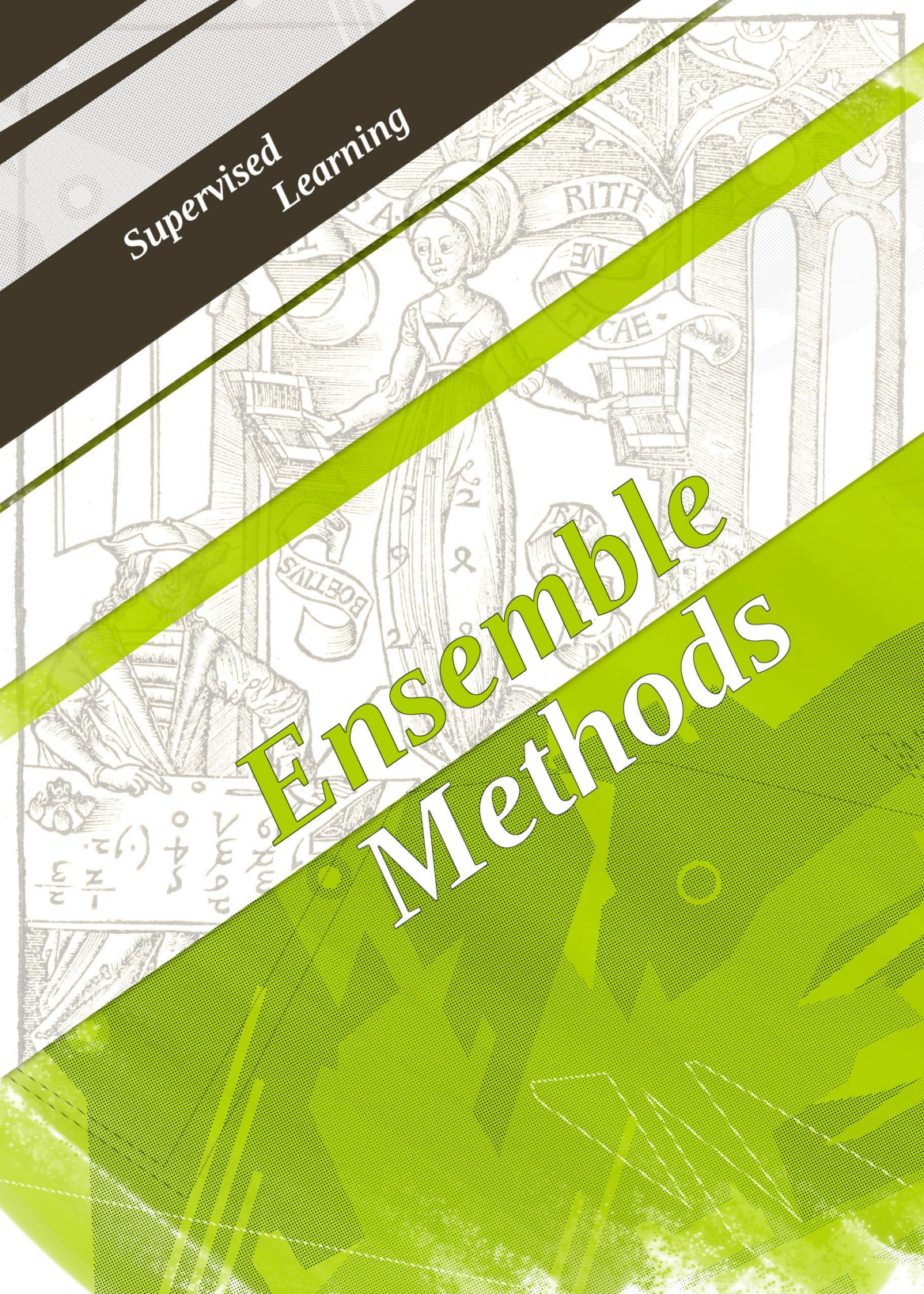
Supervised Learning

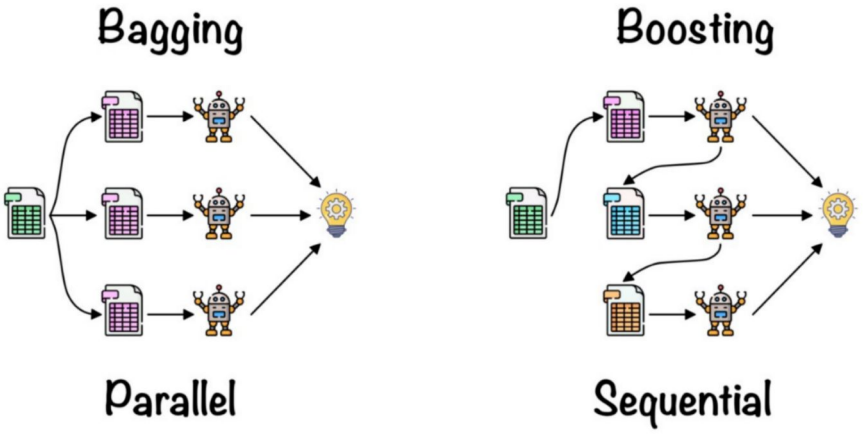
Ensemble Methods



1. Introduction to ensemble methods

Ensemble methods combine multiple WEAK learning algorithms into one optimal/strong model that is expected to outperform all of them.

To achieve such a goal, there exist two families of techniques called **Bagging** and **Boosting** that respectively combine models trained in parallel and sequentially.



Weak Learners

A **weak learner** is any learning algorithm that performs just slightly better than random guessing and has a strong variance in its results.

Such algorithms include **Decision Tree**, **Support Vector Machine**, **k-Nearest Neighbors**, **Linear regression**, **Logistic regression**...

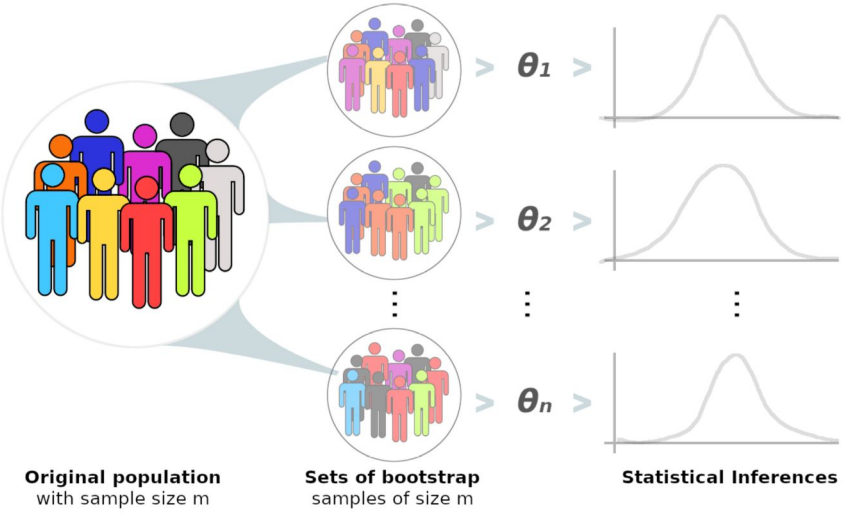
They are the prefect kind of algorithm for techniques that need to train lots of models in order to combine them because they are usually easy to create and fast to train.

In order to be efficient, the **ensemble methods** needs to have access to **weak learners** that are sufficiently different from each others. That's why the selected algorithms must have strong variance.

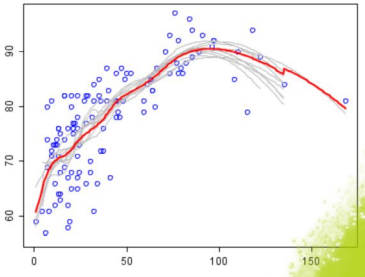
But to ensure even more difference between the models, one approach is to train them on different datasets.

Unfortunately the datasets are often quite limited in size. So, one alternative solution is to **GENERATE** *(i.e. bootstrap)* **NEW DATASETS** of the same size that has almost similar probability distribution *(the samples can be selected more than once)*.

Such a sampling method with replacement is called **Boostraping**.



Using such approach, the **Bagging** and **Boosting** reduce the overall variance and hence provide better and more stable predictions.



2. Bagging | Parallel ensemble techniques

The **bagging** *(**b**oostrap **agg**regation)*, consists in combining weak learners TRAINED IN PARALLEL on datasets generated with bootstrap method in order to get a more efficient predictive model.

