HENG Anne-Marie
MARTIN Elise
OULID AZOUZ Noureddine

ENSAE 3e année
*Apprentissage Statistique Appliqué*
*Année scolaire 2021/2022*

# TP1 : Basic functions for Supervised Machine Learning

Professeur : Mohamed HEBIRI

Chargée de TD : Clara CARLIER

# 1 PART 1 - MNIST

### Cross-validation with GridSearchCV

**Question 1.1: Complexity**

**What is the complexity for each of the three following cases ?**

- **K-NN**

When reading the Sklearn documentation, we can see there are various algorithms to compute the nearest neighbors. The default parameter to chose the algorithm is set on auto and Sklearn documentation doesn't give a clear explanation about what lies behind this default parameter. We will assume that brute force is used here, which is the method we saw in class.

Let's resume the steps computed for K-NN :
— Assume we fix the number of neighbors K : we loop over k and for each entry in the samples, compute the distance d over the n_features and select the smallest distance d (when k=1), the second smallest (when k=2) until the $k^{th}$ smallest (when k=K).
— The distance computing for an entry is in O(n_features) and this calculations is made n_samples times. So for each neighbor, the complexity is $O(n\_features * n\_samples)$.

As there is K neighbors, the final complexity is $\boxed{O(K * n_{features} * n_{samples})}$.

- **SVM**

For the SVM classifier, we use the 1.4.4 section of its Sklearn description. In the case of the linear SVC here, the algorithm used for optimization is implemented the liblinear library and "*scales almost linearly to millions of samples and/or features*".

This means that the algorithm is in $\boxed{O(n_{features} * n_{samples})}$.

- **Logistic Regression**

We can summarize the logistic regression as the optimization of the following parameters : a vectors of bias terms $b$ and the vectors of weights $W$. This vectors are of dimension $n_{features}$. To optimize this parameters, we compute for each entry the weighted sum :

$$\sigma(X_i^T W) + b$$

with $X_i$ the vector of features for entry i. For each i in range $n_{features}$, there is $n_{features}$ calculations.

Therefore the complexity of logistic regression is $\boxed{O(n_{features} * n_{samples})}$

- **GridSearch**

The GridSearch method only multiplies the complexity by the number of fits. For example, we try 5 parameters for the KNN algorithm over 3 CV, so there are 15 fits. This means multiplying the complexity by 15 (a constant) in this and changes nothing. However, we could multiply all the complexities above by $n_{fits}$ to be perfectly rigorous.

**Question 1.2: clf.fit**

**Explain in your report what happens when we run *clf.fit*.**

When we run the *fit* method , we launch the search over best combinations of parameters for our estimator. Here our estimator is a classifier : a knnclassifier model and our parameters are those we place into the variable "parameters". The data we use for the search are X_train and Y_train. A GridSearchCV is an exhaustive search, that means all combinations of parameters are tested. Each combination of parameters gives a new model. For example here we have a model with 1 neighbor, then with 2 neighbors, etc. As we specified cv = 3, we have a 3 folds cross-validation for each tested model.

**Question 1.3: Test accuracy**

**What is the test accuracy ? What would be the accuracy of random guess ?**

The test accuracy here is the proportion of observations from the X_test dataset that we have rightly classified. The accuracy of a random guess would be 0.1. Indeed, we would randomly give a label (*i.e* a number between 0 and 10) to each observation and for each one, we have a probability of 1/10 to have a right guess.

**Question 1.4: LinearSVC**

**What is LinearSVC() classifier ? Which kernel are we using ? What is C ?**

LinearSVC means Linear Support Vector Classification. Here we use a linear kernel. C is the parameter of regularization, by default there is no specific regularization, C = 1. The strength of the regularization is inversely proportional to C.

**Question 1.5: Logspace**

**What is the outcome of np.logspace(-8, 8, 17, base=2) ? More generally, what is the outcome of np.logspace(-a, b, k, base=m) ?**

The outcome of np.logspace(-8, 8, 17, base=2) is a numpy array of length 17. The numbers of this array are space evenly on a log scale beetween -8 and 8. More generally, np.logspace(-a, b, k, base=m) return numbers spaced evenly on a log scale with -a the starting value, b the stop value, k the number of values to generate and m the base of the logarithm.

**Question 1.6: Warnings**

**What is the meaning of the warnings ? What is the parameter responsible for its appearence ?**

There is a convergence warning, there is not enough iterations to have a convergence. The parameter responsible for its appareance is max_iter. The algorithm might here fail to converge in the optimization step as there is not enough iterations.

