



# MODULE 09 - Piscine Python for Data Science

## Machine Learning: Advanced

*Summary: This day will help you with advanced tasks of machine learning in Python.*

# Contents

<b>I</b>	<b>Foreword</b>	<b>2</b>
<b>II</b>	<b>Instructions</b>	<b>3</b>
<b>III</b>	<b>Specific instructions of the day</b>	<b>4</b>
<b>IV</b>	<b>Exercice 00 : Regularization</b>	<b>5</b>
<b>V</b>	<b>Exercice 01 : Decision boundaries</b>	<b>8</b>
<b>VI</b>	<b>Exercice 02 : Metrics</b>	<b>10</b>
<b>VII</b>	<b>Exercice 03 : Ensembles</b>	<b>13</b>
<b>VIII</b>	<b>Exercice 04 : Pipelines and OOP</b>	<b>16</b>

# Chapter I

## Foreword

There is real data science and data science for competitions. The difference is almost as between porter and heavy lifter. A porter can be bad at heavy lifting competitions and a heavy-lifter can underperform on a real job. The same thing is with data science: in companies, you should have a broader skill set including, for example, soft skills, you need to understand the business, you need to be focused on profit or other organization's goals. But in competitions, you need to be really good at achieving the competition target metrics. You may create a super heavy machine learning model that can be 0.00001 better than the model of your next competitor and nobody cares how long it takes to make predictions, how much interpretable the model is, and how many computational resources it requires. In business, of course, all these criteria matter.

Nevertheless, you may use competitions for improving your skills. The most popular platform is [Kaggle](#). There are lots of public datasets and competitions. You may even try to win a prize, but most likely you will earn only the experience which is not bad at all. It can be good for building your portfolio. You may organize a team and learn from each other. You can improve your collaboration skills. The platform allows you to pursue the machine learning track as well as the data exploratory track: whatever you find more attractive to you.

So think about it as a great tool to continue growing in the field, but keep in mind that there are lots of things that needed in real life besides building machine learning models and hyperparameters optimization.

# Chapter II

## Instructions

- Use this page as the only reference. Do not listen to any rumors and speculations about how to prepare your solution.
- Here and further we use Python 3 as the only correct version of Python.
- The python files for python exercises (module01, module02, module03) must have a block in the end: `if __name__ == '__main__':`
- Pay attention to the permissions of your files and directories.
- To be assessed your solution must be in your GIT repository.
- Your solutions will be evaluated by your piscine mates.
- You should not leave in your directory any other file than those explicitly specified by the exercise instructions. It is recommended that you modify your `.gitignore` to avoid accidents.
- When you need to get precise output in your programs, it is forbidden to display a precalculated output instead of performing the exercise correctly.
- Have a question? Ask your neighbor on the right. Otherwise, try with your neighbor on the left.
- Your reference manual: mates / Internet / Google.
- Remember to discuss on the Intra Piscine forum.
- Read the examples carefully. They may require things that are not otherwise specified in the subject.
- And may the Force be with you!


# Chapter III

## Specific instructions of the day

- Use Jupyter Notebook to work with your code
- For each major subtask in the list of any exercise (black bullets), your ipynb file should have an h2 heading to help your peer easily navigate in your code
- No imports allowed, except those explicitly mentioned in the section “Authorized functions” of the title block of each exercise
- You can use any built-in function, if it is not prohibited in the exercise
- Save and load all the required data in the subfolder data/
- scikit-learn (0.23.1) is the library that you need for all the machine learning tasks.
- tqdm (4.46.1) is the library that you need for tracking the progress

# Chapter IV

## Exercise 00 : Regularization

	Exercise 00
Regularization	
Turn-in directory : <i>ex00/</i>	
Files to turn in : <code>00_regularization.ipynb</code>	
Allowed functions : <b>no restrictions</b>	

In the previous day, you tried to solve different kinds of machine learning problems using different algorithms understanding the basics. During this day you will try using more advanced techniques.

