Machine Learning Engineer Nanodegree

Capstone Project

Rhys Shea

October 2018

Definition

Project Overview

This project is a recent competition from RStudio, Google Cloud and Kaggle to demonstrate the impact that thorough data analysis can have. For many businesses the 80/20 rule has proven true, where only a small percentage of customers produce most of the revenue. This challenges marketing teams to invest in the most appropriate promotional strategies and target the right customers to increase revenue. Statista.com shows that Alphabet's expenditure on marketing in 2017 was almost $13 billion[1] and a Wall Street Journal article shows that marketing budgets comprise 11% of total company budgets on average[2]. This is a phenomenal amount of money that could be going to waste according to the 80/20 statistic.

A Google Merchandise Store is the business of focus for this project. Briefly, the data used for this problem consists of a `train.csv` dataset and a `test.csv` dataset. Each contain data fields such as *fullVisitorID, date, device* etc. which provide information on users of the online website and how much, if any, they transacted with each visit. The purpose of this project is to predict what the total transactions are per unique visitor.

Problem Statement

In this competition, the challenge is to analyse a Google Merchandise Store (also known as GStore) customer dataset to predict revenue per customer. Hopefully, the outcome will be more actionable operational changes and a better use of marketing budgets for those companies who choose to use data analysis on top of Google Analytics data.
The prediction to be made is the natural log of the sum of all transactions per user. For every user in the test set, the target is:

$$y_{user} = \sum_{i=1}^{N} transaction_{user_i}$$

$$target_{user} = \ln(y_{user} + 1)$$

This is a regression problem, not a classification problem. Therefore the output should be a continuous natural log of the total transactions per user.
Alongside other features of potential importance, there exists a total revenue for each user, which will be used as a target. The output of the model will be the predicted total revenue for users where only the other features are known. The model will aim to find a pattern between the known features and the target.

Outline of Tasks

Data cleaning: There are many features in this dataset, which I envisaged would make for a very complex model to predict the desired target. Simplification of the data through data exploration was useful to determine which features appear to be important when predicting the target.

Preprocessing: The datasets consist of numerical and categorical data types. It is usually good practice to normalise numerical data of varying scales to values between 0 and 1 to aid prediction. It is also prudent to one-hot encode the data in order for stringified data to become useful. Data as strings are categorical, which many machine-learning algorithms cannot directly handle. Therefore it is necessary to convert categorical data into numbers.

Feature importance: In this stage I investigated the importance of different features using inbuilt methods of the chosen predictive model. This is a very powerful tool to see what features are having the biggest impact on the predictive capability of the model. By pursuing the most important features an adequate model can quickly be built and then further refined.

Feature creation: Alongside exploring the importance of the pre-set features, it is also a useful practice to create new features out of the existing ones, that may have a stronger relationship to the target variable. This was considered as a useful step to take, but upon exploring the dataset I could not find further features to engineer.

Benchmarking: In addition to using the mentioned benchmark model, I have created my own benchmark model using a simple linear regression model for direct comparison.

Final prediction model: I have implemented a Deep Learning regression model in order to solve this problem. In recent years, deep artificial neural networks have become powerful tools in solving complex problems. Though the methodology has been known for many years, it is only with the advancements in processing power that these methods have been put to practical use[3][4]. This has been built using the Keras library and TensorFlow backend, since a secondary goal of this project was to further my understanding of this technology.

Metrics

Submitting to Kaggle provides feedback on 30% of the data for the `test.csv` data, which is useful to see where my model stacks up in the broader competition. However, it is also useful to know how the model is performing before submitting to the competition; therefore the data was split further into *validation* data in order to test the accuracy of the model's prediction against data where the actual target is available for comparison with the prediction. I considered an 80/20 split of the training data to be split into training (80%) and validation (20%), but this did not seem appropriate in a time series problem. It is common in forecasting models to use the root-mean squared error as a means to evaluate performance. A root-mean squared error metric was used here to compute deviations between the predictions and targets. This was done for all users, N.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(\hat{y}_i - y_i)^2}{N}}$$

The baseline also employs the use of RMSE to test the performance of the model and so can be compared to my own model.

Analysis

Data Exploration

The data source is from the 'Google Analytics Customer Revenue Prediction' competition, the API of which is `kaggle competitions download –c ga-customer-revenue-prediction`. The data is also accessible through the Google Cloud Platform BigQuery as the `ga_train_set` dataset, and the `ga_test_set` dataset, under the `kaggle-public-datasets` project.

There are two datasets complimenting this project; `train.csv` and `test.csv`. These contain the data necessary to make predictions for each visitor to the store. The predicted data is to be submitted as `sample_submission.csv`.
The data is formatted according to the columns listed under Data Fields below. Each row in the dataset is one visit to the store. Not all rows in `test.csv` correspond to a row in the submission file, but all unique visitor IDs will correspond to a row in the submission. As shown below in the list, the data consists of both numerical and categorical data.
There are multiple columns that contain JSON blobs of varying depth. In one of those JSON columns, totals, the sub-column `transactionRevenue` contains the revenue information to be predicted. This sub-column exists only for the training data.
Train.csv

- 903,653 rows
- 12 columns initially, which expands to 55 columns once the JSON blobs have been flattened
- Time period: 1 Aug, 2016 to 31 July, 2017

Test.csv

- 804,684 rows
- 53 columns after data flattening
- Time period: 2 Aug, 2017 to 30 Apr, 2018

Calling the difference method between the columns of each loaded dataframe shows the missing 2 columns are `totals.transactionRevenue` and `trafficSource.campaignCode` from `test.csv`. This makes sense, as the `transactionRevenue` is the target for the test data. There is no overlap in the training and testing dataset timeline. I intend to use the full dataset provided, as there is only 2 years of data in total between both datasets.

Data Fields

- *fullVisitorId*- A unique identifier for each user of the Google Merchandise Store (numeric, *must be loaded as strings in order for all ID's to be properly unique*)
- *channelGrouping* - The channel via which the user came to the Store.
- *date* - The date on which the user visited the Store
- *device* - The specifications for the device used to access the Store.
- *geoNetwork* - This section contains information about the geography of the user (categorical)
- *sessionId* - A unique identifier for this visit to the store (numeric)
- *socialEngagementType* - Engagement type, either "Socially Engaged" or "Not Socially Engaged" (categorical)
- *totals* - This section contains aggregate values across the session (numeric)
- *trafficSource* - This section contains information about the Traffic Source from which the session originated (categorical)
- *visitId* - An identifier for this session. This is part of the value usually stored as the _utmb cookie. This is only unique to the user. For a completely unique ID, a combination of fullVisitorId and visitId should be used (numeric)

- *visitNumber* - The session number for this user. If this is the first session, then this is set to 1.
- *visitStartTime* - The timestamp (expressed as POSIX time).

Removed Data Fields

Some fields were censored to remove target leakage. The major censored fields are listed below.

- *hits* - This row and nested fields are populated for any and all types of hits. Provides a record of all page visits.
- *customDimensions* - This section contains any user-level or session-level custom dimensions that are set for a session. This is a repeated field and has an entry for each dimension that is set.
- *totals* - Multiple sub-columns were removed from the totals field.

