MACHINE LEARNING

With content adapted from the thesis PhD “Desenvolvimento de Modelos Analíticos de Apoio à Gestão de Instituições do Ensino Superior, com Recurso a Data Mining”, of Maria Martins, 2020

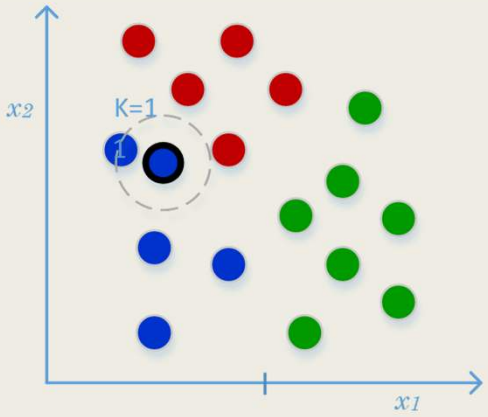
Machine Learning Algorithms

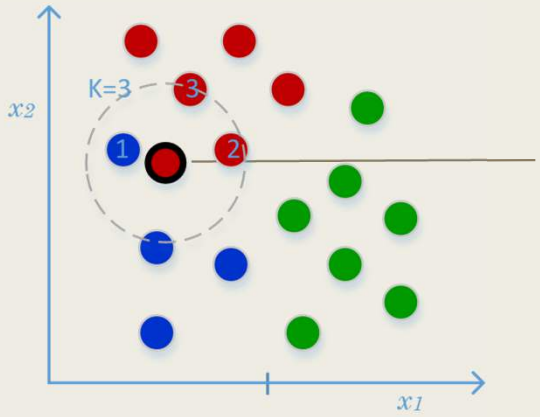
- Here are some of the most commonly used algorithms in the induction of supervised learning models
  - *Linear Regression (for regression)*
  - *Logistic Regression (for classification)*
  - *K-Nearest Neighbors (KNN)*
  - *Decision trees*
  - *Random Forests*
  - *Support Vector Machines (SVM)*
  - *Neuronal Networks (NN)*
- There are versions of most of these algorithms for either regression and classification
- All of these algorithms are available in the Scikit-learn Python package

KNN - The Nearest Neighbors K

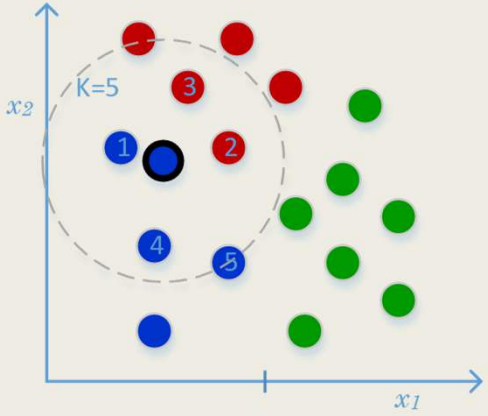
- KNN (K-Nearest Neighbors), despite being one of the simplest classification algorithms, presents, in certain problems, a performance even quite acceptable
- As its name indicates, it is a method that classifies test examples based on their proximity to training examples
  - *More specifically, in its simplest version, it assigns each test example the most frequent class among the K training examples that are closest to them*

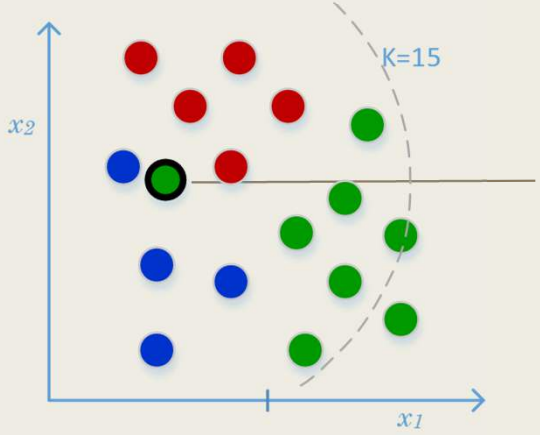
KNN illustration, for instances with 2 attributes, x1 and x2, of 3 different classes (red, green, and blue), and different K values (number of neighbours considered)





Test example classified as belonging to the "red" class, as it is the predominant class in the 3 closest training examples





Example of test classified as belonging to the class of "greens", since it is the predominant class in the 15 closest training examples