In such an algorithm, the idea is to try correcting the overfitting mistakes by relying on the most consensual prediction.

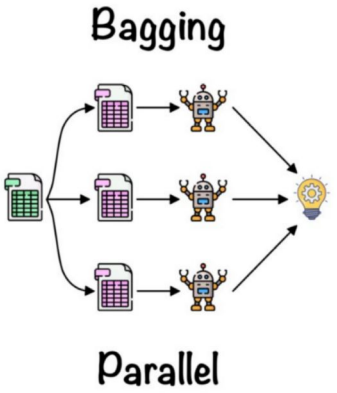
It can be viewed as the equivalent of the wisdom of the crowd effect in our day to day lives; it combines different "opinions" to make a decision that satisfies the largest part of the "voters".

The regular democratic process is a good example of such a system. And the American national election process with its Electoral College extra step, can be considered as a regularized/pondered alternative.

Each weak learner is trained on its own dataset crafted beforehand.

Then, once the weak learners are trained, the method used to make prediction depends on the problem:

- REGRESSION use the average of the weak models' predictions.
- CLASSIFICATION use the majority vote on weak models' predictions.



A **Random Forest** is an extension to the **bagging** method.

It similarly combines several weak learners TRAINED IN PARALLEL on parallelly bootstrapped datasets to get an optimal model.

However, the **weak learners** are exclusively **Decision Trees** and the **bootstraped** datasets only use a fraction of the available features.

Decision Trees usually have very good prediction power with a small computation cost, but they also tend to over-fit very easily…

So **Random Forest** tries to take advantage of such characteristics; if the trees over-fit very differently, their "mean" can be probably produce a very robust model. That's wise!

And hence **Random Forest** tries to ease the overall over-fitting to get the high predictive power of the trees without their counterpart.

To do so, the datasets also needs to be built in such a way that each decision tree produces very different results. That's the reason why this algorithm uses two random methods *(hence the name).*

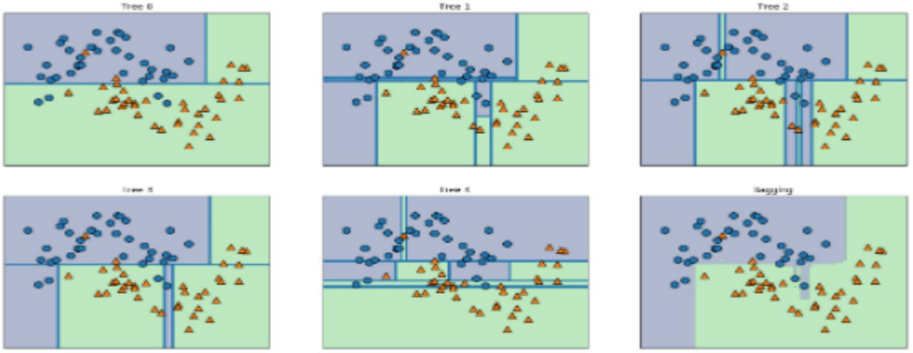
1. New datasets are randomly created with the **boostraping** method.
2. Each new dataset randomly keep a fraction of all the features.

Then the decision trees are grown using these truncated datasets *(without pruning)*. Such a process reduces the correlation between the trees and hence produce a better overall model.

Random Forest is often used in practice because it has pretty good performances with almost any problem. It is one of the first algorithms one should try when facing a new Machine Learning problem.

Once the decision trees are trained, the method used to make prediction depends on the problem:

- **REGRESSION** use the average of the weak models' predictions.
- **CLASSIFICATION** use the majority vote on weak models' predictions.



Important Hyper-parameters

The performance of such a model depends on two factors:

- the correlation between any pair of trees in the forest.
  If the correlation increases, the error increases too.
- the performance of each individual tree.
  The overall model is more likely to perform well if the individual trees performs well too. Increasing the number of selected features in the **decision trees** *will increase their performance.*

Those factors can be tuned using the following hyper-parameters:

- max_features – the maximum number of features to consider when looking for the best split. *Raising this value decreases the correlation between the decision trees and increases their own predictive power. So the overall performance increases but the model is more complex and requires more computation…*
- n_estimators – the number of trees builds in the ensemble.
  The predictive power usually increases with the number of estimators but the computational power required increases too!

