DoorDash Data Science Take-Home-Assignment Report

William Wong

March 25, 2018

1 Part I. Model Building

The work was done in a Jupyter notebook called `explore_and_build_model.ipynb`, which is submitted for evaluation.

1.1 Exploratory Analysis

1.2 Creating Additional Features

We create three additional features:

1. `created_at_hour` We extract the hour where the order is created. This is a categorical variable.

   One can see that most orders are placed late at night (from midnight to 4 AM) and during dinner time (from 7 PM to 9 PM).

   ```
   > df_csv['created_at'].dt.hour.value_counts().sort_index()

   0      12669
   1      28190
   2      36976
   3      27068
   4      15250
   5       7096
   6       1416
   7         11
   8          2
   14        40
   15       538
   16      2109
   17      3413
   18      5100
   19     13541
   20     15560
   ```

```
21    11465
22     8821
23     8163
Name: created_at, dtype: int64
```

2. `created_at_dayofweek` We extract the hour where the day of the week is created. This is a categorical variable.

3. `fractional_busy_dashers` We define this continuous variable to be `total_busy_dashers` divided by `total_onshift_dashers`. This variable should provide some insight as to how "loaded" the system is.

   One can see that the system is quite busy, as the median value of `fractional_busy_dashers` is 0.95.

```
> df_csv['created_at'].dt.dayofweek.value_counts().sort_index()

count    197421.000000
mean          0.954620
std           0.416678
min           0.000000
25%           0.846847
50%           0.946429
75%           1.000000
max          34.000000
Name: fractional_busy_dashers, dtype: float64
```

1.3 Summary of Features used in the Model

The categorical features are:

- `created_at_hour`
- `created_at_dayofweek`
- `market_id`
- `order_protocol`

The continuous features are:

- `total_items`
- `subtotal`
- `num_distinct_items`

- min_item_price
- max_item_price
- total_onshift_dashers
- total_busy_dashers
- fractional_busy_dashers
- total_outstanding_orders
- estimated_order_place_duration
- estimated_store_to_consumer_driving_duration

1.4 Removing Outliers

We remove the outliers in the features and in the outcome variable and replace the values with NaN, which will be dealt with in the next stage. We consider a data point to be an outlier if the value is outside the 99-percentile of the population. An example is shown below, where the 99-percentile value of the outcome variable (`outcome_total_delivery_time`) is 6475 seconds. However, there are outcome values that far exceed this number (e.g., `outcome_total_delivery_time` = 8.52 million seconds!).

```
> df_csv[col_outcome].describe(percentiles=
                                    [0.01, 0.1, 0.25, 0.5, 0.75, 0.90, 0.95, 0.99])

count    1.974210e+05
mean     2.908257e+03
std      1.922961e+04
min      1.010000e+02
1%       1.152000e+03
10%      1.699000e+03
25%      2.104000e+03
50%      2.660000e+03
75%      3.381000e+03
90%      4.235000e+03
95%      4.872000e+03
99%      6.474800e+03
max      8.516859e+06
Name: outcome_total_delivery_time, dtype: float64
```

As an second example (Fig. 1), note that some of the values of `total_onshift_dashers` are negative, which are wrong.