Let us start with regularization. Regularization in a broader meaning is a technique that prevents a model from overfitting. In the case of logistic regression, we have a formula with different  $X$ s (features) and coefficients. Regularization penalizes too big coefficients making the formula more robust and more ready for the unknown data that will have to work with in the future. L1 regularization makes some coefficients equal to zero. So this mode may be helpful for feature selection if there are lots of them and you need to reduce the number. L2 regularization does not make the weights equal to zero but may make them smaller. We cannot say: use only L2 regularization or L1. In the field of machine learning, there are not many things that are silver bullets. Usually, you need to try many different things on your dataset to find what suits it better.

If we talk about trees and forests, regularization is connected to the parameters that affect the number of cases in the leaves. If your tree is so deep that each leaf includes only one sample, the chances are that your tree has overfitted to the training dataset. To prevent it you may play with such parameters as `max_depth`, `min_samples_split`, `min_samples_leaf`, `max_leaf_nodes`, etc.

Many different algorithms have many different parameters of regularization. Our goal is not to cover them all. You should just know that they exist and in case of need, you will comprehend how they work for the specific algorithm.

In this exercise, you will play with some of them. The dataset will be the same and the task is still to predict the weekday for each commit having data: `uid`, `labname`, `numTrials`, `hour of the commit`.

What you need to do:

- data preparation

- read the file `dayofweek.csv` that you used in the previous day to a dataframe
- using `train_test_split` with parameters `test_size=0.2`, `random_state=21` get `X_train`, `y_train`, `X_test`, `y_test`
- use the additional parameter `stratify`
- logreg regularization
  - train a baseline model with the only parameters `random_state=21`, `fit_intercept=False`
  - use stratified K-fold cross-validation with 10 splits to evaluate the accuracy of the model
  - the result of the code where you trained and evaluated the baseline model should be exactly like this (use `%%time` to get the info about how long it took to run the cell):

```
train - 0.62902 | valid - 0.59259
train - 0.64633 | valid - 0.62963
train - 0.63479 | valid - 0.56296
train - 0.65622 | valid - 0.61481
train - 0.63397 | valid - 0.57778
train - 0.64056 | valid - 0.59259
train - 0.64138 | valid - 0.65926
train - 0.65952 | valid - 0.56296
train - 0.64333 | valid - 0.59701
train - 0.63674 | valid - 0.62687
Average accuracy on crossval is 0.60165
Std is 0.02943
CPU times: user 1.87 s, sys: 228 ms, total: 2.1 s
Wall time: 536 ms
```


- in the new cells try different values of penalty: `none`, `l1`, `l2` – you can change the values of solver too
- in the new cells try different values of the parameter `C`
- SVM regularization
  - train a baseline model with the only parameters `probability=True`, `kernel='linear'`, `random_state=21`
  - use stratified K-fold cross-validation with 10 splits to evaluate the accuracy of the model
  - the format of the result of the code where you trained and evaluated the baseline model should be similar to what you have got for the logreg
  - in the new cells try different values of the parameter `C`
- decision tree regularization
  - train a baseline model with the only parameter `max_depth=10` and `random_state=21`
  - use stratified K-fold cross-validation with 10 splits to evaluate the accuracy of the model

- the format of the result of the code where you trained and evaluated the baseline model should be similar to what you have got for the logreg
- in the new cells try different values of the parameter `max_depth`
- as a bonus play with other regularization parameters trying to find the best combination
- random forest regularization
  - train a baseline model with the only parameters `n_estimators=50`, `max_depth=14`, `random_state=21`
  - use stratified K-fold cross-validation with 10 splits to evaluate the accuracy of the model
  - the format of the result of the code where you trained and evaluated the baseline model should be similar to what you have got for the logreg
  - in the new cells try different values of the parameter `max_depth` and `n_estimators=50`
  - as a bonus play with other regularization parameters trying to find the best combination
- predictions
  - choose the best model and use it to make predictions for the test dataset
  - calculate the final accuracy
  - analyze: for which weekday your model makes the most errors (in % of the total number of samples of that class in your test dataset)
  - save the model



# Chapter V

## Exercice 01 : Decision boundaries

	Exercice 01
Decision boundaries	
Turn-in directory : <i>ex01/</i>	
Files to turn in : <code>01_binary_decision_boundaries.ipynb</code>	
Allowed functions : <b>no restrictions</b>	

We are sure that you got tired of iterating through different parameters of different models manually. You probably think that there should be a way to automate it. Yes, it exists – GridSearch.

