DoorDash Delivery Time Prediction

1 Abstract

Predicting food delivery ETA is a fascinating and interesting challenge. It is important to be able to predict accurate delivery times to increase customer satisfaction and dynamically route dashers in an optimal manner. If we quote the customer forty-five minutes for a delivery, we don't want the delivery to be there at ninety minutes, but it also shouldn't arrive at twenty minutes. Being spot-on is hard. There can be multiple factors which affect order delivery times including order characteristics, dasher characteristics & certain dynamic features such as traffic, busy merchants etc. Given a dataset of potential factors affecting the delivery time of food orders, we train a Machine Learning model which predicts the delivery time. We define a set of new historically aggregated features to input into the model. We experiment with various models such as Linear Regression, Random Forests and Gradient Boosting Regression(GBR) to model the training data. GBR outperforms other models and has superior bias/variance trade-off. Finally, we look at importance of various features and make recommendations to reduce the delivery time.

2 Data Cleaning (Training Data)

Let's look at missing row values first

```
In [8]: for col in hist_data.columns:
            null=hist_data[hist_data[col].isnull()]
            print col,len(null)
```

```
market_id 987
created_at 0
actual_delivery_time 7
store_id 0
store_primary_category 4760
order_protocol 995
total_items 0
subtotal 0
num_distinct_items 0
min_item_price 0
max_item_price 0
total_onshift_dashers 16262
total_busy_dashers 16262
```

```
total_outstanding_orders 16262
estimated_order_place_duration 0
estimated_store_to_consumer_driving_duration 526
```

Here's how data was prepared for the learning algorithm

- Missing delivery_times- There are only 7 rows which don't have an actual delivery time. Discarding these rows(as these don't have a training target value).
- Missing marketID - Applied Nearest neighbor approach to determine market_id for missing rows. Since market_id reflects the local region and hence the size of the local population, it is reasonable to use total_onshift_dashers and total_busy_dashers as the features to calculate Euclidean distance for determining the nearest neighbor. Discarding rows which have both total_onshift_dashers and market_id missing values missing. There are only 68 such rows, a tiny fraction.
- Missing Dasher Information - We are replacing missing dasher information by answering: How many dashers do you typically see in this market on a weekday/weekend at a peak/non-peak hour? Defined some time-based features: time of the day represented as a float number, hour of the day as an integer, day of the week as an integer, a binary weekday/weekend feature and a binary peak hours/non-peak hours feature. By looking at mean delivery times for each hour, 6 hours were isolated as peak hours(1,2,3,14,15,16)
- Other Missing info: We do the same as above for missing total_outstanding_orders and estimated_store_to_consumer_driving_duration
- Missing store_primary_category values - We calculate the mean delivery times for each store category. For the rows that have missing store_primary_category values, we assign them the store_primary_category which is the closest in terms of mean delivery time. Further Improvement - If we also had the duration data in which the prepared meal is waiting to be picked up by the dasher in our dataset, we could've calculated approximate meal prep time for each order by subtracting the (wait time+driving time) from the delivery time. Meal prep time would've been an even more accurate choice to assign cuisine type to missing rows, as certain cuisines have a longer preparation time than others. We can also setup a nearest neighbors model based on other store features(if available) to determine store category. However, for the missing store category values in the test dataset(in part 2), we fill in the category with the most deliveries in that market.
- Missing Order Protocol - To fill in missing values for Order Protocol, we randomly sample values from the Order Protocol column from the rest of the data.
- Outliers: We remove some extreme outliers in the data. For example, negative values for dasher numbers. We also restrict the delivery_time target to be no more than 3 hours so that the learning algorithm isn't influenced too much by outliers.

3 Data Preparation(Test Data)

On the test dataset, we replace the missing rows slightly differently and we don't discard any rows.

