Machine Learning in short

The goal of supervised Machine Learning is to build a prediction function based on historical data. This data has independent (explanatory) variables and a target variable (the variable that you want to predict).

Once a predictive model has been built, we measure its error on a separate testing data set. We do this using KPIs that allow quantifying the error of the model, for example, the Mean Square Error in a regression context (quantitative target variable) or the Accuracy in a classification context (categorical target variable).

The model with the smallest error is generally selected as the best model. Then we use this model to predict the values of the target variable by inputting the explanatory variables.

In this article, I will deep-dive into GridSearch.

Machine Learning's Two Types of Optimization

GridSearch is a tool that is used for **hyperparameter tuning**. As stated before, Machine Learning in practice comes down to comparing different models to each other and trying to find the best working model.

Apart from selecting the right data set, there are generally two aspects of optimizing a predictive model:

1. Optimize the choice of the best model
2. Optimize a model’s fit using hyperparameters tuning

Let’s now look into those to have an explanation for the need for GridSearch.

Part 1. Optimize the choice of the best model

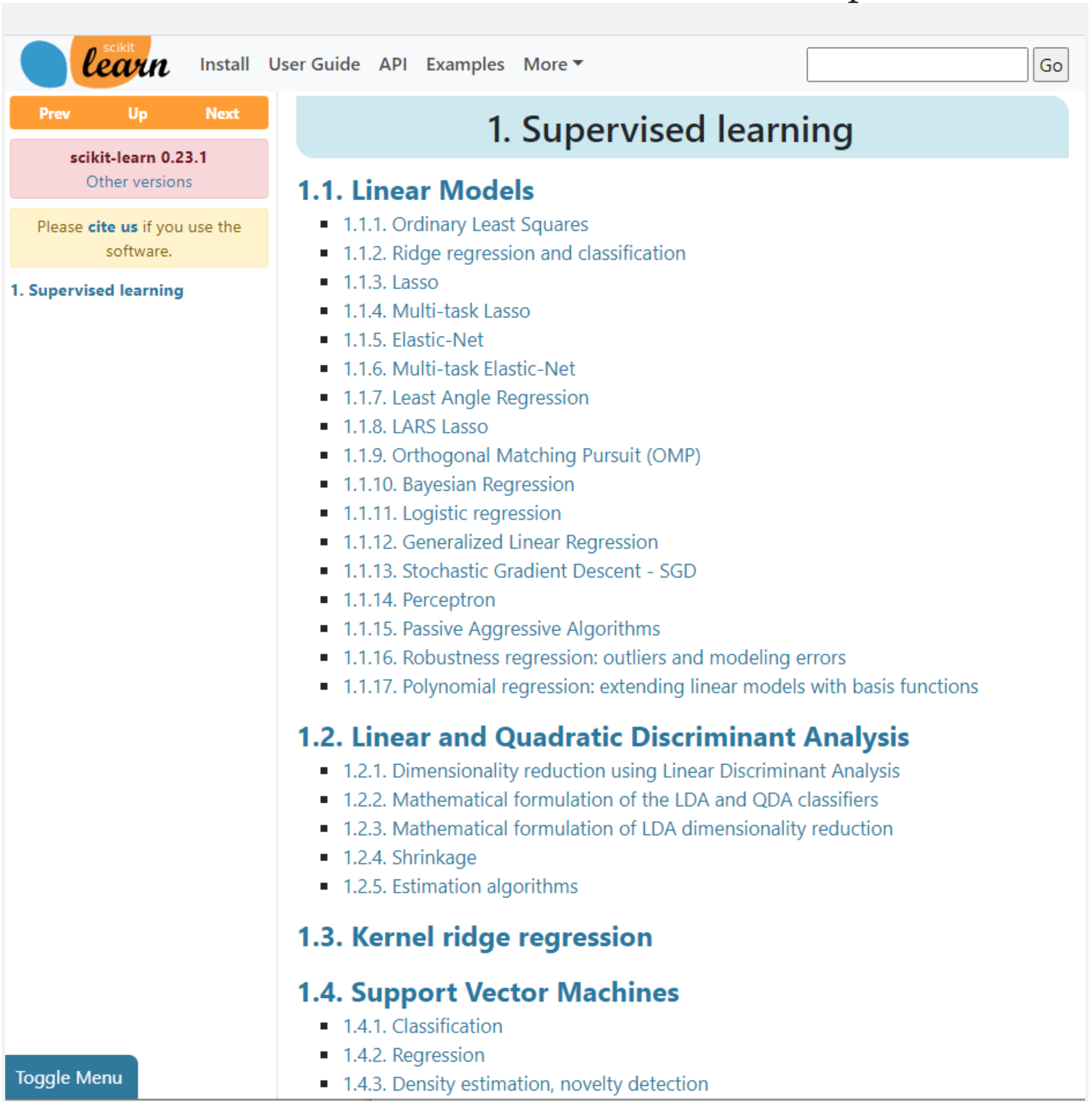
In some datasets, there may exist a simple linear relationship that can predict a target variable from the explanatory variables. In other datasets, these relationships may be more complex or highly nonlinear.