You may specify the range of values for the parameters that you want to optimize and put it to GridSearchCV. It will try all of them, calculate the metrics on cross-validation, and give you the best combination of the parameters as well as the whole results of its mini-research. It is cool, right?


What you need to do:

- data preparation
  - create a dataframe similar to the one from the previous exercise but this time do not scale continuous features (we are not going to use logreg anymore)
  - using `train_test_split` with parameters `test_size=0.2`, `random_state=21` get `X_train`, `y_train`, `X_test`, `y_test`
  - use the additional parameter `stratify`
- SVM gridsearch
  - using GridSearchCV try different parameters of kernel (linear, rbf, sigmoid), C (0.01, 0.1, 1, 1.5, 5, 10), gamma (scale, auto), class\_weight (balanced, None) use `random_state=21` and `probability=True` and get the best combination of them in terms of accuracy
  - create a dataframe from the results of the gridsearch and sort it ascendingly by the `rank_test_score`, check if there is a huge difference between different combinations (sometimes a simpler model may give a comparable result)

- decision tree gridsearch
  - using GridSearchCV try different parameters of max\_depth (from 1 to 49), class\_weight (balanced, None) and criterion (entropy and gini) and get the best combination of them in terms of accuracy
  - use random\_state=21
  - create a dataframe from the results of the gridsearch and sort it ascendingly by the rank\_test\_score, check if there is a huge difference between different combinations (sometimes a simpler model may give a comparable result)
- random forest gridsearch
  - using GridSearchCV try different parameters of n\_estimators (5, 10, 50, 100), max\_depth (from 1 to 49), class\_weight (balanced, None) and criterion (entropy and gini) and get the best combination of them in terms of accuracy
  - use random\_state=21
  - create a dataframe from the results of the gridsearch and sort it ascendingly by the rank\_test\_score, check if there is a huge difference between different combinations (sometimes a simpler model may give a comparable result)
- progress bar
  - gridsearch can be a quite long process and you may find yourself wondering when it will end
  - create a manual gridsearch for the same parameters values of random forest iterating through the list of the possible values and calculating cross\_val\_score for each combination
  - try to increase n\_jobs
  - the value cv for cross\_val\_score is 5
  - track the progress using the library tqdm.notebook
  - create a dataframe from the results of the gridsearch with the columns corresponding to the names of the parameters and mean\_accuracy and std\_accuracy
  - sort it descendingly by the mean\_accuracy, check if there is a huge difference between different combinations (sometimes a simpler model may give a comparable result)
- predictions
  - choose the best model and use it to make predictions for the test dataset
  - calculate the final accuracy

# Chapter VI

## Exercise 02 : Metrics

	Exercise 02
Metrics	
Turn-in directory : <i>ex02/</i>	
Files to turn in : <i>02_metrics.ipynb</i>	
Allowed functions : <b>no restrictions</b>	

Accuracy 90% is a good result or not? Actually, it is not easy to say. Imagine the situation when you have two classes that are unbalanced: 95% of the samples belong to the first class and 5% – to the second. The accuracy will be worse than a naive classifier when we make predictions using the most popular class. And it is not a fantasy. This case is quite popular for anti-fraud tasks, for example. The number of fraud cases is significantly lower than the number of normal cases. Is it a bad metric? What we are trying to say is that it is simple to understand and you can use it to compare different models within a task, but this metric can be misleading when you need to compare the results to a model from another task. Also, it does not say much about the errors. You just know how many of them there are. But what kind of?

There are some other metrics that can answer this question. They all come from the [confusion matrix](#). The first is precision. It is the number of correctly predicted samples of one class divided by the number of predictions of that class. Imagine that we predicted 10 days as a weekend, but only 7 of them were really weekends. The precision is 0.7.

The second is recall. It is the number of correctly predicted samples of one class divided by the true number of that class. Imagine that we again predicted 10 days as a weekend, only 7 of them were really weekends and in the dataset there were 20 weekends. The recall is 0.35 (7/20).

Precision can be good when we want to show an ad that includes some 16+ content. We want to be precise in our prediction. Recall can be good for identifying terrorists. We may want to find them all no matter how many civil people experience inconvenience because of it. Also, there is a metric that combines both of them in the harmonic mean – F1 score. You can use it when you need to optimize both of them.

