Thibault DUHAMEL / Heng SHI

IFT 712

SESSION PROJECT

https://github.com/ThibaultDuhamel/ift712/tree/project

https://trello.com/b/JeEm1SOX

Introduction

The purpose of this project is to master key notions and algorithms presented in the IFT712 course, and thereby evaluate common classification methods. In order to do so, we have implemented 6 different classification algorithms on a dataset made of 192 features from 99 types of leafs.

First of all, we need to collect data and construct a data structure out of files downloaded from Kaggle that our models will be able to use. At this point, we load raw data and directly use it, but we will apply transformations in the future.

Then, the second and main part is to implement, with the help of sklearn for python, 6 classifiers that are K-Nearest Neighbors, Ridge, Support Vector Machines, Multi-Layer Perceptron, AdaBoost with Decision Trees, and Random Forest.

After that, data transformations will be performed on the best model to assess how they impact learning and generalization.

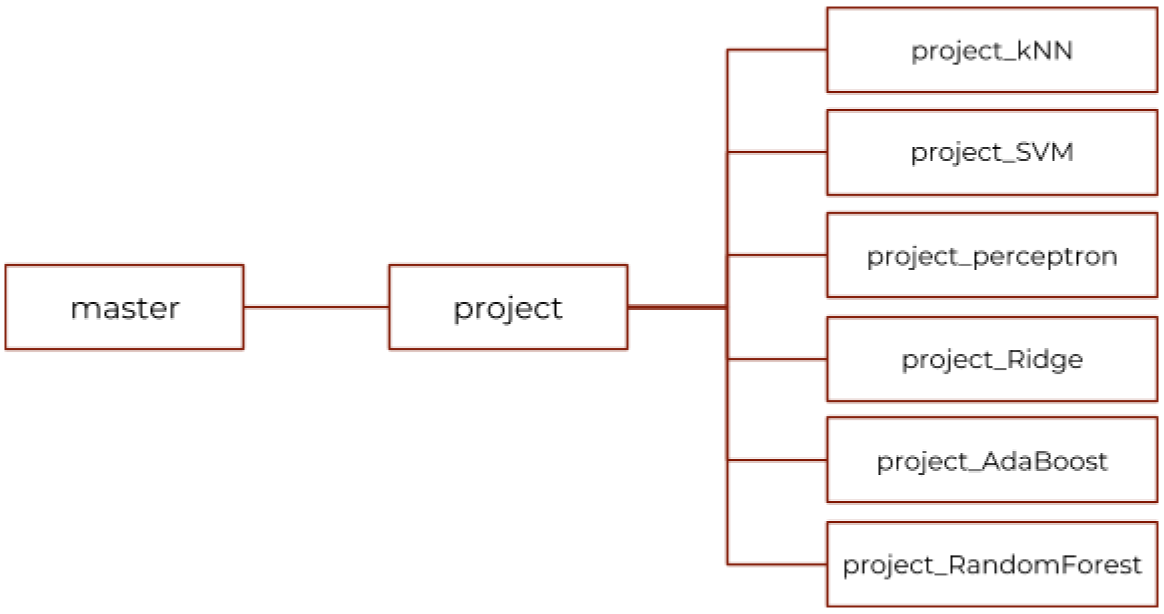
Eventually, an evaluation phase on the test set, by submitting the results of our best model with Kaggle tools, and conclusion will be mandatory to summarize the project content.

Project organization

Working in an organized environment is crucial for a team, and many tools exist to collaborate on a project. To manage our tasks at a high level, we used Trello as it offers many elementary features for free. The card system is perfect to handle the main lines, with the possibility to add a description, a checklist, but also assign members.

The python code itself was managed on a GitHub repository. Starting from a base branch named “project” (where we first created the data structure and basic parent functions used by every classifiers) 6 new sub-branches were created for each classification method. With this organization, it is possible to work on each method without interfering with the others. At the final stage of the project, when each method is tested and validated, those branches can be merged with the base project branch. Eventually, a main code file will be created to run the whole project, offering various command arguments to control which technique is to be used, along with specific parameters.

Here is a simple visualization of the GitHub project structure.



Data structure

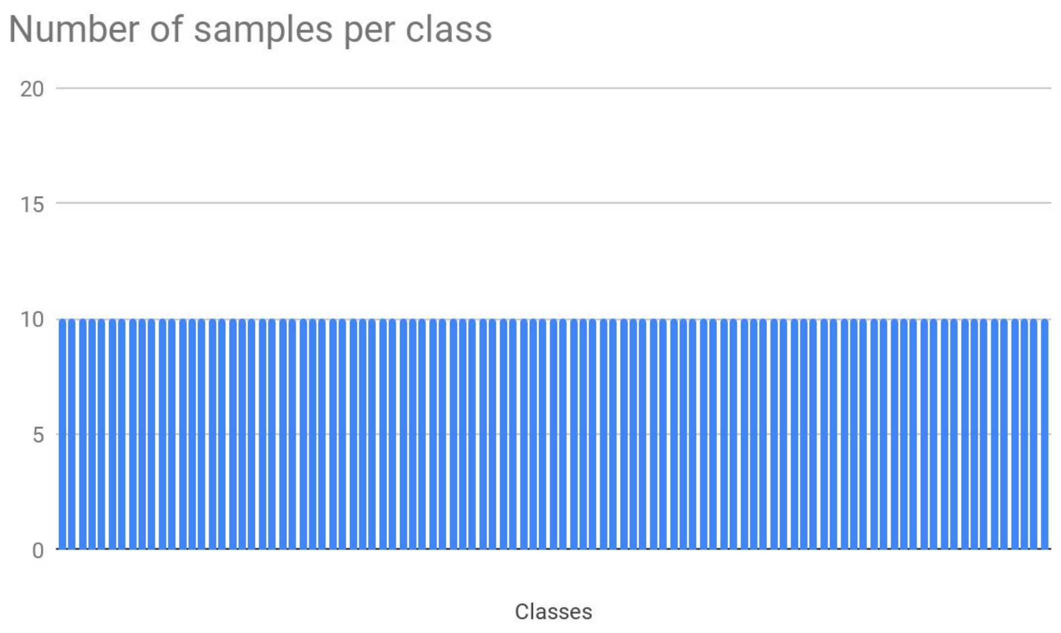
The dataset files downloaded from Kaggle are tables in the CSV format, divided in one training set and one test set (stored respectively in the files train.csv and test.csv). Each row represents an example of 192 numeric leaf features, plus additional information such as a unique ID. Training examples also have a corresponding know label, while testing examples are from unknown classes (only Kaggle has an oracle to compute a score).