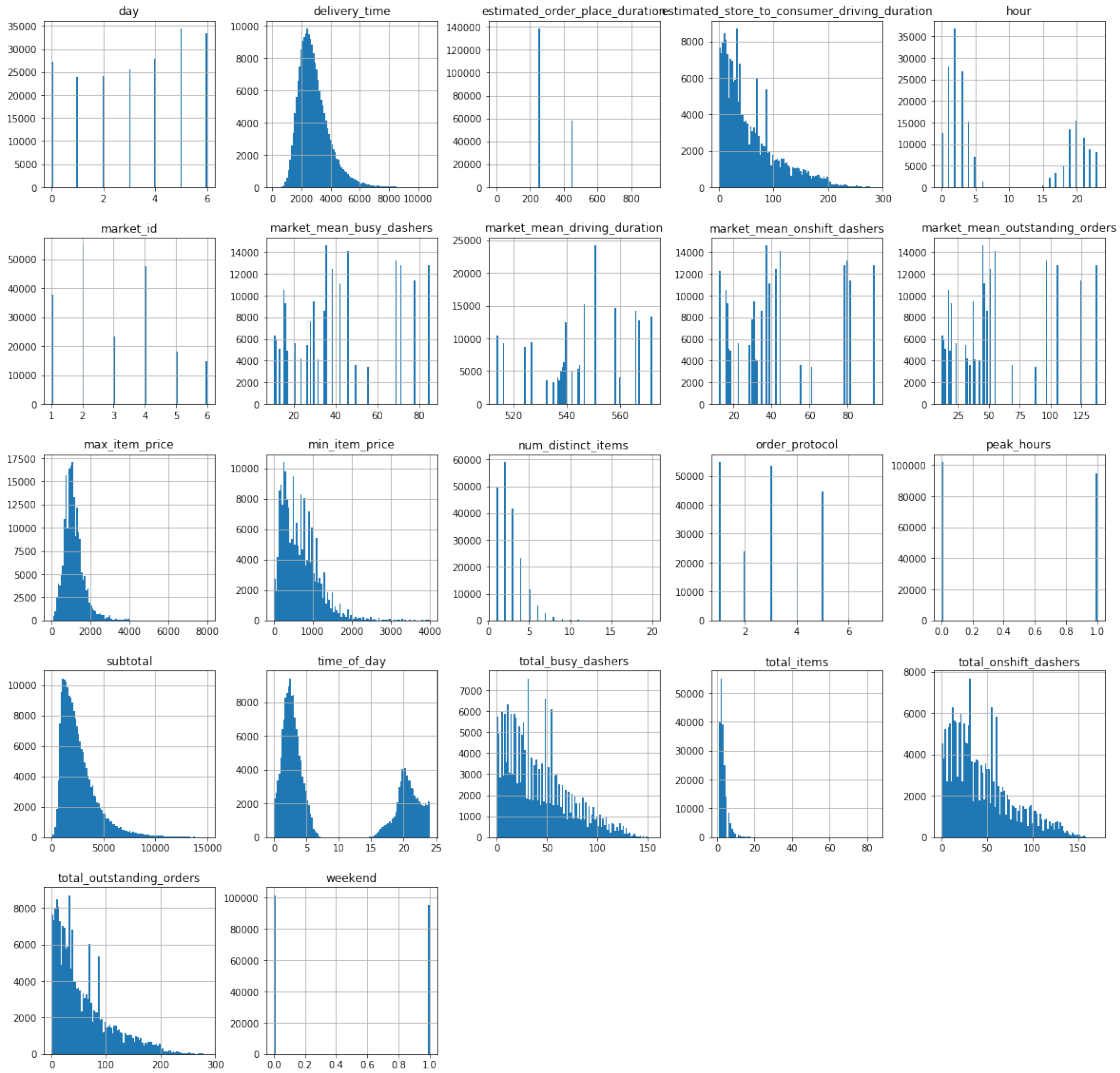
For the rows that don't have both dasher information and market information, we random sample market_id values from the rest of the dataset. In the training dataset, we discarded such rows. The process for filling the remaining missing market_id values remains the same.

To fill in the missing store category values, we find out the store category with most deliveries for each market and fill in the missing information using that row's market_id value. All other missing information is filled the same way as the training dataset.

Here's a look at the feature distributions.

```
In [28]: hist_data_clean.hist( bins=100,figsize=(20, 20))
         len(hist_data_clean)
```

```
Out[28]: 196821
```



4 Feature Engineering

To be better immune to the noise in the data, we use aggregate historical features which act as smoothed inputs to the model.

We define some aggregate historical features based on mean values aggregated over market,store,cuisine and hour called mean_market_delivery_time,mean_store_delivery_time,mean_cuisine_delivery_time, mean_hour_delivery_time respectively. mean_store_delivery_time, for instance, indicates how has a store perfomed in the past. We export these tables for later use with test_data.

There are some new store ids in the test data. To use the mean_store_delivery_time feature for these stores, we assign the mean_cuisine_delivery_times as their mean_store_delivery_times.

We use the following features for the model.

- Real Time Features
  - Hour of day to see impact of lunch / dinner times.
  - Day of week - weekend / weekday
  - Market Id
- Store Features
  - StoreId
  - Order protocol
- Order Features
  - Min Item Price
  - Max Item Price
  - Total Items
  - Number of distinct items
- Market Features
  - Onshift Dashers
  - Busy Dashers
  - Outstanding orders
- Predictions from other models
  - Estimated order place duration
  - Estimated Store to Consumer Driving Duration
- Historical Mean delivery time for
  - Market
  - Store
  - Cuisine
  - Hour of the day

We perform One Hot Encoding on the categorical features(hour,day,market_id,order_protocol) to better represent those categories.

We generate the full feature list, create the training and target datasets and split them for a 10-fold cross-validation

5 Building learning models

We evaluate our models using following:

1. Train, perform 10-fold cross validation and print the mean rmse error. After training on each cross-validation set, we plot the residual distribution and print its mean and SD. The means should be close to zero and SDs should be small.