KNN - The Nearest Neighbors K

- The proximity of the training instances to the test instance to be classified is determined by a distance measure, the most common being Euclidean
- Note that it is straightforward to interpret this type of algorithm, clearly realising which rule is used in choosing the class of each test sample
    - *However, this is not the reality of other more complex ML algorithms, such as artificial neuronal networks and support vector machines (SVM), which are therefore known as "black box" algorithms*
- In this classifier, there is not precisely the prior construction of a model from the training data
    - *Learning takes place simultaneously with the classification of test examples (learn how to sort each test sample, "looking" at the training data)*
- The prior non-construction of a model during the training phase has computational costs
    - *Much of the effort is deferred to the test phase,*
    - *when, in most ML models, it is training that requires most of the processing effort, and the testing phase is high-speed*
        - It is usually in the test phase (the application phase of the model) that execution efficiency can be assumed as a critical factor (example, real-time classification)

KNN - How many neighbors? (K=?)

In the example illustrated, k=1, K=3, K=5, and K=15 neighbors were used to classify the test sample

- In this case, it appears to have been the choices K=1 and K=5 that led to the correct classification
  - *Even the class chosen with K=3 seems acceptable, given the proximity of the test instance to the red class training*
- But the classification obtained with K=15 will be clearly wrong, given the large distance of the instance from all instances of the class to which it was associated (green class)
- Therefore, the importance of the K parameter is perceived
  - *too many neighbors favor underfitting (poor model tuning),*
  - *but few neighbors also lead, on the contrary, to overfitting, making the model quite noise-sensitive*
    - For example, for the extreme case K=1 (the chosen class is that of the nearest neighbor), the model makes its classification dependent on a single training example, probably missing in your decision whenever this example is an outlier, occupying the influence zone of another class
    - Note that with K=1 it always behaves perfectly with the training data (100% hit), but you will have some difficulty dealing with new data
- Therefore, the tuning of a KNN model essentially involves choosing the best value for the K parameter, also called hyperparameter, in the context of ML
  - *As a rule, ML algorithms include two types of parameters: those that are automatically adjusted during the training phase (learning), and hyperparameters, which we will have to adjust.*

KNN - Fine-tune with validation data

- Predicting that a model will be subject to tuning, it is advisable to partition the initial dataset into 3 subsets (unless cross-validation is used):
  - *training data – it will be with them that the model will learn (usually contains at least half of the examples of the initial dataset)*
  - *validation data – it will be with these data that the model will be evaluated in the various iterations in which different K values are tested (as these data will influence the model, they will not serve the final test)*
  - *test data – to assess the actual generalization capacity of the model (final test)*
    - these are data never before shown to the model; only they will give us an indication of their actual capacity (the final model that came was eventually influenced by the validation data)
- In general, where the developing model is to be improved in several iterations (whether by adjusting hyperparameters or other operations, and whether knn is concerned or not), we should use a specific subset of data for successive intermediate tests – which we designate validation data
- As a rule, in partitioning, the selection of examples for each of the subsets should be done randomly
  - *and stratified, particularly in unbalanced datasets (with classes much more frequent than others),*
  - *in order to ensure the same proportion of each class in the various subsets*

KNN - variants of the base model

- In the simplest version of the classification of a test sample, your class is chosen by voting
  - *being voted the most frequent among the group of k closest neighbours,*
  - *through a simple vote in which each of the neighboring k has the same weight (importance), no matter how close it is to the instance to classify (within the group of k neighbors, there are more neighbors than others...)*
- More evolved versions of the algorithm use a weighted vote, giving greater weight to the votes of the nearest neighbours
  - *such as the weight given by the inverse of the square of its distance to the instance to be classified,*
  - *thus achieving a lower sensitivity of the algorithm at the choice of K*
- Although KNN is more often illustrated with classification problems, like many other ML algorithms, regression can be easily adapted
  - *To do this, you will choose for the numerical value of the test instance the average (which can be weighted) of the response values associated with the Neighboring InstanceK (instead of selecting the predominant class)*

KNN - Application example

With the help of Scikit-learn, we will apply KNN in the classification of flowers of the dataset iris to exemplify, both the algorithm itself and the tuning process that is usually performed in search of the best model

- Let's start by loading the iris dataset

  and let's remember what's in it

```
from sklearn import datasets
iris = datasets.load_iris()
```

```
iris.data
```