At the same time, many models exist. This ranges from simple models like the Linear Regression, up to very complex models like Deep Neural Networks.

It is key to use a model that is appropriate for our data.

For example, if we use a Linear Regression on a very complex task, the model will not be performant. But if we use a Deep Neural Network on a very simple task, this will also not be performant!

To find a well-fitting Machine Learning model, the solution is to split data into train and test data, then fit many models on the training data and test each of them on the test data. The model that has the smallest error on the test data will be kept.



1. Supervised learning

1.1. Linear Models

- 1.1.1. Ordinary Least Squares
- 1.1.2. Ridge regression and classification
- 1.1.3. Lasso
- 1.1.4. Multi-task Lasso
- 1.1.5. Elastic-Net
- 1.1.6. Multi-task Elastic-Net
- 1.1.7. Least Angle Regression
- 1.1.8. LARS Lasso
- 1.1.9. Orthogonal Matching Pursuit (OMP)
- 1.1.10. Bayesian Regression
- 1.1.11. Logistic regression
- 1.1.12. Generalized Linear Regression
- 1.1.13. Stochastic Gradient Descent - SGD
- 1.1.14. Perceptron
- 1.1.15. Passive Aggressive Algorithms
- 1.1.16. Robustness regression: outliers and modeling errors
- 1.1.17. Polynomial regression: extending linear models with basis functions

1.2. Linear and Quadratic Discriminant Analysis

- 1.2.1. Dimensionality reduction using Linear Discriminant Analysis
- 1.2.2. Mathematical formulation of the LDA and QDA classifiers
- 1.2.3. Mathematical formulation of LDA dimensionality reduction
- 1.2.4. Shrinkage
- 1.2.5. Estimation algorithms

1.3. Kernel ridge regression

1.4. Support Vector Machines

- 1.4.1. Classification
- 1.4.2. Regression
- 1.4.3. Density estimation, novelty detection