The accompanying notebook provides a full exploration of the data set, an example of which is shown below. Figure 1 shows Google Chrome is clearly the preferred web browser for both visitors (left chart) and buyers (middle chart), yet Firefox users spend more on average (right chart). Further observations made from additional graphs in the notebook include:

- The desktop version of the site is used much more than mobile
- Windows is the most common OS for visitors, yet Mac users are much more likely to purchase an item
- The Americas host the greater number of visitors and buyers, in particular the United States with states such as California, New York and Washington having a higher numbers of buyers
- African buyers spend more on each transaction than any other buyers
- Google is the clear leader for traffic source but most transactions appear to be coming from within the googleplex.com domain, implying that many staff at Google are purchasing from the store

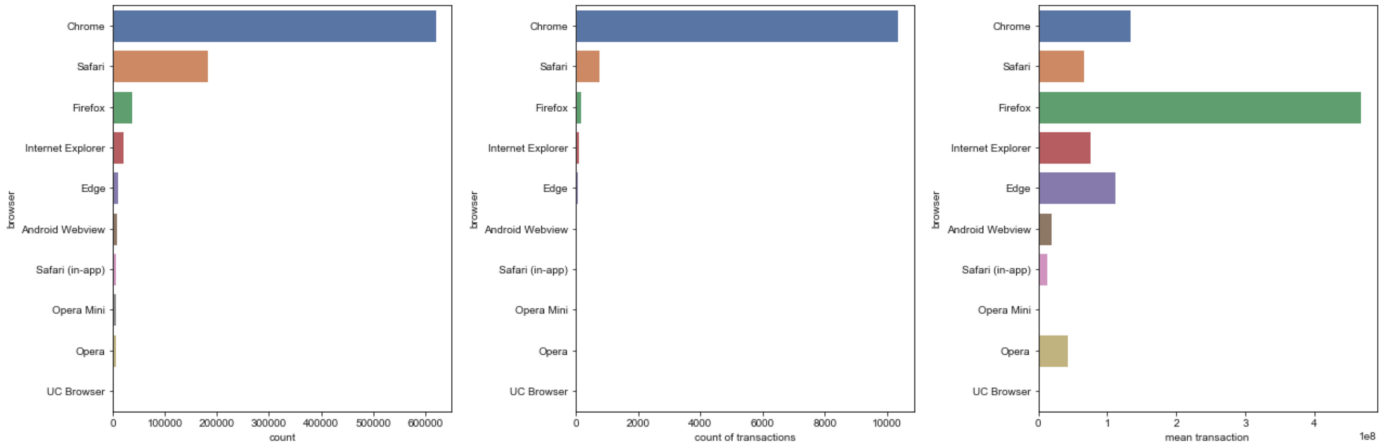


Figure 1

Figure 2 shows the number of visits to the site through time, this is taken from the `train.csv` dataset alone. There is a gradual rise in November and December 2016, likely the run up to Christmas, then a sudden drop following. In addition to this there are three clear spikes in October, February and May, possibly due to promotions and marketing campaigns. Only one of the spikes clearly matches a spike in the number of transactions in figure 3, the May spike. There appear to be no additional transactions from the October visit spike and there is a delay between the number of visits in the lead up to Christmas and a large amount of transactions in the same period. A similar phenomenon is seen in February.

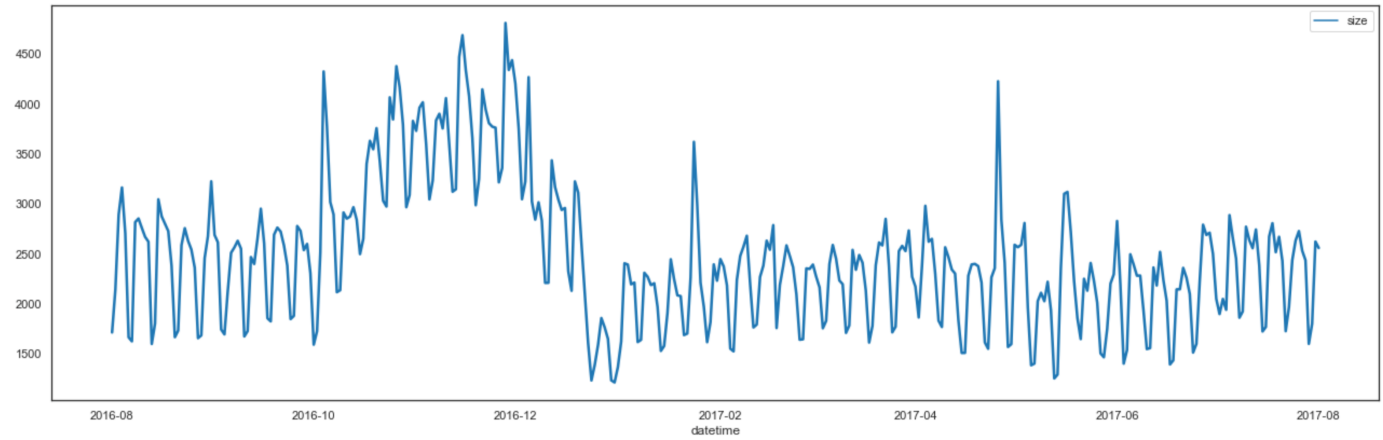


Figure 2 Number of visits to the site

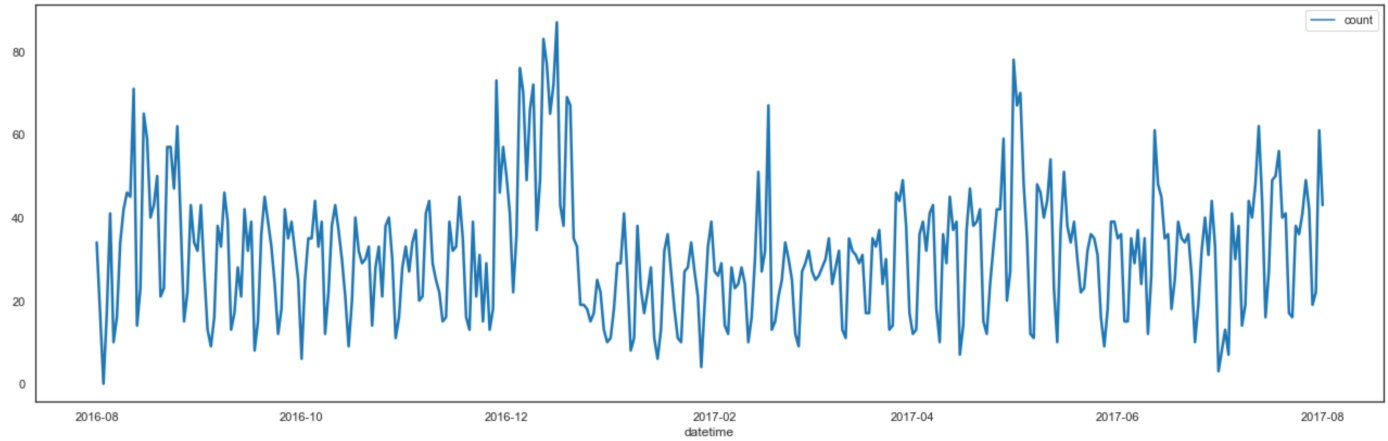


Figure 3 Number of transactions completed

Exploratory Visualisation

The figure below can be extracted from the `test.csv` dataset easily. The data is grouped by `fullVisitorId`, sorted and the natural log is taken for the sum of `transactionRevenue`. This shows a clear summary of the target data that is only supplied in the `test.csv` file.

The opening statement of the competition is "For many businesses the 80/20 rule has proven true, where only a small percentage of customers produce most of the revenue". This can clearly be seen in the figure.

In fact, the ratio is much lower than 80/20, only 1.3% of visitors have spent money when visiting the GStore.

