Part 1: Build a model to predict the total delivery duration seconds

Introduction

The target problem can be defined as a regression problem for prediction based on supervised learning. This work uses Python and PyTorch to build linear/polynomial regression and DNN regression models to predict total delivery duration seconds. This design doc includes details in Data Pre-processing, Feature Extraction & Normalization, Model design, Experiments,Conclusion and Future work, which can be used together with submitted code package.

- dataManager.py: For Data Pre-processing, Feature Extraction & Normalization
- model.py: For model design
- buildModel.py: Run experiments

Data Pre-processing

- Define target output

According to *historical_data.csv*, the total delivery duration seconds is between *actual_delivery_time* (a) and *created_at* (b), which includes *estimated_store_to_consumer_driving_duration* (c) and *estimated_order_place_duration* (d). Considering both of them come from other prediction models, in this work, define the target output as:

- *prediction = (a - b - c - d)*

Then the final total delivery duration seconds will be *prediction + c + d.*

- When *estimated_store_to_consumer_driving_durationis* is “NA”

  Set 0 seconds instead (more than 500 records)

- When *actual_delivery_time* or *created_at* are not available

  Remove records (only several records)

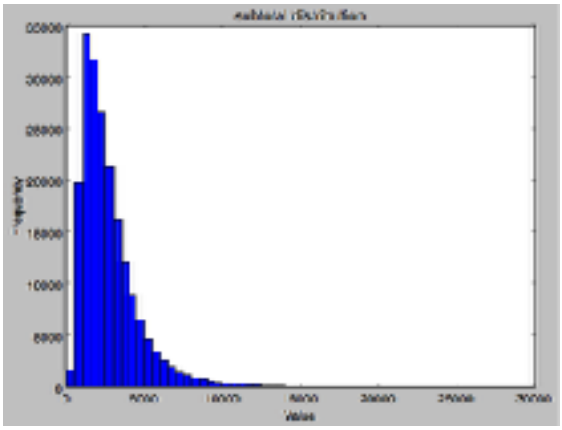
- When *total_onshift_dashers* < *total_busy_dashers*

According to data description, *total_busy_dashers* is the subset of *total_onshift_dashers,* which should be always less or equal to *total_onshift_dashers.* Set them with the same value when *total_onshift_dashers* < *total_busy_dashers. (*more than 40000 records as noise information added*)*

Feature Extraction & Normalization

- **Features with Gamma distribution (5 in total)**
    - subtotal
    - min_item_price
    - max_item_price
    - total available dashers (total_onshift_dashers - total_busy_dashers)
    - total_outstanding_orders

By analysis and data visualization, all of the above five features can be defined as Gamma probability distribution function. Below is the histogram of subtotal as example:



Based on the theory of Gamma probability distribution, use Cumulative Distribution Function (CDF) to normalize features into the range (0~1). Implementation is as follows:

```
import scipy.stats as st

def fitGamma(data):
    fit_alpha, fit_loc, fit_beta=st.gamma.fit(data)
    data = (data  - fit_loc)/fit_beta
    data=st.gamma.cdf(data, fit_alpha)
    return data
```

- **Features by embedding selection (4 in total)**
    - store_primary_category
    - order_protocol

- total_items
- num_distinct_items

It is easy to understand that *store_primary_category* and *order_protocol* are category information, which can be converted to 1-dimension vector with total categories. (*store_primary_category*: 75 categories in total, *order_protocol*: 8 categories in total)

But for *total_items* and *num_distinct_items*, although they are digital values with Gamma probability distribution, because of small value range, after fitting them by CDF, small values are converted to 0, which is useless for supervised learning. Based on this consideration, also apply embedding selection on both of them. (*total_items*: set 30 in total, *num_distinct_items*: set 20 in total).

In PyTorch, implemented by torch.nn.Embedding(EmbCategory, EmbDimension).

- **Target Normalization**

Also use Gamma CDF to normalize target value to the range (0~1) for speeding up learning process.

