AMD-SM2L joint project

Mattia Ferraretto [00072A]

Academic year 2021-2022

Contents

1 Requirements

The project is based on the analysis of the "Airline Delay and Cancellation Data, 2009 - 2018" dataset published on Kaggle. The task is to implement from scratch (without using libraries such as Scikit-learn) a classifier based on logistic regression and to train it to predict whether or not a flight will be canceled, only on the basis of information available at flight departure.

Important: the techniques used in order to infer the classifier should be time and space efficient, and scale up to larger datasets.[1]

2 Data considered

"Airline Delay and Cancellation Data, 2009 - 2018" dataset contains information about domestic flights in the United States of America between 2009 and 2018. The dataset is divided into multiple files, one for each year. Each one, stores a several number of features such as: the date of flight, airline identifier, flight number, origin, destination and many other data. For my experiments, as required, I only used basic information on departures: date of flight (`FL_DATE`), origin (`ORIGIN`), destination (`DEST`), planned departure time (`CRS_DEP_TIME`), actual departure time (`DEP_TIME`), total delay on departure in minutes (`DEP_TIME`) and flight cancelled (`CANCELLED`). I didn't use the entire dataset, in fact, I utilized for the analysis records between 2014 and 2018.

3 Data pre-processing

In this type of analysis, data must be pre-processed to obtain consistent results. To clean data I performed the following steps:

1. transforming non numeric into numeric features (such like `ORIGIN` and `DEST`): these two features column were concatenated into one, dropped duplicates, and then transformed into dictionary where key is the token and value a unique numeric value. Subsequently this map was used to replace non numeric into numeric values.

2. balancing the dataset: it is clear that the dataset has many examples which belong to the majority class (departed flights). In most cases balancing the dataset is important because, in this way, model weights aren't too much affected by the majority class, obtaining better results during classification. So, to balance the dataset, I used the undersampling balancing technique, in particular, I defined an undersampling factor that indicates how much the majority class must be big (e.g. with `undersampling_fact=0.6` the resulting datasets is composed by 60% samples belonging to the majority class and 40% samples belonging to the minority class).

3. transforming date into day of year (`FL_DATE`): in my opinion, date of flight is an important feature to reach our goal, but maybe year period in which

the airplane has taken off is more important than date itself. Considering different years, a specific date would be meaningless. Then, I decided to transform it into the day of year.

4. filling NaN values: during data analysis, I noticed, where a flight was cancelled that, clearly, there is no value for the actual departure time (`DEP_TIME`) feature. Then, there is no value also for total delay on departure (`DEP_DELAY`) because it is the subtraction results between planned departure time and actual departure time features. So, to face the problem I replaced NaN values in the actual departure time with planned departure time value and total delay on departure with 10080 (number of minutes in a week). The basic idea behind this approach is that when an airplane is ready to take off, in most cases it departs in advance or on time. This implies that when a flight has a very big delay (like one week), it never took off.
   Last, I replaced NaN values with 0 in `DEP_DELAY` where the flight has taken off on time.

Scaling up with large datasets In my proposed solution to scale up with large datasets, I take the advantage of the fact that our dataset has already been divided into multiple files. Therefore, files are loaded one at time (with the only features of interest), pre-processed, and then each feature is down-cast to the better fixed representation. What I mean is that it doesn't make sense to use 32-bit integer to represent the day of year, because values are between 1 and 365, also for planned departure and actual departure time where values are between 1 and 2400. So, likewise, is valid for all the other features. In this way we can save space to load more data. Finally, all chunks are concatenated into a single data frame.
In my opinion, there are at least two different solution that we can apply to scale up with large datasets. The first one, for example, might be to load data during the training, fit the model weights with newly loaded data and then load the consecutive data batch. The second solution, useful when your dataset is too much big to handle it with simple files, may be to use a different tool such as a relational database, in which the entire dataset is stored. In this way, fixed a batch data size, we can use a simple DBMS connector in python to fetch data from it and as well as explained in the first solution, use data to fit model weights, then fetch the subsequent data batch.[2]

4 Considered algorithms

In some application domains, such as weather prediction, a probability is preferred instead of a binary prediction. In this case our goal is learning the function $\eta(x) = \mathbb{P}(Y = 1 \mid X = x)$ in a binary classification problem. In logistic regression we train a predictor $g : X \to \mathbb{R}$ and then use $\sigma(g(x))$ where $\sigma : \mathbb{R} \to \mathbb{R}$ is defined by

$$\sigma(z) = \frac{1}{1 + e^{-z}} \in (0, 1)$$

In the experiment I computed the error with the following logarithmic loss:

$$\mathcal{L}(y, \hat{y}) = \mathbb{I}\{y = +1\} \ln \frac{1}{\hat{y}} + \mathbb{I}\{y = -1\} \ln \frac{1}{1 - \hat{y}}$$