**Question 1.7: Scaling**

**What did we change with respect to the previous run of LinearSVC() ?**

With respect to the previous run of LinearSVC(), we add a pipeline. In ML we often have to perform sequence of different transformations (examples : select only some good features, scaling, imputing missing values etc.) of raw dataset before applying final estimator. A pipeline enables us to do all these transformations in once while keeping our code clean. Each transformation is contained in a step. Here we have 2 steps : the $1^{st}$ step is a *scaler* which enables us to scale our features - it's a way to normalize our data. The second step is the *SVC* model defined earlier : SVC = LinearSVC(max_iter=5000)

**Question 1.8: Pipeline**

**Explain what happens when we execute :**

- **pipe.fit( X_train, Y_train)**
- **pipe.predict(X_train, Y_train)**

When we run the code we see that there is no warnings concerning convergence anymore so the problem we had earlier was probably due to the absence of normalization.

If we execute the 2 lines of code : we execute the 2 steps of the pipeline (that we explained earlier) on the train set with the method fit. Then our model is trained and we can predict the class of the test dataset. It is necessary to note so far that the test sample is scaled with the parameters fitted on the train set. When we apply the pipeline to our test dataset, the first step of scaling is applied to X_test and then the model of the 2nd step.

**Question 1.9: MaxAbs**

**What is the difference between StandardScaler() and MaxAbsScaler() ? What are other scaling options available in sklearn**

It is not the same way of scaling our features. StandardScaler() for a feature x in our sample$x$ is calculated as follows :

$$z = \frac{(x - \mu)}{\sigma}$$

with $\mu$ being the average observation on all individuals for a selected feature and $\sigma$ the standard deviation of the training samples. On the other hand MaxAbsScaler for a samples of observation for the feature $X = (x_1, x_2, ..., x_n)$ is calculated as :

$$Z = (\frac{x_1}{m}, \frac{x_2}{m}, ..., \frac{x_n}{m})$$

with $m = max(|x_i|)$ for i in $[\![1, n]\!]$. Other scaling options are RobustScaler (meaning robust to outliers), MinMaxScaler etc.

**Question 1.10: New classifier**

**Using the previous code as an example achieve test accuracy $\geq 0.9$ . You can use any method from sklearn package. Give a mathematical description of the selected method. Explain the range of considered hyperparamers.**

The method we chose for this part is a RandomForestClassifier. This algorithm works as a bagging method of DecisionTrees so to give a mathematical description, of this classifier, we need first to describe how DecisionTrees are trained [1].

### Decision Tree

Scikit Learn implements the CART (Classification and Regression Tree) algorithm to train a Decision Tree. The algorithm works by splitting the training set into two subsets using a single feature k and a threshold $t_k$. The splitting is done in what we call a node. Several splitting can be done during training to build a tree. To choose k and $t_k$, we search for the pair $t_k$ that gives the purest (in the Gini sense) subsets (weighted by their size). The below equation gives the cost function that the algorithm tries to minimize :

$$J(k, t_k) = \frac{n_{left}}{n} * G_{left} + \frac{n_{right}}{n} * G_{right}$$

where $G_{left/right}$ is the impurity in a subset and $n_{left/right}$ the number of instances in the two subsets. Besides, for the node i, Gini impurity is computed as follow :

$$G_i = 1 - \sum_{k=0}^{9} p_{i,k}^2$$

with $p_{i,k}$ being is the ratio of class k (classes being 0 to 9 here) instances among the training instances in the $i^{th}$ node.

A Decision Tree can therefore estimate the probability that an instance belongs to a particular class k. First it goes through the tree to find the leaf node (final node) for this instance, and then it returns the ratio of training instances for all classes in this node. It can then output the class with the highest probability and assign the instance the this class prediction.

### Bagging and RandomForest

Then, now that we now how to train one DecisionTreeClassifier, we use the bagging method to build a RandomForest. The essential idea in bagging is to average many noisy but approximately unbiased models, and hence reduce the variance. Trees are ideal candidates for bagging, since they can capture complex interaction structures in the data, and if grown sufficiently deep, have relatively low bias [2].

In this algorithm the decision tree is build only on a subpart of our sample and uses just a part of the features. Thanks to bagging on the observations level and on the features level we create trees with little variations. Thus, each tree in our forest is different. Each tree vote for the class of the observation. We then aggregate all trees together to build a classifier. The majority vote is the predicted class.

1. description from Hands on Machine Learning with Scikit-Learn, Aurélien Géron
2. see McGill Course

We only remind the way bagging works with the below algorithm, a more mathematical approach being given in the course.