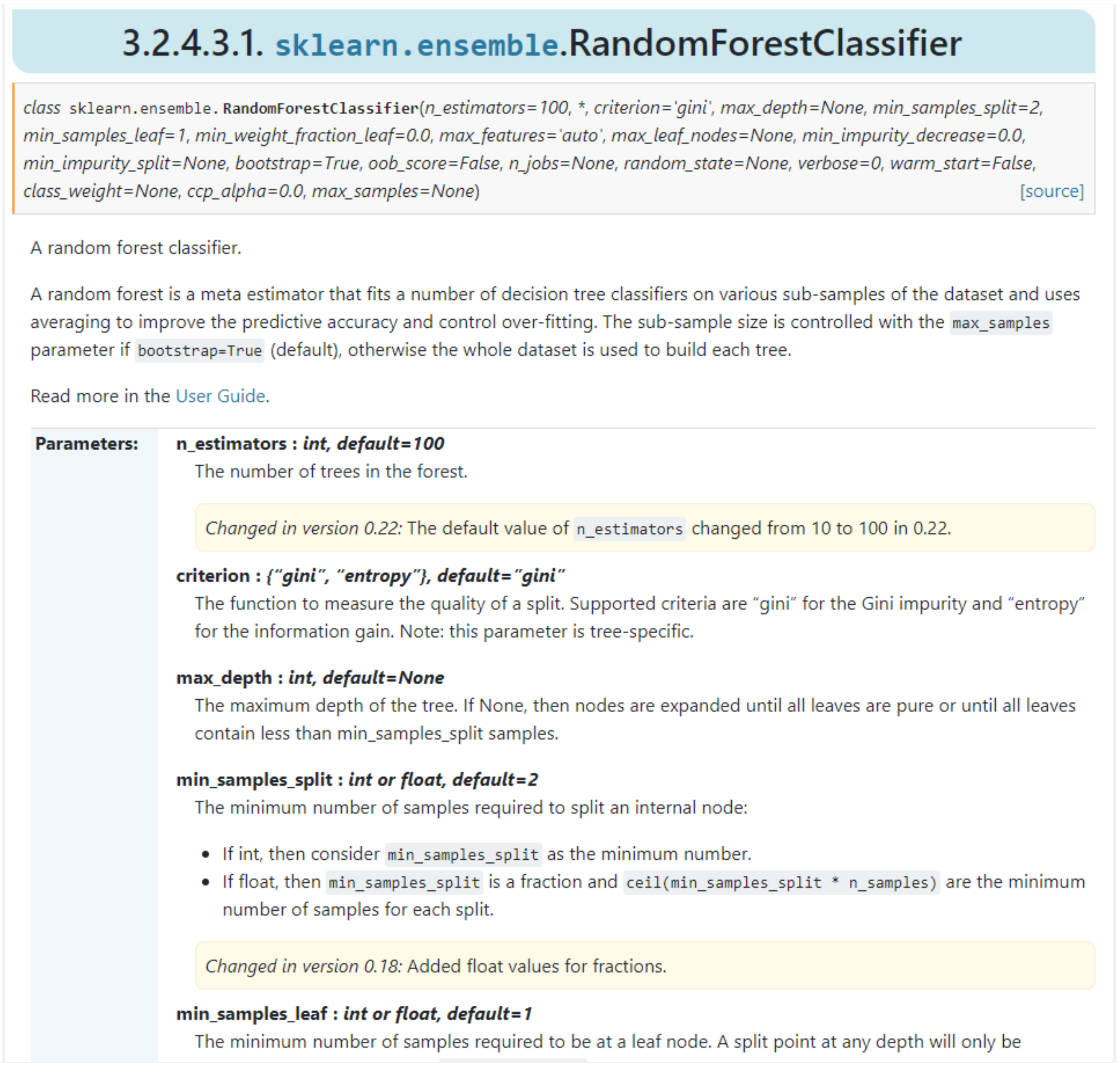
A screenshot from Scikit Learn’s list of supervised models shows that there are a lot of models to try out!

Part 2. Optimize a model’s fit using hyperparameters tuning

After choosing one well-performing model (or a few), the second thing to optimize is the hyperparameters of a model. Hyperparameters are like a configuration of the training phase of the model. They influence what a model can or cannot learn.

Tuning hyperparameters can, therefore, lower the error on the test data set even more.

The way of estimating is different for each model, and thus each model has its own hyperparameters to optimize.



3.2.4.3.1. sklearn.ensemble.RandomForestClassifier

class sklearn.ensemble.**RandomForestClassifier**(*n_estimators=100, *, criterion='gini', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None, random_state=None, verbose=0, warm_start=False, class_weight=None, ccp_alpha=0.0, max_samples=None*) [source]

A random forest classifier.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is controlled with the `max_samples` parameter if `bootstrap=True` (default), otherwise the whole dataset is used to build each tree.

Read more in the User Guide.

Parameters:

n_estimators : *int, default=100*
The number of trees in the forest.

Changed in version 0.22: The default value of `n_estimators` changed from 10 to 100 in 0.22.

criterion : *{"gini", "entropy"}, default="gini"*
The function to measure the quality of a split. Supported criteria are "gini" for the Gini impurity and "entropy" for the information gain. Note: this parameter is tree-specific.

max_depth : *int, default=None*
The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split : *int or float, default=2*
The minimum number of samples required to split an internal node:

- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.

Changed in version 0.18: Added float values for fractions.

min_samples_leaf : *int or float, default=1*
The minimum number of samples required to be at a leaf node. A split point at any depth will only be

This extract of the documentation of Scikit Learn's RandomForestClassifier shows numerous parameters that can all influence the final accuracy of your model.

One way to do a thorough search for the best hyperparameters is to use a tool called GridSearch.

What is GridSearch?

GridSearch is an optimization tool that we use when tuning hyperparameters. We define the grid of parameters that we want to search through, and we select the best combination of parameters for our data.

