M5 Forecasting - Accuracy: Estimate the unit sales of Walmart retail goods

Flavio Amurrio-Moya
George Mason University
4400 University Dr, Fairfax, VA 22030
famurrio@gmu.edu

Pyoung Kang Kim
George Mason University
4400 University Dr, Fairfax, VA 22030
pkim23@gmu.edu

Abstract

Methods based on decision trees dominate the field of tabular regressional problems and have shown to outperform many other methods such as weighted averaging and other statistical methods and even outperforming methods that are natively time series based on using networks such as LSTM/GRU. In this report, we plan to test out a pure Deep Learning approach which leverages key foundation concepts in both supervised and unsupervised learning to attempt to reach an acceptable score with minimum data manipulation, and virtually no data gathering/wrangling.

The "M5 Forecasting - Accuracy: Estimate the unit sales of Walmart retail goods" competition (will be referred to as the "M5 Forecasting Competition" and "The Competition") challenged Kaggle users to design a learning model that would be able to predict how many of a certain item would be sold on a given day. In this paper, we embark in the challenge to create such a model and improve on current strategies.

1. Introduction

The M5 Forecasting competition was created on Kaggle by "The Makridakis Open Forecasting Center (MOFC) at the University of Nicosia". This was done with the intent of being able to forecast the sales of items in order to minimize opportunity loss (such as not having enough of an item in stock) as well as to avoid stocking too much of an unpopular item. This competition aims to achieve more accurate and better-calibrated forecasts, reduce waste and be able to appreciate uncertainty and its risk implications.

In this paper, we discuss an approach to tabular regressional learning that is an end to end Deep Learning solution. We try various modifications to the network and it's loss function and discuss key points that will help achieve higher scores on the Competition.

We were provided with hierarchical sales data from Walmart to forecast daily sales for the next 28 days. The data covers stores in three US states (California, Texas, and Wisconsin) and includes item level, department, product categories, and store details. It also contains explanatory variables such as price, promotions, day of the week, and special events.

1.1. Data

We were given a star schema dataset of csv files that contained 3 files to be used for training.
calendar.csv - information regarding dates of product sales.

```
schema: date, wm_yr_wk, weekday, wday, month,
        year, d, event_name_1, event_type_1,
        event_name_2, event_type_2, snap_CA,
        snap_TX, snap_WI

row: 2011-01-29, 11101, Saturday, 1, 1, 2011,
     d_1, , , , , 0 , 0 , 0
```

sales_train_validation.csv - actual sales respective to date 'd', by hierarchical data.

```
schema: id, item_id, dept_id, cat_id,
        store_id, state_id, d_1,...d_1913

row: HOBBIES_1_001_CA_1_validation,
     HOBBIES_1_001, HOBBIES_1, HOBBIES,
     CA_1, CA, 0, ...1
```

sell_prices.csv - prices of products per a certain store and date

```
schema: store_id, item_id, wm_yr_wk, sell_price

row: CA_1, HOBBIES_1_001, 11325, 9.58
```

1.2. Closer Look At The Data

Graphing the data allows us to visualize some of the shopping trends.

In Figure 1 we can visualize the number of items per category. We can see a certain drop towards the end of each year which we'll see in later graphs.

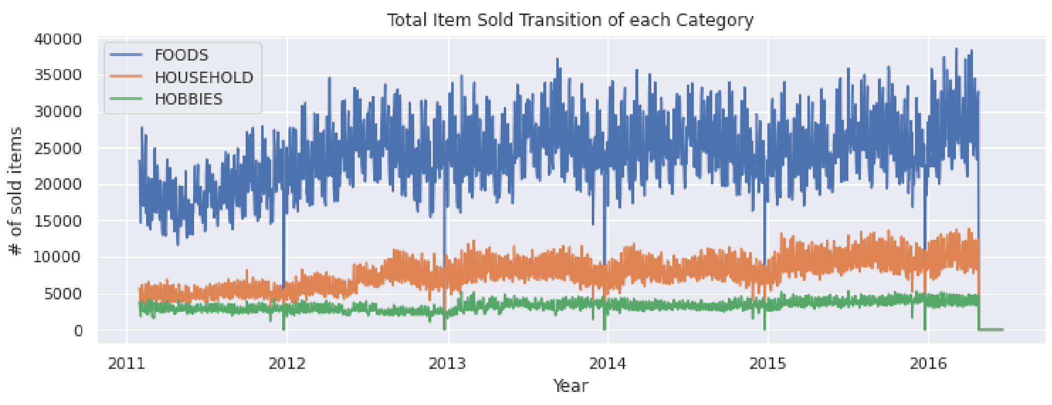

Figure 1. Total items sold of each category.

