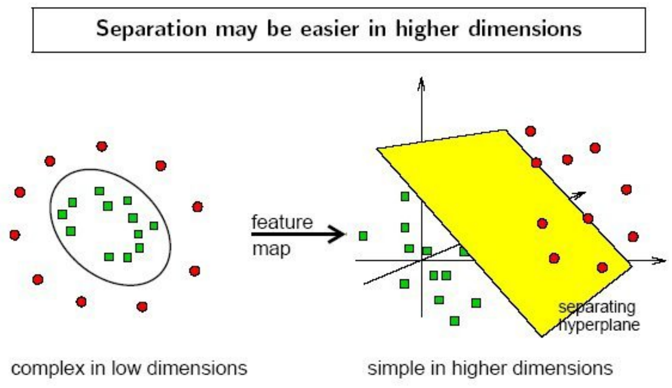

Separation may be easier in higher dimensions
feature map
separating hyperplane
complex in low dimensions
simple in higher dimensions

Figure 18: Mapping Φ from input space to feature space in order to find separating SVM hyperplane.

2.3 K-Nearest Neighbors

The K-Nearest Neighbors algorithm is a supervised machine learning model that can be used to solve classification problems. The idea behind the KNN algorithm is that similar things exist in close proximity, which means that similar things are near to each other.

The learning process of the KNN algorithm has the following steps:

1. initialize K to the chosen number of neighbors
2. for each example, calculate the distance between the query example and the current example from the data;
3. add the distance and the index of the example to an ordered collection and sort the ordered collection of distances and indices from smallest to largest by the distances;
4. pick the first K entries from the sorted collection and get the labels of the selected K entries;

5. return the mode of the K labels (which corresponds to the prediction for the classification task).

In Figure 19 examples of the decision boundaries of KNN classifier are represented. As we decrease the value of K to 1, our predictions become less stable. Inversely, as we increase the value of K, our predictions become more stable due to majority voting, and thus, more likely to make more accurate predictions (up to a certain limit). The number of errors in the training dataset could increase by increasing the value of K, but the classifier would be less linked to the training data and the decision boundaries more general.

KNN does not make any underlying assumption about the data and it is able to achieve a relatively high accuracy in many classification tasks. One advantage is that with the addition of more data points, the classifier constantly evolves and is capable of quickly adapting to the changes in input dataset.

KNN has also some limitations since it is very sensitive to outliers and as the dataset grows, the classification becomes slower. In this case the number of examples is quite high, but the algorithm is capable to train in reasonable time.

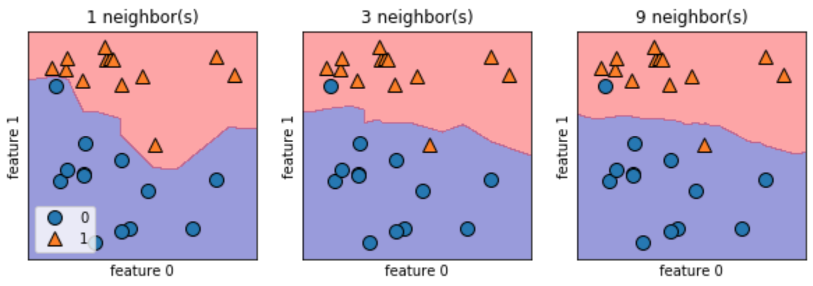


Figure 19: KNN decision boundaries using increasing K values.

3 Results

In this last section the results of the different classification techniques are shown. In particular, the advantages of using ensemble techniques on decision trees (when dealing with the bias-variance trade-off) will be illustrated. Moreover, the classifiers will be compared in order to show the relative strengths and weaknesses when dealing with an imbalance dataset.

For all the classifiers the best parameters are found by performing a **cross-validated grid-search** and extracting the best model to calculate the scores and visualize the results.

The metrics that are used for the evaluation are the **accuracy score** and the **F1 score**, more appropriate to investigate the classification results for a specific class.

The code is available on my GitHub at:
`https://github.com/giuliaTortoioli/MML_project_2021`.

3.1 Decision Tree and Ensemble Techniques

First of all, DecisionTreeClassifier (from *scikit-learn* library) is trained by using the Gini impurity index and the performances are evaluated by using different values for the parameter *max_depth*.