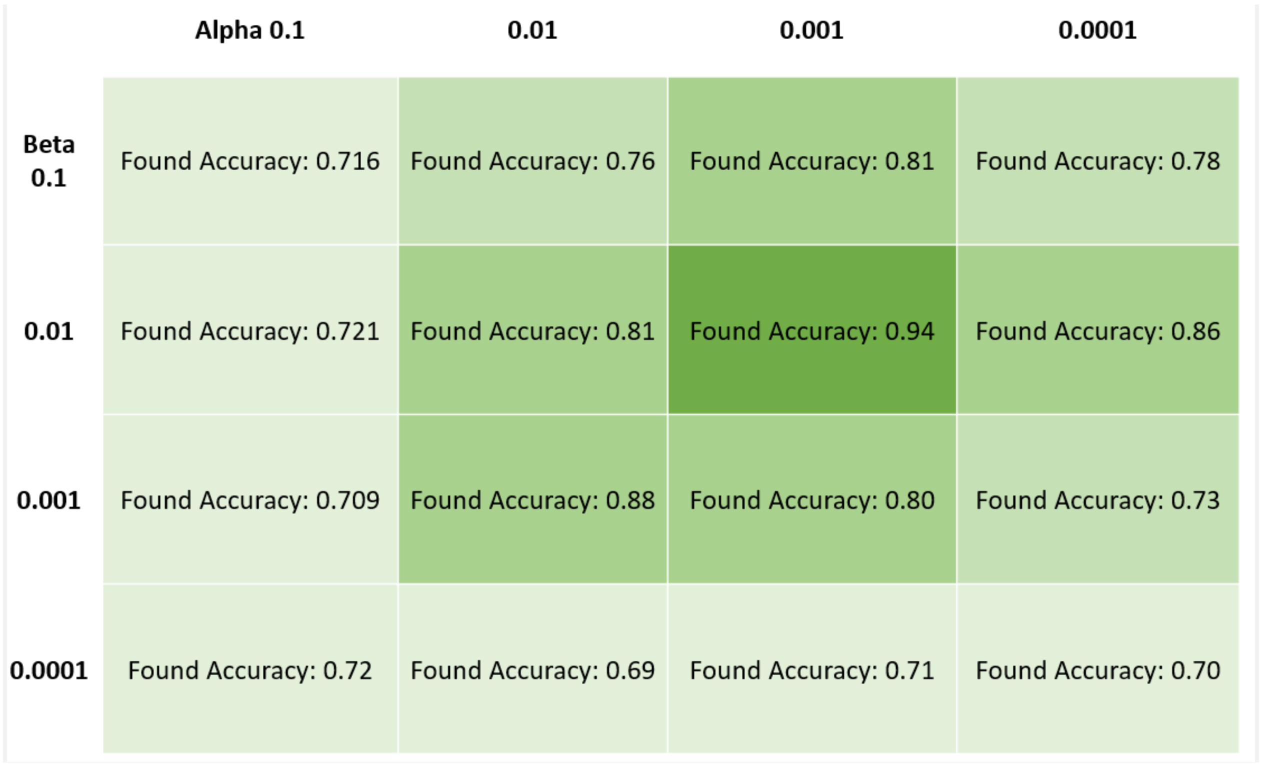
The “Search” in GridSearch

The hypothesis is that there is a specific combination of values of the different hyperparameters that will minimize the error of our predictive model. Our goal using GridSearch is to find this specific combination of parameters.

The “Grid” in GridSearch

GridSearch’s idea for finding this best parameter combination is is simple: just test each parameter combination possible and select the best one!

Not really each combination possible though, since for a continuous scale there would be infinitely many combinations to test. The solution for this is to define a Grid. This Grid defines for each hyperparameter, which values should be tested.



	Alpha 0.1	0.01	0.001	0.0001
Beta 0.1	Found Accuracy: 0.716	Found Accuracy: 0.76	Found Accuracy: 0.81	Found Accuracy: 0.78
0.01	Found Accuracy: 0.721	Found Accuracy: 0.81	Found Accuracy: 0.94	Found Accuracy: 0.86
0.001	Found Accuracy: 0.709	Found Accuracy: 0.88	Found Accuracy: 0.80	Found Accuracy: 0.73
0.0001	Found Accuracy: 0.72	Found Accuracy: 0.69	Found Accuracy: 0.71	Found Accuracy: 0.70

A schematic overview of GridSearch on two hyperparameters Alpha and Beta (graphics by author)

In an example case where two hyperparameters — Alpha and Beta— are tuned: we could give both of them the values [0.1, 0.01, 0.001, 0.0001] resulting in the following “Grid” of values. At each crossing point, our GridSearch will fit the model to see what the error at this point is.