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2
       [4.6
```

values of explanatory variables

```
iris.data.shape
```

```
(150, 4)
```

```
iris.feature_names
```

```
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

```
iris.target
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
              2, 2, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

response variable values

```
iris.target_names
```

```
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')
```

```
iris.target.shape
```

```
(150,)
```

Application example - Partitioning the dataset

- We represent the explanatory variables by X and the response variables by y

```
X=iris.data
y=iris.target
```

- Since we intend to fine-tune the classification model, trying to find the best value for the K hyperparameter (i.e., which maximises model performance), we should start by partitioning the initial dataset into 3 subsets: training, validation, and testing
  - *These 3 subsets can be obtained by performing the train_test_split() function twice (in the 1st, the initial dataset is divided into two installments; in the 2nd is the second instalment that is divided into two new installments)*

```
from sklearn.model_selection import train_test_split
```

We start by removing 50% of the examples for training

```
Xtreino,Xrest,ytreino,yrest = train_test_split(X, y, test_size=0.5,
random_state=1234, stratify=y)
print(Xtreino.shape, Xrest.shape, ytreino.shape, yrest.shape)
```

to ensure the same y ratios across the various partitions

to ensure that it always generates the same partitions

```
(75, 4) (75, 4) (75,) (75,)
```

Then, half of the rest go to validation (25%) and the other half for testing (25%)

```
Xvalid,Xteste,yvalid,yteste = train_test_split(Xrest, yrest, test_size = 0.5,
random_state=1234, stratify=yrest)
print(Xvalid.shape, Xteste.shape, yvalid.shape, yteste.shape)
```

```
(37, 4) (38, 4) (37,) (38,)
```

```
del X, y, Xrest, yrest
```

variables that are no longer needed

Application example - Tuning of the model

- Scikit-learn's KNeighborsClassifier class allows us to easily create a classification model based on the KNN algorithm

```
from sklearn.neighbors import KNeighborsClassifier
```

- It is in its instantiation that we should set the value of the Hyperparameter K, indicating the amount of neighbors to use in the classification
- As we intend to optimize the model, we evaluate its performance for different K values

```
for k in range(1,11):
    modelo = KNeighborsClassifier(n_neighbors=k)
    modelo.fit(X=Xtreino, y=ytreino)
    acuracia=modelo.score(X=Xvalid, y=yvalid)
    print("Tx de acerto considerando k={} vizinhos: {:.1f}%".format(k,acuracia*100))
```