Model Design

All model design and implementation are based on PyTorch nn.Module package. Model and normalization parameters will be saved after training.

- **Linear model**

Straight-forward, design by torch.nn.Linear(inputSize, outputSize)

- **Polynomial (Quadratic) model**

Design quadratic terms by torch.nn.Bilinear(inputSize, inputSize, outputSize), then add linear terms together as output

- **DNN model**

The designed model supports multiple layers, that can be as many layers as you want, and also supports to use Sigmoid, Relu, Tanh and SELU activation functions for each layer.

- **MSE as Cost Function**

All models use MSE (from nn.Module) as cost function to monitor model performance, and also calculate RMSE on validation set.

In order to calculate RMSE between prediction and real target values, need to do reverse CDF of prediction, then calculate RMSE with real target values. Implementation of reverse CDF is as follows:

import scipy.stats as st

def reverseCDF(data, gammaPara):

fit_alpha,fit_loc,fit_beta=gammaPara[0],gammaPara[1],gammaPara[2]

data=st.gamma.ppf(data, fit_alpha) fit_beta +fit_loc*

return data

- **Cross-validation to evaluate model performance**

After shuffling the data from historical_data.csv, the whole data is split into train data (train_data.csv) and validation data (validation_data.csv) by the ratio of 90% and 10%.

The data in train_data.csv is used for training of model parameters, and the data in validation_data.csv is used for validation to get better hyper-parameters and models.

The most important thing of using train data and validation data is it will be much easier to monitor the issue of under-fitting or overfitting, then users can find good ways to fix.

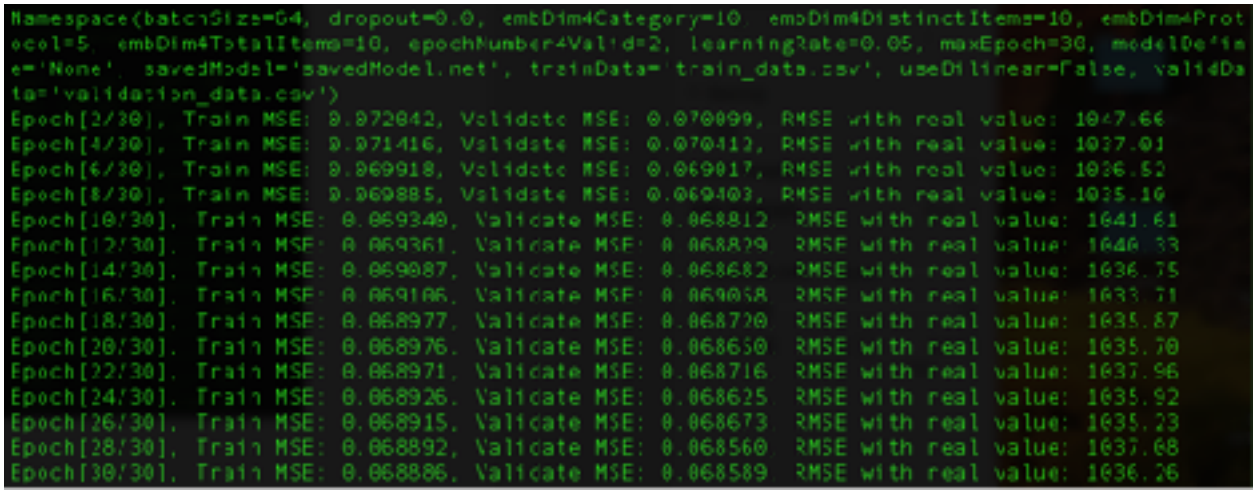
- **Use SGD and mini-batch for optimization**

Experiments

When run buildModel.py in command line, the combination of inputs of *modelDefine* and *useBilinear* can control which kind of model will be built.