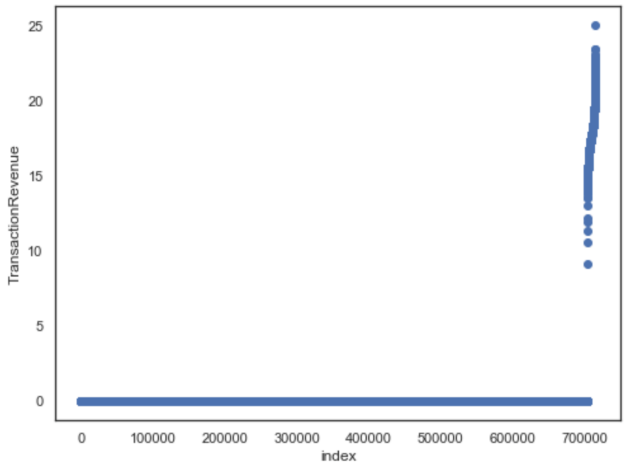
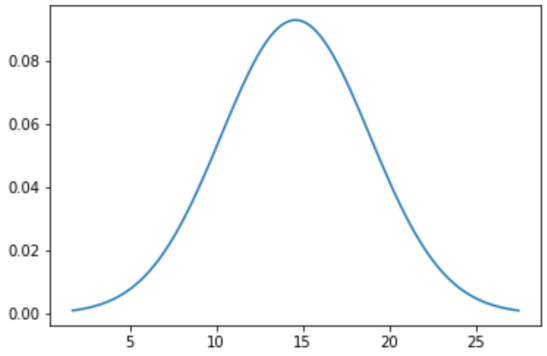


Figure 4 Transaction Revenue (y axis) per visitor to the store.

Additional statistics to be taken from this plot are:

Minimum	0.0
Maximum	25.06
Mean	14.58
Standard deviation	18.50

This gives an indication of where we should expect the predictions to be, which can be graphically shown as:



Algorithms and Techniques

Numpy and Pandas have been used throughout the project to easily handle the large arrays of data, making use of inbuilt functionality such as array subtraction, lognormal transformation and dataframe grouping for summarising the data. A formula of note is the algorithm used to load the data and flatten the JSON blobs that exist in some of the columns. This was taken and modified slightly from a kernel on Kaggle and can be found in the beginning of the notebook.

The methodology of choice to tackle this problem is a Deep Learning Regression prediction model, as there are many features to be explored. Deep artificial neural networks have become powerful tools in solving complex problems since the recent advancements in processing power thanks to graphical processing units from the gaming industry [3][4]. This is using the Keras library of TensorFlow, which has a straightforward process of implementing layers to create a neural network as well as callback functionality for more involved models. The parameters to be optimised for this model include 'number of epochs', 'learning rate', 'shape of neural network layers', 'batch size' and 'dropout percentage' as well as various optimisers and neural network structures.

In order to analyse the performance of the model ahead of having to expose the model to the `test.csv` data, the `train.csv` data has been split into k-folds by means of a time sequential formula. The process of which is described in the methodology and the exact formula can be found in the accompanying notebook.

The root-mean squared error formula is used for performance analysis and has been defined in the metrics section.

Benchmark

Alongside the benchmark models from Kaggle, I have built a simple Decision Tree Regression model. This provided a simple approach to ensure the data pre-processing and train-test split worked correctly and the data could be used for a prediction model, before committing the data to a time consuming and complex DNN. Additionally, the model provides a target RMSE score for the DNN model to improve upon.

The method for splitting the data into training and validation data will be elaborated on in the Data Preprocessing section. For now, know that the data was split with linear time in mind. Therefore the model

is tested on several runs through time as can be seen below. Interestingly the model shows a sudden improvement in the second k-fold run, however there is a lot of volatility in the first few runs in the time series. This appears to flatten out somewhat when more data is used with the final 2 runs producing a RMSE of 2.16 and 2.14. The RMSE for the test data upon uploading to Kaggle was 1.94, which is my first baseline.

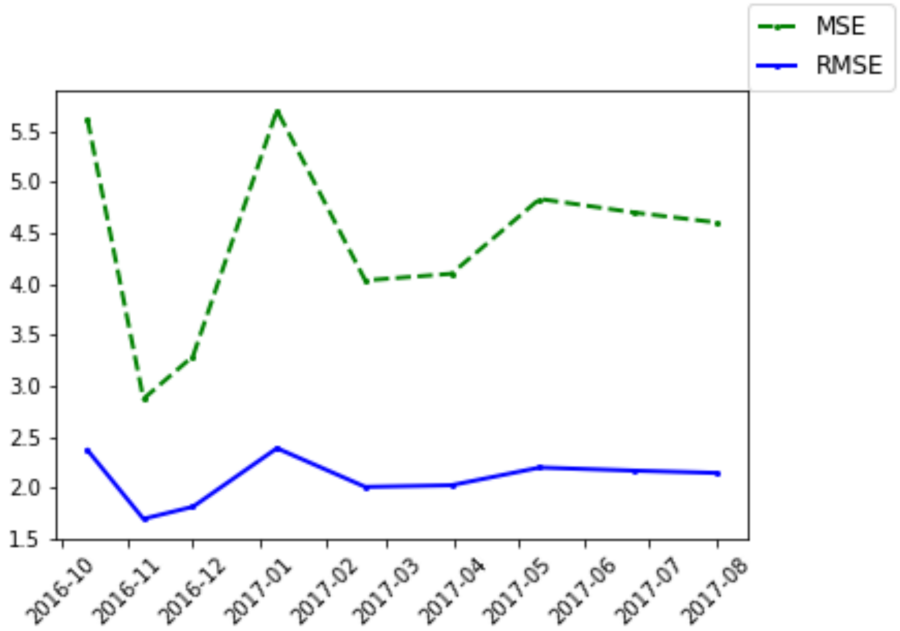


Figure 5

Using a Decision Tree Regressor as a baseline model also allowed for additional feature analysis. The below figures shows the SHAP[10] value for the most impactful features for every datapoint depending on the feature having a higher or lower value. What the plot shows is that higher page views increases the total revenue. Likewise, being in the United States increases revenue and not being in the United States decreases the revenue. Hits, has a spread of high (red) values and a concentration of low (blue) values around zero. This implies that the feature is not telling us much information and the total revenue can be smaller or higher than average with high hits. Some other features such as operating system have blue and red points mixed together, this means that some other features are having an impact on this feature.