With the content of those tables we chose to define a python class DataManager to load the features, transform them, and store them into 2D numpy arrays. This class has 7 attributes:

- x_train is a 2D numpy array, whose rows contains the 192 features for each training leaf example. As there are 990 training examples in the Kaggle dataset, this attribute has a shape of (990, 192).
- y_train is a 2D numpy array, whose rows are one-hot vectors for each training leaf example. As there are 99 different types of leafs, a row is made of 98 zeros, and a one for the value corresponding to the real label. It can be seen as a probability vector.
- y_train_strings is a simple numpy array with string labels, in the same order as y_train.
- y_train_integer is a numpy array with integer labels, in the same order as y_train. We just assign a unique integer value to each leaf class. Then, we translate the class of each row in the training dataset to the corresponding integer.
- x_test is a 2D numpy array, whose rows follow the same structure as x_train, but for the testing leaf examples. There are no labels associated with this set.
- ids_test is a numpy array with leaf ids corresponding the the x_test features. Those values will be used when writing the Kaggle submission file.
- string_labels is a numpy array with the 99 unique leaf names, also used while creating the submission file.

Note that there are 3 possible labels format : one-hot, strings, and integers. They both can be used equivalently, depending on algorithms implementations requirements.

After loading such data, we may now wonder about the quality of the training dataset. To check if classes are balanced, we count the number of examples par class, which gives us exactly 10 leafs for each label:



This means that the dataset is perfectly balanced, which is a really good observation to start with. Those observations allow us to define the following metrics, without a risk of training dataset bias.

Performance metric

In this classification project, we use 2 metrics, the accuracy and the cross-entropy loss function (also named logloss) as our performance quantifiers.

The accuracy metric is defined as the ratio between the number of correct predictions and the number of total predictions on a given set of examples. In the case of probability and one-hot vectors, a prediction is said to be correct if the highest probability is the one of the true class (if there is a tie, we consider the first value: as the training set is uniformly distributed, it is in practice equivalent to a random draw). In the case of string labels or integer labels, a prediction is simply said to be correct if the predicted label matches the true one.

We use the usual definition of the cross-entropy function.

An important point before starting is to analyse what should be "bad" values for those metrics. As we have 99 classes, the accuracy of a random model would be ~1.01% and its loss would be -ln(1/99) = ~4.59. After browsing the net, it seems that the first place on the leaderboard in 2017 was held by Ivan Sosnovik, with a loss of 0.0, which means that all his predictions were correct. Of course, the scope of this session project is not to beat state-of-the-art algorithms, but we will get quite strong results.

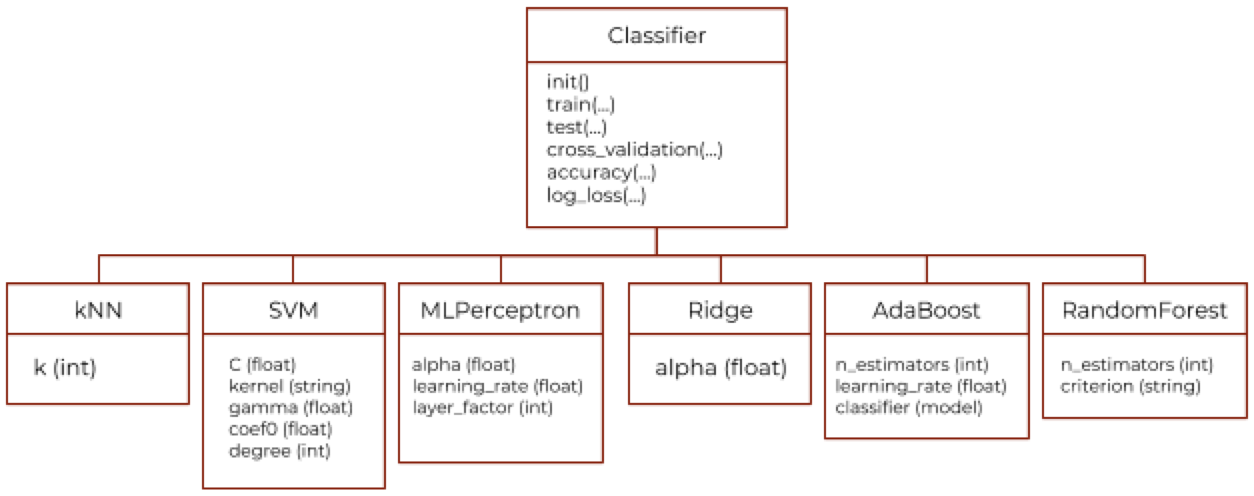
Classification methods

As we want to build 6 differents methods of classification, we created a general parent class inherited by 6 specific classifiers. In theory, this parent class is an interface, because it cannot be instantiated. Python, however, does not make any difference. This parent, named Classifier, has different functions to handle the basics of a classifier:

- In the class initialization function, parameters and hyperparameters can be defined and stored to be used later, with the python self instance.
- The function train(x_train, y_train) is called to tune and optimize the class parameters of the classification model. Each child classifier shall, of course, have a different implementation. It is important to note that, as mentioned above, the format of y_train is here depending on the model requirements.
- The function test(x_test) is called to perform predictions on a test set. The return value of this function also has a format depending on the model requirements, that is, the same shape as y_train used for training.
- The function cross_validation(x_train, y_train) is really important since it is where the values of the class hyperparameters are optimized. One common way to find their best values is to perform a k-fold cross-validation and select the best validation accuracy mean. At the end of this function, the model is trained one final time with the best values found.
- The function accuracy(predicted_labels, true_labels) evaluates the percentage of correct predictions.