Advantages

- **FEW OVERFITTING** – because it's a combination of decision trees with low correlations, **Random Forest** almost never overfit.
- **GOOD PERFORMANCE** – it has very good performance on most problems, even without tuning any parameters.
- **LARGE DATASETS** – because of the parallel nature of the bagging methods, and the low complexity of the Decision Trees, the Random Forest can compute several trees at the same time, and hence handle very large datasets with reasonable resources.
- **MEMORY** – it needs the same memory space as the size of the data.
- **FEATURE IMPORTANCE / INTERPRETABILITY** – considering that the higher a feature is in the tree, the more it contributes training dataset, one can estimate the importance of such features and hence select the most important ones before training or try to interpret the model.
- **MANY USAGES** – **it** allows many applications such as **clustering**, **outlier detection**, **feature selection**, **classification**, **regression**…

Disadvantages

- **TIME** – the training and prediction time can be quite long in comparison to other algorithms. At first, it has to train all decision trees and then it has to make a prediction using each of them to generate output for a given input data.
- **INTERPRETABILITY** – a **Random Forest** is less easy to interpret than a single **decision tree**.
- **TOO EASY** – it is such an easy & good solution that one may forget that even better solutions exist… keep trying other algorithms!

3. Boosting | Sequential ensemble techniques

The **boosting** methods consists in combining several weak learners TRAINED SEQUENTIALLY on datasets generated with bootstrap method in order to get a more efficient predictive model.

In such an algorithm, the idea is to try correcting the mistakes of the previous models at each new step of the sequence.

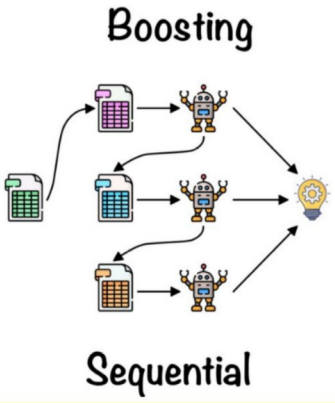
It can be interpreted as the equivalent of the knowledge transfer between generations in our societies.

For example; *Newton's equations* led to *Maxwell's equations*, which led to *Lorentz transformation*, which in turn led to the *general relativity*... And this theory is the core of new theories and applications such as GPS etc

The original dataset is used to train the first step. Then after each step of the sequence, a weighted dataset is crafted to give more importance to the samples that went wrong in the last iteration. And this new dataset is used to train the next step of the sequence.

Finally, the predictions of all weak learners are assembled and pondered using the weights computed at each step of the sequence.

- REGRESSION use $G(x)=\sum_{i=1}^{M} \alpha_m G_m(x)$
- CLASSIFICATION use $G(x)=sign[\sum_{i=1}^{M} \alpha_m G_m(x)]$



AdaBoost

AdaBoost *(additive boosting)* is the historical algorithm of the **boosting** family, it has been designed to solve binary classification.

Afterwards this version can be seen as a special case of **gradient boosting** *(see below)* with a particular loss function. But it evolved to solve multi-class classification and egressions problems.

Also, even if it can use any weak learner, it often use one-level decision trees *(decision stump)* that are very very fast and light.

Hence, this algorithm is now a very interesting method, but **gradient boosting** is still much more flexible. So let's discover it.

Gradient Boosting

Gradient boosting is a generalization of the **AdaBoost** method because it uses the **functional gradient descent** to minimize the error in function space and hence can use any loss function.

- REGRESSION often use
  - the least squares: $1/2(h_\theta(x^{(i)})-y^{(i)})^2$
  - the L1 norm: $|h_\theta(x^{(i)})-y^{(i)}|$
- CLASSIFICATION often use:
  - the exp loss *(same as AdaBoost)*: $\exp(-\ h_\theta\ (x)\ \ y)$
  - the binomial deviance: $\log(1+\exp(-2\ \ h_\theta(x)\ \ y))$

It similarly combines several weak learners TRAINED SEQUENTIALLY on weighted boostrapped datasets to get an optimal model. And even if those weak learners are often **decision trees** in practice *(with 4 to 8 levels rather than stumps)*, it can in fact be any weak learner.