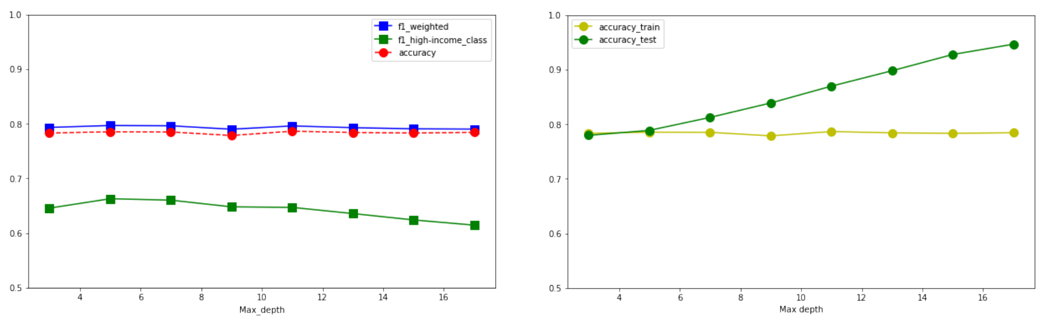


Figure 20: On the right the performances of the Decision Tree classifier on the test set. On the left the comparison of accuracy values between train and test set. The classifier is evaluated using different values of *max_depth*.

The plots in Figure 20 show how the classifier works when increasing the maximum depth. The accuracy and F1 score (weighted for the two classes) remain practically constant on the test set when changing the values of *max_depth*. For this reason the behaviour of the classifier is investigated by analysing the scores achieved in the train set while changing the depth parameter. It can be seen in the right graph in Figure 20 that for higher values of *max_depth* the model starts to overfit on the training data, since the accuracy is increasing in the training set but not in the test set. These results on the train set explain also the decrease of the F1 score for the *high-income* class when increasing the *max_depth* value, since the classifier tends to overfit on the training data which contain less samples belonging to the *high-income* class.

It is evident that a single weak tree is prone to poorly generalize as the tree increase in size. Moreover, a single decision tree does not overcome 80% accuracy score on the test set.

Thus, the performances of ensembles of trees are analysed: RandomForestClassifier, BaggingClassifier and AdaBoostClassifier from *scikit-learn* with DecisionTreeClassifier as base estimator with Gini Index as impurity measure.

All the classifiers are trained by running a grid search on the parameters and the resulting confusion matrices on the test set are shown in Figure 21.

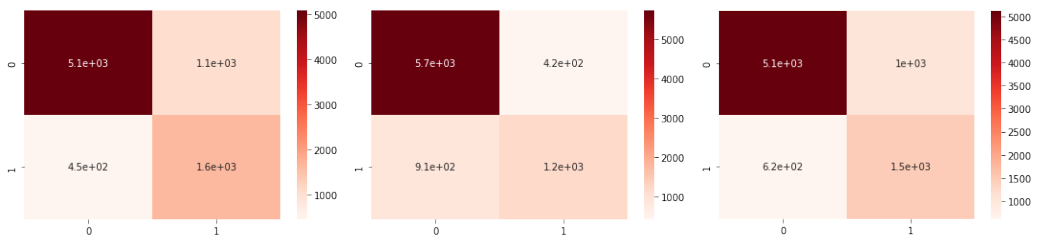


Figure 21: Confusion matrix of Bagging classifier (on the left), Random Forest classifier (on the centre) and AdaBoost classifier (on the right).

Between the three models the best results in terms of weighted F1 score and accuracy are reached by the Random Forest classifier (accuracy score: 83.93%, F1 score: 83.18%). Despite Random Forest has better results in terms of score, Bagging classifier is the one that reached the most balanced results (precision and recall results, as well as accuracy scores, are more comparable between the two classes).

Finally, all the different methods based on the Decision Tree classifier are compared and the evaluation is done by changing the values of *max_depth* and using the number of estimators that were found to be the best from the grid search.

