# Data Science Lab
Lab #6

Politecnico di Torino

## Intro

In this laboratory you will learn about classification problems and how they can be approached using a category of tree-based models. In particular, you will use a decision tree from scikit-learn. You will see it in action with different datasets and understand its points of strength and weaknesses. Then, you will implement your own version of a random forest, starting from scikit-learn's decision trees.

## 1 Preliminary steps

### 1.1 Useful libraries

The main library you will need for this laboratory is scikit-learn. You should already have it from previous labs. If not, you can install it using `pip`.

### 1.2 Datasets

For this laboratory, you will both use datasets already available from scikit-learn, and a synthetic dataset you can download from github.

#### 1.2.1 Wine dataset

The Wine dataset is a famous dataset available on the UCI ML repository. The data is the result of a chemical analysis of wines grown in the same region in Italy, but derived from three different cultivars. The analysis determined the quantities of 13 constituents found in each of the three types of wines. From these 13 constituents (features), your goal is to predict the target class (the cultivars).

You can either download the dataset from UCI, or you can get it directly from scikit-learn.

```
from sklearn.datasets import load_wine

dataset = load_wine()
X = dataset["data"]
y = dataset["target"]
feature_names = dataset["feature_names"]
```

#### 1.2.2 Synthetic 2d dataset

This is a very simple 2d dataset that will help you understand some of the limitations of decision trees. It contains 2 synthetic features, each ranging from 0 to 10, and a target class (0 or 1).

You can download it from the following link as a CSV file.

`https://raw.githubusercontent.com/dbdmg/data-science-lab/master/datasets/2d-synthetic.csv`

### 1.2.3 MNIST

You have already met MNIST in the first lab. In that occasion, we used a dataset of 10,000 digits: that was the MNIST test set. A training set of 60,000 digits is also available.

You can download the entire MNIST dataset either from the original source original source, or you can use sklearn's `fetch_openml` function.

```
from sklearn.datasets import fetch_openml

dataset = fetch_openml("mnist_784")
X = dataset["data"]
y = dataset["target"]
```

**Info:** While very convenient, it might happen (if no caching occurs) that `fetch_openml` will need to download the dataset multiple times (i.e. at each execution).

# 2 Exercises

Note that exercises marked with a (*) are optional, you should focus on completing the other ones first.

## 2.1 Wine classification

In this exercise, you will use sklearn's `DecisionTreeClassifier` to build a decision tree for the wine dataset. You can read more about this class on the official documentation.

1. Load the dataset from sklearn, as described in Subsec. 1.2.1. Then, based on your $X$ and $y$, answer the following questions:
   - How many records are available?
   - Are there missing values?
   - How many elements does each class contain?
2. Create a `DecisionTreeClassifier` object with the default configuration (i.e. without passing any parameters to the constructor). Train the classifier using your $X$ and $y$.
3. Now that you have created a tree, you can visualize it. Sklearn offers two functions to visualize decision trees. The first one, `plot_tree()`, plots the tree in a matplotlib-based, interactive window. An alternative way is using `export_graphviz()`. This function exports the tree as a DOT file. DOT is a language for describing graph (and, as a consequence, trees). From the DOT code, you can generate the resulting visual representation either using specific Python libraries, or by using any online tools (such as Webgraphviz). We recommend using the latter approach, where you paste the string returned by `export_graphviz` (which is the DOT file) directly into Webgraphviz. If, instead, you would rather run it locally, you can install `pydot` (Python package) and graphviz (a graph visualization software). Then, you can plot a graph with the following code snippet:

   ```
   import pydot
   from IPython.display import Image
   from sklearn.tree import export_graphviz

   clf = DecisionTreeClassifier(...)
   ...
   # here, features is a list of names, one for each feature
   # this makes the resulting tree visualization more comprehensible
   dot_code = export_graphviz(clf, feature_names=features)
   graph = pydot.graph_from_dot_data(dot_code)
   Image(graph[0].create_png())
   ```

   After you successfully plotted a tree, you can take a closer look at the result and draw some conclusions. In particular, what information is contained in each node? Take a closer look at the leaf nodes. Based on what you know about overfitting, what can you learn from these nodes?
4. Given the dataset $X$, you can get the predictions of the classifier (one for each entry in $X$) by calling the `predict()` of `DecisionTreeClassifier`. Then, use the `accuracy_score()` function (which you can import from `sklearn.metrics`) to compute the accuracy between two lists of values (`y_true`, the list of "correct" labels, and `y_pred`, the list of predictions made by the classifier). Since you already have both these lists (`y` for the ground truth, and the result of the `predict()` method for the prediction), you can already compute the accuracy of your classifier. What result do you get? Does this result seem particularly high/low? Why do you think that is?
5. Now, we can split our dataset into a training set and a test set. We will use the training set to train a model, and to assess its performance with the test set. Sklearn offers the `train_test_split()` function to split any number of arrays (all having the same length on the first dimension) into two sets. You can refer to the official documentation to understand how it can be used. You can use an 80/20 train/test split. If used correctly, you will get 4 arrays: `X_train, X_test, y_train, y_test`.

