CS376 Term Project - Team 21

Clément Marcou Gabriel Lima Jaewoong Bae
Myeonghoe Song

December 13, 2018

1 Model Descriptions

Our model is a XGBoost Regressor with parameter optimization done by random search. Our data preprocessing is composed by many methods that will be discussed in Section 2. Figure 1 shows a representation of our model.

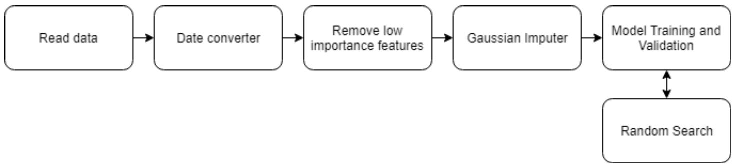

Figure 1: Representation of our model.

We start our model by reading the data into a numpy array, converting dates to the number of days passed from January 1st, 1980 until the date provided. We then delete some low importance features and apply our proposed Gaussian imputer. We finally optimize our hyperparameters by Random search.

2 Unique Methods

2.1 Deleting low importance features

We first ran the XGBoost regressor without any kind of parameter optimization and analyzed the importance of each one of the features. Figure 2 shows the graph provided by XGBoost.

After analyzing the graphic, we decided to remove features 4, 9, 13 and 19, which, according to the project notice, are respectively: 1st class region ID, angle, whether a vehicle can enter the parking lot, and built year.

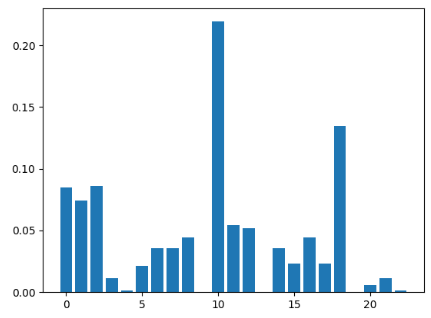

Figure 2: Feature importance using XGBRegressor.

Our method is unique because we first ran the model without any optimization or pre-processing to understand which features are useful to the regression task, and then considered that the low importance features actually decreased the accuracy of the model rather than just poorly participating.

2.2 Gaussian imputer

We propose a Gaussian imputer to fill the missing values in the provided data. For each feature, we compute μ_j, which is the mean of the feature j in the dataset. Then, for all j, we input the missing values according to a Gaussian distribution centered at μ_j and with a bandwidth of γ, which is a hyperparameter. Let x_{ij} be the the feature j of the data point i, and N be the number of data points that has their feature j not missing.

$$\mu_j = \frac{\sum_i^N x_{ij}}{N}, \forall x_{ij} \neq NaN \tag{1}$$

$$x_{ij} \sim Gaussian(\mu_j, \gamma), \forall x_{ij} = NaN \tag{2}$$

Our method is unique because we implemented a new form of imputing missing values following a Gaussian distribution centered at the feature mean instead of the usual imputer fixed on the mean. Also our method requires a hyperparameter γ to impute the missing values. The goal here is to mimic the reality more accurately.

2.3 Random search

We also implemented random search for hyperparameter optimization. We ran 100 epochs for 4 hyperparameters in the XGBRegressor: learning rate (lr), maximum depth

(max_depth), regularizer (λ) and minimum child weight (min_child_weight). Each hyperparameter was searched in a defined interval:

$$
\begin{gathered}
lr \in [0.05, 0.15[\\
max\_depth \in [8, 18[\\
\lambda \in [0.005, 0.020[\\
min\_child\_weight \in [0, 2[
\end{gathered}
$$

Our method is unique because we set the extremes of our values and ran the random search with a chosen random seed.

3 Libraries

3.1 Numpy

Version: 1.15.4
Purpose: Data manipulation, numerical calculations and random number generator.

3.2 Pandas

Version: 0.23.4
Purpose: CSV file reading for easier date conversion.

3.3 XGBoost

Version: 0.81
Purpose: Regression boosting algorithm implementation.

3.4 Sklearn

Version: 0.20.0
Purpose: Training and testing data splitting.

3.5 Datetime

Version: Included with Python 3.6.7
Purpose: Date to number conversion.

3.6 Math

Version: Included with Python 3.6.7
Purpose: Floor and ceil functions.

3.7 Time

Version: Included with Python 3.6.7
Purpose: Measure time elapsed during model training and validation.

3.8 OS

Version: Included with Python 3.6.7
Purpose: Check if the model has been saved.

4 Source codes

4.1 Read Data

First of all, we implemented $read\_data(training)$ and $date\_parser(data)$ functions. The $read\_data(training)$ function gets train or test data from a csv file using pandas. If $training$ is true, this function reads training data; if set to false, testing data is read. The $date\_parser(data)$ function changes datetime format into float type. This data represents the number of days passed since January 1st, 1980. The $divide\_data(data)$ function divides data into X and y, which are respectively the features and target values.