```
Tx de acerto considerando k=1 vizinhos: 94.6%
Tx de acerto considerando k=2 vizinhos: 94.6%
Tx de acerto considerando k=3 vizinhos: 94.6%
Tx de acerto considerando k=4 vizinhos: 97.3%
Tx de acerto considerando k=5 vizinhos: 97.3%
Tx de acerto considerando k=6 vizinhos: 100.0%
Tx de acerto considerando k=7 vizinhos: 97.3%
Tx de acerto considerando k=8 vizinhos: 97.3%
Tx de acerto considerando k=9 vizinhos: 97.3%
Tx de acerto considerando k=10 vizinhos: 97.3%
```

- From the results obtained it is noticed that the best performance is achieved with 6 neighbors (K=6)

For performance metrics, the hit rate was used. Being 100%, it means that all flowers in the validation set were correctly classified (in one of the 3 species)

Example - Model Test

- To be able to evaluate the generalisation capacity of the found model, we need to test it with data that we have never seen...
  - *These data were protected, precisely in the variables Xteste and yteste, so that they can now be used*

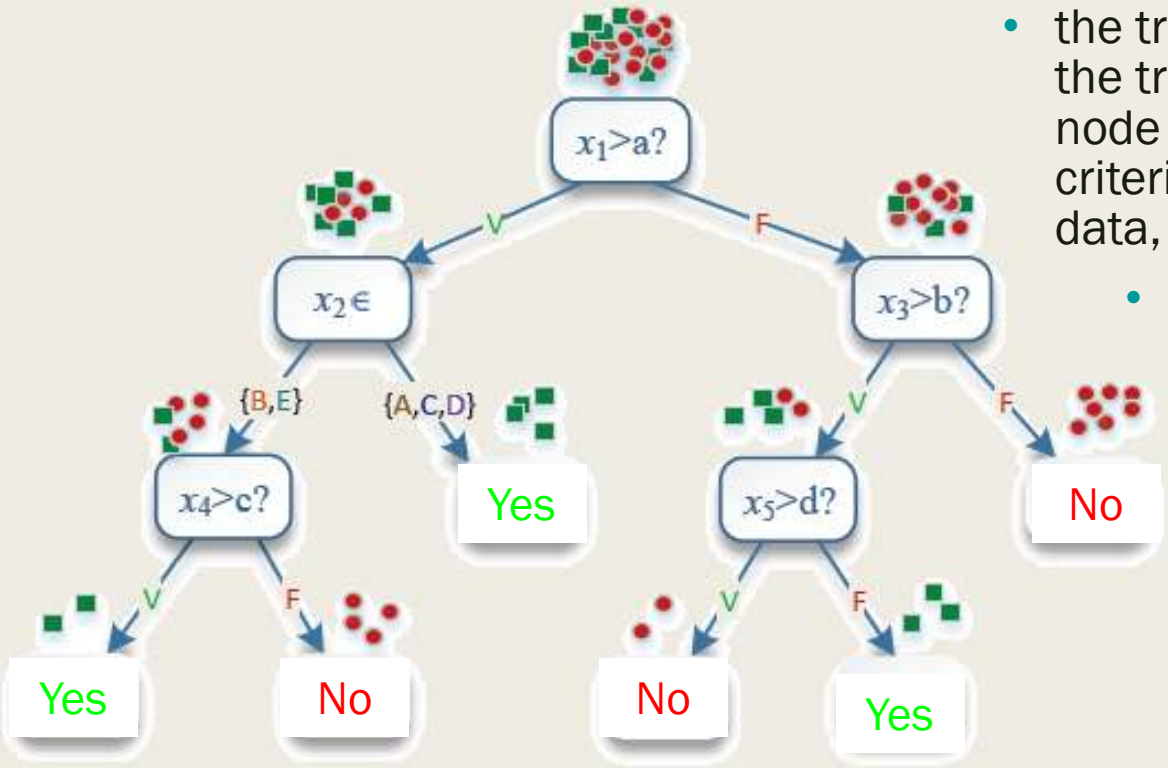
```
k=6
modelo = KNeighborsClassifier(n_neighbors=k)
modelo.fit(X=Xtreino, y=ytreino)
acuracia=modelo.score(X=Xteste, y=yteste)
print("Tx de acerto do melhor modelo (k={}), no conjunto de teste: {:.1f}%".format(k,
acuracia*100))
```