XGBoost *(Extrem Gradient Boosting)* is a specific version of Gradient Boosting with highly optimized methods.

Important Hyper-parameters

The performance of such model depends on two factors:

- the tendency to overfit of the boosting methods.
  Boosting methods tends to overfit because each extra step in the sequence keep minimizing the cost function… So at some point it can overemphasize outliers and cause overfitting.
- the performance of each individual tree.
  The overall model is more likely to perform well if the individual learners performs well too.

Those factors can be tuned using the following hyper-parameters:

- n_estimators – the number of trees builds in the ensemble.
  The predictive power usually increases with the number of estimators but it also easily tends to overfit. So it is usually a good idea to reduce the learning rate in order to compensate.
- learning_rate – it controls the amount of contribution that each model has on the ensemble. *A low value usually makes the model perform better. But there is a trade-off with the number of trees. Hence lower values may require more decision trees in the ensemble, and hence more computation power.*
- max_depth – the maximum depth of the trees in the ensemble.
  It controls how general or overfit each tree is to the training set. This method usually performs well with trees that are neither too small (high-bias) nor too large (high-variance).
- min_samples_split – the number of samples used to split the trees.
  A high value in this parameter tends to protect against overfitting, but if the value is too high, under-fitting can arise.
- many others – in fact **gradient boosting** is very sensible to a lot of hyper-parameters. It is highly recommended to use grid-search.

Advantages

- **GOOD PERFORMANCE** – it has very good performance on most problems, given that the hyper-parameters are correctly set in order to avoid the overfitting. It even performs better than **Random Forest** with small / medium dataset *(< 4000)*.
- **NO DATA PREPROCESSING** – this method often works well with both categorical and numerical values without any special care. It also handle missing values, so imputation is usually not required.
- **FEATURE IMPORTANCE / INTERPRETABILITY** – if the weak learners are decision trees, considering that the higher a feature is in the tree, the more it contributes training dataset, one can estimate the importance of such features and hence select the most important ones before training or try to interpret the model.
- **MANY USAGES** – **it** allows many applications such as **classification**, **regression**, **clustering**, **feature selection**, **outlier detection***(XGBOD)*

Disadvantages

- **EASILY OVERFIT** – as it keeps descending the functional gradient at each step, it can overemphasize outliers and cause overfitting.
- **TIME** – the training and prediction time can be quite long. At first, it has to train all weak learners, then it has to make a prediction using each of them to generate output for a given input data.
- **COMPUTATIONALLY EXPENSIVE** – often require many trees *(>1000)* which can be time and memory exhaustive.
- **INTERPRETABILITY** – **it** is less easy to interpret than a single weak learner *(decision tree, logistic or linear regression…)*
- **LOT'S OF HYPER-PARAMETERS** – in order to get a high predictive power and yet avoid overfitting, there is a lot of hyper-parameters to correctly set… Using a grid search is highy recommended.

CODE with sklearn & XGBoost

Bagging & Boosting Classifications with sklearn

```
from sklearn.ensemble import BaggingClassifier <or any other>
# … load,  split and prepare data (WITHOUT feature normalization)…

  # Train the model
  # in practice one use hundreds or thousands of estimators
clf = BaggingClassifier(
                        n_estimators=500 , base_estimator=SVC(),
                        max_features=10, random_state=0, n_jobs=-1 )
  # the default estimator is DecisionTreeClassifier because it's fast
  # or
clf = RandomForestClassifier(
                        n_estimators=500, max_features=5,
                        max_depth=6, random_state=0, n_jobs=-1 )
  # or
clf = AdaBoostClassifier(
                        n_estimators=500,
                        learning_rate=0.5, random_state=0)
  # or
clf = GradientBoostingClassifier(
                        n_estimators=500,
                        learning_rate=0.5, max_features=None,
                        max_depth=4, random_state=0)

  # Fit the training set
clf.fit( X_train, y_train )

  # Get default evaluation score
accuracy = clf.score( X_test, y_test )

  # Get weak estimators
weak_estimators = clf.estimators_

  # Make predictions
weak_estimators[0].predict( X_new )
y_pred = clf.predict( X_new )                  # return 0 or 1
y_pred = clf.predict_proba(X_new )[:,1]    # return probability of getting 1
```

If is highly recommended to use grid-search or at least cross-validation along with all those algorithms.