6. Now, train a new model using `(X_train, y_train)`. Then, compute the accuracy with `(X_test, y_test)`. How does this value compare to the previously computed one? Is this a more reasonable value? Why? This should give you a good idea as to why training and testing on the same dataset returns meaningless results. You can also compute other metrics (e.g. precision, recall, $F_1$ score) using the respective functions (`precision_score`, `recall_score`, `f1_score`). Note that, since these three metrics are all based on a single class, you can either compute the value for a single class, aggregate the results into a single value, or receive the results for all three classes. Check the `average` parameter on the documentation to learn more about this. You can also use the `classification_report` function, which returns various metrics (including the previously mentioned ones) for each of the classes of the problem.

7. So far, you have only used "default" decision trees (i.e. decision trees using the default configuration). The "default" decision tree might not be the best option in terms of performance to fit our dataset. In this exercise, you will perform a "grid search": you will define a set of possible configurations and, for each configuration, build a classifier. Then, you will test the performance of each classifier and identify that configuration that produces the best model.

   On the official documentation for `DecisionTreeClassifier` you can find a list of all parameters you can modify. Identify some of the parameters that, based on your theoretical knowledge of decision trees, might affect the performance of the tree. For each of these parameters, define a set of possible values (the official documentation provides additional information about the possible values that can be used). For example, we can identify these two parameters:

   - `max_depth`, which defines the maximum depth of the decision tree, can be set to `None` (i.e. unbounded depth), or to values such as 2, 4, 8 (we already know from previous exercises the approximate depth the tree can reach with this dataset)
   - `splitter`, which can be set to either `best` (in which case, for each split, the algorithm will try all possible splits), or `random` (in this case, the algorithm will try $N$ random splits on various features and select the best one)

   You can and should identify additional parameters and possible values for them. Then, you can build a parameter dictionary (i.e. a dictionary where keys are parameter names and values are lists of candidate values). Using the `ParameterGrid` class offered by scikit-learn, you can generate a list of all possible configurations that can be obtained from the parameter dictionary. The following is an example with the two parameters we identified:

```python
from sklearn.model_selection import ParameterGrid

params = {
    "max_depth": [None, 2, 4, 8],
    "splitter": ["best", "random"]
}
for config in ParameterGrid(params):
    print(config)
```

   Which returns the following output:

```
{'max_depth': None, 'splitter': 'best'}
{'max_depth': None, 'splitter': 'random'}
{'max_depth': 2, 'splitter': 'best'}
{'max_depth': 2, 'splitter': 'random'}
{'max_depth': 4, 'splitter': 'best'}
{'max_depth': 4, 'splitter': 'random'}
{'max_depth': 8, 'splitter': 'best'}
{'max_depth': 8, 'splitter': 'random'}
```

   For each configuration `config`, we can train a separate model with our training data, and validate it with our test data: for each configuration, compute the resulting accuracy on the test data. Then, select the parameter configuration having highest accuracy.

**Info:** In Python you can use the `**` operator to pass a dictionary as keyword (named) parameters to a function. To further understand this syntax, you can read more about *args and **kwargs. In this specific case, for each `config` dictionary, you can create a decision tree with the following code:

```
clf = DecisionTreeClassifier(**config)
```

**Info:** What we referred to as "parameters" typically goes by the name of "hyperparameters". These are parameters that are set *before* the training of the model. A model's "parameters", instead, are those value that are learned during the training phase (for a decision tree, for example, the features to split and the threshold values for the splits are parameters).

8. In the previous exercise, you searched for the best configuration among a list of possible alternatives. Since we used our test data to select the model, you may be overfitting on the test data (you may have selected the configuration that works best for the test set, but which may not be as good on new data). Typically, you do not want to use the test set for tuning the model's hyperparameters, since the test set should only be used as a final evaluation.

   For this reason, datasets are typically splitted into

   - *Training set*: used to create the model.
   - *Validation set*: used to assess how good each configuration of a classifier is.
   - *Test set*: used at the end of the hyperparameter tuning, to assess how good our final model is.