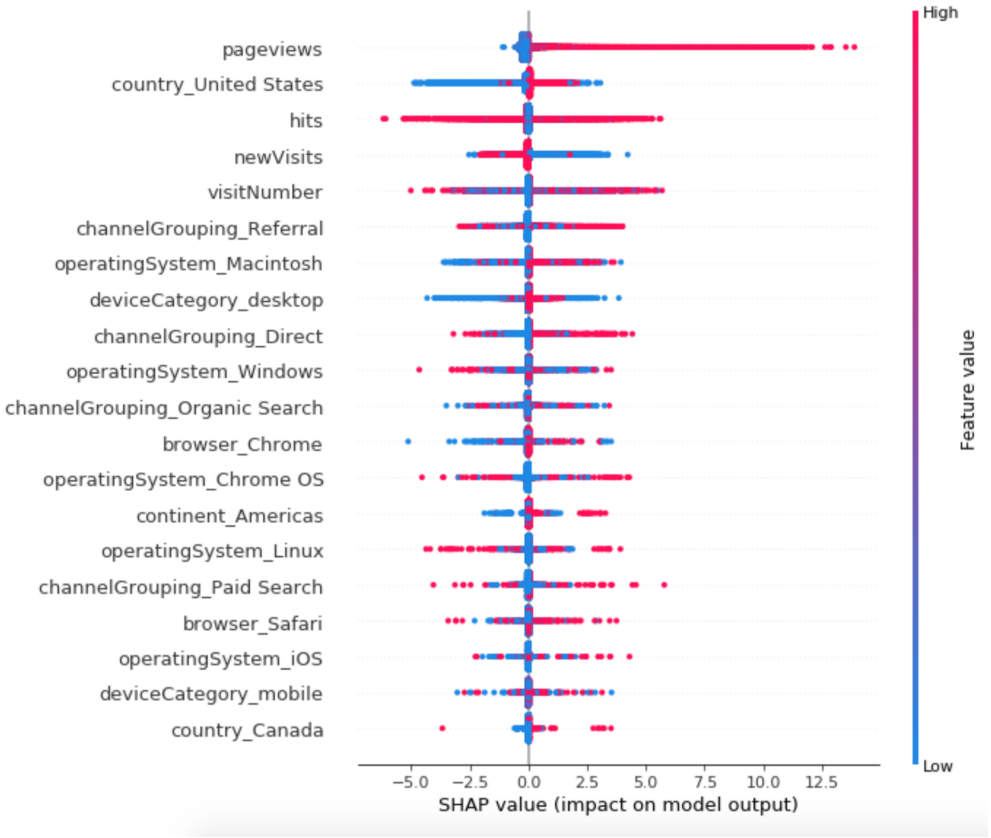


Figure 6

Two other baselines I chose were taken from Kaggle kernels for exploring the data. This kernel[5] has been used as a useful baseline in order to explore the data and begin to make adequate predictions. Using a Gradient Boost Machine it achieves a RMSE of 1.70 with LightGBM. Additionally this kernel[6] also uses deep neural networks for predicting the total revenue and served as the starting point for my own model structure. It only uses 4 densely connected layers, uses no dropout and achieves a RMSE of 1.82.

Lastly, in figure 4 it was demonstrated that nearly 99% of visitors had a total revenue of 0. If the test data follows a similar pattern then submitting 0 for every entry would achieve an accuracy of 99%. Testing this on the Kaggle submission, and thus 30% of the test data, reveals this is not the case as the RMSE for an all-zeroes baseline is 1.78. However this is a better result than the other baseline models and should absolutely be a target to beat with the final deep learning model.

Methodology

Data Preprocessing

A brief overview of the dataset revealed that there are null values throughout. Upon inspection it is apparent that there are entire columns that contain information that is not useful such as null, NaN or strings like ‘not available in demo dataset’. The code below has been used to show which columns do not contain useful data.

```
Const_cols = [c for c in train_df.columns if train_df[c].nunique(dropna=False)==1]
```

The removal of these columns saved on time and memory for easier loading and analysis.

Likewise, there are columns with many missing values such as null or NaN but that still contain other useful information. This includes the target variable totalRevenue therefore should not be removed. However, many formulas do not handle such entries very well therefore these values have been replaced with zeroes.

The next step was to sort the dataset by the date in order to correctly split the data into training and validation data. Once the data is sorted then the target variable can be extracted, as it will remain in the correct order. As the purpose of this challenge is to predict the natural log of the total revenue then the following code was applied:

```
target = np.log1p(target)
```

The `log1p` call for Numpy is equivalent to `ln(x + 1)`, as described in the problem statement.
The user IDs were extracted from the `test.csv` data, as they are not to be used in the model since the algorithm will try to use these large numbers to make a prediction. The IDs are required for later to associate with the predictions to be submitted for the challenge.

Beyond extracting the columns with little value, I have gone one step further to remove columns that I do not believe will be of interest. This follows the decision tree regressor SHAP step, that showed the most important features, as well as columns that should not be represented in the prediction model such as the visitor ID and date. A list of these columns is shown below:

```
reduce_columns = ["bounces", "Unnamed: 0", "fullVisitorId", "date", "sessionId",
            "visitId", "visitStartTime", "isMobile", "city", "metro",
            "networkDomain", "region", "adContent",
            "adwordsClickInfo.adNetworkType",  "adwordsClickInfo.gclId",
            "adwordsClickInfo.isVideoAd", "adwordsClickInfo.page",
            "adwordsClickInfo.slot", "campaign", "isTrueDirect", "keyword",
            "medium", "referralPath", "source", "datetime"]
```

Of the remaining data there are numerical and categorical types. Firstly the categorical columns needed to be one hot encoded so the algorithm can use them. This involved splitting each column into several columns equal to the number of unique entries and allocating a 1 or 0 to one of the columns depending on the original value. This horizontally expands the dataset a great deal, from fewer than 30 columns to more than 400. Finally the numerical data that isn't one-hot encoded data must be normalised. This is to prevent explosive predictions or certain parameters having larger weights because they have greater values. The numerical data was normalised to values between 0 and 1, which aids prediction.

Once the data was pre-processed the final step was to split it into training and validation groups. A common method for this is to use the train-test-split method as it randomises the split and prevents overfitting. This method is not appropriate in this case as it is a time series and may become more or less predictable through time, which is an important factor. The following methodology was employed instead, a time-series k-fold split. This splits the entire dataset into 10 groups in this case. Then the model is trained on group 1 and validated on group 2. Next group 2 also becomes part of the training set and the validation is done using group 3. This goes on until the end when the entire dataset is used, the first 90% as training and the final 10% as validation. Therefore the model is run 9 times using 10 segments of the data.

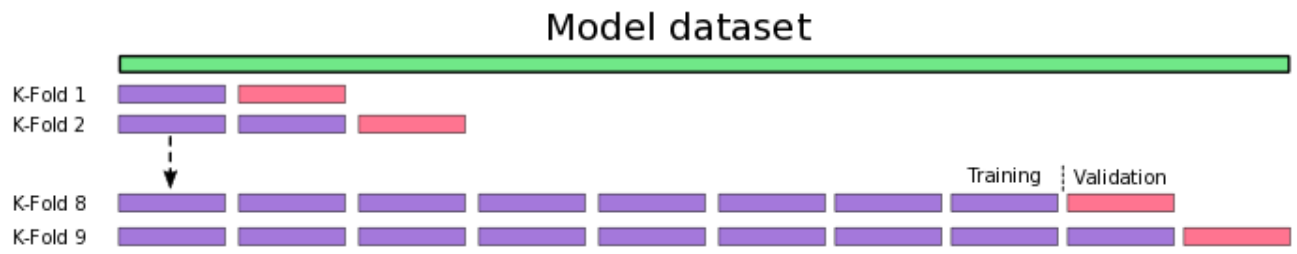


Figure 7

Implementation

During data pre-processing there were several instances of data that didn't work since there were null, NaN and string values where I did not expect them. This occurred before the decision was made to dramatically reduce the number of columns in the dataset. Initially I reduced the dataset based on only having single useless values throughout, but some columns had several values that were not useful therefore didn't flag up the first time. This led to removing several columns in the reduce_columns section.

The number of epochs to run was previously stated as 500 in the proposal, however realising the time the runs take when using a complex model on a volume of data such as this, and over several k-folds, prompted me to decrease the number of epochs to no more than 100. This was further reduced after the introduction of early stopping because several runs did not reach 50 epochs.