Also, there is the [ROC-curve](#). When you make predictions, usually you have probabilities. And the final classification is made by comparing them to the threshold. For example, if we see that for that given day the probability of being a weekend is 0.2 and

the threshold is 0.5, we can say that it is not a weekend. But if you change the threshold to 0.1, the same sample will get the prediction “weekend”. Imagine now, that for each threshold we calculate recall and also the number of how many working days we predicted as weekends divided by the real number of working days. Both those values we can put on a plot and we will get the ROC-curve. The higher it is, the better. Comparing curves can be not that convenient. That is why we can use another metric – AUC (area under the curve). Precision, recall, AUC are good when we want to compare the performance of different models from different tasks. And they tell us something about the errors. Everything that we told you here was connected to binary classification. But it can be used with some adjustments for multiclass and multilabel classification. You can read it [here](#), for example.

What you need to do:

- data preparation
  - create the same dataframe as in the previous exercise
  - using `train_test_split` with parameters `test_size=0.2`, `random_state=21` get `X_train`, `y_train`, `X_test`, `y_test`
  - use the additional parameter `stratify`
- SVM
  - take the best parameters from the previous exercise and train the model
  - your code from the cell should calculate accuracy, precision, recall, ROC AUC
  - precision and recall should be calculated for each class (use `average='weighted'`)
  - ROC AUC should be calculated for each class against any other class (all possible pairwise combinations) and then weighted average should be applied for the final metric
  - the code in the cell should display the result as below:


```
accuracy is 0.88757
precision is 0.89267
recall is 0.88757
roc_auc is 0.97878
```

- Decision tree
  - take the best parameters from the previous exercise and train the model
  - your code from the cell should calculate accuracy, precision, recall, ROC AUC
  - precision and recall should be calculated for each class (use `average='weighted'`)
  - ROC AUC should be calculated for each class against any other class (all possible pairwise combinations) and then weighted average should be applied for the final metric
  - the code in the cell should display the result as in the previous example
- Random forest

- take the best parameters from the previous exercise and train the model
- your code from the cell should calculate accuracy, precision, recall, ROC AUC
- precision and recall should be calculated for each class (use average='weighted')
- ROC AUC should be calculated for each class against any other class (all possible pairwise combinations) and then weighted average should be applied for the final metric
- the code in the cell should display the result as in the previous example
- predictions
  - choose the best model
  - analyze: for which weekday your model makes the most errors (in % of the total number of samples of that class in your full dataset), for which labname and for which users
  - save the model
- write a function that takes a list of different models and a corresponding list of parameters (dicts) and returns a dict that contains all the 4 metrics for each model

# Chapter VII

## Exercice 03 : Ensembles

	Exercise 03
Ensembles	
Turn-in directory : <i>ex03/</i>	
Files to turn in : <i>03_ensembles.ipynb</i>	
Allowed functions : <b>no restrictions</b>	

You already know that the random forest is an ensemble of many different trees. But actually you can create an ensemble from any type of model. In this exercise, you will try several approaches: voting classifier, bagging classifier, and stacking classifier. Who knows maybe it will help you increase the quality of your predictions.

What you need to do:

- data preparation
  - create the same dataframe as in the previous exercise
  - using `train_test_split` with parameters `test_size=0.2`, `random_state=21` get `X_train`, `y_train`, `X_test`, `y_test` and then get `X_train`, `y_train`, `X_valid`, `y_valid` from the previous `X_train`, `y_train`
  - use the additional parameter `stratify`
- individual classifiers
  - train SVM, decision tree and random forest again with the best parameters that you got from one of the previous exercises, `random_state=21` for all of them
  - evaluate accuracy, precision, and recall for them on the validation set
  - the result of each cell of the section should look like this:

```
accuracy is 0.87778
precision is 0.88162
recall is 0.87778
```