**Algorithme 1 :** RandomForest for classification

**Input:** $B$ the number of boostraps
**Output:** An estimator
1) **For** b in $[\![1, B]\!]$
Draw a bootstrap sample $Z$ of size $N_b$ from the training data
Grow a Decision Tree $T_b$ on the bootstrapped data, by recursively repeating the following steps for each terminal node of the tree, until the minimum node size $n_{min}$ is reached :
i) Select m variables at random from the p variables.
ii) Pick the best variable/split-point among the m.
iii) Split the node into two daughter nodes.
**end for**
2) Output the ensemble of trees $\{T\}_b^B$

To classify a new point x :

Let $\hat{C}_b(x)$ be the class prediction of the b-th DecisionTree.
Then $\hat{C}_{rf}^B(x) = majority\ vote\{\hat{C}_b(x)\}^B$

Note that the predicted class of an input sample is a vote by the trees in the forest, weighted by their probability estimates. That is, the predicted class is the one with highest mean probability estimate across the trees. This means :

$$\hat{C}_{rf}^B(x) = \arg\max_k \sum_{b=1}^{B} w_b \hat{p}_{k,b}$$

where $\hat{p}_{k,b}$ is the probability estimate from the $b^{th}$ classification tree for the $k^{th}$ class. The weights are usually set as $\frac{1}{B}$.

### Range of parameters

To define a good range of parameters, we first inspect the parameters to tune when using RandomForest in the Sklearn documentation. As we saw in our notebook, we can keep things simple as the default parameters gives a test accuracy greather than 0.9. As a consequence we know that the default parameters already give good performances. We can try to consider a range of hyperparameters close to these. Furthermore, regarding the max_features parameter, a good rule of thumb is the try a range of features near the square root of the number of features we have. As we don't have a huge dataset we can let the default parameters for max_depth, min_samples_leaf and min_samples_splitAs we don't have a huge dataset we can let the default parameters for $\max_depth, min_samples_leaf and min_samples_split$".

Note : Here, our classifier achieves a better performance on the test set than on the train set. This phenomenon is quiete rare but could be explained by the relative simple data we are working with. Indeed MNIST nowaydays an easy dataset for many classification algorithms such as random forests.

## Visualizing errors

**Question 1.11: Mistake**

**There is a mistake in the following chunk of code. Fix it.**

For this question, please refer to the notebook.

### Changing the Loss function

**Question 1.12: Balanced accuracy**

**What is balanced_accuracy_score ? Write its mathematical description.**

The balanced accuracy is a metric to evaluate the performance of our model. It is often used in case of imbalanced datasets in classification problems. Indeed take for example a binary classification problem where we have an imbalanced dataset with 90% of our observations in class 0 and 10% in class 1. If we take a dummy estimator that always predicts the class 0 we'll have an accuracy of 0.9. Thus, in that case, the accuracy score is an inappropriate metric. Mathematically, the balanced accuracy score is defined as the average of recall obtained on each class. The best value is 1 and the worst value is 0 when adjusted=False (parameter by default). The mathematical formula is given below.

If $y_i$ is the true value of the $i^{th}$ entry, and $w_i$ its corresponding sample weight, then we adjust the sample weight to :

$$\hat{w}_i = \frac{w_i}{\sum_j \mathbf{1}(y_j = y_i) w_j}$$

where $1(x)$ is the indicator function. Givin predicted $\hat{y}_i$ for sample $i$, balanced accuracy is defined as :

$$balanced\_accuracy(y, \hat{y}, w) = \frac{1}{\sum \hat{w}_i} \sum_i \mathbf{1}(\hat{y}_i = y_i) \hat{w}_i$$

**Question 1.13: Confusion matrix**

**What is the confusion matrix ? What are the conclusions that we can draw ?**

The confusion matrix is a matrix with each row $i$ corresponding to the true class and with each column $j$ corresponding to the predicted class. Entry $i, j$ in a confusion matrix is the number of observations actually in group $i$ and predicted to be in group $j$. Thus, the diagonal corresponds to the good predictions.

The conclusions we can draw are that the classes where we make most of the mistakes are :

- There are 3 observations that are classified as 5 whereas their true label is 8.
- There are also 3 observations that are classified as 5 whereas their true label is 3.
- There are 3 observations that are classified as 3 whereas their true label is 1.
- The classes we have the most trouble to rightly predict are 3 and 8. For both, we have 6 observations that are wrongly predicted.