- The function log_loss(predicted_labels, true_labels) evaluates the cross-entropy loss for the predictions, as defined in the Kaggle evaluation section.

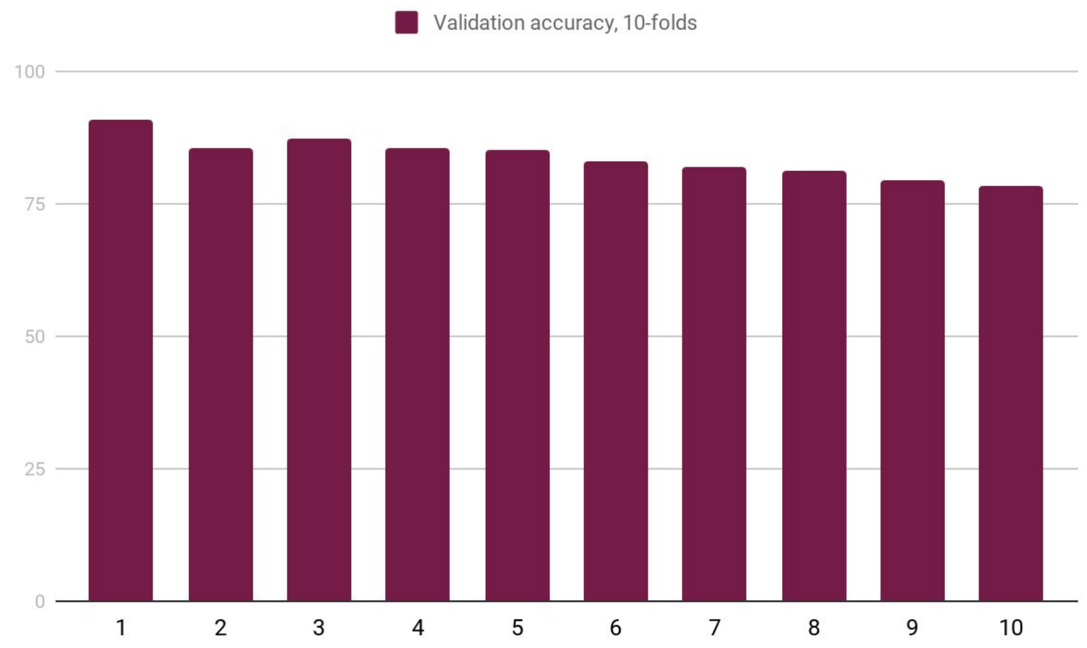
With such a class structure, each class inheriting Classifier can freely have its own implementation of those functions, depending on its model and hyperparameters. Here is a visual representation of the class structure:



1) k-Nearest Neighbours

The k-Nearest Neighbours method is one of the simplest and most intuitive classifier. The training phase simply consists in storing the training set and training labels, representing voters. During the testing phase, we compute the L2 distances of a test example with every training example. Then, the k examples with the minimal distances vote for their respective class, and the class with the highest number of votes is selected. In case of ties, random selections are made. We use, for this method, string labels.

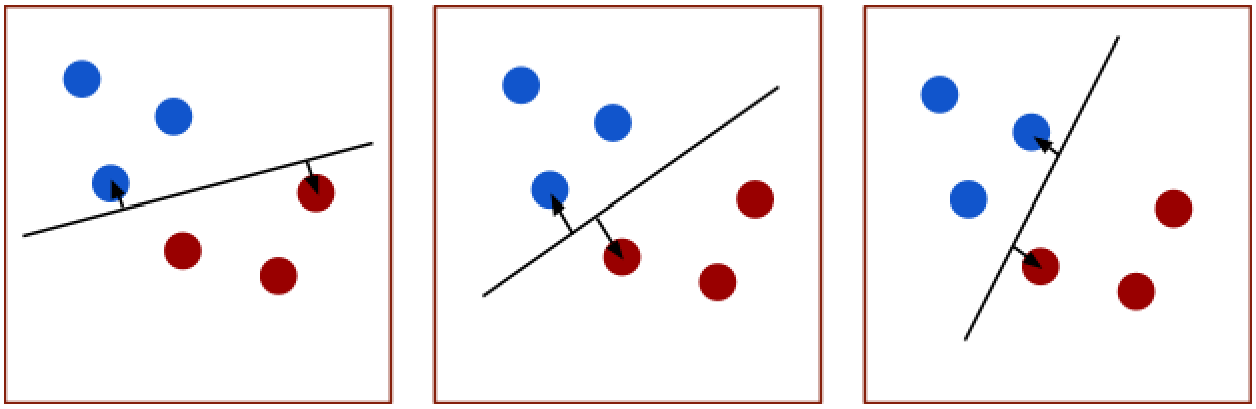
Of course, there is an hyperparameter, k. Let us us cross validate our model to find its best possible value. As there are 990 leaf examples in the training set, and 99 classes, we saw that there are 10 examples for each class. Hence, theoretically, it is possible to assume that a maximum reasonable value of k should be 10. To cross validate the model, we perform a 10-fold cross-validation for each value of k. The global validation accuracy for one value of k is computed as the mean of the 10 different folds accuracies. Here is a graphic evolution of the validation accuracy against k:



It is clear that the higher the value of k, the less the model is able to generalize to an unseen set. The best validation accuracy value in this case is 90,9%, so it may be a good choice to select k=1 on the testing set.