```
Tx de acerto do melhor modelo (k=6), no conjunto de teste: 97.4%
```

- As the hit rate in the test data is 97.4%, it can be concluded that the model found has a good generalisation capability (which is what has always prospered in this type of model),
  - *Maintains very high performance even when applied to new data (its accuracy has decreased only 3%)*
- However, it should be noted that there is always some degree of subjectivity in the results of the ML models, and should therefore not be interpreted with excessive rigidity...
  - *given the number of examples in datasets rarely be optimal (150 examples of flower classifications in the iris dataset is manifestly little)*
  - *and due to the stochastic component that usually characterises both the partitioning of datasets and the ML algorithms themselves*

Decision trees

- A decision tree presents a set of classification rules organised according to a tree-shaped structure
- The rules, being organised hierarchically, are recursively so.
  - *Each tree node represents a separation criterion based on the value of a specific attribute*
  - *and each branch that comes out of this node represents one of the possible results of this criterion.*



Decision tree for binary classification

- the training (automatic construction) of the tree begins by choosing for the root node of the tree the attribute and criterion that best separates the training data,
  - *after which, and according to this separation criterion, the data are divided into different subsets.*
    - Then, for each of these subsets, a new node is created, descending from the first, and the same procedure is applied, choosing the attribute and criterion that best separates the subgroup. And so on, until the subset data is all part of the same class or if it meets any other stop criterion.

Operation of a decision tree

- For a better understanding of the functioning of this type of classifier, let's focus on the decision tree for binary classification illustrated in the figure:
  - *Each variable x_i represents one of the attributes that characterise the instances of the dataset.*
  - *The attribute used in each node and its separation criterion is chosen to maximise the separation, in two subgroups, of the positive class instances of the negative class.*
  - *An attribute can occur more than once in the same tree, which can be numeric or categorical.*
  - *In the example, the variable x_2 represents a categorical attribute that can assume 5 distinct classes (A, B, C, D, and E), and all others are necessarily numeric or ordinal attributes.*
  - *In the illustration, a set of positive and negative instances is added to each node to illustrate the successive divisions suffering the subsets of data as the respective separation criteria are applied to them.*
  - *In this example, the tree performs its function perfectly, completely separating the two classes*
    - But because it is not usually possible to achieve perfect separation of classes, the algorithm has to include additional stop criteria, such as imposing a maximum number of levels that the tree can reach or when it becomes impossible to further separate instances from the subset.

Training and use of a decision tree

- The decision tree was one of the first structures to support machine learning
  - *each node that is automatically added to it, based on the training data, translates into more knowledge that it acquires*
  - *it is expected that the tree, after construction, in this supervised training process that will be added nodes one by one, will be empowered to classify future instances with a high degree of hit*
- Already in the prediction phase, the decision tree will classify each example according to the path that satisfies the conditions from the root node to the terminal node, and the example is classified according to the class associated with the latter node.
  - *Sometimes, after the creation of the tree, "pruning" techniques are applied to it to purge it of possible impurities, thus contributing to the fact that only the information considered relevant is used in decision-making*

Advantages and disadvantages of decision trees

- Advantages
  - *produce classification rules that are easy to interpret*
  - *can be adapted to regression problems*
  - *are quite efficient in the construction of the models*
  - *are not dependent on the scale of the variables*
  - *robustness to the presence of outliers and redundant or irrelevant attributes.*
- Disadvantage
  - *disturbances in the training set can cause considerable changes in the induced model*

Decision Trees - Application example

Let's reclassify the flowers from the dataset iris, but now using a decision tree (AD) to exemplify both the algorithm itself and the use of the cross-validation method (VC) in the tuning process of the respective model

- *Scikit-learn provides us with both the AD algorithm and the tool for tuning with VC*

- Let's start by creating the variables X and y, with the predictors and the variable predicting, respectively.

```
from sklearn import datasets
iris=datasets.load_iris()
X=iris.data; y=iris.target
```

- Since we will use VC, we should not create a specific subset of data for validation since the VC algorithm itself uses the training data, either for training or for validation

```
from sklearn.model_selection import train_test_split
Xtreino,Xteste,ytreino,yteste = train_test_split(X, y, test_size=0.25,
random_state=1234, stratify=y)
print(Xtreino.shape, Xteste.shape, ytreino.shape, yteste.shape)
```

```
(112, 4) (38, 4) (112,) (38,)
```

Application Example - A 1st Test

- Scikit-learn's DecisionTreeClassifier class allows us to create an AD-based classifier easily

```
from sklearn.tree import DecisionTreeClassifier
modelo = DecisionTreeClassifier()
```

- We can start with a prelimit test

```
modelo.fit(Xtreino, ytreino)
modelo.score(X=Xteste, y=yteste)
```

```
0.9473684210526315
```

- Even without tuning, this model already performs quite interestingly
  - *However, we will try to optimise the model even more so that we can understand, with this example, how the VC tuning process can be performed in Scikit-learn*

  (of course, this VC tuning process can then be replicated in the development of any other ML model)

Application example - Tuning of the model

- As we intend to optimise the model, we evaluate its performance to different values of some of its hyperparameters
  - *The GridSearchCV class helps us in this process, using in the search for the best model all possible combinations of the values of the hyperparameters we give it and ensuring that the evaluations in this process are performed by VC*

```
from sklearn.model_selection import GridSearchCV
```

- We provide you, in dictionary form, with the values of the hyperparameters we belong to test

```
grelha_valores={'criterion':['gini','entropy'], 'max_depth':[2,3,4,5,10]}
```

function that scares the quality of the separation, used for criterion in each node

maximum tree height

```
modelo = DecisionTreeClassifier()
procura_modelo = GridSearchCV(modelo,param_grid=grelha_valores,cv=5)
procura_modelo.fit(X=Xtreino, y=ytreino)
```

number of partitions used in VC

Application example - The final test

- Optimal hyperparameters can then be queried via the best_params_

```
procura_modelo.best_params_
```

```
{'criterion': 'gini', 'max_depth': 2}
```

And it's model in the attribute best_estimator_

```
modelo_otimo=procura_modelo.best_estimator_
```

- The generalisation capacity of the found model can finally be evaluated with the test data

