RAMP project: Predicting Air Passenger Traffic

Team IP-Rion - Errajraji Aya & El Alami Khouloud

Final Submission - AK–212

Ecole Polytechnique

Fall 2020

Python for Data Science

December 4th, 2020

Introduction

This project is part of a collaborative data science challenge on the RAMP platform. The goal consists of expanding the limited dataset, which was donated by an unnamed airline company handling flight ticket reservations all across the U.S, and building a machine learning model to predict the number of air passengers. The prediction of air passengers at a given time will enable to efficiently allocate planes amongst possible routes. This was achieved by first adding external data to the original dataset, then exploring several ensemble methods to achieve high predictive performance. Finally, the quality of our model was measured by the root mean square error (RMSE), which we attempted to minimize by manipulating our data, trying out several machine learning models and tuning their parameters.

I. Data Processing

The original data includes the following features: *date of departure,* airports of *departure* and *arrival,* the mean as well as the standard deviation of the number of weeks between the reservation and the departure date denoted respectively by *WeeksToDeparture* and *std_wtd,* and finally the variable of interest *log_Pax*, which is related to the number of passengers. The starting kit also provided us with additional data pertaining to weather conditions, which we merged to the original one.

Spatiotemporal Features

After inspecting the *requirements.txt* document, we identified the *'geopy'* and *'holidays'* libraries to be useful for creating additional variables. Therefore, we imported the airports' **latitude** and **longitude** from The Global Airport Database to retrieve their geolocations.

We then used the *holidays* library to create a new covariate indicating whether the flight date falls within a **state holiday** or not, as people are more susceptible to travel during their time off. After establishing that Christmas, New Year's and Thanksgiving were the most important holidays nationwide, we created a new variable to identify their timeframe. Considering the fact that people tend to take days off before or after a holiday to extend their vacations, we also created another variable to describe the **proximity** between the flight date and the closest state holiday. Moreover, we also deemed **weekends** to be a considerable factor in people's decision to travel. We thus added a covariate defining whether the flight date is on the weekend or not, including Friday as people are likely to travel right after work or school. Finally, we believe **summer** to be the season of predilection for travelling. In this regard, we created a new variable defining whether the flight date falls within the hot months or not.

Economic and Logistic Features

In our effort to determine the elements that influence the number of passengers during a flight, we identified airport traffic and GDP to be impactful. A new variable **traffic** was added, giving more weight to the busiest airports with the largest number of passengers per year. We also took into consideration state **GDP** measured at PPP, as it provides information on the purchasing power of the population and thus, the ability of travellers to afford plane tickets.

Data Merger

We merged the general dataset that we created with the original data in such a way that we were able to access all the information related to the airports of departure and arrival of each flight. Once the merger was done, we removed the duplicate columns that were identical for departure and arrival. Then, we applied geopy's *distance* function to compute flight **distance** based on the geolocation of departure and arrival airports, as people are more likely to travel by plane over long distances.

Feature Interaction

Interaction effects occur when the effect of one variable depends on the value of another one, which is the case in our final dataset given the fact that it contains the same features for both airports of departure and arrival. Consequently, closer attention was given to the *interaction* between the departure and arrival versions of the same variable, which is mathematically defined as their product. From there, we created a new covariate describing the interaction between traffic volume of both airports, as a larger number of passengers tend to be concentrated in flights where both the departure and arrival airports are busy. We did the same thing for the other variables but not all the interactions were actually relevant. For instance, we had taken into consideration interactions between weather variables, but they resulted in a lower performance of our model. Also, the interaction between the coordinates of each airport was already represented by the computed distance between them. Finally, the most significant interaction effects turned out to be the ones for the *Traffic, GDP, is_State_Holiday* and *DistanceToClosestHoliday* covariates.

Data Encoding

Once our data was complete, we had to preprocess it for the regressors. *DateOfDeparture* was converted to separate datetime objects (*year*, *month*, *day*...) to facilitate its use. After that, these date information were turned into binary features by using the *'get_dummies'* function so that they would not be considered as ordinally encoded by the model. Moreover, given the sensitivity of ensemble methods to categorical data, One Hot encoding was performed on *Departure* and *Arrival* to provide a better representation of these string features.