2) Support vector machine (SVM)

The objective of an SVM classifier is to divide the data so that the margin of the frontier is maximal. The frontier built by an SVM model is an hyperplane, which means that its dimension is the dimension of the features space minus 1. For instance, here is a 2D toy example with a few points distributed between 2 classes, red and blue:



With this specific set of points, an infinite amount of lines (hyperplanes of the 2D space) can be drawn to separate the 2 classes. However, the best solution for an SVM is the one in the center, because this line maximizes the margin around itself (represented by arrows on each side). The intuition behind this method is that a bigger margin will generalize better during the testing phase.

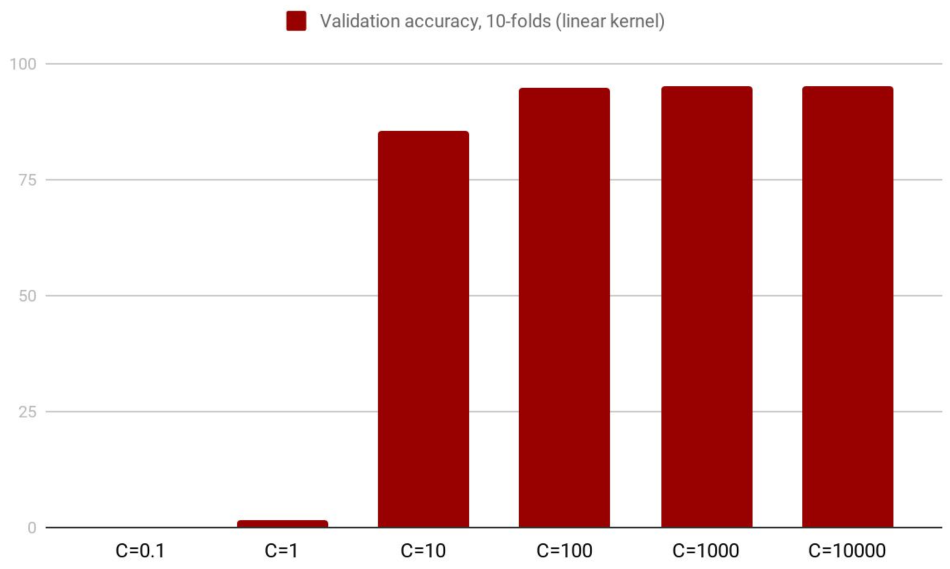
To implement this method, we used sklearn, because it already has ready-to-use functions such as fit (to train the model) or predict (to output a prediction for a test element). This implementation uses the dual formulation of the optimization problem, with kernels. We use, for this method, string labels.

The development of an SVM model requires several hyperparameters.

- C is used to control how soft the frontier should be, and thus, how the model would generalize. A high value would highly penalize misclassified data, thus possibly leading to overfitting. A low value would not consider misclassified data, possibly leading to underfitting.
- The kernel function used by the classifier can be linear, polynomial, sigmoidal, or rbf.
- If the selected kernel is polynomial, sigmoidal or rbf, the gamma hyperparameter controls the kernel function coefficient, and the coef0 hyperparameter controls the bias value.
- If the selected kernel is polynomial, another hyperparameter is required, which is the degree of the polynomial function.

To cross validate our model, there are a lot of possible combinations. To do so, we define a range of possible values for every hyperparameters. In some cases, kernels do not require every hyperparameters, so we let unused arbitrary values and do not loop over those. The ranges are defined after manually training and testing the model a few times. There are 4 different kernel functions to test: linear, rbf, polynomial and sigmoidal. We allow C to vary between 0.1 and 10000, on a logarithmic scale, gamma from 10e-4 to 1, on a logarithmic scale again, degree from 1 to 8, and coef0 from -4 to 4. For each possible combinations, a 10-folds validation accuracy mean is computed.

When using a linear kernel, there is only one hyperparameter, so it possible to plot the accuracy against C:



Low values for C (less than 10) often perform poorly (giving a validation accuracy of zero most of the time) and high values usually perform very well (with a validation accuracy around 95%). With such big C values, we should be careful and wonder if the model is overfitting to the training data. However, as the accuracy shown in the chart is for validation sets, so C=100 should be a perfect choice. Moreover, we will later see that with data transformation lower C values will perform well too.

For other kernels, it is not possible to present a visual representation on a graphic with that much dimensions (there are at most 5 hyperparameters). However some recurrent behaviors can be described. The rbf, polynomial, and sigmoidal kernels seem to be very sensitive to gamma and C. Indeed, with low values (gamma < 0.001, C<10), the validation accuracy always drop to almost 0%. Higher values tend to perform similarly to the linear kernel (validation accuracy of ~95%).