2. Plot a learning curve (RMSE Vs Size of the Dataset) to explore the bias-variance tradeoff

We first evaluate the performance of a very basic model which predicts the mean delivery time of the training dataset for each test dataset row.

Mean residuals vary a lot and are not centered around zero. The mean RMSE is more than 1000s. We try Linear Regression next.

Both Ridge and Lasso Regressions show similar residuals with means close to zero and an average RMSE of around 890s.

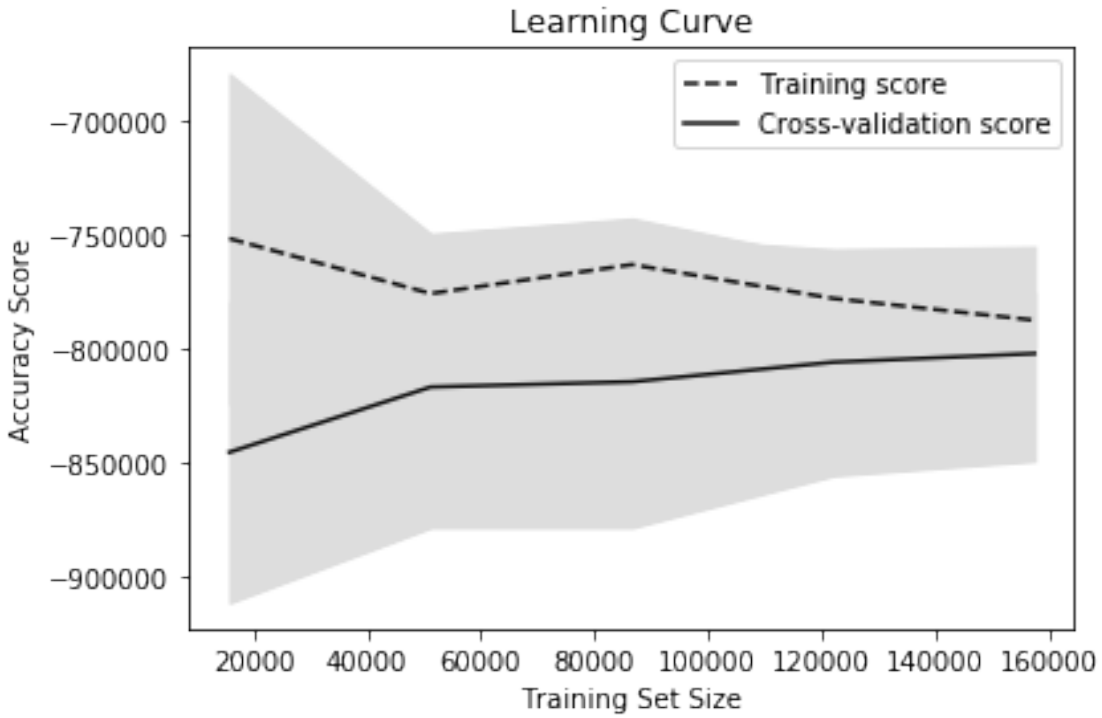
Next we train a Random Forest Model. Random Forest yields an average RMSE of 880s.

Finally, we try a Gradient Boosted Regressor using the XGBoost library. GBR gives us an average RMSE of 860s.

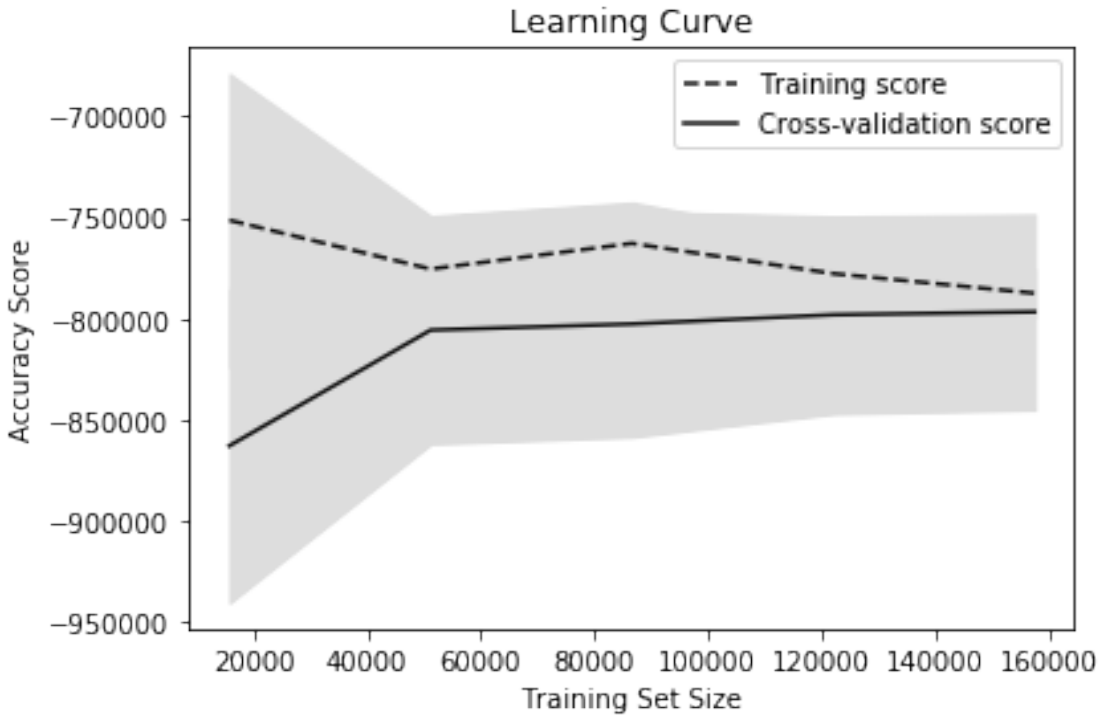
6 Evaluating models using learning curves

