INF442 Project : Hacking the Paris Metro

Stanislas DOZIAS, Xavier DURAND

1 Introduction

Our project is about the RATP traffic in Paris. Due to timing issues, we couldn't have access to the real time data. Furthermore a lot of files was quite hard to understand, and so we focalised our project on one task : try to anticipate, via a regression, the effect of the Covid on the frequencies of the traffic hour by hour.

For that we did first a preprocessing of the data included in "historical.csv" (which contains some data before the pandemic in 2019) and "stop_times.csv", containing some data in the pandemic in 2021. After that we retook the TD7 in C++ about Linear Regression to make some predictions, and we compared it with the results obtained by some more sophisticated models using the python library : SkLearn.

We also used github to share our code : https://github.com/StanislasDOZIAS/INF442/.

Contents

2 Preprocessing

2.1 Steps of the preprocessing

The easiest way to do prepossessing was to use the library Pandas from Python. For further details, please open the notebook "preprocessing.ipynb".

Here are the steps of our preprocessing :

- first we deleted the non-useful data : we kept only the arrival times (which had to be converted into numerical numbers) of each trains and its station

- for both "historical.csv" and "stop_times.csv', we calculated the frequencies at each hour of the day per station.
- at the end we joined both datasets on the stations (there are only 92 stations in common, which is quite low but good enough), before deleting the columns of the stations, to have only numbers is the dataset when we exported it into .csv.

We have however one comment about the way we calculated the frequencies. For "historical.csv", we have the true frequencies because we know well the total day range, and it is the same for all stations. For "stop_times" it is more complicated, because there is no date in the data. But by doing some joining with other dataset, we can calculated a number of days. However depending on which stations we consider, this number changes, so we cannot really have the real frequencies (and if we take the largest day range, we have a factor 10 or even more between the frequencies of "historical.csv" and these, which is quite too much). So at the end we end up by making that the frequencies will be at the same magnitude that "historical.csv" by putting $number\_of\_days$ to 10. This is not the reality and we know it, but because it is a project and there will be no application in real life, we think it is a good decision.

2.2 Final format of the dataset

The format of the dataset at the end is like this : 48 columns, separated in two categories :

- $X = (X_i)_{0 \leqslant i \leqslant 23}$, which are the frequencies at each our before the pandemic
- $Y = (Y_i)_{0 \leqslant i \leqslant 23}$, which are the frequencies at each our during the pandemic

Each row of the dataset represent a station amoung the 92 that we have.

3 Linear Regression in C++

3.1 Presentation of the models

We tried 3 differents models in the Linear Regression. Let's say that $d = 24$ (the number of frequencies in one day) and n the number of samples. We search A so that $Y_{pred} = XA$ is close to Y, with $Y \in \mathcal{M}_{n,d}(\mathbb{R})$, $A \in \mathcal{M}_{d+1,d}(\mathbb{R})$ and $X \in \mathcal{M}_{n,d+1}(\mathbb{R})$ (we add one column to X for the bias)

- In a first approach we can assume that $A = \begin{pmatrix} 0 & \\ \vdots & aI_d \\ 0 & \end{pmatrix}$ with $a \in \mathbb{R}$. that would mean that the effect of the pandemic is the same and consists to have just a fraction of the traffic (for example they are all divided by 2 and we have $a = 0.5$). To compute is we just make compare the both means of all frequencies and put $a = \dfrac{mean(Y)}{mean(X)}$
- In a second approach we assume that $A = \begin{pmatrix} c_1 & d_1 & & \\ \vdots & & \ddots & \\ c_d & & & d_d \end{pmatrix}$ with $(c_1, \ldots, c_d), (d_1, \ldots, d_d) \in \mathbb{R}^d$. That means that each frequency in 2019 influences only the same frequency in 2021. To compute it we had to make 24 regressions with in each one column X_i and a constant column to get c_i and d_i, and then merge the results in the matrix.
- Finally the last model is the more general : A is without any property. That means that the frequencies are inter dependant.

Of course the first will compute faster than the second and the third, but with a lost of precision. The goal of this comparison is to see if it is worth to have a complicated model (the third) or if a simpler does the job and save us some time.

3.2 Expectations

Before analysing the results, let's predict a little what we will have.

For us the first model is too simple and will not fit very good with the data, because the frequencies in the middle of the day are far more impacted by the pandemic than the early morning.

The second model will have a worst training score than the last one (because the model is simpler), but will be better in the test-score, because for us the dependencies between the different frequencies is too little and will only implies over-fitting.

But let's stop prediction and begin the analysis of the results.