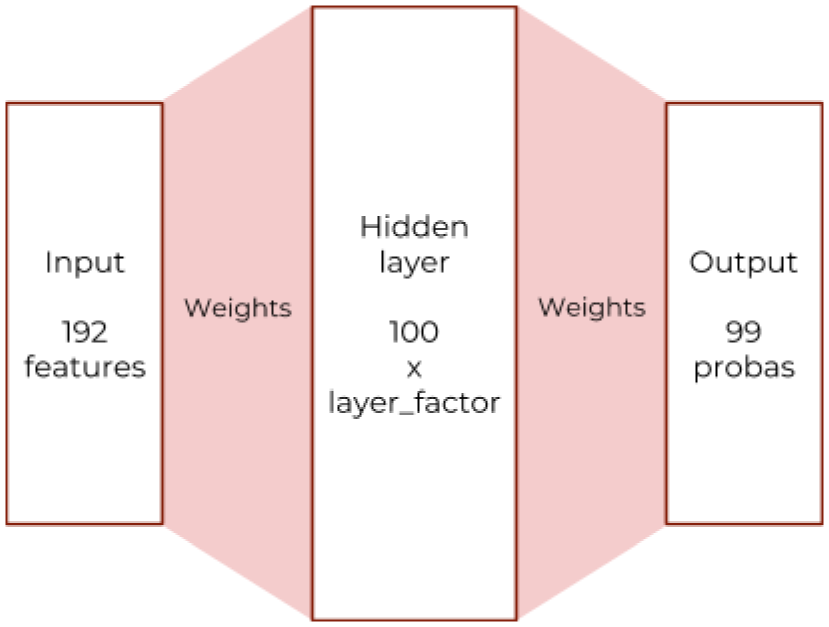
After testing every possible combinations (during approximately 30 minutes), the best values found are kernel=polynomial, gamma=1, coef0=2, C=10, and degree=4. The validation accuracy for those hyperparameters values is 95.4%, which is already quite high.

3) Multi Layer Perceptron (MLP)

A classic perceptron classifier computes a dot product between the input and a weight vector, combined with an activation function to separate the data between 2 classes. Here, however, we do have 99 classes. The sklearn perceptron implementation does only provide support for 2 classes, so we had to use the multi layer perceptron classifier. Even though it is still named perceptron, it is already a simple neural network. The input of our network is the vector of the 192 leaf features, while the output is a 99 vector of probabilities, using a softmax activation function.

To implement this method, we used sklearn, because it already has ready-to-use functions such as fit (to train the model) or predict_proba (to output a prediction for a test element with a softmax activation). We use, for this method, one-hot vectors labels.

We aimed to keep the structure as simple as possible, to stay close to a simple perceptron. To do so, we decided to build a network with only one hidden layer, and defined an hyperparameter layer_factor to control how large this layer is. The number of units is 100 x layer_factor.



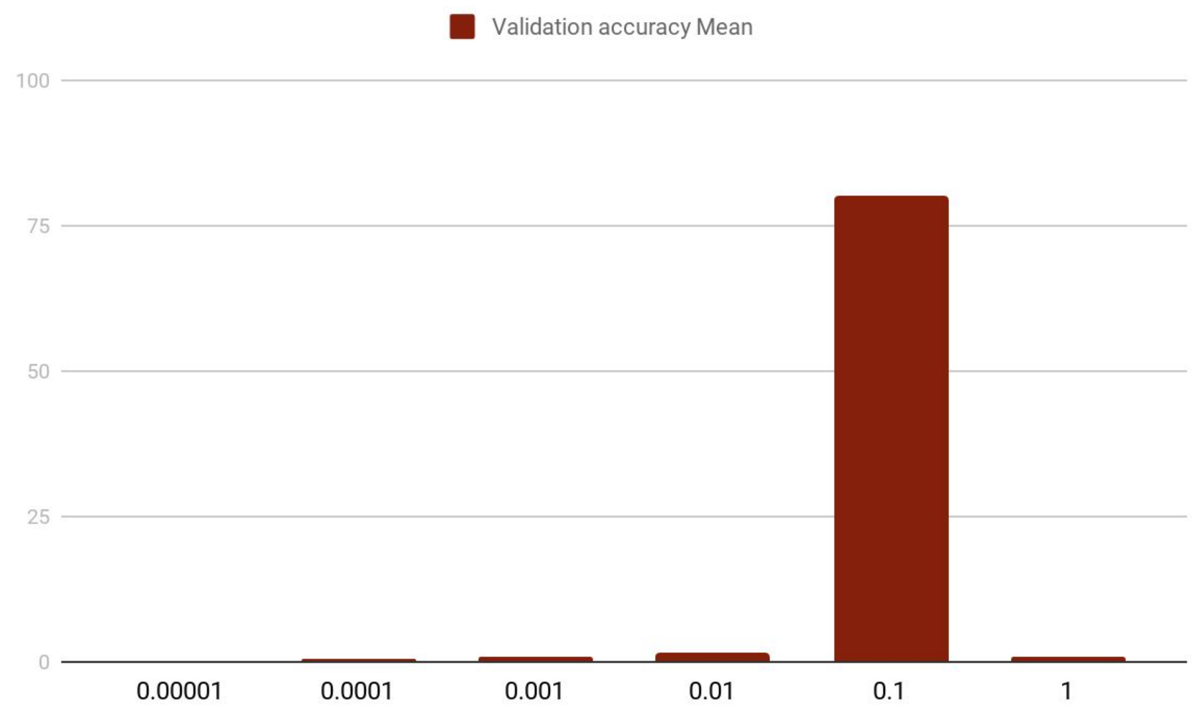
As there are almost 200 input features and 99 output probabilities, it is reasonable to assume that a correct unit amount could be at least 200. We hence allow our hyperparameter layer_factor to vary between 2 and 32. Of course, with such a structure, lower values will underfit and higher values will overfit.

Sklearn also let us play with 2 more main hyperparameters:

- alpha is the constant associated to the L2 regularization penalty. We let it vary between 1e-5 and 1 on a logarithmic scale.
- learning_rate is the value controlling how fast the network will learn, that is, how big its gradient steps will be. We keep the default adam optimizer, widely used in literature, with default beta1 and beta2 values, as they do not clearly change the results in our context.