   However, it often happens that only a limited amount of data is available. In these cases, it is wasteful to only use a small fraction of the dataset for the actual training. In these cases, *cross-validation* can be used to "get rid" of the validation set. You can read more about *cross-validation* on the course slides. One popular method of is the *k-fold cross-validation*. In this, the training set is split into $k$ partitions. $k-1$ are used for the training, the other one is used validation. This is repeated until all partitions have been used once for validation.

   Sklearn offers the `KFold` class for doing k-fold cross-validation. You can use this class as follows:

```
from sklearn.model_selection import KFold

# Split the datasets into two:
# - X_train_valid: the dataset used for the k-fold cross-validation
# - X_test: the dataset used for the final testing (this will NOT
#   be seen by the classifier during the training/validation phases)
X_train_valid, X_test, y_train_valid, y_test = train_test_split(...)
kf = KFold(5) # 5-fold cross-validation
# X and y are the arrays to be split
for train_indices, validation_indices in kf.split(X_train_valid):
    X_train = X_train_valid[train_indices]
    X_valid = X_train_valid[validation_indices]
    y_train = y_train_valid[train_indices]
    y_valid = y_train_valid[validation_indices]
```

   Notice that `kf.split()` returns a list of tuples, where the first value of each tuple are the indices that should be used for training, the second are the indices used for validation. Then, since $X$ and $y$ are NumPy arrays, we can extract the values we are interested in with fancy indexing.

   For each fold, you can use the training data (i.e. `X_train` in the example above) to train each classifier (i.e. decision trees with different configurations) and measure the performance on the validation data (i.e. `X_valid`). You can then aggregate the information extracted (e.g. by computing the overall accuracy from the accuracies on each fold) and select the best performing model. After you select one model, you can assess its performance on never-before-seen data (i.e. your test set).

9. (*) Given a decision tree, we can assign an importance to each split of the tree. The importance of a split can be computed as the decrease in impurity achieved by it. The following are some definitions we will use to define the impurity decrease of a node $P$:

   - $i_P$ is the impurity (e.g. GINI index) of the node (parent)
   - $i_L$ and $i_R$ are respectively the impurities of the left and right children of $P$
   - $|P|$ is the cardinality of the parent node (i.e. the number of elements contained)
   - $|L|$ and $|R|$ are the cardinalities of the left and right children
   - $N$ is the total number of observations in the dataset

   A possible way of computing the impurity decrease $I(P)$ of $P$ is the following:

   $$I(P) = \frac{|P|}{N} i_P - \frac{|L|}{N} i_L - \frac{|R|}{N} i_R \tag{1}$$

   That is, the impurity of the parent node minus the impurity of the children, each weighted by the fraction of elements contained within. The higher this impurity decrease, the better the split is at creating "pure" children.

   This defines how important each node is. We can also define how important any feature is, by summing the importance of the splits that use that feature. We can do this for each of the features of our dataset. Then, we can normalize these weights so that they sum to 1.

   In this exercise you will build a function that, given a tree, extracts all the feature importances. To do this, you should have some prior knowledge of how binary trees work and, in particular, how a pre-order tree traversal works. This is because the `DecisionTreeClassifier` class has an attribute, `clf.tree_`, which contains both the features used at each split (`clf.tree_.feature`) and the impurity for each split (`clf.tree_.impurity`). These are arrays of the pre-order traversal of the tree. From these, you should build the feature importance for each split.

   **Info:** Please note that this exercise requires some extensive knowledge on data structures you may or may note have acquired during your studies (depending on your background). Even if you do not complete this exercise, please do spend some time understanding how the feature importance is computed (that part only requires concepts from this course), and keep in mind that sklearn already computes the feature importances of its trees, you can find it at `clf.feature_importances_`.

## 2.2 Synthetic dataset

In this exercise, you will apply some of the steps you have already applied in Exercise 1 on a different dataset. This will highlight some of the weaknesses of decision trees.

1. Load the synthetic dataset you have previously downloaded. This dataset has two features and a class label. Use matplotlib's `scatter()` function to plot the dataset on a 2D plane and color the points based on their class label. How do you expect a decision tree to approach data distributed in this way?

2. Build a "default" decision tree using the entire dataset, then visualize the learned model. What is the tree learning, and why?

3. (*) Identify a preprocessing step that would make the decision tree "correctly" approach this problem.