From the accuracy values in the graph in Figure 22 we can extract some insights:

- Random Forest and Boosting classifiers perform generally better than the others, especially if increasing the *max_depth* value;
- boosting with AdaBoostClassifier works better with threes with a lower

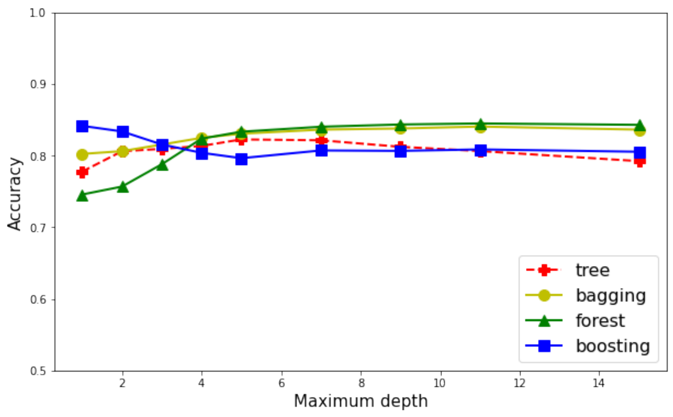


Figure 22: Ensemble trees performances varying maximum depth.

maximum depth, in particular with maximum depths 1 or 2 (blue squared line);

- the singular DecisionTreeClassifier has quite stable performances and on average presents a lower accuracy with respect to the ensemble techniques;

3.2 Support Vector Machines Classifier

Differently to the Decision Tree, SVMs classifier is commonly known as a strong learner, so it could be able to generalize quite well on a large training dataset.

In this case the cross-validated grid-search is used in order to choose the best C (penalty) and type of kernel parameters between *linear* and *rbf*. The single base model with the best configuration found (*'rbf'* kernel and $C = 10$) reaches an accuracy value of 80.81%.

In Figure 23 SVMs classifier performances are shown when varying the value of C. The performances decrease when the regulation term C exceeds 10, especially the results on the test data belonging to the *high-income* class (green line in the graph).

The weighted F1 values are slightly higher then the accuracy values, this means that, despite the accuracy reached by the classifier is not very high, the results are quite balanced between the classes.

3.3 K-Nearest Neighbors Classifier

Finally, KNN classifier is used and evaluated for the classification task. The parameters chosen in the grid-search are the number of neighbors and the

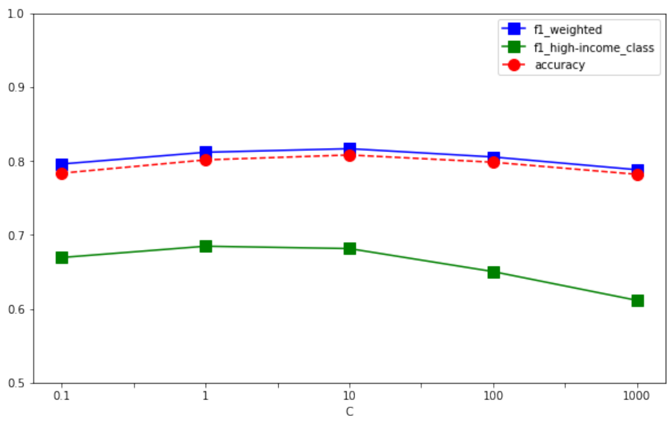


Figure 23: SVMs classifier's performances varying maximum depth.

weight function used in the prediction. It can take either the value *'uniform'*, if the weights are equal for all the neighbors, or the value *'distance'*, if the weights are an inverse function of the distance of the query point with the neighbor.

The best parameters are: 19 neighbors and 'uniform' distance. With these parameters the classifier reaches a quite high accuracy score of 83.27% on the test set. The number of neighbors is high and this could be due to the large number of training samples. By looking at Figure 24 it can be stated that a quite high number of neighbors is needed to allow a generalization on the data distribution.

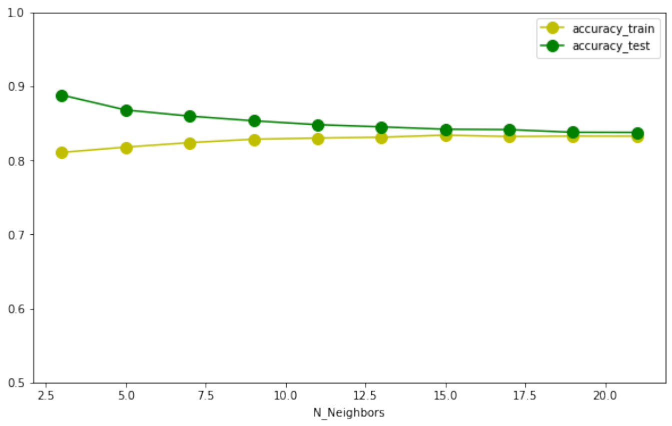


Figure 24: Comparison of accuracy values between train and test set using KNN classifier.

References

[1] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[2] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning - From Theory to Algorithms.* Cambridge University Press, 2014.