We evaluate our learning models using learning curves to explore the bias/variance trade-off.

```
In [64]: learning_curve(ridge_reg,hist_data_subset_X,hist_data_subset_Y)
```

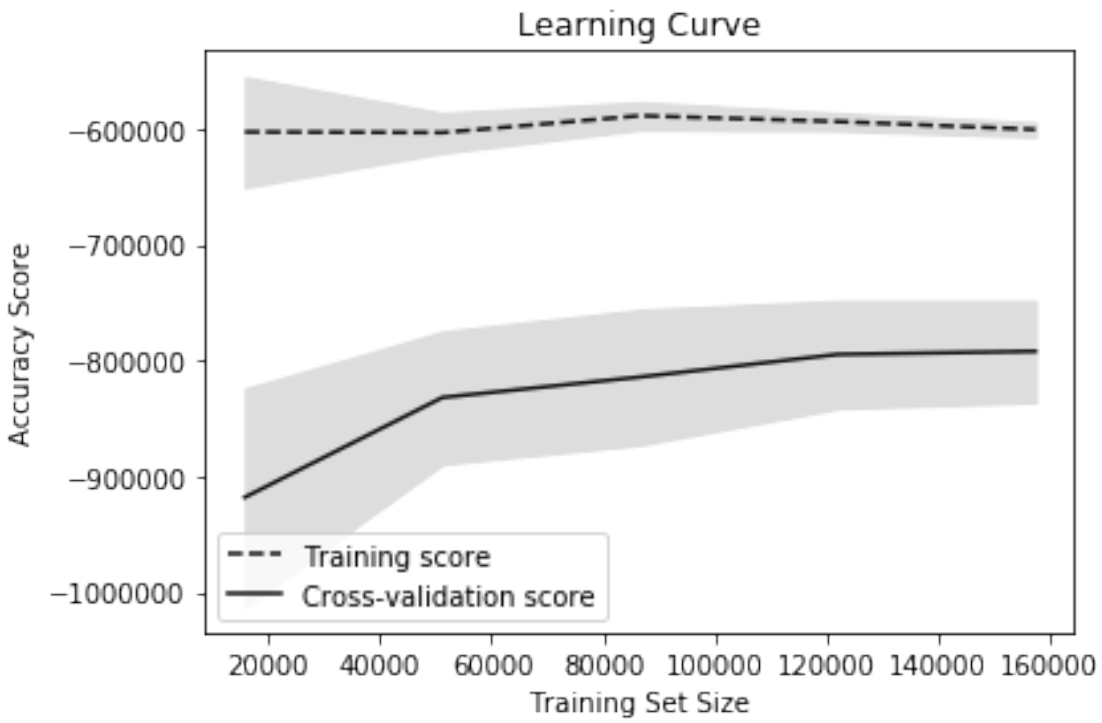


In [65]: `learning_curve(lasso_reg,hist_data_subset_X,hist_data_subset_Y)`



Both the linear regression models have a lot of bias as indicated by the close training and cross-validation score as the data size increases. A simple linear model is not complex enough to model the problem