Bagging & Boosting Regressions with sklearn

```
from sklearn.ensemble import BaggingRegressor <or any other>
# … load,  split and prepare data (WITHOUT feature normalization)…

  # Train the model
  # in practice one use hundreds or thousands of estimators
clf = BaggingRegressor(
                        n_estimators=500 , base_estimator=SVC(),
                        max_features=10, random_state=0, n_jobs=-1 )
  # the default estimator is DecisionTreeClassifier because it's fast
  # or
clf = RandomForestRegressor(
                        n_estimators=500, max_features=5,
                        max_depth=6, random_state=0, n_jobs=-1 )
  # or
clf = AdaBoostRegressor(
                        n_estimators=500,
                        learning_rate=0.5, random_state=0)
  # or
cls = GradientBoostingRegressor(
                        n_estimators=500,
                        learning_rate=0.5, max_features=None,
                        max_depth=4, random_state=0)

  # Fit the training set
reg.fit( X_train, y_train )

  # Get default evaluation score
r2 = reg.score( X_test, y_test )

  # Get weak estimators
weak_estimators = reg.estimators_

  # Make predictions
weak_estimators[0].predict( X_new )
y_pred = reg.predict( X_new )
```

Boosting Classification with XGBoost

```
from xgboost import XGBClassifier
# … load,  split and prepare data (WITHOUT feature normalization)…

 # Train the model
clf = XGBClassifier( n_estimators = 500, eval_metric='auc', … )
clf.fit( X_train, y_train )

 # Get default evaluation score
accuracy = cls.score( X_test, y_test )

 # Make prediction with the ensemble
y_pred = clf.predict( X_new )                 # return 0 or 1
y_pred = clf.predict_proba(X_new )[:,1]   # return probability of getting 1

 # Get ROC auc and curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred)
rocauc = auc(fpr, tpr)
```

Boosting Regression with XGBoost

```
from sklearn.ensemble import BaggingRegressor
# … load,  split and prepare data (WITHOUT feature normalization)…

 # Train the model
reg = xgb.XGBRegressor(n_estimators=500, max_depth=6, … )
reg.fit( X_train, y_train )

 # Get default evaluation score
r2 = reg.score( X_test, y_test )

 # Make prediction with the ensemble
y_pred = reg.predict( X_new )
```

XGBoost also provides RandomForest implementations for both classification *(XGBRFClassifier)* and regression (*XGBRFRegressor*).

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import SelectFromModel
# … load,  split and prepare data (WITHOUT feature normalization)…

 # Train a basic model
cls = RandomForestClassifier(n_estimators=500, oob_score=True)
clf.fit( X_train, y_train )

 # Select important Features
select = SelectFromModel(cls, prefit=True, threshold=0.0025)

X_train_select = select.transform(X_train)
X_test_select = select.transform(X_test)

 # Train a new model on the selected features
cls = RandomForestClassifier(n_estimators=500, oob_score=True)
clf.fit( X_train_select, y_train )