4. (*) Sklearn's decision trees store, for each split they do, the information about the feature they are using for the split and the threshold value used in the comparison. You can find this information in `clf.tree_.feature` and `clf.tree_.threshold` (the order of the elements in those arrays is the one you get with a pre-order visit of the decision tree). With this information, plot the decision boundaries (i.e. features' thresholds) applied by the decision tree on the dataset. Ideally, you should have a 2D scatter plot with vertical and horizontal lines that divide the plane into subregions (one for each leaf of the tree). You can use matplotlib's `axvline` and `axhline` to plot vertical and horizontal lines in your plot.

## 2.3 Random forest

In this exercise, you will implement your own version of a random forest, using the trees available from scikit-learn. You will then train the random forest using the MNIST dataset and assess its performance compared to decision trees.

1. Load the MNIST dataset into memory. Divide the 70,000 digits you have into a training set (60,000 digits) and a test set (10,000 digits).

2. Train a single decision tree (with the default parameters) on the training set, then compute its accuracy on the test set.

3. For this next exercise, you will implement your own version of a random forest. A random forest is an *ensemble* approach: it trains multiple trees on different portions of the dataset. This lowers the chance of overfitting on the dataset (the single tree might overfit its portion of data, but the overall "forest" will likely not). Each tree of the random forest is trained on $N$ points extracted, with replacement, from the entire dataset (comprised of $N$ points). Note that, since the extraction of the points is done *with replacement*, selecting $N$ points does not necessarily extract *all* points of the dataset. Indeed, only approximately 63.2% of all points will be extracted for each tree[1].

   Each tree, additionally, bases each split decision using a subset of all features. The size of this subset, $B$, is often selected to be the square root of the total number of features available, but different random forest may adopt different values. This parameter can be defined for each decision tree through the `max_features` parameter. When building a tree, a random sample of `max_features` features will be extracted and used to select the split.

   Another important parameter for random forests is the number of trees used. We will call this parameter `n_estimators`. During training, each of these trees (or estimators) is trained with its subset of data. During the prediction of a new list of points, each tree of the random forest will make its prediction. Then, through majority voting, the overall label assignment is made. Majority voting is just a fancy way of saying that the class selected by the highest number of trees is selected.

   With these information about random forest, you can now implement your very own. The following is the structure your random forest should have.

   ```
   class MyRandomForestClassifier():
       def __init__(self, n_estimators, max_features):
           pass

       # train the trees of this random forest using subsets of X (and y)
       def fit(self, X, y):
           pass

       # predict the label for each point in X
       def predict(self, X):
           pass
   ```

4. Now train your random forest with the 60,000 points of the training set and compute its accuracy against the test set. How does the random forest behave? How does it compare to a decision tree? How does this performance vary as the number of estimators grow? Try values from 10 to 100 (with steps of 10) for `n_estimators`.

5. Scikit-learn implements its own version of a random forest classifier, which is unsurprisingly called `RandomForestClassifier` (from `sklearn.ensemble`). Answer the same questions as the previous exercise. How does your implementation of the random forest compare to sklearn's?

6. Much like for decision trees, sklearn's random forests can compute the importance of the features used. It does this by aggregating the feature importance of the trees into a single value. If $I_{ab}$ is

[1] The proof of this statement is left to you

the feature importance of the $a^{th}$ feature according to the $b^{th}$ tree, the feature importance for $a$ according to the random forest can be computed as follows:

$$I_a = \frac{\sum_j I_{aj}}{\sum_i \sum_j I_{ij}} \tag{2}$$

That is, the overall feature importance for any feature is given by the sum of the feature importance for that feature across all trees, divided by the sum of the feature importances across all trees. This makes it so that $\sum_i I_i = 1$. Compute the feature importance for the 784 features of MNIST according to your random forest (to compute the feature importance of each tree, you can either use sklearn's precomputed feature importance, `tree.feature_importances_`, or you can use your own implementation from Exercise 1.

7. (*) From the previous exercise, you should now have an array with 784 feature importances, one for each of the features in MNIST. You can reshape this array to a $28 \times 28$ matrix of values (remember that MNIST images are $28 \times 28$ black and white images). You can use the `seaborn` library to visualize a heatmap of this matrix (i.e. a 2D grid where elements have different colors based on their value). The following code snippet does exactly this.

```
import seaborn as sns

# This is the result from the previous exercise
feature_importances = get_feature_importances(clf)

sns.heatmap(np.reshape(feature_importances, (28,28)), cmap='binary')
```

Now train a random forest from sklearn, extract its feature importance (`rf.feature_importances_`) and visualize it. Does it resemble your results? What are the most important features? From Lab 1, you should have some idea as to which features are most relevant to distinguish 0's from 1's. Are those pixels also relevant for the 10 classes problem?