The activation function is left to its default value, ReLU (rectified linear unit), because it is also widely used in literature, and others are now mostly deprecated because of the vanishing gradient issue. The batch size for gradient optimization steps is left to its default value, because it theoretically does not change the final convergence point.

Again, for each combination of values, with compute the validation accuracy mean on 10 folds. During this cross validation process, some interesting behaviors happen. Here is the mean of the validation accuracy, for every combination of layer_factor and alpha, depending on the learning_rate:



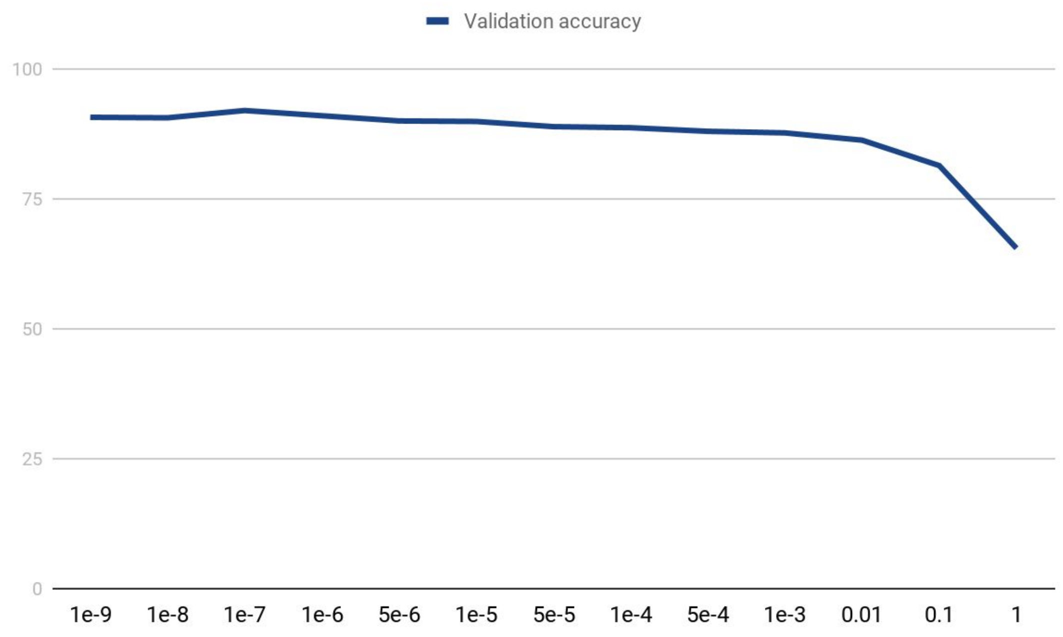
Those results shows that the learning_rate is crucial for the network to learn and generalize efficiently, and that 0.1 is almost always its best value. As the maximum number of iterations was a constant number set to 40, it means that for low values, the network takes too much time to converge, while for big values it does not converge at all. Then, with this best value found, both layer_factor and alpha seems to determine how the network will generalize to a unknown test set. Different combinations result in a validation accuracy usually ranging between 50% and 95% (in some cases, it can even drop below 5%). The best combination found is layer_factor=8, alpha=0.001, learning_rate=0.1 for which the validation accuracy is 92.7%.

4) Ridge

We can see our classification problem as a regression problem and use the Ridge regression method as a classifier. The trick is that the output of our function is a vector of 99 values, instead of one single value.

To implement this method, we used sklearn, because it already has ready-to-use functions such as fit (to train the model) or predict (to output a prediction for a test element). We use, for this method, one-hot vectors labels as target for our regression function.

The Ridge model has only one hyperparameter, alpha, which is as before the L2 regularization penalty factor. We allow it to vary between 1e-9 and 1, on a logarithmic scale. Again, for each combination of values, we compute the validation accuracy mean on 10 folds. Here is a graphic representation of the validation accuracy against the hyperparameter alpha:



It is clear that higher values deprive the model of learning and generalization capabilities. A good value for alpha would be around 1e-7, and the corresponding validation accuracy is 91%.

5) AdaBoost with Decision Trees

AdaBoost is a particular method to train a boosted classifier, using a sequence of weak classification algorithms (estimators), resulting in a robust classifier. A weighted majority vote takes place, and the classifiers are combined to produce the final prediction. Moreover, in the process, the weighted vote will give a higher influence to the more accurate classifiers of the sequence.

In this method, we introduce AdaBoostClassifier and DecisionTreeClassifier from sklearn.

The main hyperparameters of the AdaBoostClassifier used in the algorithm are as follows:

- base_estimator is the base estimator with which the boosted ensemble is built. In our context, its value is an instance of a DecisionTreeClassifier.
- n_estimators is the maximum number of estimators to use, that is, the number of voters.
- learning_rate shrinks the contribution of each classifier. There is a trade-off between n_estimators and learning_rate.

Since we use an instance of DecisionTreeClassifier as base_estimator, it's necessary to mention some of key hyperparameters.

- criterion is the measure of quality of a split. It supports 'gini' and 'entropy'.
- max_depth is the maximum depth of the decision tree.

To cross validate the model, there are various ways to combine these hyperparameters from the two classifiers. We tried several types of combination with a 10-folds cross-validation.

At first, let's check the effects on the two hyperparameters of the DecisionTreeClassifier. The best validation accuracy we got from a Decision Tree alone is 73.15%, with the criterion set to "entropy", max_depth to 11. With such a base estimator, we then searched for the best n_estimators and learning_rate for AdaBoost (note that we here had to separate the cross-validation phases of the two methods because of their computation time, though sub-optimal). The resulting validation accuracy is 73.34%, which almost doesn't change from the accuracy of Decision Tree.