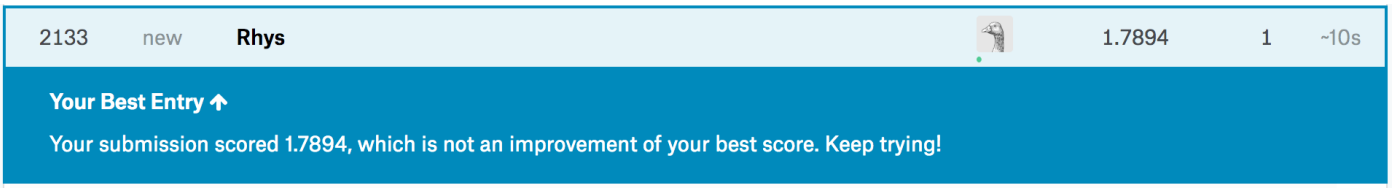
Once I had implemented the `k_fold_split` formula I realised it would be difficult to view this in time, having removed the date column. Instead I separated the date column and fed it into the same formula so it could be split accordingly, allowing me to plot the RMSE score through time. This yielded some interesting results as the predictability of the data varies through time.

Once I'd reached the stage of implementing the DNN model it became apparent I did not have the processing power to handle this computation. Therefore I had to setup a virtual machine using Amazon Web Services. The AMI used is the Deep Learning AMI Ubuntu Linux p2.xlarge machine. This caused some issues in replicating the techniques I'd already written due to differences in dependencies such as the pandas libraries. One example being that the DNN would no longer accept the data directly from the `k_fold_split` algorithm. The solution to this being that the data had to be 'converted' to a numpy array first.

Following several runs of various models I became aware of a problem that I believe was related to the learning rate. I had previously only been graphing the final k-fold run of loss and val_loss, which did not demonstrate the desired steep drop of val_loss but instead appeared quite flat. I then incorporated the graphing function into the main model loop so a graph would be plotted after each iteration. This shed more light on what was happening. In some runs there was a definite drop in validation loss and converged to the training loss in some cases. However, in other cases both loss functions remained relatively flat. This caused concern that the decay rate set in the optimizer input was not working as intended and raised questions as to whether the learning rate was resetting after each k-fold. I decided to change the input method for the decay rate, using a callback and the LearningRateScheduler instead, as well as printing the learning rate at each epoch to confirm it was reducing as I expected. This did show improvement, however not across all runs. Clearly the model is quite sensitive to the varying windows of data and requires different parameters to perform well in each k-fold.

Refinement

The initial run was a deep neural network consisting of 5 fully connected layers of decreasing size. Each layer used a ReLU activation function, the final layer had a shape of 1 for the regression mode and output a continuous stream of numbers equivalent to total revenue predictions. This model produced an RMSE score of 1.75 when using the validation data, this is the final k-split group therefore is a comparison of the first 90% of the train data to the final 10%.
Uploading the produced `submission.csv` from the `test.csv` data revealed a score of 1.7894. More importantly the validation run had a better score than the deep learning baseline model that had an RMSE of 1.82 and my own Decision Tree Regressor model with a RMSE of 1.98 when uploaded to Kaggle.



The second model had several additions, to run the model longer and hopefully converge more to the target whilst preventing overfitting. Improvements made to model:

- Reduced learning rate from 0.01 to 0.0001
- Increased epochs from 10 to 100
- Included 3 dropout constants following the largest 3 dense layers at 40%, 30%, 20%

The second model achieved an RMSE score of 1.7025 on Kaggle, a noticeable improvement on the first. However, this model took a great deal of time, as one might expect considering the granularity of the learning rate and the number of epochs. The RMSE from the validation data through time is noticeably much smaller than previous (figure below), now in the region of 0.30, yet on the test data receives a score of 1.70, suggesting the model has overfit to the data.

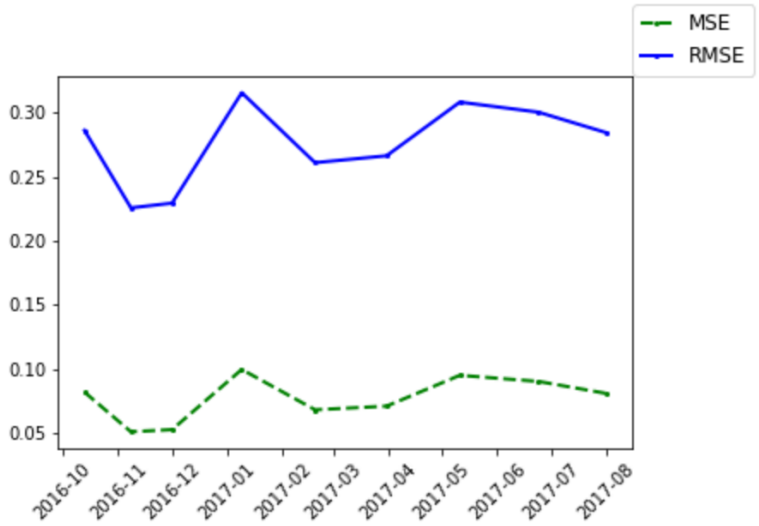


Figure 8

The total runtime came to approximately 9 hours, yet inspecting the loss function through the epochs reveals little improvement. An early stopping function was implemented to prevent this. Initially I set the early stopping function to have a patience of 5 i.e. if an improvement is not seen over 5 epochs the algorithm terminates but this did not seem to improve the runtime much beyond the first couple of k-fold

runs. I used a patience of 3 instead, which seemed more reasonable from inspecting the outputs, without unnecessarily stopping the model too early and potentially losing accuracy. Alongside early stopping, a learning rate decay was also implemented to progress quicker in the earlier epochs. The learning rate decay prevented plots as demonstrated in figure 9 where the validation loss is very erratic.