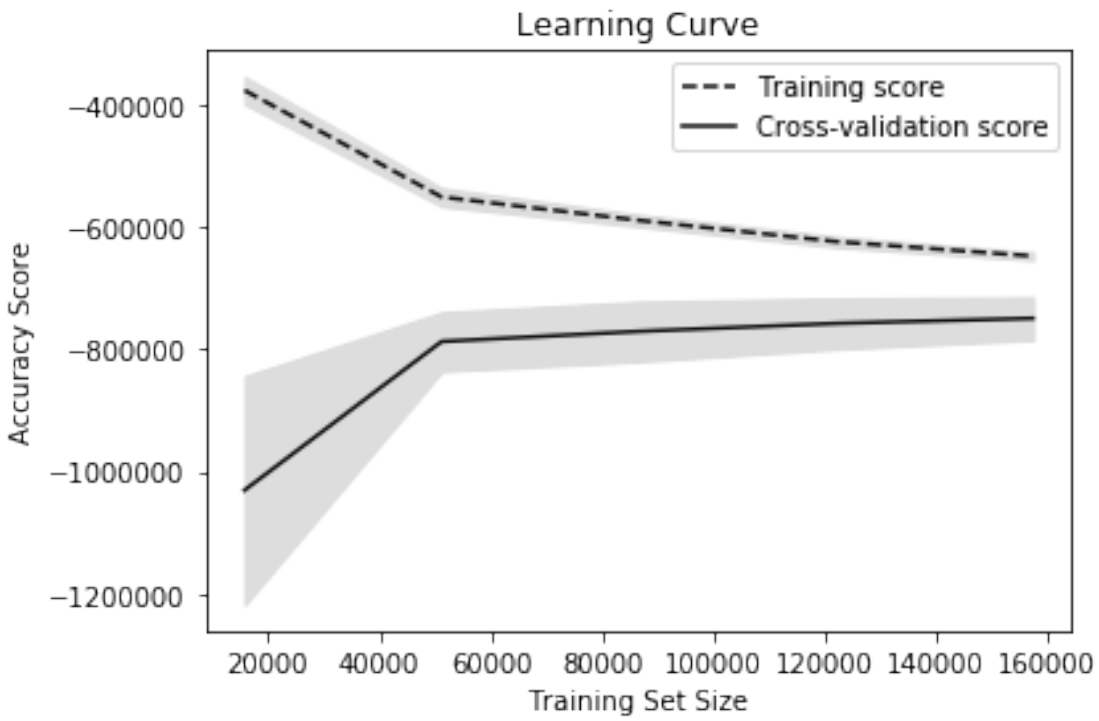
In [66]: `learning_curve(rf,hist_data_subset_X,hist_data_subset_Y)`



The large gap between the training and cross-validation errors indicates a lot of variance in the Random Forest model.

```
In [58]: learning_curve(xgb_reg,hist_data_subset_X,hist_data_subset_Y)
         #9

/Users/souvikroy/anaconda2/lib/python2.7/site-packages/sklearn/model_selection/_validation.py:
  if np.issubdtype(train_sizes_abs.dtype, np.float):
```



GBR performs the best in terms of bias and variance trade-off. There is some gap between the two curves as the datasize increases, but not too excessive.

7 Summary

Here's a summary of model performances measured by average RMSE on the cross-validation sets.

- Baseline - 1057s
- Ridge Linear Regression - 890s
- Lasso Regression - 892s
- Random Forest - 880s
- GBR - 860s

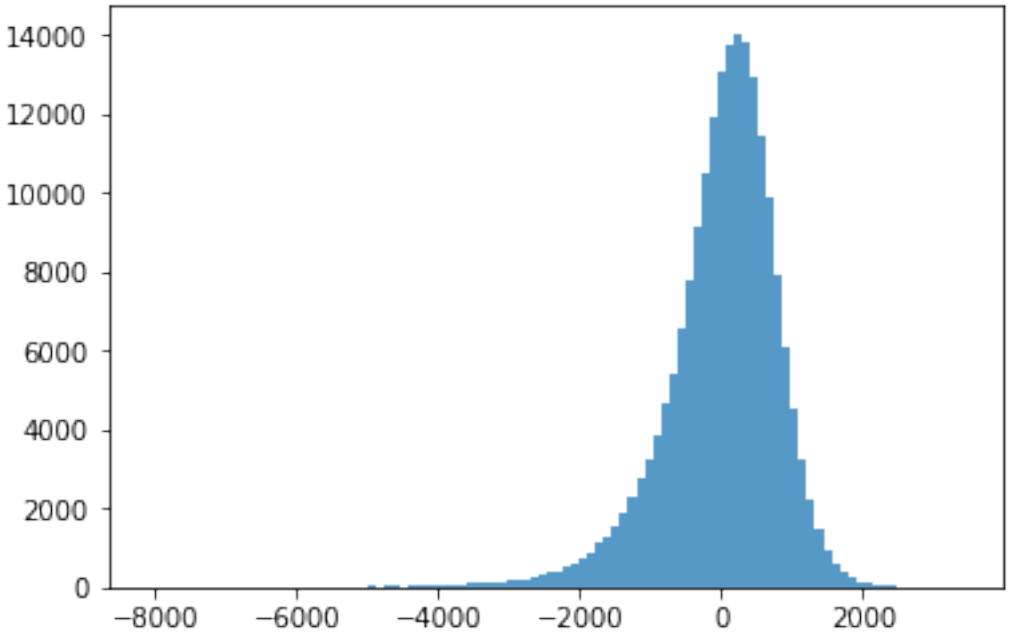
8 Optimizing hyper-parameters

To optimize the Gradient Boosting hyper-parameters, we perform a grid search and choose the model parameters with the best 5-fold cross-validation score.