4.2 Preprocessing

There are two functions used for preprocessing. We implemented $gaussianImputer(X, bandwidth)$ to impute missing values. This function imputes missing values using Gaussian distribution centered at μ with spread γ. The $remove\_NImp\_features(X)$ function removes the 4 least important features which are 1st class region ID, angle, whether a vehicle can enter the parking lot and built year.

4.3 Training

To train the model, we use the $training(X\_train, y\_train, random\_split, unique)$ function. There are two parameters: one that decides how to split the training and the validation set, and another that determines whether unique methods are going to be used or not. If $random\_split$ is set to true, then the training and validation set are separated randomly. Otherwise the validation set will be composed of only future data. If $unique$ is set to true, the model will be trained with our optimized hyperparameters and unique methods. If not, the regressor will be trained with the default hyperparameters and without our unique methods. To get better performance, we had to to optimize the hyperparameters used in XGBRegressor. To do so, we implemented the $random\_search(X, y, random\_split, epochs)$ function. For each epoch, it randomly generates hyperparameters values and computes the performance using them, finding the best combination of hyperparameters.

Thanks to this method, we found the following hyperparameters that maximize the accuracy.

$$\begin{gathered} bandwidth\colon 1 \times 10^{-10} \\ lr\colon 0.097 \\ max\_depth\colon 13 \\ \lambda\colon 0.005 \\ min\_child\_weight\colon 0 \end{gathered}$$

4.4 Testing

To test the model, the $testing(X, y, unique)$ and $read\_data(training)$ functions are used. First you have to get the test dataset using $X\_test = read\_data(training = False)$ and change the name of the input file in the $read\_data(training)$ function. Then, you can save the predictions by calling $testing(X\_train, y\_train, X\_test, unique)$. Setting $unique = True$ will predict target values with our unique methods, and without with $unique = False$. Finally, if you want to check performance, you can use $performance\_metric(actual, predicted)$ function. This function calculates performance using Equation 3, where N is the number of data, A_t is the actual value and F_t is the predicted value.

$$performance = 1 - \frac{1}{N}\sum_{i}^{N} \mid \frac{A_t - F_t}{A_t} \mid \tag{3}$$

5 Performance

We split the training data for validation in two ways. First, we used the train_test_split function from sklearn, which separates the data randomly. We also used the function $get\_train\_test(data)$, written by us, that orders the data in increasing order of date and gets the last part of the data points for validation (the future). The former method will be called as $random\_split$ and the latter as $future\_split$. We used 5% of the data for validation in both methods. Table 1 shows the results using the two methods with our unique methods and Table 2 shows the performance without our unique methods.

Performance	$random\_split$	$future\_split$
Training	0.9718	0.9718
Validation	0.9560	0.9461

Table 1: Performance results of our model **with** our unique methods.

Performance	$random\_split$	$future\_split$
Training	0.8744	0.8760
Validation	0.8754	0.8664

Table 2: Performance results of our model **without** our unique methods.

The time elapsed during training and validation in seconds is shown in Table 3. The computer used has a i5 8250-U and 8GB of RAM.

Time (s)	$random\_split$	$future\_split$
With our unique methods	113	114
Without our unique methods	37	38.8

Table 3: Time elapsed during training and validation in seconds.

The rationale behind using $future\_split$ is because it was told that most of the test data is to predict the future. However, the test data also had many 'past' data points, thus we also decided to use $random\_split$ for validation.

As seen in Table 1 and Table 2, both splitting methods achieved great results in terms of the performance on training data (in Table 2, $future\_split$ even outperforms $random\_split$). However, $future\_split$ did worse than $random\_split$ in validation. This is because with $future\_split$, the model predicts the future only, while with $random\_split$, the model predicts both the future and the past. Our hypothesis is that with $random\_split$, our model learns well to predict the past, since most of the training data is from the past. However, with $future\_split$, our model cannot take the same advantage of predicting the past by learning from the past, since the validation data is only composed of future data points.

As for the performance improvement with our unique methods, our unique methods are verified to improve the prediction performance by approximately 0.08 in validation. The main source of this improvement was hyperparameter tuning of XGBoost regressor with our random search. Deleting low importance features and Gaussian imputer contributed less, but also did improve the performance.

XGBoost regressor already achieved good results without our unique methods (Table 2), but deepening XGBoost model's decision tree by our random search greatly improved the prediction performance. Deleting low importance features let us remove redundant features(e.g. Feature 19 (built year) is practically the same as Feature 18 (construction completion date)) and some features that did not or badly contributed to the prediction performance(Features 4, 9, and 13). Thus, we could obtain a small performance improvement by deleting low importance features. Finally, Gaussian imputer did not contribute to the performance improvement as much as the first two, but we could randomize the missing values a bit using this method, making it to more relatable to real data.