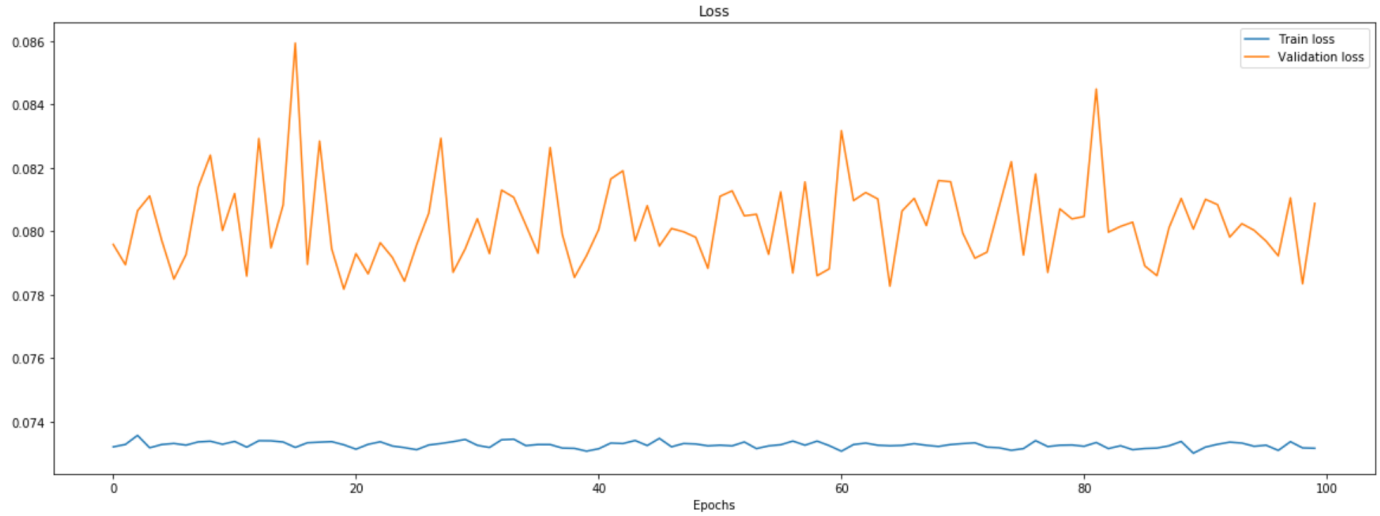


Figure 9

When introducing a decaying learning rate I set the initial learning rate to 0.1, expecting it to reduce dramatically. This turned out to be too high a value as the validation loss remained unchanged and flattened out. I eluded to issues I found with plotting the validation loss and training loss in the implementation section. To elaborate on this, once I had begun to graph the losses after each k-fold iteration I found some odd behaviour. Figure 10 shows the first run, where the desired steep drop in the loss function is seen, however the validation loss is lower than the training loss. This could be due to the model using several dropout layers, which are not used in the validation stage therefore the model performs better. However, in later iterations when more data is used, the steep drop in loss is no longer seen, plus the validation loss becomes greater than the training loss. The gap between the two also increases, which led me to believe that the model is starting to overfit to the data. Therefore I decided to introduce a larger dropout.

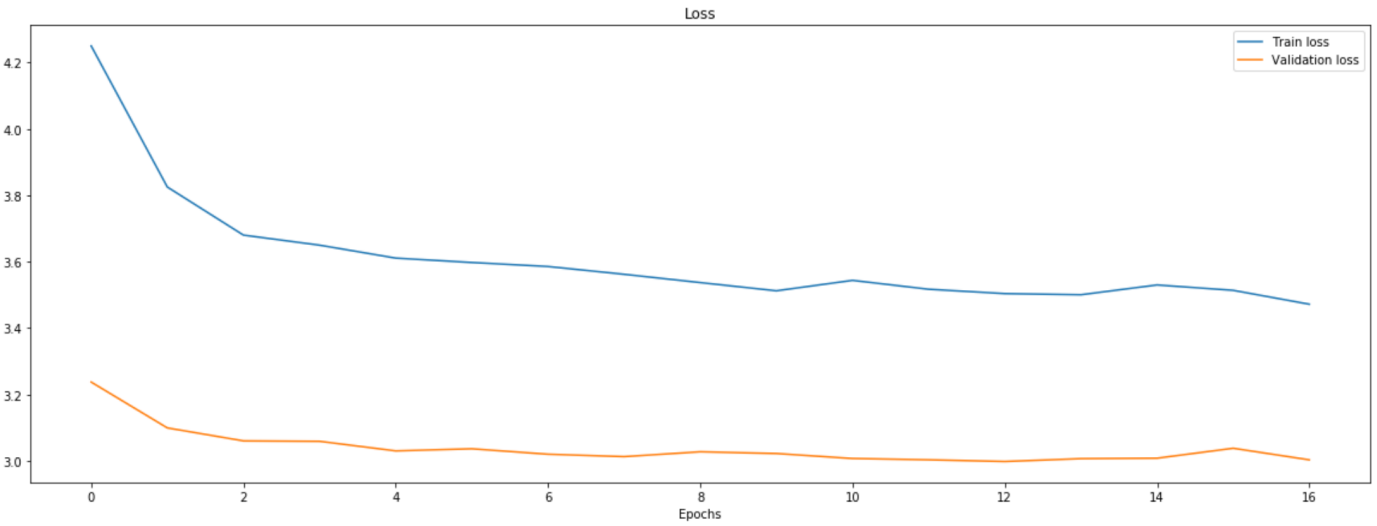


Figure 10

I also reconsidered the size of the neural network because the initial first layer had a size of 256 neurons, however there were 425 parameters for consideration. It is good practice to have an initial layer that can accommodate all parameters plus a bias weight. Therefore I increased the first layer and then halved each subsequent hidden layer. The output dense layer remained as 1 since this is a regression problem. This did not appear to have a great deal of impact on the outcome, less so than the learning rate.

Next I wanted to investigate the role of the ReLU activation function, since I had used it in every model so far. Briefly, ReLU returns a value of 0 for negative input or the positive input, in other words the maximum value of (x, 0). This is a common activation function to use in deep learning and seemed appropriate here since I do not want to output negative values for total revenue, it should be positive or 0. However, I was concerned that using ReLU throughout would limit the output data through an issue known as the dying ReLU problem, because it can zero out many weights. Therefore I decided to implement the Leaky ReLU activation function, which gives a small value alpha to the negative values as seen in the figure below. Then any negative outputs after the last dense layer can be dealt with outside of the model.

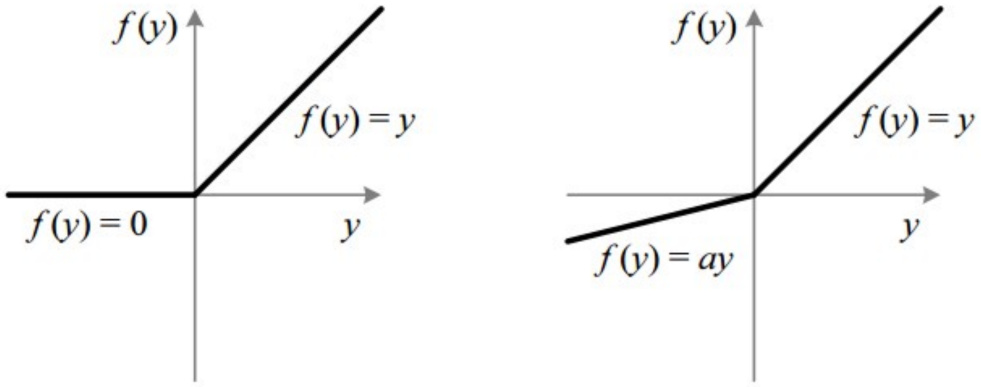


Figure 11 Standard ReLU on the left against Leaky ReLU on the right

After experimentation on several aspects of the model, it became clear that not much improvement was being made. I backtracked to the models with better scores such as model 3, firstly replicating the model to ensure the score achieved was similar (it is expected to be slightly different due to the use of dropout) and then attempting to fine-tune the parameters. I chose model 3 over model 6 because the score achieved was almost identical, but model 3 had a smaller network and hence trained faster.

I also tested different numbers of k-fold splits to see if I could reach an optimum split. The initial choice of 10 was a generic first choice to balance the number of splits with the time it would take to run the entire model. I halved this amount to reduce the runtime of the model and found results to be the same, if not slightly better. A similar result came from 15 k-folds, but the total runtime was much longer. This was disheartening to see that the training-validation split did not have a huge impact but showed that the model is stable under different scenarios of data input.

With little progress being made in terms of the RMSE score, I experimented with the batch-size, in the hope I'd better utilise the processing power of the virtual machine but again, saw no noteworthy improvement. A particularly aggravating aspect of the process was seeing that the model would continually run with the validation loss reducing 0.0001 with each epoch, but this was counted as an improvement. I do not disagree with this, however the learning rate was decaying therefore the model was unlikely to improve a great deal after reaching this point. To alleviate this issue I introduced a minimum delta to the early stopping callback, with a value of 0.0005, meaning that the model had to improve by at

least 0.0005 within a patience of 3 or the model would stop. This method reduced the runtime by a significant amount and produced an equally satisfying result.