After fitting the training data on the optimal GBR model, we calculate mean RMSE on the full training dataset and compare training and predicted delivery times distributions

```
In [89]: y_predicted=xgb_reg.predict(training_dataframe[all_features])
         y_train=training_dataframe['delivery_time'].values
```

```
n, bins, patches = plt.hist(y_predicted- y_train, 100, normed=0, alpha=0.75)
plt.show()
error=math.sqrt(np.mean((y_predicted- y_train) ** 2))
print("Mean Residual: %.2f"
      % np.mean(y_predicted- y_train))
print("Residual SD: %.2f"
      % np.std((y_predicted- y_train)))
print("RMSE: %.2f"
      % error)
```




```
Mean Residual: 0.14
Residual SD: 811.57
RMSE: 811.57
```

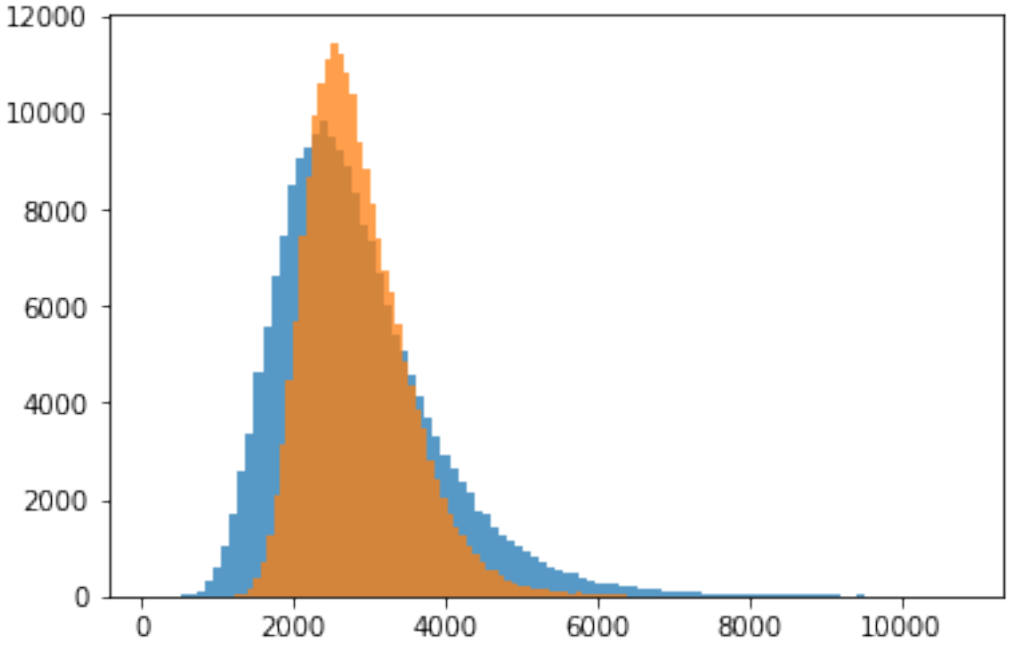
```
In [94]: n, bins, patches = plt.hist(y_train, 100, normed=0, alpha=0.75)
         #plt.show()

         print("Historical Data Mean: %.2f"
               % np.mean(y_train))
         print("Historical Data SD: %.2f"
               % np.std((y_train)))

         n, bins, patches = plt.hist(y_predicted, 100, normed=0, alpha=0.75)
         plt.show()
```

```
print("Predicted Mean: %.2f"
      % np.mean(y_predicted))
print("Predicted SD: %.2f"
      % np.std((y_predicted)))
```

```
Historical Data Mean: 2851.52
Historical Data SD: 1082.23
```




```
Predicted Mean: 2851.66
Predicted SD: 675.63
```