Then, we can rewrite it as function known as logistic loss,

$$\mathcal{L}(y, \hat{y}) = \ln(1 + e^{-y\hat{y}}), \quad \hat{y} = g(x)$$

Assuming now that $g(x)$ is a linear model $w^T x$. Given a training set $S = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, let $\mathcal{L}(w) = \ln(1 + e^{-yw^T x})$ and $s = w^T x$, we compute the gradient as follows

$$\begin{aligned} \nabla \mathcal{L}(w) &= \left(\frac{d}{ds} \ln(1 + e^{-ys}) \Big|_{s = w^T x} \right) x \\ &= \frac{-y}{e^{ys} + 1} x \\ &= -y\sigma(-yw^T x)x \end{aligned} \tag{1}$$

Therefore, the gradient descent update can be written as

$$w_{t+1} = w_t + \eta_t \sigma(-y_t w^T x_t) y_t x_t$$

where η_t is the learning rate.[3]
Alternative approach considered in my proposed solution, was the mini-batch stochastic gradient descent. In this way, we neither consider all training examples nor single example (such as SGD) at each step, but we consider a subset of examples of fixed size at each step. In my opinion this technique might be the right trade-off when we want to scale our solution with large datasets.

I considered to evaluate a learning algorithm using the external cross validation. First of all, we assume that our hyperparameter θ is fixed. Then, let S the entire dataset, we partition S in K subsets $D_1, \ldots, D_K$ where each has dimension m/K. Now let $S^{(i)} \equiv S \setminus D_i$, we call D_i the testing part of the i-th fold while $S^{(i)}$ is the training part.
Cross validation estimates $\mathbb{E}[\mathcal{L}_{\mathcal{D}}(A)]$ on S (where A it the learning algorithm). CV is computed as follows: we run A on each training part $S^{(i)}$ of the folds

$i = 1, \ldots, K$ and obtain the predictors $h_1 = A(S^{(1)}), \ldots, h_k = A(S^{(K)})$. Then, we compute the (rescaled) error on the testing part of each fold

$$\mathcal{L}_{\mathcal{D}}(h_i) = \frac{K}{m} \sum_{(x,y) \in D_i} \mathcal{L}(y, h_i(x))$$

Finally, we compute the CV estimate by averaging these errors

$$\mathcal{L}_S^{CV}(A) = \frac{1}{K} \sum_{i=1}^{K} \mathcal{L}_{D_i}(h_i)$$

[4]. In my implementation I saved the predictor that minimizes the error.

Another problem faced in the experiment was how to tune hyperparameters to obtain a predictor with small risk. Given Θ as a set of all possible hyperparameters, which could be very large also infinite, risk minimization is generally done over a suitably chosen subset $\Theta_0 \subset \Theta$. If S is our training set, then we want to find $\theta^* \in \Theta_0$ such that

$$\mathcal{L}_{\mathcal{D}}(A_{\theta^*}(S)) = \min_{\theta \in \Theta_0} \mathcal{L}_{\mathcal{D}}(A_\theta(S))$$

So, to estimate

$$\mathbb{E}\left[\min_{\theta \in \Theta_0} \mathcal{L}_{\mathcal{D}}(A_\theta)\right]$$

I considered two possible solutions: the first one is to use the best CV over $\{A_\theta : \theta \in \Theta_0\}$,

$$\min_{\theta \in \Theta_0} \mathcal{L}_S^{CV}(A_\theta)$$

and the second one is applying the nested cross validation algorithm:

Require: Dataset S
  Split S into folds $D_1, \ldots, D_K$
  for $i = 1, \ldots, K$ **do**
    Compute training part of i-th fold: $S^{(i)} \equiv S \setminus D_i$
    Run CV on $S^{(i)}$ for each $\theta \in \Theta_0$ and find $\theta_i = \text{argmin}_{\theta \in \Theta_0} \mathcal{L}_{S^{(i)}}^{CV}(A_\theta)$

    Re-train A_{θ_i} on $S^{(i)}$: $h_i = A_{\theta_i}(S^{(i)})$
    Compute error of i-th fold: $\varepsilon_i = \mathcal{L}_{D_i}(h_i)$
  end for
  return $(\varepsilon_1 + \ldots + \varepsilon_K)/K$

---

[4]

Finally, I implemented a dynamic learning rate. The basic idea was to have a higher learning rate at the beginning and then slowly decrease it over the course

of training. Therefore, given a training set S with size m, the learning rate is adjusted at each step as follows

$$\eta_t = \frac{\eta_t}{0.01 \cdot \sqrt[6]{t}}, \quad t = 1, \ldots, m$$

To evaluate the model performance I used the following metrics:

- **Precision:** is the fraction of relevant instances among the retrieved instances and is defined as follows

$$Precision = \frac{TP}{TP + FP}$$

- **Recall:** is the fraction of relevant instances that were retrieved and is defined as follows

$$Recall = \frac{TP}{TP + FN}$$

- $\mathbf{F_1}$ **score:** is the harmonic mean of precision and recall and is defined as follows

$$F_1 \; score = \frac{2 \cdot TP}{2 \cdot TP + FP + FN}$$

- **ROC:** is a curve created by plotting true positive rate against false positive rate at various threshold settings. The area under the curve is the main parameter in determining whether the model performs well: the higher the value, the better the model performs.

5 Description of the experiment

In this section I will describe experiments performed in the analysis. Before, I want to introduce you the first common steps necessary to run experiments:

1. load and pre-processes data
2. compute average and variance of each features
3. normalize data

Experiment 1 The first experiment consist of building a logistic regression model on the data using specific hyperparameters. In this case, average, variance and training arguments must be specified during the model initialization. Next, the dataset is divided into train and test set. At this point, the fit function is used to tune model weights in a proper manner. Finally, model scores and curves (e.g. training and validation loss, ROC curve) are shown.

Experiment 2 In the second experiment, a logistic regression model was built using cross validation algorithm. Again, average, variance and training arguments must be specified at the beginning. After that, the cross validation function is called. Now, model performance is evaluated by loading additional data, then, scores are shown.

Experiment 3 The goal of the third experiment is to find the best hyperparameters given a well-chosen subset of them. In this instance we use the best CV method. This one is less computationally expensive than other techniques and tends to underestimate, but generally the difference is small [4]. At the end, model results are valuated with additional data and scores are shown.

Experiment 4 In the last experiment, I tried to find the best hyperparameter through the nested cross-validation technique. Even in this case, the algorithm needs to take as input a subset of appropriately chosen hyperparameters. Again, model performance is evaluated trough additional data and scores are shown.

6 Comment on the experiment results

In this section I will show you experiments results. Firs of all, I want to describe the environment in which the whole analysis was performed. Then, all attempts was done in Jupyter Lab 3.2.9 with Python 3.8.3 on a Macbook Pro 2018 (i7 quad-core 2.7 GHz and 16 GB of RAM) running macOS Big Sur 11.6.6. The data considered for training (and evaluation in first experiment) were loaded from `2017.csv.zip` and `2018.csv.zip` with an undersampling factor of 0.65, while for evaluation, additional data were loaded from `2015.csv.zip` and `2016.csv.zip` in the next experiments with an undersampling factor of 1.

Experiment 1 In the first experiment, I built a logistic regression model with ten different hyperparameter tuples (epochs, learning rate, batch size). The point is to show the difference between using mini-batch stochastic gradient descent and gradient descent. Table 1 below shows the performance of the models according to the evaluation metrics defined in the section 4

#	Hypers. (eps, lr, bs)	Precision	Recall	F_1 score	ROC	Time (s)
1	150, 0.01, 2048	0.7393	0.9666	0.8378	0.9787	110.57
2	150, 0.02, 2048	0.9519	1.0	0.9753	1.0	104.52
3	150, 0.02, 4096	0.9767	1.0	0.9882	1.0	51.91
4	250, 0.02, 4096	0.9978	1.0	0.9989	1.0	103.28
5	300, 0.031, 8192	1.0	1.0	1.0	1.0	59.76
6	300, 0.031, 0	0.4477	0.7771	0.5681	0.7975	7.66
7	600, 0.031, 0	0.4758	0.8097	0.5994	0.8472	15.22
8	1200, 0.038, 0	0.6129	0.858	0.715	0.9218	29.39
9	1800, 0.04, 0	0.7395	0.8804	0.8039	0.947	46.58
10	2400, 0.038, 0	1.0	1.0	1.0	1.0	65.21

Table 1: Model performance

#	Training loss	Validation loss	#	Training loss	Validation loss
1	0.5701	0.5117	6	2.4427	2.4315
2	0.4893	0.4865	7	2.3977	2.4091
3	0.4643	0.4687	8	1.1192	1.1187
4	0.3939	0.3973	9	0.6567	0.6426
5	0.2708	0.2710	10	7.0632e-05	7.0937e-05

Table 2: losses

As can be seen, the first five attempts were done using mini-batch stochastic gradient descent (MBSGD), while the last five were done with gradient descent (because the batch size was set to 0). Focusing only on the first five, we notice

that increasing learning rate, model performance improves (#1 and #2) while losses decrease. Then, looking at attempts #2 and #3, in which numbers of epochs and learning rate were remained the same, we find out that model results are improving and computational time has been reduced by 51%. A small decrease in losses is also observed in this case. Fixing learning rate and batch size, but increasing number of epochs (attempts #3 and #4) we obtain a little bit improvement in model results, while losses were diminished significantly. In the last attempt, with mini-batch stochastic gradient descent (#5), all hyperparameters values were increased and as we can see from the tables above, we have got the best classifier.
Now, talking about gradient descent technique (attempts from #6 to #10), we can say that increasing hyperparameters values model performance improves, losses values decrease and computational time increases.