3.3 Results

Here you can this the results of the regressions, with for each model the true value and the prediction of the first sample :

- First model :

```
Time of execution = 0.006358

Testing the Estimate method on the first sample of test_file
The true value of y for the first sample of test_file is:
1 0 0 0 0 2 2.6 3.2 3.4 3.3 3.1 3.1 3.1 3.1 3.1 3.1 3 3.4 3.3 3.1 2.2 1.9 1.9 1.9
The estimated value of y for the first sample of test_file is:
0.423415 0.62045 0 0 0 0 1.47566 2.28057 2.85071 3.0184 3.0184 2.81718 2.81718 2.81718 2.81718 2.81718 2.81718 2.61595 3.0184 3.0184 2.61595 1.67689 1.47566 1.29959 1.2409

Sums of Squares wrt training set:
 ESS=2930.88
 RSS=1068.8
 TSS=2823.46
 R2=1.03804
 MSE=14.641
Sums of Squares wrt regression set:
 ESS=28824.3
 RSS=338.33
 TSS=25791.9
 MSE=17.8068
```

- Second model :

```
Time of execution = 0.017234

Testing the Estimate method on the first sample of test_file
The true value of y for the first sample of test_file is:
1 0 0 0 0 2 2.6 3.2 3.4 3.3 3.1 3.1 3.1 3.1 3.1 3.1 3 3.4 3.3 3.1 2.2 1.9 1.9 1.9
The estimated value of y for the first sample of test_file is:
0.86574 0.0120705 0 0 -1.21364e-17 1.92535 2.7634 3.30622 3.44667 3.43884 3.15231 3.08489 3.0727 3.0727 3.07109 3.08925 2.95005 3.46576 3.45071 3.09584 2.08426 1.84566 1.72062 1.67347

Sums of Squares wrt training set:
 ESS=2790.63
 RSS=32.8331
 TSS=2823.46
 R2=0.988371
 MSE=0.449769
Sums of Squares wrt regression set:
 ESS=25684.3
 RSS=6.28665
 TSS=25791.9
 MSE=0.330876
```

- Third model :

```
Testing the Estimate method on the first sample of test_file
The true value of y for the first sample of test_file is:
1 0 0 0 0 2 2.6 3.2 3.4 3.3 3.1 3.1 3.1 3.1 3.1 3.1 3 3.4 3.3 3.1 2.2 1.9 1.9 1.9
The estimated value of y for the first sample of test_file is:
0.842125 0.0843008 0 0 -2.13537e-17 1.85717 2.62767 3.26034 3.44958 3.36564 3.11026 3.08272 3.08509 3.08509 3.08509 3.09293 3.13647 3.44764 3.52895 3.27183 2.14671 1.7937 1.75838 1.76408

Sums of Squares wrt training set:
 ESS=2813.51
 RSS=9.95081
 TSS=2823.46
 R2=0.996476
 MSE=0.136312
Sums of Squares wrt regression set:
 ESS=25680.6
 RSS=2.42357
 TSS=25791.9
 MSE=0.127556
```

To summarize the scores :

- First model : $time = 0.0064s,\ MSE_{train} = 14.641,\ MSE_{test} = 17.807$
- Second model : $time = 0.017s,\ MSE_{train} = 0.450,\ MSE_{test} = 0.331$

- Third model : $time = 0.0095s,\ MSE_{train} = 0.136,\ MSE_{test} = 0.128$

As predicted, the first model is too simple and cannot shape good with the data. But we see that having a full matrix and not a diagonal matrix improve a lot the precision (from 0.45 to 0.128 !) and does not take longer (maybe because our implementation for the diagonal matrix is not optimal). So the last model is the best and we can conclude that there is some dependencies between the frequencies.

4 More sophisticated models with SkLearn

For Further details, please open the notebook "statitics.ipynb". In order to make some comparisons, we built some models with SkLearn.

4.1 Linear

```
model = LinearRegression()
model.fit(X_train, Y_train)

pred = model.predict(X_train)
print("score MSE on the train_set : ",(np.linalg.norm(pred-Y_train)**2/(Y_train.shape[0])))

pred = model.predict(X_test)
print("score MSE on the test_set : ", (np.linalg.norm(pred-Y_test)**2/(Y_test.shape[0])))
```

```
score MSE on the train_set :  0.19172069399492322
score MSE on the test_set :  0.8708515038988245
```

The linear model of SkLearn has a lot more over-fitting than the one that we coded ourselves, which is quite surprising.

