Cherkaoui Hamza, Corbière Charles

AXA Data Challenge Report

Content table

1. Exploring and visualizing dataset

1.1 Cleaning data and preprocessing

Initial dataset file weight more than 3 Gb, which makes it unrunable, RAM-ly speaking, on our usual computers. First thing is to reduce the size of this file. The submission file only contains 3 columns : DATE, ASS_ASSIGNMENT and CSPL_RECEIVED_CALLS. We concluded that we do not need other columns now from original dataset as they are not included in the submission file. We have code a function to read the raw data and select only the chosen date and columns:

```
def get_raw_data(data_path, date_regex, max_lines=None):
```

Then the idea is to `groupby` by 30 minutes period and sum the number of calls. We have a specific function for this which take the output of `get_raw_data` and create a proper Pandas DataFrame with as much columns than assignement:

```
def preprocessing(X_df):
```

After this first pass we have three differents files:

```
-rw-rw-r-- 1 hcherkaoui hcherkaoui 2,3M janv.  6 23:49 2011.csv
-rw-rw-r-- 1 hcherkaoui hcherkaoui 2,3M janv.  6 23:49 2012.csv
-rw-rw-r-- 1 hcherkaoui hcherkaoui 1,8M janv.  6 23:49 2013.csv
```

As you can see those files are much lighter, but still holding all the necesseray information. For each file the corresponding DataFrame is:

index Dates	CMS	Crises	Domicile	...	Tech. Total
datetime 1	x	x	x	...	x

where x hold for the number of CSPL_RECEIVED_CALLS for that specific datetime period.

1.2 Generate statistics

Before starting prediction, we want to get a first impression on the evolution. To do so, we have compute some statistic for the assignement:

- max call per assignement
- min per assignement
- mean per assignement
- 80% percentile per assignement
- 90% percentile per assignement
- variance per assignement

Those statistics were produced along three time windows: Years, Months and Days. We also produce separated statistics regarding day or night period. Finally, we end up by producing 18 files. Each refer to a DataFrame index by the time period and with 28 * 6 = 168 columns: 6 statistics per assignements.