 # Get default evaluation score
accuracy = cls.score( X_test, y_test )

 # Get weak estimators
weak_estimators = cls.estimators_

 # Make predictions
y_pred_one = cls.estimators_[0].predict( X_new )
y_pred_all = cls.predict( X_new )
```

References & Credits

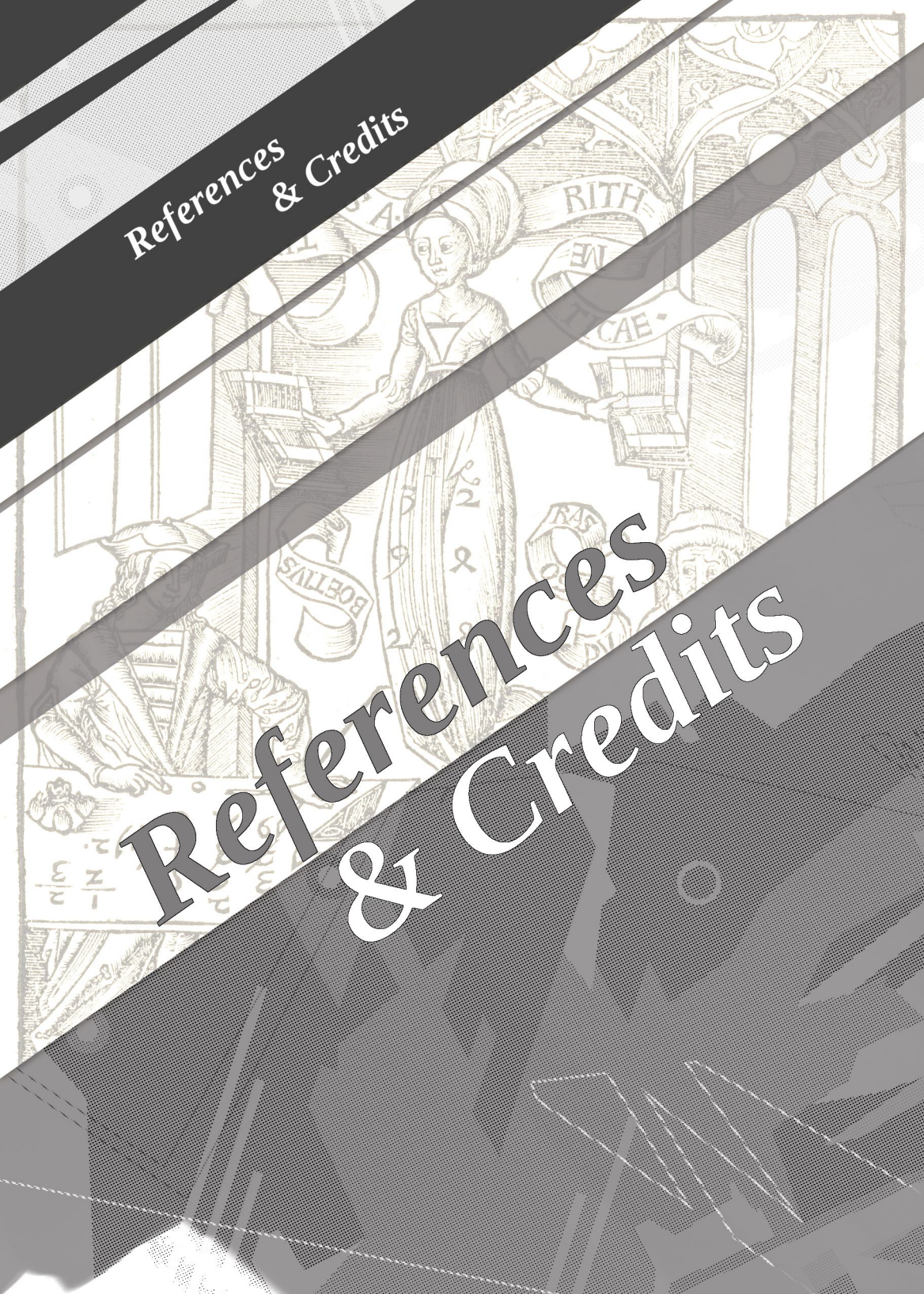


Illustration from Margarita philosophica, 1503, by **Gregor Reisch** (d. 1525). Typ 520.03.736, Houghton Library, Harvard University

Yannis Chaouche - OpenClassrooms.
"Modélisez vos données avec les méthodes ensemblistes."
Short URL. https://bit.ly/3g356ZZ
Web. https://openclassrooms.com/fr/courses/4470521-modelisez-vos-donnees-avec-les-methodes-ensemblistes/

Sruthi E R
"Understanding Random Forest."
Short URL. https://bit.ly/3r4BhhL
Web. https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/

Sharoon Saxena
"A Beginner's Guide to Random Forest Hyperparameter Tuning."
Short URL. https://bit.ly/33ZPCDt
Web. https://www.analyticsvidhya.com/blog/2020/03/beginners-guide-random-forest-hyperparameter-tuning/

scikit-learn.org documentation
"Ensemble methods."
Short URL. https://bit.ly/3L3VxYI
Web. https://scikit-learn.org/stable/modules/ensemble.html

Dr. Hasthika Rupasinghe
"Confidence Intervals: The Bootstrap."
Short URL. https://bit.ly/3GksbSD
Web. https://hasthika.github.io/STT3850/Lecture%20Notes/Ch_5_MSWR_Notes.html