However, we tried to fix the criterion to "gini" and an interesting phenomenon happened. The validation accuracy of the Decision Tree alone dropped to 33.4%, and trying different values of max_depth did not help. However, the validation accuracy of Adaboost skyrocketed to 94.4%. It suggests that the criterion significantly influences the performances of the resulting global vote. It shows that AdaBoost benefits from a set of underfitting estimators, while it seems not to improve an already strong classifier. Therefore, we kept those hyperparameters as base_estimator. Combining them together with a vote system highly enhanced the validation accuracy. However, when the learning_rate is too high, the validation accuracy sometimes drops below 40%, which shows that the weighted majority vote plays a crucial role in the AdaBoost algorithm.

Finally, after cross-validating our model, and based on these experiments, the best validation accuracy is 94.4%, with n_estimator = 200 and learning_rate = 0.01.

6) Random Forest

Random Forest is an cluster of Decision Trees. In other words, it merges multiple decision trees together to get an accurate and stable prediction, to limit overfitting. Not so different from AdaBoost, the main idea of the random forest method is to create random subsets of data and build smaller trees using these subsets, to finally combine them.

The RandomForestClassifier in the sklearn library offers the ability to fit a number of decision tree classifiers on different sub-datasets and use a vote to improve the prediction accuracy. Therefore, we introduce those hyperparameters to train the model:

- n_estimator is the number of trees to use in the forest. Usually, a bigger number of estimators improves the accuracy, up to a certain limit.
- criterion is the quality measure of a split, and can be “gini” or “entropy”. This parameter is the same as the one in the DecisionTreeClassifier.

During the process of cross validation with 10-folds, we combined various values of n_estimator and criterion and to train and validate the model. After several manual trials, we narrow n_estimator in a range from 10 to 100, with a step size of 10, and we set criterion as “gini” or “entropy”. Compared to the results of Adaboost, the validation accuracy of Random Forest is more reliable as it does not drop to excessively low values. As expected, lower numbers of estimators show lower accuracies, and higher values tend towards a limit. The best hyperparameters found are n_estimators = 90 , criterion = “gini”, for which the validation accuracy is 97.58%.

Data transformations

Now that our 6 methods are ready to use, our purpose is now to manipulate the data or, in other words, pre-process it and assess how different transformations could improve the learning capabilities of our best models so far, thas is, the SVM and Random Forest classifiers.

1) Center and normalize

The first and well-known step toward achieving better results is, of course, centering and normalizing data. To do such a transformation, we subtract for x_train its mean, and divide it by its standard deviation. Of course, we do the same with x_test. The results are already remarkable: after performing a cross-validation on the training set, the best validation accuracy of the SVM raises up to 98.7%, which is already high, while the Random Forest is not affected. In addition, for the SVM, lower values of C perform quite well too, so that means the transformed data is more easily separable with a softer frontier than the original one. We can thus expect the model to generalize better on the test set.

2) Image features

As images were provided, we tried to exploit basic features from them. We load images with OpenCV for python, and add the width, height, ratio (width/height), square (width*height), and orientation (width > height) to the existing set of features. This idea comes from the competition winner, so we decided to give it a try. After adding such features, the validation accuracy of the SVM does increases a bit up to 98.9%, and the Random Forest one goes up to 98.6%. Sadly, we will see that the score on Kaggle becomes worse.

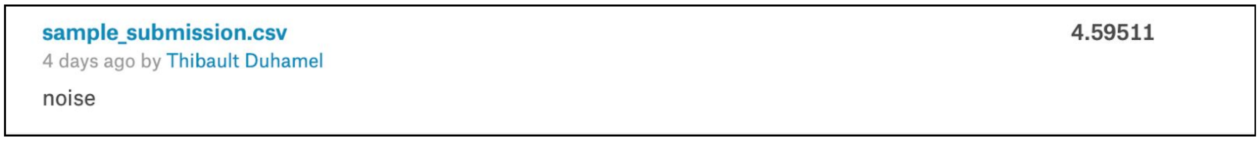
3) Feature pruning

We now aim to evaluate how our models would perform with a subset of features. We use the sklearn feature selection module and use a recursive feature elimination (RFE) to gradually remove useless features from the train and test set. We first reduced the dimension of the input to 180 features and the training accuracy of our SVM did not change, which proves that some features may not help to separate the data (because their variance may be low). So, we continued to decrease the number of features until it actually affected the training accuracy. We assessed, after testing several values, that the training accuracy would start to go down when reaching the threshold of 10 features. However, we also plotted the test accuracy for the values we tried (in the next section), and we will see that a too small amount of features impairs the model from generalizing.

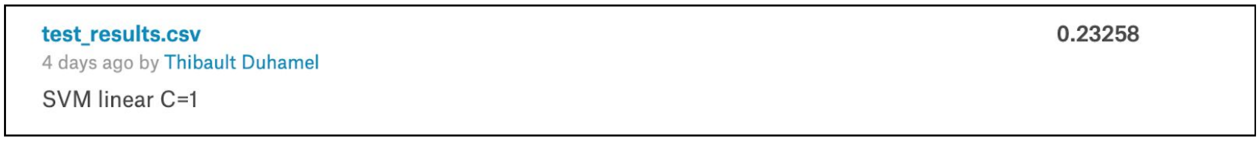
Kaggle test evaluation

The submission file has a specific csv format, in which each row is made of a leaf id and 99 class probabilities for this leaf. We then use the Kaggle API to upload it and get a score. The Kaggle online system only outputs one score, the cross-entropy loss value.