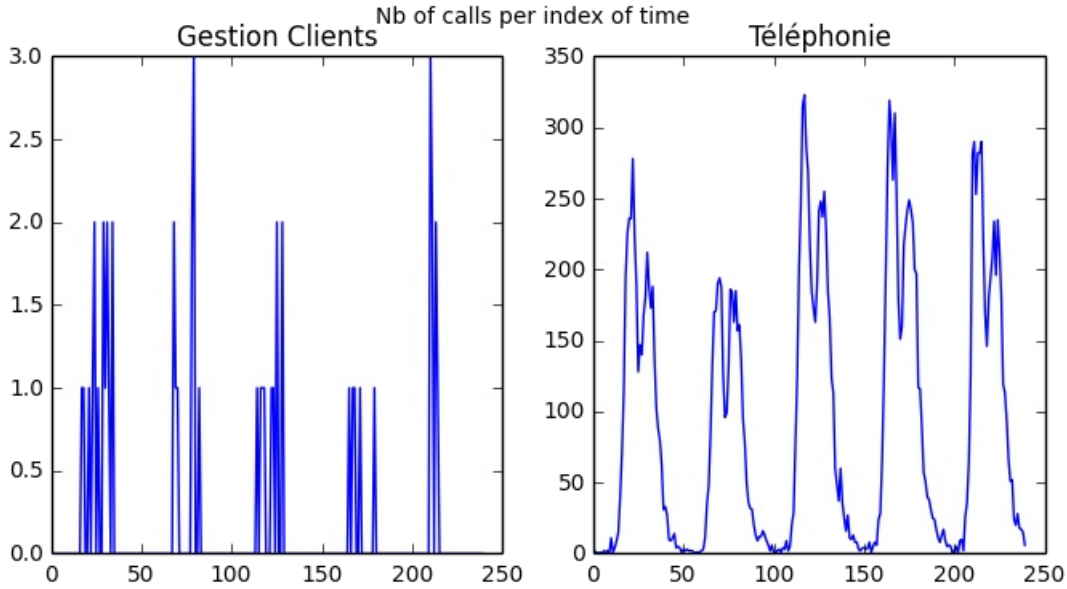
1.3 Visualization

Finding a pattern

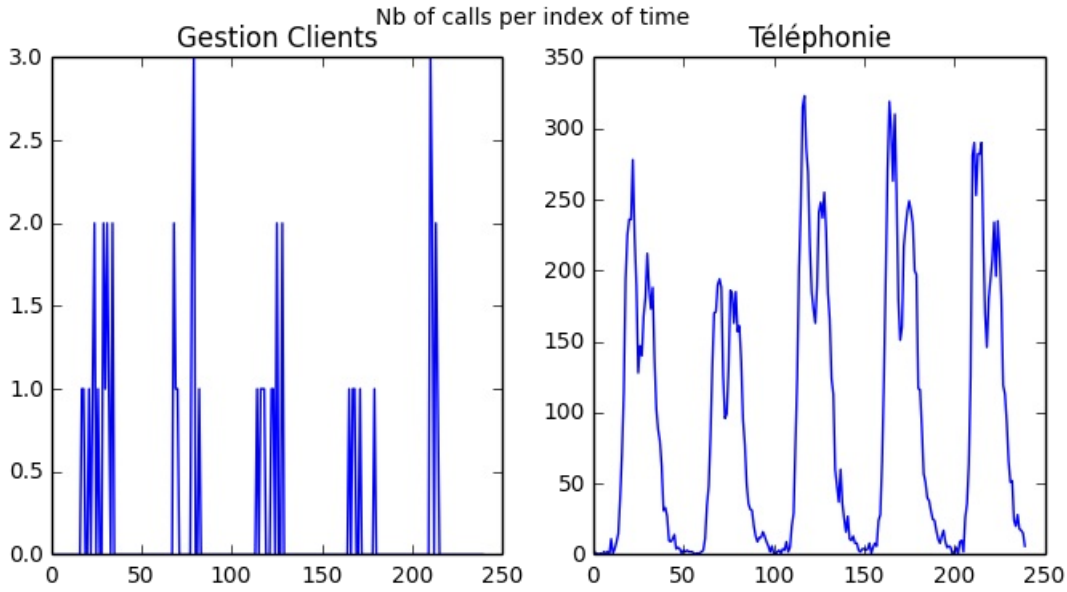
Let's first plot 2013 raw data for two chosen assignements: Gestion Clients and Téléphonie. In our code, the function used is

```
def plot_2013_raw_data(assignements, type_periods, n_periods=5):
```

Here's a plot by week:



And here's a plot by month:

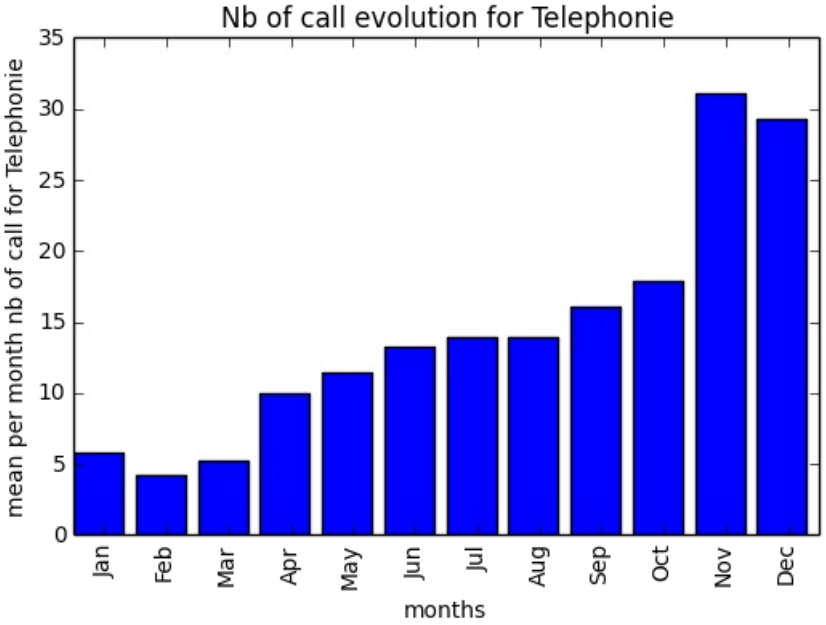


- For 'Téléphonie', we can see that the calls follows a logical pattern. During the night there are less calls, as for the week end. And on each Monday morning we have the week highest activity. We can also observe that there is a slight tendency to decrease through the week.
- For 'Gestions clients', we have a more difficult pattern, we do not find the same week pattern as in 'Téléphonie'.

Those two examples were chosen to underline the fact that each assignements have a specific behaviour and order of magnitude. Even if most of assignements look like 'Gestion Clients', we have differents pattern.

Year-to-Year evolution

Let's plot for example the evolution of the number of received call for Téléphonie during the year 2011:



Even during a year, some assignement average heavily increasing. At the end of the year, we have in average a 500% increasing for 'Téléphonie'.
Those obsevation were also done for all the assignements, all years. And we noticed that heavy increasing are commons.

Temporary conclusions

Based on our previous plots here, and the exploration of the data we made we can make several conclusion:

- time evolution is not smooth at all, meaning all the technics based on functional analysis, derivation, etc..., should be useless here.
- we observe a huge year-o-year and month-to-month evolution that are vitals to catch during our prediction.
- For each assignement, even if the mean change, we have a periodic-stable behaviour with noise. Our predictor will then be able to learn at least this behaviour from the previous month (for example).

To conclude, the main objectif of the challenge will be to capture the random law type of the number of received calls and also its mean (which will evolve through the year) for each assignement. We can think of this problem as each assignement follow a Gaussian distribution and the variance and the mean evolve throught the year.

2. Prediction

2.1 First dummy submission and LinExp loss consideration

For our first submission, we wanted to test our pipline and see how the testing serveur responds.

We chose to take only one mean, here's the mean on 2012 for all assignement and use it for every assignements, just to test. Our score were to big to be save... And we then chose to take **half of the maximum means on all assignements** for 2012 and use it as a number of calls prediction for all the assignements. Our score were about a hundred.

What we can conclude about this is that the LinExp is very sensible to under-prediction. For example, for 'Téléphonie' our guess is about hundred of calls. Suppose you miss 100 calls, then you will pay it $e^{(0.1*100)} = 22026$... I means that we will have to **over estimate** a lot.

2.2 Regressor on raw data

Our second idea was to try directly a predictor on the raw data. Meaning, giving 'something' that sum up the date, predict a specific assignment number of calls. We simply choose to do regression on the feature 'epoch':

```
epoch = X_df.index.astype(int)
```

It was quite brutal approch and despite trying many differents regressors and hyperparameters we couldn't improve our score. We added then a few other time columns as Night/Day or Weekend but not enough improvement to notice.

2.3 Interpolation and repeating the past

After those first tries, we use a more manual approch. Because of the assignement evolution not being smooth, we couldn't try usual 'interpolling the past' to predict. Yet, because the behaviour repeat itself for each assignement one can think of just **repeating the past** with a ground base correction.
Base on this idea we developped two predictors:

```
def pred_same_last_week(dates, X_df):

def pred_same_last_year(dates, X_df):
```

Both are based on:

```
def get_last(date, X_df, period_type, nlast=1, nbtry=4):
```

In order to test it, we have directly submit those two predictors and as expected they behave wrongly since they do not take account of the ground base evolution, `pred_same_last_year` being the best.
So, we try to take advantage of all the statistics that we have previously produce, the `datasets/` directory, but we had not better result.