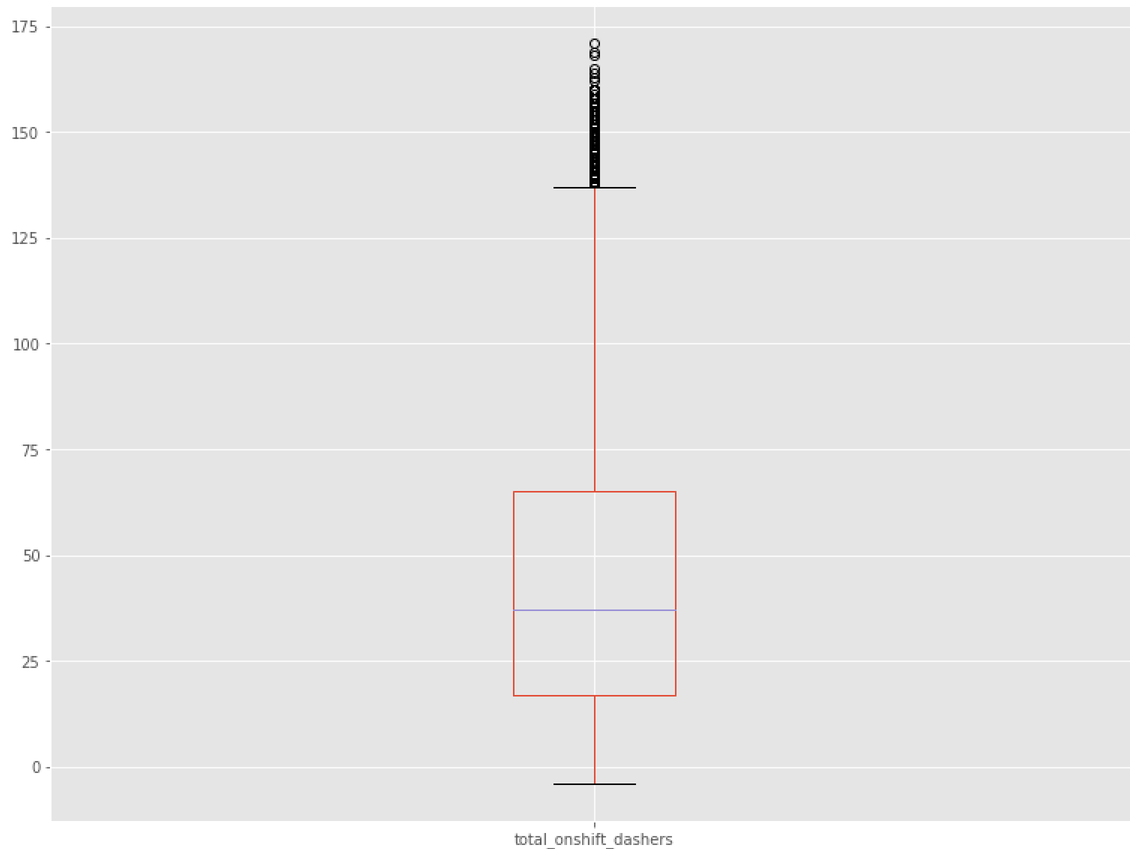

Figure 1: Box plot of the feature `total_onshift_dashers`.

1.5 Imputating Missing/Null Values

We replace missing categorical values with the mode, and missing continuous values with the median.

We remember these operations and will repeat them during the production run.

1.6 Training and Testing the Model

To get the best performance, we model the outcome variable using random-forest regression with 200 trees. We plot the observed outcome variable versus the predicted outcome variable, and the residual versus the predicted outcome variable in Fig. 2. The root-mean-squared error (RMSE) is found to be 900 seconds $\pm$ 5%, which is decent (about 15 minutes).

We split the dataset randomly into training (90%) and test (10%). We train the model using the training set and evaluate the model in terms of RMSE using the test set.

The trained model is serialized to a file using Pickle.

Figure 2: Plot of the observed outcome variable versus the predicted outcome variable, and plot of the residual versus the predicted outcome variable.

2 Part II. The Production Run

2.1 The Code Base

We de-serialize the model we trained in Part I in Python and apply it to the held-out data.

The highlights in our production-quality code in Python are

- object-oriented programming (using classes and member functions). The library code resides in `model.py`.
- Python logging
- unit testing using the `unittest` framework.

  To run a unit test,

```
$ python test_model.py
.
----------------------------------------------------------------------
Ran 1 test in 24.288s
```

```
OK
```

In this unit test, we apply a previously trained model on *training data* to compute the RMSE. We ensure that the RMSE stays below some threshold value.

2.2 Cleaning of Model Features

Since we remember the previous data-cleaning operations, we will apply them to the features during the production run.

```
for col in self.cols_features:
    logger.info( 'Working on column ' + str(col) )
    column = self.features_dict[col]
    df_csv.loc[ df_csv[col] < column.value_low, col]  = column.value_default
    df_csv.loc[ df_csv[col] > column.value_high, col] = column.value_default
```

2.3 Prediction of the Holdout Data Set

We apply the trained model to the holdout data set by calling the driver code.

```
$ python run_model.py
```

The tab-separated output is called `output.tsv`. It consists of two columns `delivery_id` and `predicted_delivery_seconds` and 54,778 records.

3 Discussions

From Part I, since the median value of `fractional_busy_dashers` is 0.95, we suggest that we make more dashers available to reduce the total delivery time.