```
modelo_otimo.score(X=Xteste, y=yteste)
```

```
0.9736842105263158
```

- Therefore, with the tuning of the model, its performance was further improved, achieving the hit rate that had already been obtained with the KNN algorithm.

- As the exhaustive search performed by the GridSearchCV class is not always feasible, due to the computational effort involved (when used several hyperparameters with many values), we also can opt for a more efficient (albeit slightly less assertive) search, using the RandomizedSearchCV class.

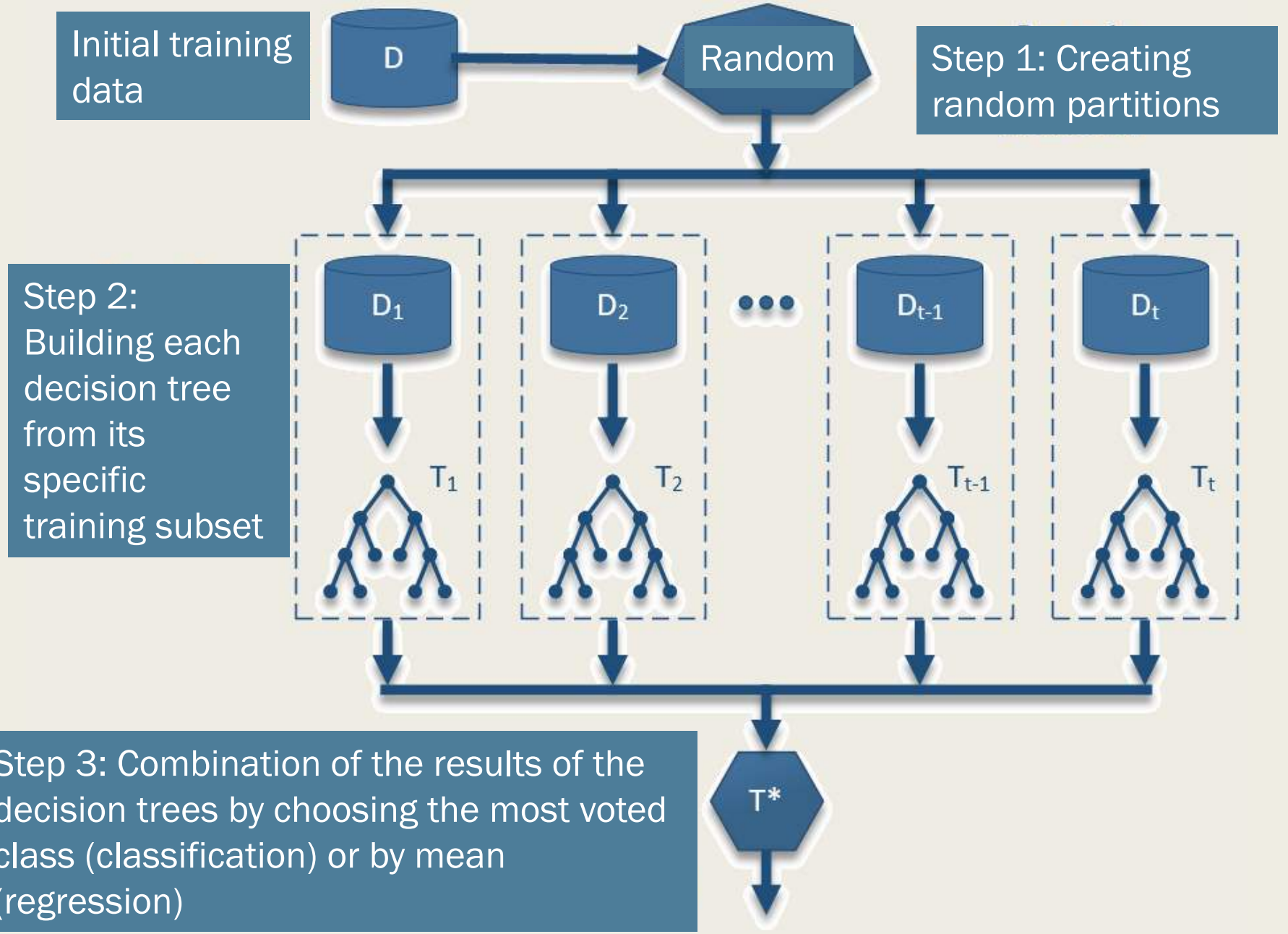
Random Forest

- The random forest (RF) algorithm, proposed by Breiman[1], is a learning method based on comités[2], that generates multiple decision trees during training
  - *It is based on the premise that a set of weak classifiers can create a robust classifier*
  - *RF combines the results of multiple individually trained decision trees with different error patterns to try to optimise overall predictive performance*
- As illustrated in the following schema, to induce each forest tree individually, the algorithm randomly partitions the initial training data set D (which contains n examples and d attributes) into multiple smaller training subsets, $D_1.D_t$,
  - *each obtained independently and by random resampling with replacement of the original set.*
  - *Each of these multiple partitions, which contains examples and i attributes, in which m<n and i<d, is used to induce a different forest tree*
  - *Examples of the initial training set D that do not appear in the training subset of an individual tree, the so-called out-of-bag data, are used as test data to estimate the performance of that tree during the training phase, thus achieving a more reliable estimate, since it is new data*
  - *The random sampling process, both of the examples and the attributes, will cause the increase of the variability of the forest trees, thus achieving to reduce the variance of the final model and keep low the bias of the error in the generalisation.*

[1] *Leo Breiman. Random forests. Machine learning, 45(1):5–32, 2001*

[2] *also called assembly methods or even mixing of specialists*

Random Forest



Advantages and disadvantages of random forests

- Advantages
  - *generate low-skewed estimates*
  - *the ability to model high-scale nonlinear relationships*
  - *resistance to overfitting*
  - *the ability to deal with categorical and continuous data*
  - *the possibility of estimating the importance of predictive variables*
  - *the speed of construction and predictive efficiency, even in sets of high dimension and dimensionality*