Results

Model Evaluation and Validation

This table shows a summary of each model and the corresponding score, determined by the root mean squared error on 30% of the test data when uploading to Kaggle. I have built several models with varying parameters, some of which have made a significant improvement and others less so.

Table 1 Summary of the parameters in each model. Dense layers has in brackets the number of nodes in the input layer. LRS in the Decay row refers to the Learning Rate Scheduler callback of Keras

Parameter	Model											
	1	2	3	4	5	6	7	8	9	10	11	12
K-folds	10	10	10	10	10	10	10	10	5	5	15	5
Dense layers	5(256)	5 (256)	5 (256)	5 (256)	5 (426)	5 (426)	5 (426)	5 (426)	5 (256)	5 (256)	5 (256)	5 (256)
Activation	ReLU	ReLU	ReLU	Tanh	ReLU	ReLU	ReLU	L ReLU	ReLU	ReLU	ReLU	ReLU
Dropout	None	.4,.3, .2	.4,.3, .2	.4,.3, .2	.4,.3, .2	.4,.3, .2	.5,.5, .5	.4,.3, .2	.4,.3, .2	.4,.3, .2	.4,.3, .2	.4,.3, .2
Epochs	10	100	100	50	50	50	50	50	50	50	50	50
Learning rate	0.01	0.0001	0.01	0.1	0.1	0.001	0.001	0.001	0.001	0.001	0.001	0.01
Batch size	64	64	64	64	64	64	64	64	64	128	64	64
Patience	None	5	3	3	3	3	3	3	3	3	3	4
Min delta	None	None	None	None	None	None	None	None	None	None	0.0005	0.0005
Decay	None	None	0.1	0.1	0.1	0.1	LRS	LRS	0.01	0.01	0.01	0.01

Table 2 *refers to number of cross validation groups were used

Model	Test RMSE	Model	Test RMSE	Model	Test RMSE
Baseline DTR	1.9443	**Model 1**	1.7894	**Model 7**	1.5332
Baseline DTR (5 k*)	1.9615	**Model 2**	1.7025	**Model 8**	1.5169
Baseline zeroes	1.7804	**Model 3**	1.5143	**Model 9**	1.5093
Baseline DNN	1.82	**Model 4**	1.5324	**Model 10**	1.5098
Baseline LGBM	1.70	**Model 5**	1.7493	**Model 11**	1.5121
		Model 6	1.5128	**Model 12**	**1.5006**

The final model chosen is the one with the smallest RMSE score, meaning the predictions made by the model and the actual targets have the least difference between them. The parameters with the largest influence on the score were the introduction of dropout layers and the correct balance between the initial learning rate and its decay. The results are in-line with many other models on the Kaggle competition and a general improvement is seen, though it became very difficult to make much progress beyond the 1.51 RMSE score. However, the models that produced the better scores were run several times and proved to achieve a similar result, which is promising. An exact result should not be expected since these models are training using dropout, therefore the results will vary slightly. Additionally, changing the k-fold split was a good way of evaluating the stability of the model. Though the score varied during each k-fold split, the

outcome at the end of the entire run was largely the same, regardless of splitting the data into 5, 10 or 15 groups. The greatest affect was the time it took to run the model, so 5 groups became the more desirable option.

Therefore the final model looks like this…

- Number of k-folds / cross validation groups: 5
- Number of densely connected layers: 5
  - Number of nodes: 256, 128, 64, 16, 1
- Activation functions on each layer: ReLU
- Dropout layers: 3
  - Dropout factor: 0.4, 0.3, 0.2
- Batch size: 64
- Initial learning rate: 0.01
- Number of epochs: 50
- Early stop patience: 4
- Minimum delta of performance: 0.0005
- Decay of learning rate: 0.001