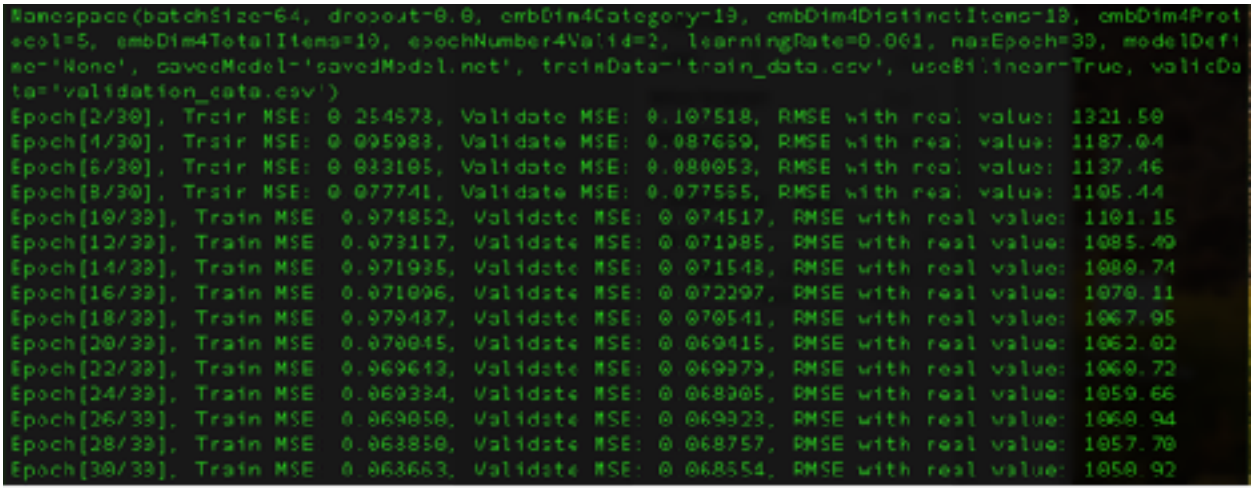
- **Linear model**

Run: python buildModel.py --modelDefine 'None' --learningRate 0.05



- **Polynomial (Quadratic) model**

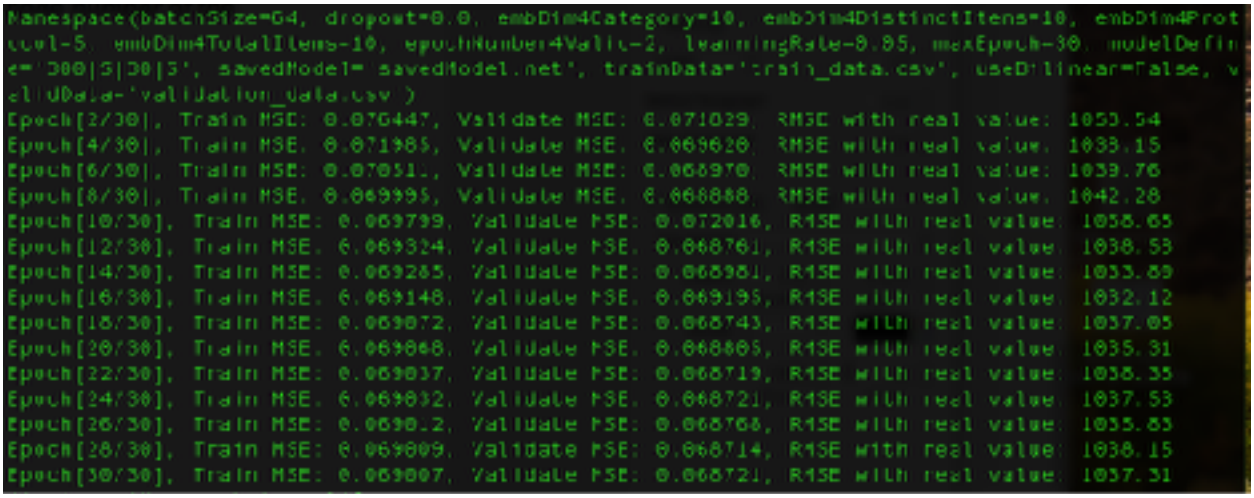
Run: python buildModel.py --modelDefine 'None' --learningRate 0.001 --useBilinear