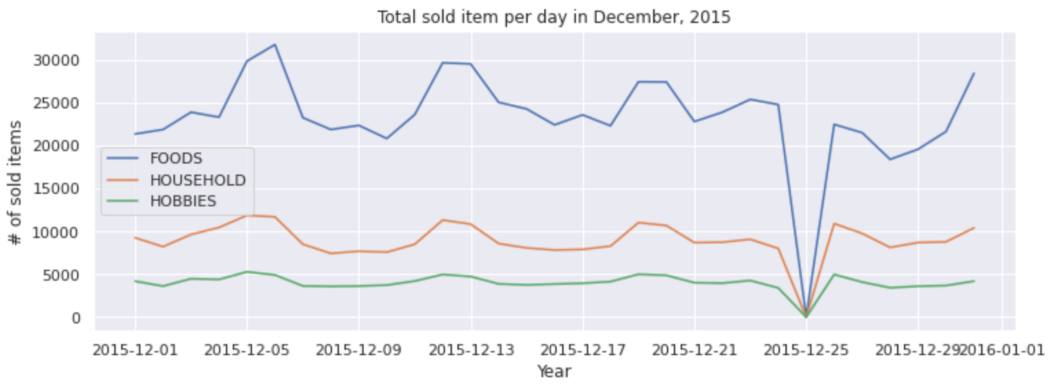

Figure 2. Total items sold per day in December 2015.

Figure 2 shows the sales for the month of December in 2015. Here it shows that the day of the drop is December 25 with is Christmas.

Figure 3 shows the number of items sold per day of the week. Here we can see how people tend to do grocery shopping on weekends and sales is lower towards the middle of the week. This is correlates to many stores doing restock on Tuesdays.

Lastly, Figure 4 graphs the items sold in California, Texas and Wisconsin showing, as we should expect, that a bigger population means more sales.

2. Approach

The approach corresponds with the files attached to the BlackBoard submission, which contains all the source code used to generate files, train, and run evaluation on.

2.1. Dataset

In order to train a Neural Network that can learn to correctly predict the number of sales per the hierarchical data shown above, i.e. item in a department, in a particular store, in a particular state, for a particular range of dates, we collapsed the star schema into a singular table containing all this particular information, with the dates and corresponding sales both becoming respondent columns. The training file ended up looking as such.

sales.csv - training file for the neural network, containing labels and its respective metadata.

```
schema: sales, id, item_id, dept_id, cat_id,
        store_id, state_id, d, date,
        wm_yr_wk, weekday, wday, month,
        year, event_name_1, event_type_1,
        event_name_2, event_type_2, snap_CA,
        snap_TX, snap_WI, sell_price

row: 0, HOBBIES_1_001_CA_1_validation,
     HOBBIES_1_001, HOBBIES_1, HOBBIES, CA_1,
     CA, d_1, 2011-01-29, 11101, Saturday, 1,
     1, 2011, , , , , 0, 0, 0,
```

The column sales correspond to the number of products sold at that certain date per the hierarchical item. In order to develop such a training set, it is necessary to check for the