- voting classifier
  - using VotingClassifier and the three models that you have just trained, calculate the accuracy, precision, and recall on the validation set
    - \* try hard voting
    - \* try soft voting
    - \* try soft voting with different weights of the models
    - \* try hard voting with different weights of the models
  - optimize the weights and the voting mode
    - \* create the list of lists of different weight values using list comprehension, where each individual weight is an integer in the range between 1 and 5: [1, 1, 1], [1, 1, 2], ...
    - \* use tqdm.notebook for tracking the progress of the process
  - calculate the accuracy, precision and recall on the test set for the model with the best weights in terms of accuracy (if there are several of them with equal values, choose the one with the higher precision)
- bagging classifier
  - using BaggingClassifier and SVM with the best parameters create an ensemble, try different values of the n\_estimators, use random\_state=21
  - find the best n\_estimators trying the following values [5, 10, 50, 100]
  - use GridSearchCV with cv=5 and verbose=2 for tracking the progress
  - calculate the accuracy, precision, and recall for the model with the best parameters (in terms of accuracy) on the test set (if there are several of them with equal values, choose the one with the higher precision)
- stacking classifier
  - to achieve reproducibility in this case you will have to create an object of cross-validation generator: StratifiedKFold(n\_splits=n, shuffle=True, random\_state=21), where n you will try to optimize (the details are below)
  - using StackingClassifier and the three models that you have recently trained, calculate the accuracy, precision and recall on the validation set, try different values of n\_splits [2, 3, 4, 5, 6, 7] in the cross-validation generator and parameter passthrough in the classifier itself
  - calculate the accuracy, precision, and recall for the model with the best parameters (in terms of accuracy) on the test set (if there are several of them with equal values, choose the one with the higher precision)
- predictions
  - choose the best model in terms of accuracy (if there are several of them with equal values, choose the one with the higher precision)

- analyze: for which weekday your model makes the most errors (in % of the total number of samples of that class in your full dataset), for which labname and for which users
- save the model



# Chapter VIII

## Exercise 04 : Pipelines and OOP

	Exercise 04
Pipelines and OOP	
Turn-in directory : <i>ex04/</i>	
Files to turn in : <i>04_pipelines.ipynb</i>	
Allowed functions : <b>no restrictions</b>	

Trying to solve the problem you made a lot of different actions: prepared the data, tried different models, tried different metrics, optimized their hyperparameters, tried different kinds of ensembles. Now it probably looks a bit chaotic: your code in different notebooks that require a lot of scrolling. In this exercise, that will be the final of the day, you will make it look a bit cleaner, more organized. Why can it be important? In real life you may want to share it with your colleagues, you may want to make it a part of your portfolio or you may look after yourself in the future. The chances are that if you get back to that code in several months, you will think: who made that mess?

In this exercise, you will try to apply the OOP approach to data analysis. The first part of your notebook will contain only the imports, classes and methods. The second part will be your “main program”. You will work with the initial data and you will go through most of the steps that you made before.

What you need to do:

- data preparation pipeline
  - create three custom transformers, the first two out of which will be used within a [Pipeline](#)
  - FeatureExtractor() class
    - \* takes a dataframe with uid, labname, numTrials, timestamp from the checker table of the database without admins and checking statuses
    - \* extracts hour from timestamp
    - \* extracts weekday from timestamp
    - \* drops the timestamp column

- \* returns the new dataframe
- `MyOneHotEncoder()` class
  - \* takes the dataframe from the result of the previous transformation and the name of the target column
  - \* identifies all the categorical features and transforms them with `OneHotEncoder()`
    - if the target column is categorical too, then the transformation should not apply to it
  - \* drops the initial categorical features
  - \* returns the dataframe with the features and the series with the target column
- `TrainValidationTest()` class
  - \* takes X and y
  - \* returns `X_train`, `X_valid`, `X_test`, `y_train`, `y_valid`, `y_test`
  - \* `test_size=0.2`, `random_state=21`, `stratified`
- model selection
  - `ModelSelection()` class
    - \* takes a list of `GridSearchCV` instances and a dict where the keys are the indexes from that list and the values are the names of the models, the example is below in the reverse order
      - `ModelSelection(grids, grid_dict)`
      - `grids = [gs_svm, gs_tree, gs_rf]`
      - `gs_svm = GridSearchCV(estimator=svm, param_grid=svm_params, scoring='accuracy', cv=2, n_jobs=jobs)`, where jobs you can specify by yourself
      - `svm_params = {'kernel':('linear', 'rbf', 'sigmoid'), 'C':[0.01, 0.1, 1, 1.5, 5, 10], 'gamma': ['scale', 'auto'], 'class_weight':('balanced', None), 'random_state':[21], 'probability':[True]}`
    - \* method `choose()` takes `X_train`, `y_train`, `X_valid`, `y_valid` and returns the name of the best classifier among all the models on the validation set
    - \* method `best_results()` returns a dataframe with the columns `model`, `params`, `valid_score` where the rows are the best models within each class of models

```

model  params  valid_score
0   SVM   {'C': 10, 'class_weight': None, 'gamma': 'auto... 0.772727
1  Decision Tree  {'class_weight': 'balanced', 'criterion': 'gin... 0.801484
2  Random Forest  {'class_weight': None, 'criterion': 'entropy',... 0.855288