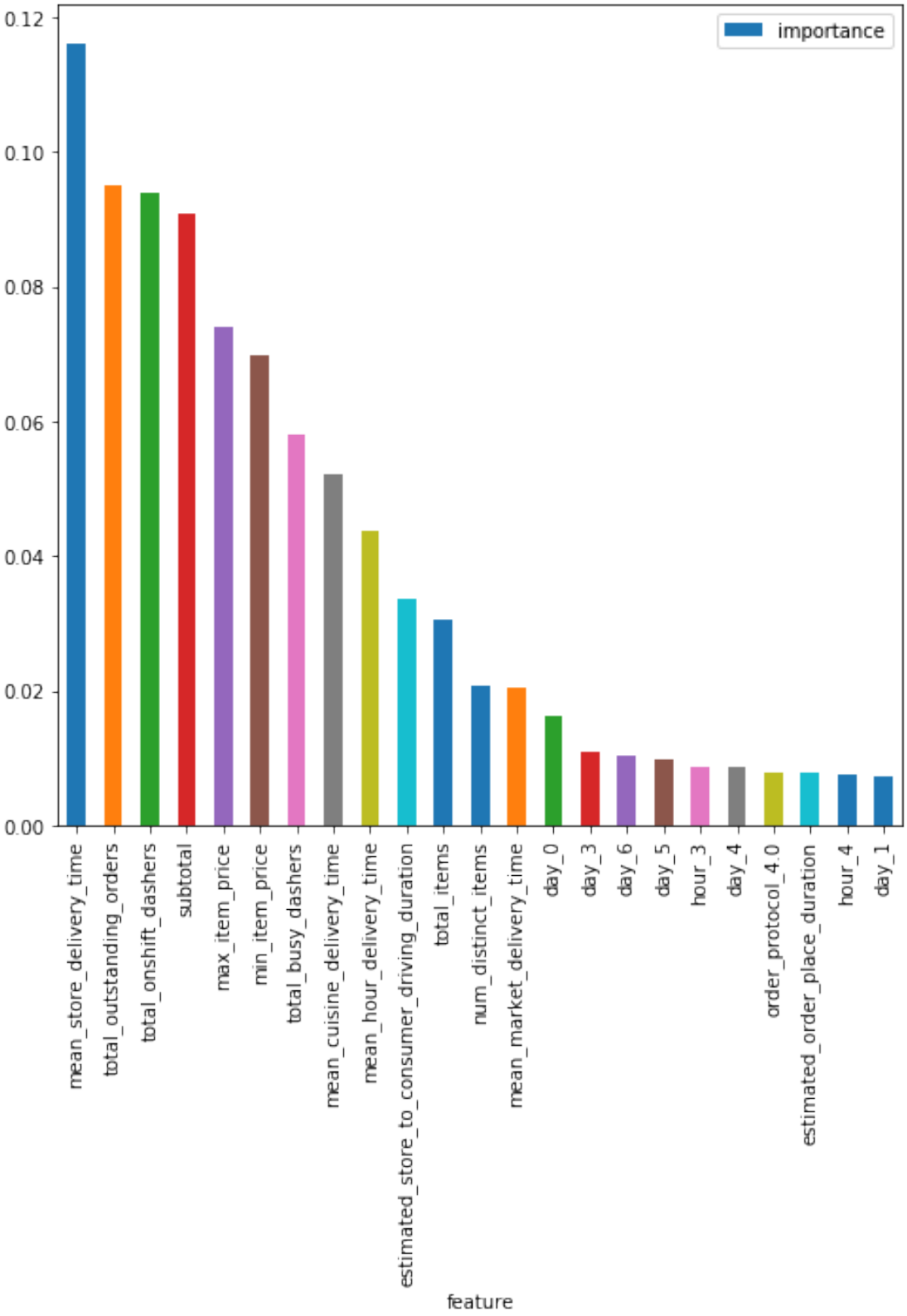
The predicted delivery times and training data delivery times have means which are very close, but the training delivery times have a much wider spread, possibly due to noise in the training data.

9 Model Insights

We plot the feature importance values on a bar plot.