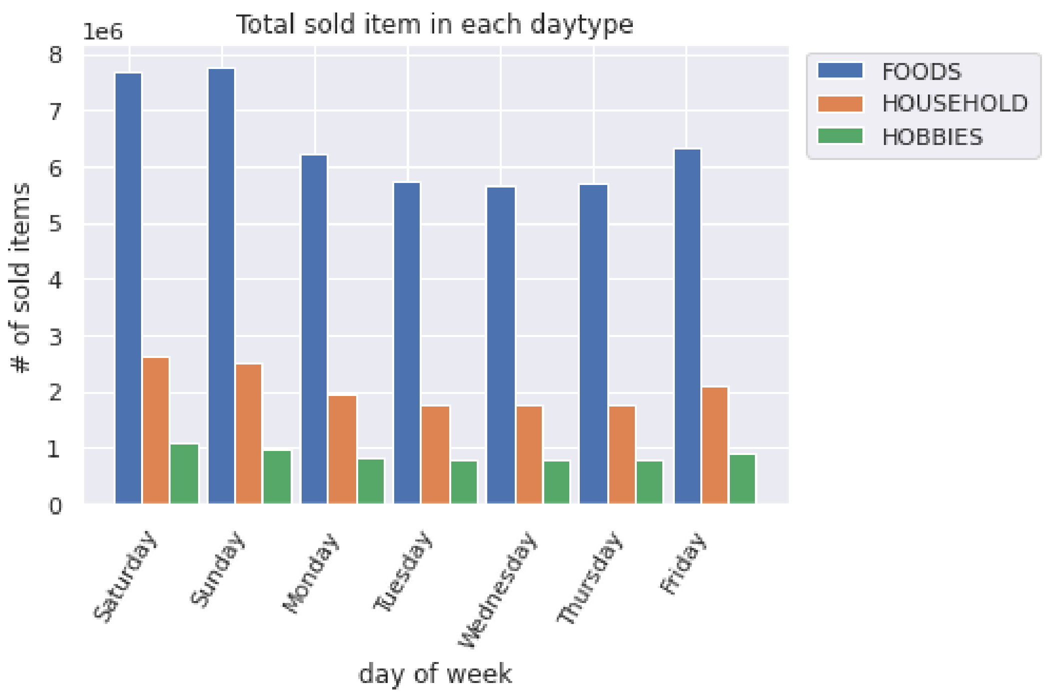

Figure 3. Total items sold each day of the week

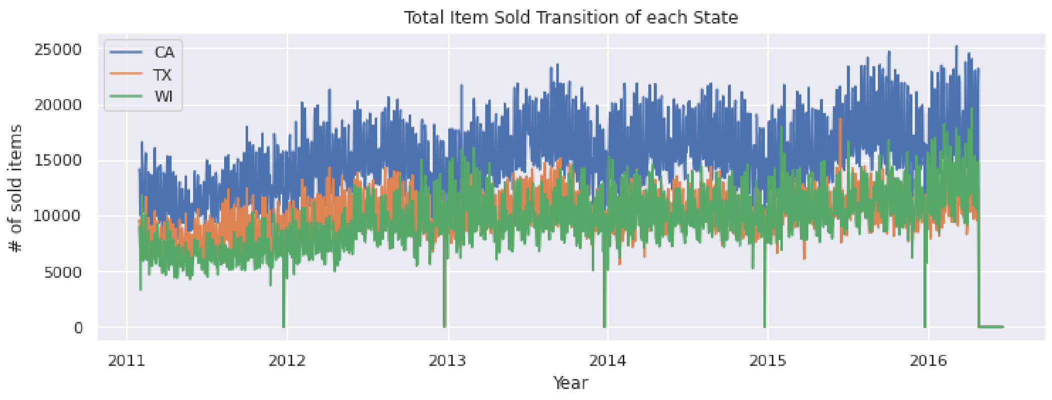

Figure 4. Total items sold in each state.

correctness of the data. We made sure there are no duplicate columns/rows, and that all data was filled correctly for a correct merge operation between the respective fields across the tables. All the joins were done using outer joins, so come resulting rows had missing fields such as a pricing per item at a particular date

The sales_train_validation.csv contained 30490 entries, however, with the sales.csv having rolled out the dates into columns of their own, and having merged with its surrounding tables, it became over 60 million rows, a job for a server computer with at least 50gb of ram. At the time, we didn't have access to a server computer, thus partitioned data and then finally concatenated them together. Once we had access to a server computer with 128gb of ram, we were able to load the dataset into memory. Loading the file into a pandas' dataframe took over 4 minutes. However, after manually typing the pandas' columns to smaller data types, the load took somewhere around 3-4 minutes. The types were define as shown on Table 2.1

2.2. Model/Data

The model we used was a tabular model from a popular repository on GitHub, https://github.com/fastai/fastai. The

Column Name	Type
snap_CA	int8
snap_TX	int8
snap_WI	int8
sales	float32
wm_yr_wk	int16
wday	int8
month	int8
year	int8
sell_price	float32

Table 1. Types Table.

underlying details worked as so. We defined the dataset to be fed to the model as a TabularDataBunch, which is a hierarchical abstraction to the Dataloader and Dataset classes and manipulation as given in PyTorch for tabular datasets. It bunches the data up to be trained upon and customized upon, as well as preprocessing. Our sales.csv confirmed to their API and was loaded successfully. The DataBunch API allowed us to fill in the previously mentioned data such as pricing, by using the median of the values across the column. We also normalize the pricing column using Standardization. The prices ranged from .01 to 107.32, before Standardization.

$$Z = \frac{x - u}{\phi} \tag{1}$$

Z = standard score

x = observed value

u = mean of the sample

ϕ = standard deviation of the samples

The model uses embeddings under the hood in order to learn meaningful vectors for categorical variables during training, that can be referenced during evaluations. Then subsequent layers are defined to learn the interaction between these embeddings and the dependent variable, i.e. sales. It contains BatchNorm layers, to speed up training as well as adding an extra layer against overfitting, and a ReLU layer for nonlinearity. The dropout rate can be specified for both embeddings and the subsequent layers.