Comparing the proposed techniques, it can be seen that with mini-batch stochastic gradient descent we reach convergence with only 300 epochs, losses are not very close to 0 and we can control computational time defining the batch size. On the other hand, with gradient descent approach, to reach the convergence we have to see our training examples at least 2400 times, we cannot control the computational times, but with this technique losses are very close to 0.

In my opinion, the best model obtained is #5, for two main reasons: the first one is that we got the best classifier with hyperparameters values relatively low (in general with MBSGD, model performance are good also with lower hyperparameters values), and the second one is that using MBSGD might be the right trade-off if you want to scale up our solution with large datasets obtaining acceptable computational time.

Experiment 2 In the second experiment I performed cross-validation with different values of K (number of folds). In this experiment, hyperparameter values were set to: `epochs=300`, `learning_rate=0.031` and `batch_size=8192` (best predictor found). Results are shown in the following tables.

#	K	Precision	Recall	F_1 score	ROC	Time (s)	CV loss
1	2	0.9911	1.0	0.9956	1.0	3.50	0.3309
2	4	1.0	1.0	1.0	1.0	9.86	0.2711
3	8	1.0	1.0	1.0	1.0	25.26	0.2489
4	10	1.0	1.0	1.0	1.0	32.58	0.2452
5	12	1.0	1.0	1.0	1.0	42.73	0.2426

Table 3: Cross validation results

In the Table 3 we can observe that with hyperparameters fixed and increasing number of folds (K), the cross-validation loss decreases, computational time increases, while models performances remain very similar at each attempt. So, the right trade-off between models results and computational time

might be the attempt #3. In conclusion, we can say that setting `epochs=300`, `learning_rate=0.031` and `batch_size=8192` is a good choice for our learning problem.

Experiment 3 Third experiment tries to build the best classifier (trough best cross validation method) with `K=8` and the following hyperparameters lists:

- `epochs=[150, 250, 300]`
- `learning_rate=[0.01, 0.02, 0.31]`
- `batch_size=[2048, 4096, 8192]`

Hypers. (eps, lr, bs)	Precision	Recall	F_1 score	ROC	Time (s)	Loss
300, 0.031, 8192	1.0	1.0	1.0	1.0	612.83	0.4006

Table 4: Best cross validation result

In the table below (4) is shown the best classifier found by the BCV method. From table, we find out that time has increased a lots. Again, we can reconfirm our hyperparameters choice although the loss is not close to 0.

Experiment 4 Last experiment consist of building the model using nested cross validation with `K=8` and the following hyperparameters lists:

- `epochs=[150, 250, 300]`
- `learning_rate=[0.01, 0.02, 0.31]`
- `batch_size=[2048, 4096, 8192]`

In the table below results are shown.

Hypers. (eps, lr, bs)	Precision	Recall	F_1 score	ROC	Time (s)	Loss
150, 0.02, 4096	1.0	1.0	1.0	1.0	4306.35	0.2489

Table 5: Nested cross validation result

Looking at results of nested cross-validation, the first thing you notice is the computational time. Respect to the best cross-validation method (under the same conditions) time has increased by 603%. Focusing now on the loss, we see that in the last experiment loss is decreases significantly, which means that best CV method tends to underestimate as expected. Another interesting and unexpected result is that nested cross-validation method has found `epochs=150`, `learning_rate=0.02` and `batch_size=4096` as the best values for the hyperparameters, maybe due to the random split of the training set.

7 Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

References

[1] "Algorithms for massive datasets", "Statistical methods for ML" joint project, 2021 - 2022.

[2] Pandas documentation, Scaling to large datasets, https://pandas.pydata.org/docs/user_guide/scale.html

[3] Nicolò Cesa-Bianchi, Logistic regression and surrogate loss functions, https://cesa-bianchi.di.unimi.it/MSA/Notes/surrogate.pdf, June 11, 2022.

[4] Nicolò Cesa-Bianchi, Hyperparameter tuning and risk estimates, https://cesa-bianchi.di.unimi.it/MSA/Notes/crossVal.pdf, March 31, 2022.