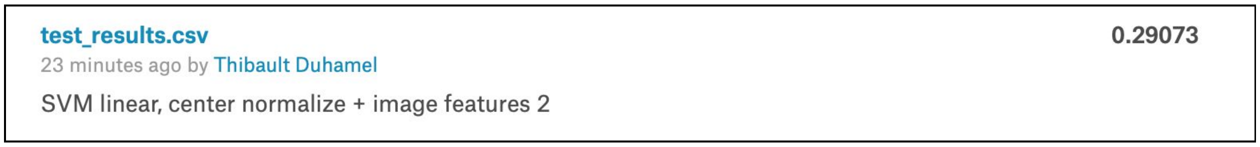
Let us first create a file made of noise, that is, with each probability being 1/99 as a sanity check. The score obtained with such a uniform distribution is exactly -ln(1/99)=~4.59, as expected.


sample_submission.csv 4.59511
4 days ago by Thibault Duhamel
noise

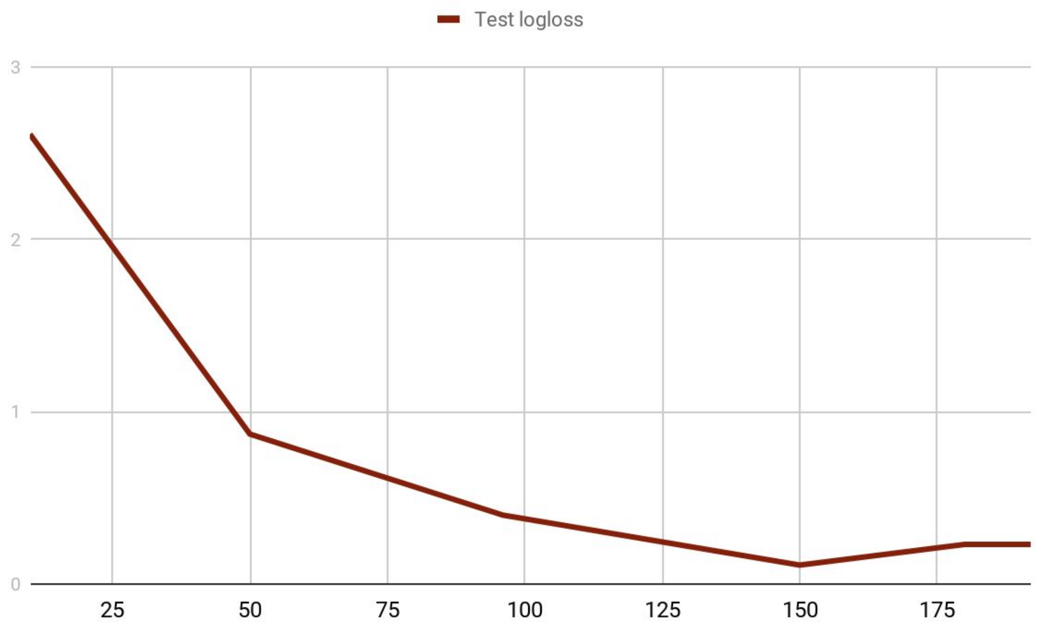
Then, we aim to evaluate our best model, which is the SVM, with centered and normalized data, but not image features. The 10-folds validation accuracy for this method was 98.7%, and the validation cross-entropy value was 0.45. When uploading the submission file, we get a test set score of 0.23. As 0<0.23<0.45, we can estimate a test accuracy around approximately 99%, which confirms that our model is not overfitting on the training data, and that the test data is quite similar to the training set.


test_results.csv 0.23258
4 days ago by Thibault Duhamel
SVM linear C=1

Now, we add images features and upload a new submission file. This time, the loss increases, which is bad news. The images features may hence not help to generalize.

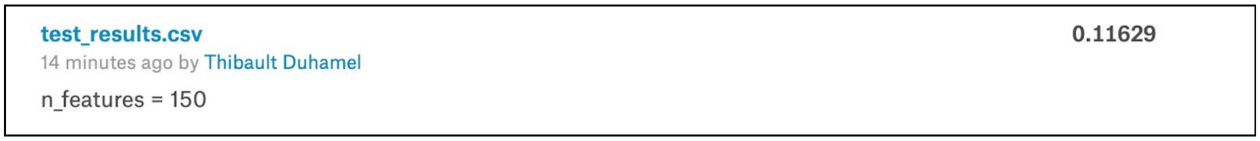

test_results.csv 0.29073
23 minutes ago by Thibault Duhamel
SVM linear, center normalize + image features 2

Finally, we upload our results for different levels of feature pruning. Let's plot the graph of the test loss score against the number of features:



We here observe 2 things:

- It is clear that an excessively low number of features indeed impairs the network from generalizing, as we could expect.
- The minimal loss score we could get using this feature pruning technique is 0.11, even lower than our best one so far. This result is extremely interesting because it means that some features are not only useless, but also dangerous to learn from.


test_results.csv 0.11629
14 minutes ago by Thibault Duhamel
n_features = 150

Conclusion

We have implemented 6 classification methods on a leaf dataset and provided analysis of our work results. We cross-validated our models to tune their hyperparameters, but also transformed the raw data (centering and normalizing, using images, pruning features...) so that it could be separated in a more effective way.

Eventually, we evaluated the test results of our best model, the SVM classifier, with all those transformations, and it has performed quite well with a test accuracy above 99% and a Kaggle cross-entropy score of 0.116.

We believe that even better results could be achieved using a convolutional neural network on the leaf images (along with data augmentation, that is, scaling, rotating, translating, mirroring...), but this kind of architecture was just too complex and too long to train on a laptop in the scope of this project. Feature extraction using a deep-learning architecture could also help separate the data in a more effective way.

References

Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

Trevor, H., Robert, T., & JH, F. (2009). The elements of statistical learning: data mining, inference, and prediction.

Sklearn:

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.RidgeClassifier.html https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html