```

- \* when you iterate through the parameters of a model class, print the name of that class and show the progress using `tqdm.notebook`, in the end of the cycle print the best model of that class

```

Estimator: SVM
100%
125/125 [01:32<00:00, 1.36it/s]
Best params: {'C': 10, 'class_weight': None, 'gamma': 'auto', 'kernel': 'rbf', 'probability': True, 'random_state': 21}
Best training accuracy: 0.773
Validation set accuracy score for best params: 0.878

Estimator: Decision Tree
100%
57/57 [01:07<00:00, 1.22it/s]
Best params: {'class_weight': 'balanced', 'criterion': 'gini', 'max_depth': 21, 'random_state': 21}
Best training accuracy: 0.801
Validation set accuracy score for best params: 0.867

Estimator: Random Forest
100%
284/284 [06:47<00:00, 1.13s/it]
Best params: {'class_weight': None, 'criterion': 'entropy', 'max_depth': 22, 'n_estimators': 50, 'random_state': 21}
Best training accuracy: 0.855
Validation set accuracy score for best params: 0.907

Classifier with best validation set accuracy: Random Forest

```

- ensemble selection
  - BestEnsemble() class
    - \* takes a list of estimators where each element is a tuple like this ('svm', model\_1), parameters for the voting classifier (voting and its possible values and weights and their possible values) and parameters for stacking classifier (passthrough and its possible values, cv and its possible values, final\_estimator)
    - \* methods choose() takes X\_train, y\_train, X\_valid, y\_valid and returns the name of the best classifier among all the ensembles on the validation set
    - \* method best\_results() returns a dataframe with the columns ensemble, params, valid\_score where the rows are the best ensembles within each class of ensembles (voting or stacking)
    - \* when you iterate through the parameters of an ensemble class, print the name of that ensemble type and show the progress using tqdm.notebook, at the end of the cycle print the best model of that ensemble
- finalization
  - Finalize() class
    - \* takes an estimator
    - \* method final\_score() takes X\_train, y\_train, X\_test, y\_test and returns the accuracy of the model as in the example below:
 

```

final.final_score(X_train, y_train, X_test, y_test)
Accuracy of the final model is 0.908284023668639
              
```
    - \* method save\_model() takes a path, saves the model to this path and prints that the model was successfully saved

- main program
  - load the data from the checker table of the database
  - create the pipeline preprocessing that consists of two custom transformers: `FeatureExtractor()` and `MyOneHotEncoder()`
    - \* `preprocessing = Pipeline([('feature_extractor', FeatureExtractor()), ('one-hot_encoder', MyOneHotEncoder('dayofweek'))])`
  - use that pipeline and its method `fit_transform()` on the initial dataset
    - \* `data = preprocessing.fit_transform(df)`
  - get `X_train`, `X_valid`, `X_test`, `y_train`, `y_valid`, `y_test` using `TrainValidationTest()` and the result of the pipeline
  - create an instance of `ModelSelection()`, use the method `choose()` applying it to the models that you want and parameters that you want, get the dataframe of the best results
  - create an instance of `BestEnsemble()`, use the method `choose()` applying it to the models that you want and parameters that you want, get the dataframe of the best results
  - create an instance of `Finalize()` with your best model, use method `final_score()` and save the model in the format: `name_of_the_ensemble_accuracy` on test dataset.sav
  - that is it, congrats!