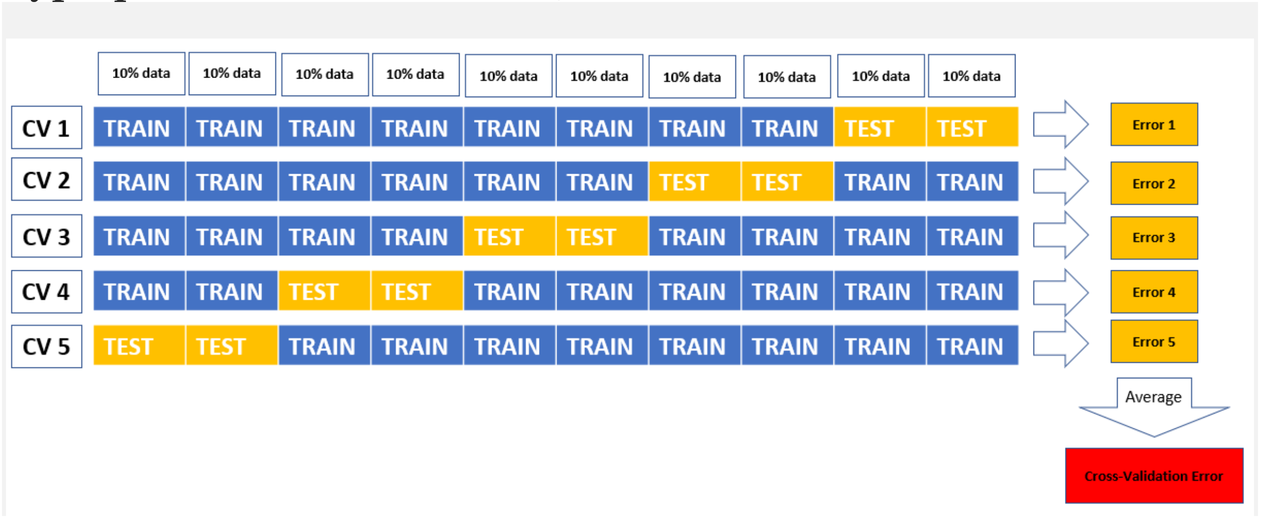
And after checking all the grid points, we know which parameter combination is best for our prediction.

The “Cross-Validation” in GridSearch

At this point, only one thing remains to be added: the **Cross-Validation Error.**

When testing the performance of a model with each combination of hyperparameters, there could be a risk of overfitting. This means that just by pure chance, only the training data set corresponded well to this particular hyperparameter combination! The performance on new, real-life data, could be much worse!

To get a more reliable estimate of the performances of a hyperparameter combination, we take the Cross Validation Error.



A schematic overview of Cross-Validation (graphics by author)

In Cross-Validation, the data is split in multiple parts. For example 5 parts. Then the model is fit 5 times while leaving out one-fifth of the data. This one-fifth left-out data is used to measure the performances.

For one combination of hyperparameter values, the average of the 5 errors constitutes the cross-validation error. This makes the selection of the final combination more reliable.

What makes GridSearch so important?

GridSearch allows us to find the best model given a data set very easily. It actually makes the Machine Learning part of the Data Scientists role much easier by automating the search.

On the Machine Learning side, some things that still remain to be done is deciding on the right way to measure error, deciding on which models to try out and which hyperparameters to test for. And the most important part, the work on data preparation, is also left for the data scientist.

Thanks to the GridSearch approach, the Data Scientist can focus on the data wrangling work, while automating repetitive tasks of model comparison. This makes the work more interesting and allows the Data Scientist to add value where he’s most needed: working with data.

A number of alternatives for GridSearch exist, including Random Search, Bayesian Optimization, Genetic Algorithms, and more. I will write an article about those soon, so don’t hesitate to stay tuned. Thanks for reading!