# 2 PART 2 - PROBLEM

**Question 2.1: Loss function problem**

**Propose a loss function that would address our needs. Explain your choice**

## 2.1 Description of the loss

Assume we have the following confusion matrix.

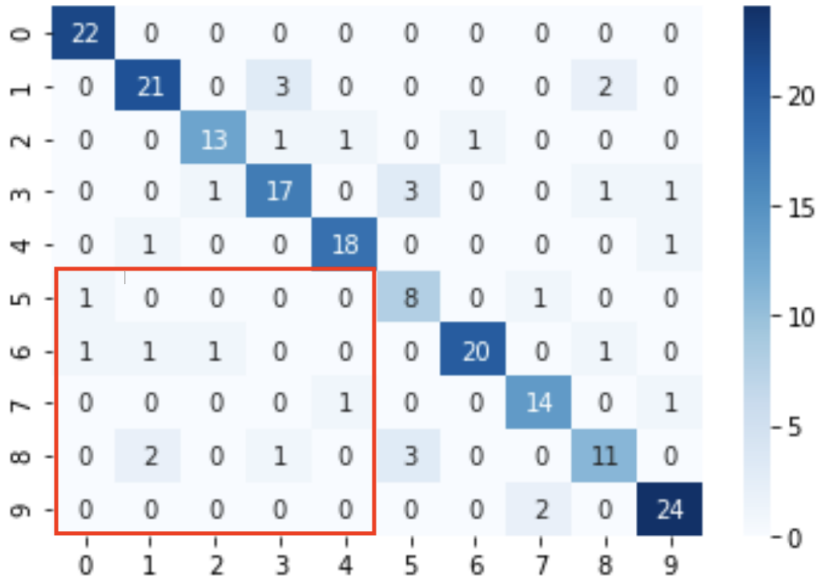


FIGURE 1 – Confusion Matrix

Let $x_{ij}$ be the coefficient of the $i^{th}$ row and the $j^{th}$ column.

The red area in the confusion matrix represents all the mistakes we **really** want to avoid, that is to say the digits from $\{5, 6, 7, 8, 9\}$ that are predicted to be from $\{0, 1, 2, 3, 4\}$.

The red area is the ensemble $E_1 = \{x_{ij}$ with i $\in \{5, 6, 7, 8, 9\}$ and j $\in \{0, 1, 2, 3, 4\}\}$.

In the diagonal we have all the observations that we have correctly classified.

The remaining coefficients that is to say the coefficients in the ensemble $E_2 = \{x_{ij}$ with $i \in \{0, 1, 2, 3, 4\}$ and j $\in \{5, 6, 7, 8, 9\}$, i $\neq$ j$\}$

The idea is that if we sum the coefficients of $E_1$ (let call this sum $S_1$) we obtain the number of errors we **really** want to avoid. So the worst errors, let call them type 1 errors.

Let call the other type of errors, type 2 errors, and the sum the coefficients of $E_2$ is noted $S_2$.

We want to minimize $S_1$ but yet still control for $S_2$.
Thus we have a loss function like this :

$$l(y, y') = \alpha S_1 + \beta S_2$$

with $\alpha > \beta$ because type 1 errors are worst than type 2 errors.

**Question 2.2: New loss classifiers and pipeline**

**Following above examples, make an ML pipeline that uses your loss function and finds appropriate classifiers.**

## 2.2 Description of the pipeline

We use the same type of pipeline as before. As decision trees do not care of feature scaling, this is an unecessary step and we can therefore directly compute the training and the prediction. For a better estimation of the error, we will also use a 3-folds cross-validation.

## 2.3 Description of the algorithm

The best algorithm has already been described in question 1.10.

**General comments about errors :**

Once again, the model that obtains the best performace with our new score function is the RandomForest. Here one might see that the fine-tuning step returned different hyperparameters than when trained with the accuracy score. In short, we did reduce the type 1 errors from 8 to 5 with our new score. Surprisingly, the model fine tuned on new score does a little better than the older one. This could be due to randomness or the fact that we also increase the overall accuracy by focusing on some kinds of error that the models do most !

Rather than changing the loss, other approaches could improve our models on these errors. For example, we could try to gather more training data for digits that look like 1s, 2s, 3s and 4s (but are not) so that the classifier can learn to distinguish better. Or we could engineer new features that would help the classifier — for example, writing an algorithm to count the number of closed loops (e.g., 8 has two, 6 has one, some digits in $[\![1, 4]\!]$ has none ! ). Or we could preprocess the images (e.g., using Scikit-Image, Pillow, or OpenCV) to make some patterns, such as closed loops, stand out more.