4.2 Ridge

The ridge regressor is a linear regressor with a penalty on the L_2 norm of the matrix A.

```
model = MultiOutputRegressor(Ridge(random_state=0))
model.fit(X_train, Y_train)

pred = model.predict(X_train)
print("score MSE on the train_set : ",(np.linalg.norm(pred-Y_train)**2/(Y_train.shape[0])))

pred = model.predict(X_test)
print("score MSE on the test_set : ", (np.linalg.norm(pred-Y_test)**2/(Y_test.shape[0])))
```

```
score MSE on the train_set :  0.22296139320128236
score MSE on the test_set :  0.367754094322364
```

The ridge model is worst on the train set but has much less over-fitting and so has a good score on the test-set. We made a GridSearchCV to optimize the hyperparameters and get sclicy better scores :

```
hyper_params={'estimator__fit_intercept' : [True, False],
              'estimator__alpha' : [0.01,0.1, 1, 10],
              'estimator__tol' : [0.0001,0.001,0.01]
             }


model = GridSearchCV(MultiOutputRegressor(Ridge()), param_grid=hyper_params,
                                          scoring='neg_root_mean_squared_error', cv=5)

model.fit(X_train, Y_train)

print(model.best_params_)


pred = model.predict(X_train)
print("score MSE on the train_set : ",(np.linalg.norm(pred-Y_train)**2/(Y_train.shape[0])))

pred = model.predict(X_test)
print("score MSE on the test_set : ", (np.linalg.norm(pred-Y_test)**2/(Y_test.shape[0])))
```

```
{'estimator__alpha': 1, 'estimator__fit_intercept': False, 'estimator__tol': 0.0001}
score MSE on the train_set :  0.23823728598994234
score MSE on the test_set :  0.3497778488769704
```

4.3 RandomForest

```
model = RandomForestRegressor()
model.fit(X_train, Y_train)

pred = model.predict(X_train)
print("score MSE on the train_set : ",(np.linalg.norm(pred-Y_train)**2/(Y_train.shape[0])))

pred = model.predict(X_test)
print("score MSE on the test_set : ", (np.linalg.norm(pred-Y_test)**2/(Y_test.shape[0])))
```

```
score MSE on the train_set :  0.10078502739725952
score MSE on the test_set :  0.870514947368425
```

The Random Forest model is very good on the training set but has a serious problem of over-fitting, making it as bad as the Linear model. We can reduce it a little with a GridSearchCV, but not so much.

```
hyper_params={'n_estimators' : [200,400,800],
              'min_samples_split': [2,3],
              'max_features' : [0.4,0.8],
              'max_depth' : [4,6,10],
              'min_samples_leaf' : [1,2],
              'bootstrap' : [False, True],
             }


model = GridSearchCV(RandomForestRegressor(n_jobs=1), param_grid=hyper_params,
                                       scoring='neg_root_mean_squared_error', cv=5)

model.fit(X_train, Y_train)

print(model.best_params_)

pred = model.predict(X_train)
print("score MSE on the train_set : ",(np.linalg.norm(pred-Y_train)**2/(Y_train.shape[0])))

pred = model.predict(X_test)
print("score MSE on the test_set : ", (np.linalg.norm(pred-Y_test)**2/(Y_test.shape[0])))
```

```
{'bootstrap': False, 'max_depth': 10, 'max_features': 0.4, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 200}
score MSE on the train_set :  0.003575178322772086
score MSE on the test_set :  0.8391574353654975
```

4.4 SVM

```
model = MultiOutputRegressor(svm.SVR())
model.fit(X_train, Y_train)

pred = model.predict(X_train)
print("score MSE on the train_set : ",(np.linalg.norm(pred-Y_train)**2/(Y_train.shape[0])))

pred = model.predict(X_test)
print("score MSE on the test_set : ", (np.linalg.norm(pred-Y_test)**2/(Y_test.shape[0])))
```

```
score MSE on the train_set :  0.7639282187484986
score MSE on the test_set :  0.9628736535640292
```

SVM is quite bad without GridSearchCV, but almost as good as Ridge with it

```
hyper_params={'estimator__kernel' : ['linear','rbf'],
              'estimator__epsilon' : [0.001,0.01],
              'estimator__C' : [0.01, 0.1,1],
              'estimator__gamma' : ['auto', 'scale']
             }


model = GridSearchCV(MultiOutputRegressor(svm.SVR()), param_grid=hyper_params,
                     scoring='neg_root_mean_squared_error', cv=5)

model.fit(X_train, Y_train)

print(model.best_params_)

pred = model.predict(X_train)
print("score MSE on the train_set : ",(np.linalg.norm(pred-Y_train)**2/(Y_train.shape[0])))

pred = model.predict(X_test)
print("score MSE on the test_set : ", (np.linalg.norm(pred-Y_test)**2/(Y_test.shape[0])))
```

```
{'estimator__C': 0.1, 'estimator__epsilon': 0.01, 'estimator__gamma': 'auto', 'estimator__kernel': 'linear'}
score MSE on the train_set :  0.27652590966572116
score MSE on the test_set :  0.36802575278701727
```