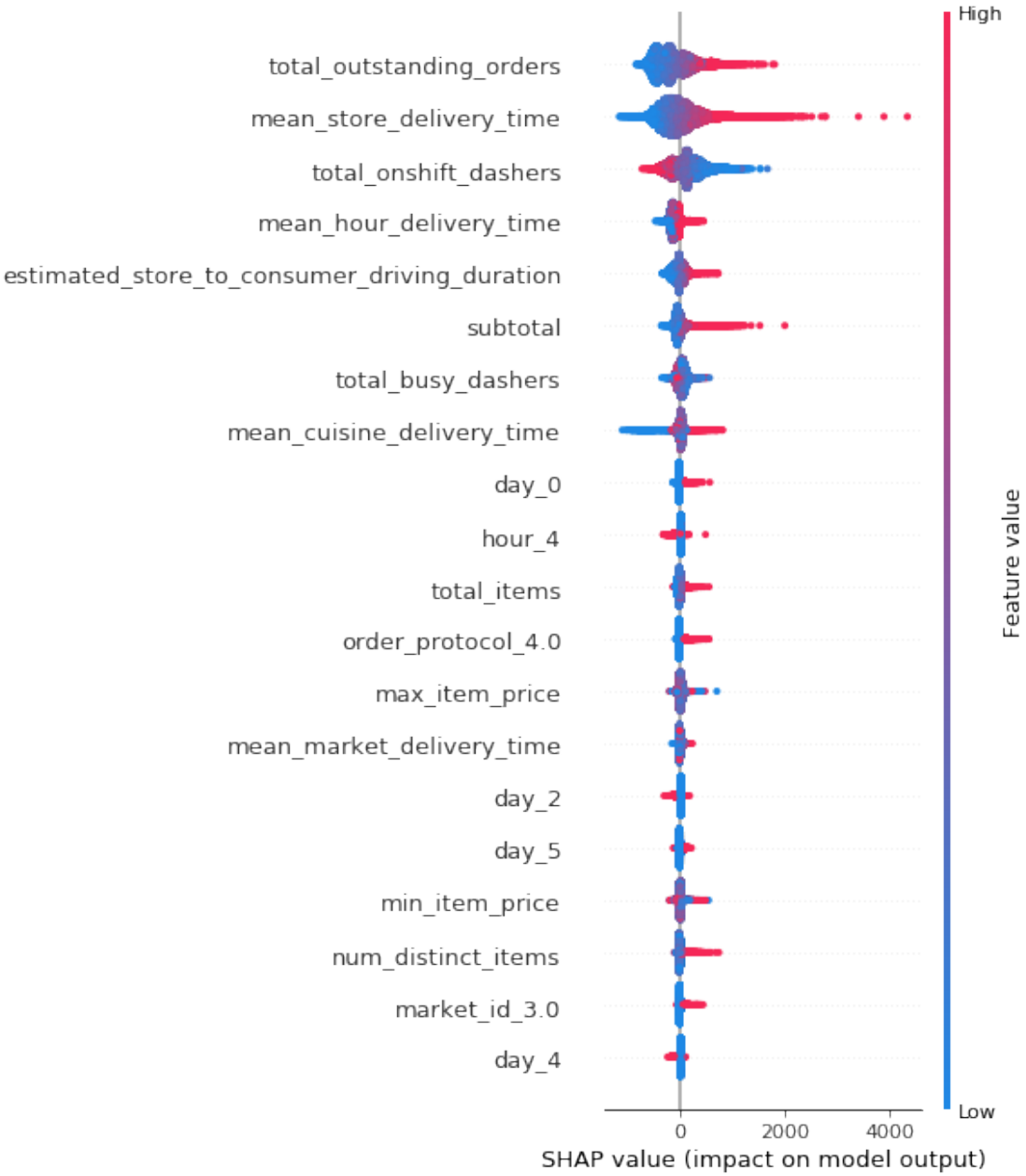
```
In [110]: feature_importance_xgb=pd.DataFrame(columns=['feature','importance'])
          feature_importance_xgb['feature']=all_features
          feature_importance_xgb['importance']=xgb_reg.feature_importances_
          feature_importance_xgb=feature_importance_xgb.sort_values('importance',ascending=Fals
          feature_importance_xgb[feature_importance_xgb['importance']>0.007].plot.bar('feature
```

Out[110]: <matplotlib.axes._subplots.AxesSubplot at 0x1a22fedd90>



```
In [73]: shap.initjs()
         shap_values = shap.TreeExplainer(xgb_reg).shap_values(training_dataframe[all_features]
         shap.summary_plot(shap_values, training_dataframe[all_features][1:40000])
```

```
<IPython.core.display.HTML object>
```



We use the SHAP library(https://github.com/slundberg/shap) "to get an overview of which features are most important for a model we can plot the SHAP values of every feature for every sample. The plot below sorts features by the sum of SHAP value magnitudes over all samples, and uses SHAP values to show the distribution of the impacts each feature has on the model output. The color represents the feature value (red high, blue low)." This reveals for example that a high total_outstanding_orders increases the predicted delivery time.

10 Recommendations

- Looking at the feature importance values, historical mean_store_delivery_time ranks very high. Stores which have historically had long delivery times should be taken into special consideration. Maybe the store is slightly far from where the orders come from. Or maybe dashers struggle to find parking at the store. Maybe the store is busier than other stores and take a longer time to prepare foods. Indentifying causes for stores with high mean_store_delivery_time can help reduce delivery times for these stores significantly.

- Total_onshift_dashers and total_outstanding_orders are among the most influential factors. We can increase the number of onshift dashers by incentivizing dashers when total_outstanding_orders values exceeds a certain threshold and ensure that the number of onshift dashers remains high.

- Subtotal has a strong influence on delivery times. Higher sub totals lead to higher delivery times. We can tweak the dispatch algorithm to ensure that the dasher always arrives slightly earlier when subtotal is higher than historical thresholds. So, there is no additional wait time (the food waits for a dasher to be picked up).

- Total items and number of distinct items have a similar effect on delivery time. We can adopt a similar strategy flagging orders which have too many items and prioritizing them for dasher dispatch over others.

- As expected day0(Sunday) and day6(Saturday) affect the delivery time the strongest. day3(Wednesday) ranks next and is as high as the day05(Friday). Getting more dashers during weekends will help reduce the delivery times. Paying dashers more on these days could get more dashers available and may reduce the delivery times. Similarly, for hour3 and hour4.