Actually there were one thing missing from our exploration conclusion: catching the evolution rate from year to year or week to week. In order to address it, we choose to produce those two functions:

```
def pred_reg_two_weeks(dates, X_df):

def pred_reg_two_years(dates, X_df):
```

Using only of them separatly gave us bad results aswell. The idea now was to mix all those predictions together in a clever way. At first try, we average all the predictions but it gives us bad results since the predictors do not try to over-estimate the prediction (cf 2.1). So we try different 'mergers': the 90% percentile, the 80% percentile, the maximum. It ends that the best way of merging was taking the **maximum of the four prediction**.

2.4 A clever prediction merging

At this point we realize that we could simply **replace the merger by a learner**. Indeed, we can guess that our four "first pass" predictor produce differents errors per assignement. So, we update our code such that we fit on all the dates (except the one that we want to predict and a week before) with the output of the four predictors and the epoch feature. Thank to that approch we pass under the 0.6 linexp error. At this point we also refactor the code to have a solid and clean base of code. We have different Python files and the prediction are made by a `Regressor` class. Our regressor call the `GradientBoostingRegressor` from Scikit-Learn with `n_estimators=1000` and `learning_rate=0.2`.

```
class Regressor(BaseEstimator):
    def __init__(self):

    def _fit_simple_predictors_(self, dates):

    def preprocess_data(self, df, full=False):

    def fit(self, dates):

    def predict(self, dates):
```

2.5 Tuning the final pipline

We have try several hyperparameters for our regressor, and to force the over-estimation we also multiplie our prediction by a coefficient (the first pass and the merged prediction). We end up by tuning **two coefficients and the `GradientBoostingRegressor` hyperparameters**. This allow us to make our best prediction. Yet we suppose that at some point we have ovefitted on the test datasets, which is recommendable.

3. Appends

3.1 Bug on the server

We have lost almost **13h** in finding what appear to us as a bug: We produce such submission file.

```
DATE    ASS_ASSIGNMENT  prediction
2012-12-28 00:00:00 CMS     0.0
```

Meaning that we do not put the millisecond since it's the default Pandas behaviour.
We are not sure why it's important to refuse a submission that do not display the millisecond. We cast this behaviour as a bug. Anyway the fact that the server only print a maximum error when the submission failed lure us for 13h... At some point we understand that it was not our predictor that create a huge error base on the LinExp loss function but it was 'only' a "Wrong file format error"... The server should have been more wordy about that. So since that we always manually corrected our submission file.

3.2 Local error: an Hold-Out and a V-fold script try

We have try to create a local estimation error. First we have try to compute a Hold-Out error estimation. We try to predict a week before what was ask on the server. The code run smoothly, yet we it was about to estimate the error a 0.36 server error was estimate at $10^3 6$... we have no clue why since expression of the LinExp loss function is not that difficult. We have also try to make a V-fold function. But again the error estimation was huge..

3.3 Tuning parameters: an `grid_search` function try

In parallel of our effort on the error estimation we code a `grid_search` function hoping that when the error estimation will be solve we will be able to tune up our hyperparameters. But since the estimation error is still there, the `grid_search` function never work. This grid search is implemented in our `script_gridsearch.py` script. It is based on the function implemented in scikit-learn.

4. Conclusion

The main difficulty for this challenge was to made a robust code base to handle differents points:

- a cleaning and a preprocessing step of the data
- differents layers of predictors, and the ability to easily update/change it
- a script to create a proper submission file (almost working at the end)
- (optional) a script to estimate the error locally (not working...)
- (option) a script to make `grid_search`(not working...)

The main problem for us was that we were **not able to estimate the error locall** (and so to do `grid_search`). It would have been perfect if we were able to validate a new technique locally and massively search the best hyperparameters. But instead we slowly try different technique.
The second problem was that it seems to have many **random noise on the number of calls**. Which mean that even with many efforts this can't be 'catch' by our learner.
The last 'problem' was the loss function, the **slightest error make the score increase dreadfully**. But, also, it was one of the interess of the challenge.