  - *it can be said that random forests are one of the algorithms that best results for a vast set of applications, even if they involve large data sets (many instances) and high dimensionality (many attributes)*
- Disadvantages
  - *the lack of reproducibility for different datasets*
  - *do not handle categorical attributes well with a high number of distinct classes*

Categorical predictive variables

- Many of the ML algorithms cannot handle categorical-type predictors
  - *this is the case of neuronal networks and SVM since they involve techniques that wait as inputs only numerical values*
  - *Decision tree-based algorithms typically work well with categorical variables*
- But if we are using Scikit-learn, then this restriction is extended to all its algorithms
  - *all of them require the inputs to be numeric*
- What then do we do to categorical variables?
  - *Simply eliminating them is not a solution because, despite their nature, they can, like numerical ones, play a relevant role in explaining the response variable*
  - *The only reasonable option then is to convert them, in some way, to numerical values*

Categorical ordinal vs nominal

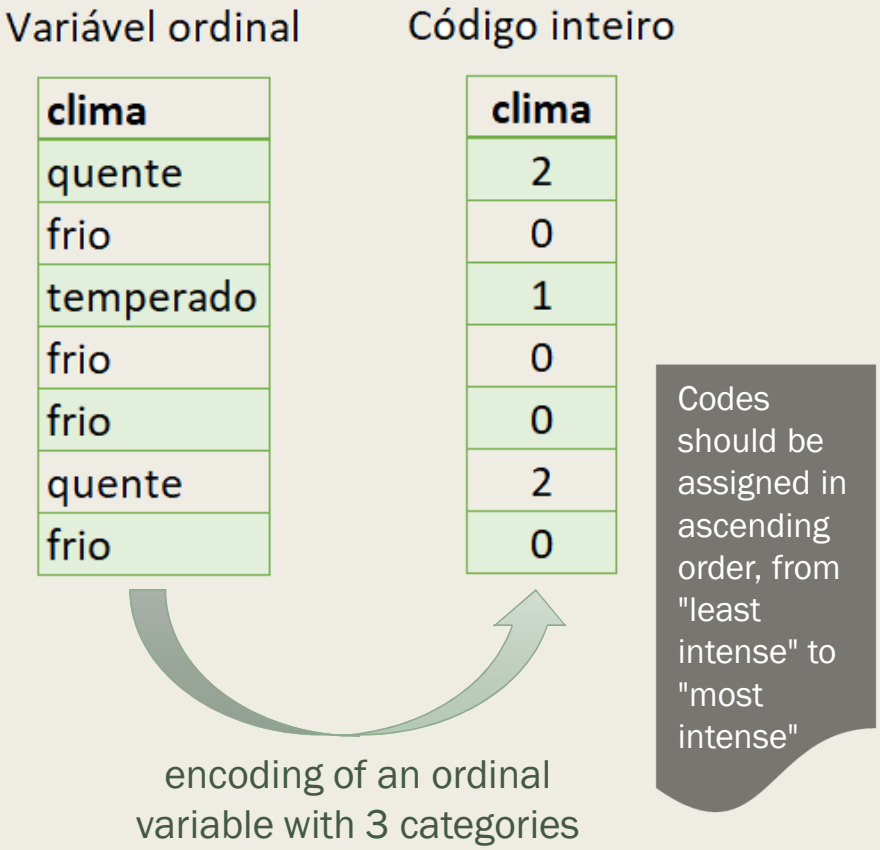
- According to their nature, we can also classify the categorical variables themselves into two different typologies
  - *ordinals – when there is a relationship of order between the various categories (labels/classes) assumed by the variable, to the point where we can sort and compare them (Example: 'Bad' < 'Insufficient' < 'Enough' < 'Good' < 'Very Good' < 'Excellent')*
  - *and nominalones – when there is no order relationship between the various categories (Example: 'Red', 'Green', 'Blue', 'White', 'Black')*

- If the conversion to numeric is done simply in the case of ordinal variables,
  - *simply map to a sequence of integers, in an orderly manner, their categories (e.g.: 'Bad' → 0; 'Insufficient' → 1; 'Enough' → 2 ; 'Good' → 3; 'Very Good' → 4; 'Excellent' → 5),*

the same is no longer the case with nominal

  - *For conversion of nominal categorical variables, a more elaborate coding scheme, called 'one-hot encoding', is usually used, and we will then describe*

Ordinal coding can be performed with the help of the OrdinalEncoder of the module sklearn.preprocessing.



encoding of an ordinal variable with 3 categories

One-hot encoding

- Note that if we mapped the nominal categories directly to whole codes we would mislead the ML model