Embeddings can have a vector size associated with them, and this is defined from a heuristic that the authors seem to have come up with through empirical means. The embeddings also have an extra size to their vocab(number of categories), in order to have another vector for an unknown field that is encountered, i.e. #na.

Below is a view of the model of the best results on our Kaggle submissions, as we have tried increasing the layers on the model to no avail, perhaps due to the limited training time we had. Again, the Embeddings are for our categorical variables + #na, whereas our only continuous variable 'sell_prices' are fed in as the normalized value directly. The method to do it is that in the forward pass of the model, it takes all the vectors of the embeddings, flattens them into one vector, then appends the continuous variable to the end of the value, thus having a single vector as input to the subsequent layers.

```
TabularModel(
  (embeds): ModuleList(
    (0): Embedding(6, 4)
    (1): Embedding(2, 2)
    (2): Embedding(2, 2)
    (3): Embedding(2, 2)
    (4): Embedding(2, 2)
    (5): Embedding(1532, 97)
    (6): Embedding(8, 5)
    (7): Embedding(8, 5)
    (8): Embedding(13, 7)
    (9): Embedding(6, 4)
    (10): Embedding(31, 11)
    (11): Embedding(5, 4)
    (12): Embedding(5, 4)
    (13): Embedding(3, 3)
    (14): Embedding(3, 3)
    (15): Embedding(3, 3)
    (16): Embedding(3, 3)
    (17): Embedding(3, 3)
  )
  (emb_drop): Dropout(p=0.04,
                      inplace=False)
  (bn_cont): BatchNorm1d(1,
                         eps=1e-05,
                         momentum=0.1,
                         affine=True,
                         track_running_stats=True)
  (layers): Sequential(
    (0): Linear(in_features=165,
                out_features=1000,
                bias=True)
    (1): ReLU(inplace=True)
    (2): BatchNorm1d(1000,
                     eps=1e-05,
                     momentum=0.1,
                     affine=True,
                     track_running_stats=True)
    (3): Dropout(p=0.001,
                 inplace=False)
    (4): Linear(in_features=1000,
                out_features=500,
                bias=True)
    (5): ReLU(inplace=True)
    (6): BatchNorm1d(500,
                     eps=1e-05,
                     momentum=0.1,
                     affine=True,
                     track_running_stats=True)
    (7): Dropout(p=0.01, inplace=False)
    (8): Linear(in_features=500,
                out_features=1,
                bias=True)
  )
)
```

2.3. Loss Functions/Forward Methods

There were two different options for the Forward functions. We tried both.

1. A simple output through the layers of the neural network, with no modifications to the output variable.

2. A squashed and re-expanded output by means of using the sigmoid function with a specified dependent variable range.

A simple output through the layers of the neural network, with no modifications to the output variable.

$$(y_1 - y_0) * sigmoid(x) + y_0 \tag{2}$$

This method is particularly useful when you are more interested in ratios of the output being close to each other, especially when paired with a method of logging the dependent variable and inversing the logging for submission. However, the mean(1.0312) and scale(0-763) of the dependent was not appropriate for this forward pass. The intuition is that sigmoid tends to saturate towards 0 and 1, and having outliers that raises the scale makes it a harder task for the Neural Network. The results, as will be discussed in the Results/Extras section shows this.

The loss for both the forward passes is a mean squared error loss function.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y_i})^2 \tag{3}$$

This is typical of regression tasks which tries to minimize the residual sum of squares.

The metric used in the paper is Weighted Root Mean Squared Scaled Error (RMSSE), however in terms of training, it's fine to use a MSE.

2.4. Training

Training the Neural Network can be a tedious task, with iterative and manual methods to try and converge the train/valid/test losses down to zero. However, we took a much more scientific approach as given from the API of FastAi. We used a learning rate finder algorithm, which employs a callback while increasing the learning rate, to see when the losses would diverge. Then we take the learning rate right before the diversion, and scale it down by a factor of 10-100 and choose that as the learning rate.

For the actual SGD process, we used a fit one-cycle method on top of the Adam Optimizer. The adam optimizer from PyTorch were given betas of (.95, .99) for the calculation of moving averages of the gradients, and the default epsilon value of 1e-08 to the denominator for numerical stability.

3. Results

The results get explained here.

4. Related Work

The relate work goes here

5. Summary/Discussion/Conclusion

The Summary Goes here

References