II. Model Selection

Reference Model & Feature Selection

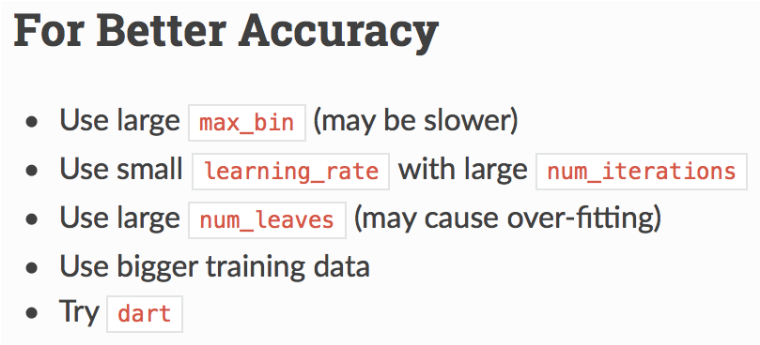
Given the complexity of the problem, we decided to focus on ensemble methods to achieve a higher predictive performance. Scikit-learn's *RandomForestRegressor* was used as a benchmark model for evaluating and comparing the different ensemble methods. We checked the feature importances using the function *sklearn.inspection.permutation_importances* on the said model to identify the most impactful variables. Features with null importance were then excluded from the subsequent models and weather data was reduced to the most influential covariates: *''Max TemperatureC', 'MeanDew PointC', 'Max Humidity' and 'WindDirDegrees'*, which resulted in a lower RMSE and thus, a better prediction performance.

Ensemble Methods

We attempted to build a performant model using boosting methods, which train weak learners sequentially, each trying to correct its predecessor. **Adaboost**, was first used as a boosting approach. In this algorithm, the output of the "weak learners" is combined into a weighted sum that represents the final output of the boosted regressor. We then sought to apply **Gradient Boosting**, which is known for its predictive accuracy. After that, **XGboost** was performed, which yielded the lowest root mean square error out of all the boosting methods, but with the highest training time. We also performed bagging with **BaggingRegressor**, but it predictably resulted in a higher RMSE than boosting. Finally, after further investigation of ensemble methods, we discovered **Light Gradient Boosting** methods, which achieved a better score in a smaller amount of time compared to **XGboost**. Overall, **LGBM** was more efficient, stable and accurate than the other methods.

Parameter Tuning

After selecting Light Gradient Boosting Methods to be our final model, we tried tuning our parameters using Scikit-learn's GridSearchCV but due to several bugs, we failed to implement it. Eventually, we ended up manually selecting our parameters using LightGBM's documentation.



In an effort to improve our model's overall performance, we tried to compromise between a higher accuracy and a lower training time. Therefore, amongst other things, we decided not to use a large max_bin as advised.

Finally, we tuned our parameters as follows:

1. **boosting-type = 'dart' –** which stands for Dropouts meet Multiple Additive Regression Trees. Despite being the slowest, DART produces the best results amongst all the boosting types. The algorithm employs dropouts on multiple additive regression trees (MART), which is an ensemble model of boosted regression trees. DART overcomes the issue of over-specialization of MART in which trees added at later interactions impact the prediction of only a few instances, resulting in having a negligible impact on the remaining instances. By adding dropouts, DART makes it harder for the trees added at later iterations to specialize in only a few samples, which considerably improves the model performance.
2. **n_estimators = 10,000 –** we noticed that RMSE decreased as the number of estimators increased. However, it stopped decreasing past 10,000, which was an indicator that it was the optimal value for this parameter given our dataset.
3. **learning_rate = 0.1 –** following the documentation's recommendation to use a small learning_rate with a high number of iterations, 0.1 was the smallest value that yielded the lowest RMSE.
4. **max_depth = -1 –** we kept the default value as it produced the best results. This parameter is usually used to deal with overfitting when the dataset is small, but since it is not our case, we did not need any limitation.
5. **num_leaves = 16 –** this parameter is important as it controls the complexity of the model. A large number of leaves may increase accuracy but also the risk for overfitting along with it, so we used the smallest number that maximized accuracy without overfitting the model. Following the documentation, this number should be strictly lower than 2^{max_depth}. However, considering that our max_depth is -1, we used the smallest power of 2 that yielded the best performance.
6. **subsample = 0.9 –** it specifies the percentage of rows used per tree building iteration, meaning that different rows will be selected every time for fitting each tree, which improves generalization and speed of training. 90% produced the best results for our model.
7. **colsample_bytree = 0.9 –** it deals with the sampling of our datatset's columns. Basically, our model will select a random subset containing 90% of the features before training each tree at every iteration. This parameter, along with the previous one, allow our model to be less sensitive to noise while testing.
8. **subsample_freq = 1 –** it is the frequency for bagging. Setting it to 1 means bagging at every single iteration.
9. **uniform_drop = True –** it enables better randomness and thus, better accuracy and generalization.

Conclusion

After a lot of investigation and experimentation, we managed to optimize our model as much as we could, which got us to the podium of the competition, with a relatively acceptable training time.