  - *in this situation, the algorithm would assume that there is a natural order between the categories, when in fact this does not happen – all categories are at the same level (we would be, in essence, 'deceiving' the model)*
- The 'one-hot encoding' is then the scheme that is normally used in ML to convert to numeric the nominal categorical variables
- In this coding scheme, the nominal variable is replaced by a set of binary variables, one per category (label/class)
  - *More specifically, a nominal variable of n categories is replaced by n variables of 0s and 1s*
  - *That is, each category (label/class) gives rise to a one-hot variable, leaving this variable at 1 only in the dataset examples that assume that category*
- The new variables resulting from one-hot encoding are thus mutually exclusive
  - *meaning that for each example in the dataset, only one of them will be at 1 (all the rest will be at 0)*

Variável nominal

fruto
pera
pera
laranja
banana
banana
pera
maca

One-hot encoding

banana	laranja	maca	pera
0	0	0	1
0	0	0	1
0	1	0	0
1	0	0	0
1	0	0	0
0	0	0	1
0	0	1	0

encoding a nominal variable with 4 categories

For coding one-*hot* can be used class OneHotEncoder of the module sklearn.preprocessing.

Dummy variables

- We saw that one-hot encoding creates a binary variable for each category
- This form of representation involves some redundancy
  - *As they are mutually exclusive to each other, one of them turns out to be expendable*
  - *For example, the value of the 1st variable can always be inferred from the others (it will be 1 when all the others are 0)*
- When one of the one-hot variables is excluded, we have a set of n-1 variables (being n the number of categories), which is called dummy variables
  - *In addition to reducing redundancy, some ML models require using these variables, such as linear regression*

- Should boolean or binary variables also be applied to one-hot encoding?
  - *No, since they are already in the dummy representation (a variable of 0s and 1s representing two categories)*
- In Scikit-learn it is customary for datasets to represent nominal variables with integer codes
  - *Special attention should be paid to these situations in order to prevent algorithms from interpreting codes as numerical quantities*

Variável nominal | variáveis *dummy*

fruto
pera
pera
laranja
banana
banana
pera
maca

laranja	maca	pera
0	0	1
0	0	1
1	0	0
0	0	0
0	0	0
0	0	1
0	1	0

encoding a nominal variable with 4 categories

Coding for dummy variables can also be performed by the OneHotEncoder class of the sklearn.preprocessing module, using the drop='first' parameter

Continue solving exercise #18

from the book of exercises