- ***DNN model with linear features***

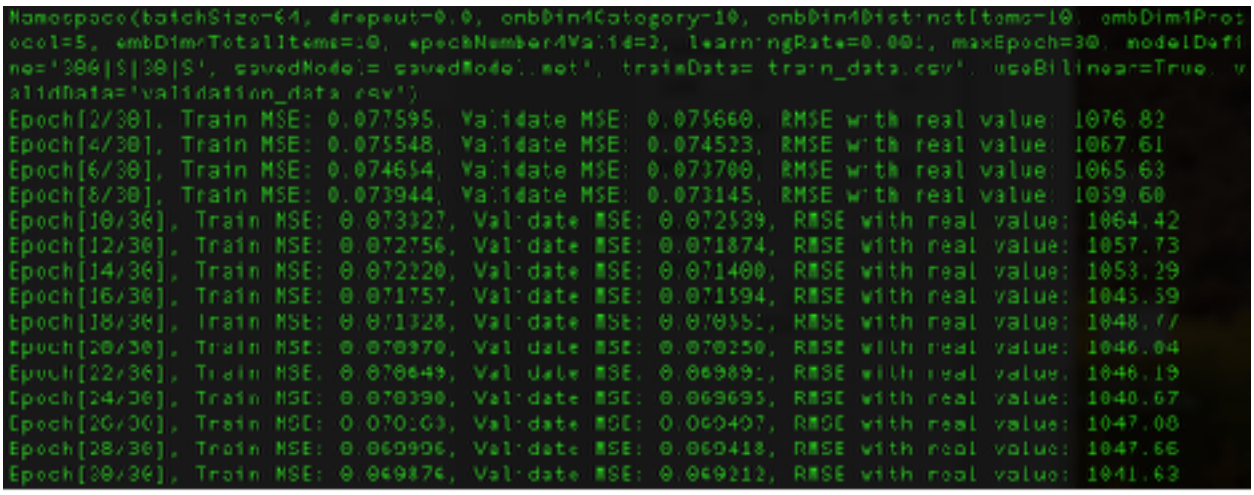
Run: python buildModel.py --modelDefine '300|S|30|S' --learningRate 0.05

'300|S|30|S' means there is two hidden layers, 300 and 30 represent the node number of the first and second hidden layer, S represents Sigmoid activation function.



- **DNN model with linear and quadratic features**

Run: python buildModel.py --modelDefine '300|S|30|S' --learningRate 0.001 --useBilinear



- **Hyper-pamameters Tuning**

When do tuning on buildModel.py, hyper-parameters include:

  - learningRate
  - batchSize
  - maxEpoch and epochNumber4Valid
  - embDim4Category
  - embDim4Protocol
  - embDim4TotalItems
  - embDim4DistinctItems
  - dropout

And also including modelDefine for DNN layers.

Conclusion and Future work

- From experiments, MSE of all models on train data and validation data after 30 epochs are similar, which means models are just fine without overfitting or under-fitting based on available data.
- The performance of all models are better than the baseline on validation set after 30 epochs.
- Linear model and DNN model with linear feature are a little better and faster than models with quadratic terms of features. And it needs to set a much smaller learning rate for the training with quadratic terms. The reason might be there is no normalization for quadratic terms, so all features have different scales.

- The model performance do not have significant improvement by hyper-parameter tuning. The reason might come from data with too much noisy information.
- In future, with more clean and complete data, DNN models should be more powerful to get more accurate prediction results, considering the main advantage of DNN is to get more complex relationship between features from training.
- Also in future, consider to add feature to present different days (weekday or weekend) and different time in each day, which should also be valuable information on the prediction.