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_11 (Dense)             (None, 256)               109056
_________________________________________________________________
dropout_7 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_12 (Dense)             (None, 128)               32896
_________________________________________________________________
dropout_8 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_13 (Dense)             (None, 64)                8256
_________________________________________________________________
dropout_9 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_14 (Dense)             (None, 16)                1040
_________________________________________________________________
dense_15 (Dense)             (None, 1)                 17
=================================================================
Total params: 151,265
Trainable params: 151,265
Non-trainable params: 0
_________________________________________________________________
```

Justification

The final model, and many other deep neural networks before reaching a final model, has shown an improvement on all the baseline models. This includes the linear regression baseline, the Kaggle DNN baseline and the all zeroes submission. An important baseline to beat was the all zeroes baseline as this has no predictive logic behind it at all and is similar to a random guess.

It is difficult to say concretely whether this solution has ‘solved’ the problem. That is something very difficult to do in this type of analysis. However improvement has been shown to converge towards the optimum solution. There are other models out there that have shown better performance and further learning and improvement will be necessary to get my model to that level.

Conclusion

Free-Form Visualization

Below are two plots of the differences between the predictions and the targets. This was achieved by subtracting the prediction from the target, meaning a negative value represents a prediction that was greater than the target and a positive number is a prediction smaller than the target. Both charts below show a greater proportion of positive differences, therefore the predictions are generally not large enough. A comforting fact is the range that the predictions are in, with no outlandish predictions of very high total revenue, which would show as a largely negative value. An interesting observation here is the difference between both charts. The first is from the first of 5 cross validation runs, the second is from the final run i.e. the last segment of the training dataset. There are clearly more instances of larger differences in the second chart, showing the greater difficulty the model is having in predicting the end of the dataset compared to the beginning, when using more training data. There also appears to be grouping in both visualisations, shown by thicker 'bands' of white or blue. This is an odd phenomenon and might suggest the model is making similar predictions on consecutive datapoints.

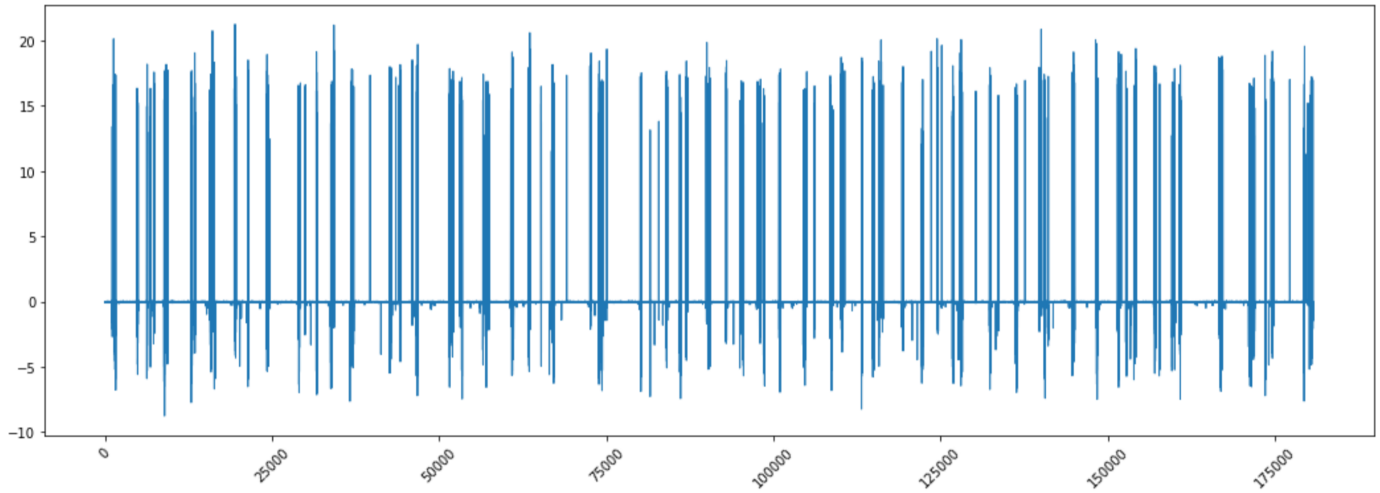


Figure 12

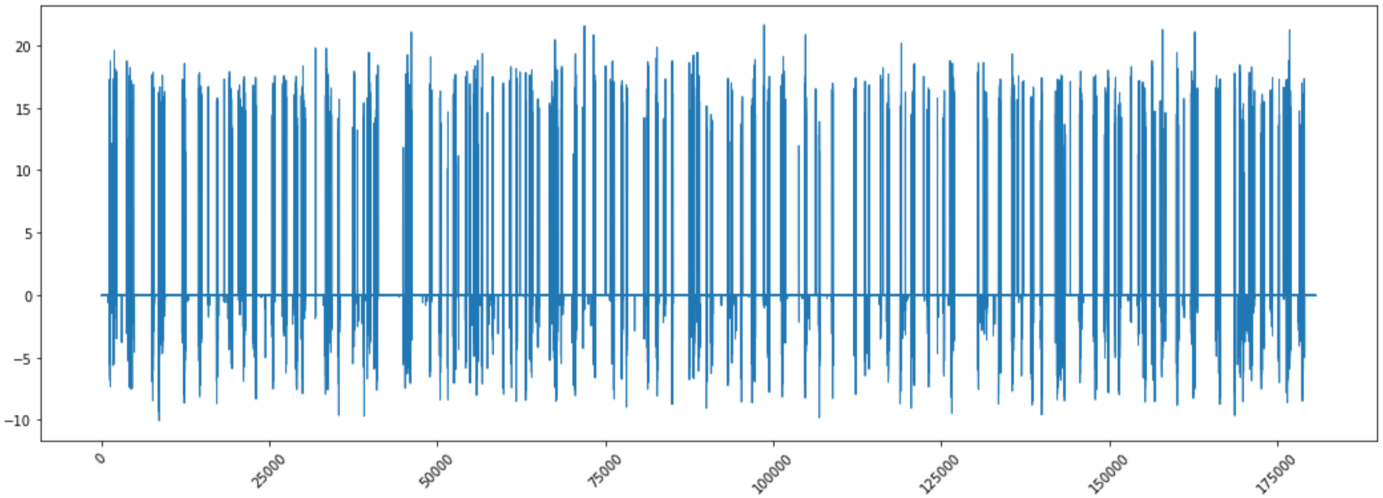


Figure 13

Reflection

The problem of interest here is a regression prediction problem, with a focus on predicting the total revenue per visitor to a Google Merchandise Store. This solution can be split into three main sections:

- data exploration
- simple prediction testing
- complex prediction testing

The first section refers to exploring the data for trends, empty data points, cleaning the data and understanding what the different parameters are. Firstly I discovered that following the flattening of the raw JSON data, there were 903,653 rows and 55 columns in train.csv and 804,684 rows and 53 columns in test.csv. Before any preprocessing of the data such as one-hot encoding, this is a great deal of data. This could easily be reduced due to the amount of blank data points and parameters that only contained a couple of useless values, this made handling the data much easier. There did not seem to be any obvious features that could be engineered either, because there were so many parameters and sub-parameters already, many of which had no association with one another. Exploration of the data showed the most important features to be generally around the themes of devices, browsers, location and page visits.

The simple prediction testing involved building a simple decision tree regression model to make an initial baseline prediction of the data. This did not perform well against the other more complex baseline models, as expected. This model allowed for further analysis in the form of a SHAP function. The SHAP analysis showed a similar conclusion to the previous data exploration, with page views, hits and country of origin having a large impact on the total revenue of each visitor.

The final stage was the creation and improvement of the deep neural network, the main model to be used for making predictions in the Kaggle competition. The initial model was taken from another Kaggle kernel, which was used as a baseline model. This consisted of several simple densely connected layers. To assess the performance of each model thereafter, the submission file created by the models predictions was submitted to Kaggle to compare the RMSE of the predictions with the targets. Several models were created in an attempt to improve on previous scores. Parameters such as the learning rate, decay rate, dropout layers and structure of neural network were tested. The use of a decay rate and refining the learning rate are the primary factors that appear to have made the biggest improvement.

An interesting aspect of the project has been learning the finer points of callbacks in Keras. Though I did not achieve a great deal of improvement in the score, delving deeper into the callback functionality has given me a greater understanding of the functionality and will allow me to improve on my models in the future.

I think the most difficult part of this project was trying to implement and experiment on so many different parameters, given the small timeframe to do so, especially considering the runtime of the model. The plots of training and validation loss for each k-fold run did not show much improvement in some cases, which knocked my confidence as the previous exercises for Deep Learning showed the kind of progress a good-fit model should make through epochs. It has certainly been a learning experience as I have come to realise how difficult it is to achieve these perfect answers with imperfect data.

This model could be used in a general setting; a relatively good answer is achieved but, as is always the case, it can be improved. Clear progress has been made throughout this process and I have achieved what I set out to do; firstly to beat the benchmarks that I had set and secondly to learn more about deep neural networks, TensorFlow and Keras. This model has been proven to be better than simpler linear models and

random guessing, and though it may not be the ideal solution, it is certainly a step in the right direction. I have seen better scores achieved in the competition, many of which seem to be using gradient boosting machines (GBM). This is certainly something I will be investigating further for the sakes of improving my machine learning knowledge, but is not in scope for this deep learning project.

Improvement

The k-fold split was a difficult part to tackle, the RMSE scores for each segment of the data were wildly different with clear sections that were harder or easier to predict. I attempted to eradicate this issue by varying the number of k-folds and hence the size of them.
I would like to implement a formula that determines what the best learning rate range is for the model and data, since it clearly varies what the optimal parameters are depending on each segment of the data. The problem with this is that I was running the model over different intervals so the data was constantly changing meaning this would have to be determined for each k-fold run. This would be computationally heavy but is one way of optimising an important parameter. An example of this method has been included and referenced in the attached notebook.
Another improvement I'd have like to make in this project is predicting at a visitor level. What I mean by this is between the training and testing data there are repeat visitors. I think being able to track each visitor through training so the model knows what the total revenue is individually so far would aid prediction in the testing stage, as it would be a sum of the revenue at training and the predicted additional revenue. I am continuing to work on the model and this is my next step to accomplish.

References

[1] https://www.statista.com/statistics/507853/alphabet-marketing-spending/
[2] https://deloitte.wsj.com/cmo/2017/01/24/who-has-the-biggest-marketing-budgets/
[3] Schmidhuber, J. Deep learning in neural networks: An overview. Neural Networks 61 (2015) 85–117
[4] http://www.deeplearningbook.org/
[5] https://www.kaggle.com/sudalairajkumar/simple-exploration-baseline-ga-customer-revenue
[6] https://www.kaggle.com/dimitreoliveira/deep-learning-keras-ga-revenue-prediction
[7] https://www.kaggle.com/julian3833/1-quick-start-read-csv-and-flatten-json-fields/notebook
[8] https://medium.com/mlreview/a-simple-deep-learning-model-for-stock-price-prediction-using-tensorflow-30505541d877
[9] Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research 15 (2014) 1929-1958
[10] https://github.com/slundberg/shap