4.5 Gradient Boosting

```
model = MultiOutputRegressor(GradientBoostingRegressor())

model.fit(X_train, Y_train)


pred = model.predict(X_train)
print("score MSE on the train_set : ",(np.linalg.norm(pred-Y_train)**2/(Y_train.shape[0])))

pred = model.predict(X_test)
print("score MSE on the test_set : ", (np.linalg.norm(pred-Y_test)**2/(Y_test.shape[0])))
```

```
score MSE on the train_set :  0.004204198531889425
score MSE on the test_set :  0.7667222740024954
```

Like the Random Forest and the Linear, it performs very good on the train set but can't generalize its performance on the test set. The GridSearchCV upgrades the results a lot, and this is thanks to the choice of the initialization model :

```
hyper_params={'estimator__n_estimators' : [500,1000],
              'estimator__max_depth' : [2,3,4],
              'estimator__learning_rate' : [0.05, 0.1],
              'estimator__subsample' : [0.5,1],
              'estimator__init' : [Ridge(), RandomForestRegressor(), svm.SVR()]
             }



model = GridSearchCV(MultiOutputRegressor(GradientBoostingRegressor(init=Ridge())), param_grid=hyper_params,
                     scoring='neg_root_mean_squared_error', cv=5)

model.fit(X_train, Y_train)

print(model.best_params_)

pred = model.predict(X_train)
print("score MSE on the train_set : ",(np.linalg.norm(pred-Y_train)**2/(Y_train.shape[0])))

pred = model.predict(X_test)
print("score MSE on the test_set : ", (np.linalg.norm(pred-Y_test)**2/(Y_test.shape[0])))
```

```
{'estimator__init': Ridge(), 'estimator__learning_rate': 0.05, 'estimator__max_depth': 2, 'estimator__n_estimators': 500, 'esti
mator__subsample': 0.5}
score MSE on the train_set :  0.006503495029480505
score MSE on the test_set :  0.2740283409694716
```

We see that the best initialization is the Ridge model, which gave us the best results so far. Let's choose as initialization the GridSearchCV of the Ridge model, and then make a GridSearchCV on the hyper parameters of the Gradient Boosting.

```
hyper_params_ridge={'fit_intercept' : [True,False],
              'alpha' : [0.1, 1, 10],
              'tol' : [0.0001,0.01],
              'max_iter' : [100, 200,500]
             }



ridge = GridSearchCV(Ridge(), param_grid=hyper_params_ridge,
                                       scoring='neg_root_mean_squared_error', cv=5)

ridge.fit(X_train, Y_train)

hyper_params={'estimator__n_estimators' : [500,1000],
              'estimator__max_depth' : [2,4],
              'estimator__learning_rate' : [0.05, 0.1],
              'estimator__subsample' : [0.5,1],
              'estimator__min_samples_leaf' : [1,2]
             }


model = GridSearchCV(MultiOutputRegressor(GradientBoostingRegressor(init=ridge)), param_grid=hyper_params,
                     scoring='neg_root_mean_squared_error', cv=5)
model.fit(X_train, Y_train)

print(model.best_params_)

pred = model.predict(X_train)
print("score MSE on the train_set : ",(np.linalg.norm(pred-Y_train)**2/(Y_train.shape[0])))

pred = model.predict(X_test)
print("score MSE on the test_set : ", (np.linalg.norm(pred-Y_test)**2/(Y_test.shape[0])))
```

```
{'estimator__learning_rate': 0.05, 'estimator__max_depth': 4, 'estimator__min_samples_leaf': 1, 'estimator__n_estimators': 500,
'estimator__subsample': 0.5}
score MSE on the train_set :  7.559931152345061e-05
score MSE on the test_set :  0.2556659574055599
```

Unfortunately we only update our score by a little.

4.6 Conclusion

To conclude we are pretty surprised that all the model tested with SkLearn are not better that the Linear Regression made in C++, which has a MSE on the test set twice lower as the best model. We can also note that the computation of the linear model on C++ is very fast, and the training of the model with SkLearn could take a while (especially with GridSearchCV which can take 30min).

The fact that our models have so much over-fitting is certainly due to the lack of data (the train set has 73 stations and the test set only 19).

About the precision itself of the frequencies : having a 0.128 MSE score is very good, because the quantities that we predict are of the order of the unity, so it's a 10% error (approximately), which is reasonable for the service of the RATP.

We can conclude that the stations are globally impacted in the same way by the pandemic, because we can make quite good predictions from some stations to other stations.