Machine Learning Engineer Nanodegree

Capstone Project Report

Nathaniel Watkins
September 17th, 2018

Kaggle Competition - New York City Taxi Fare Prediction

Competition:
https://www.kaggle.com/c/new-york-city-taxi-fare-prediction

Dataset:
Available directly from the Kaggle competition or Big Query
https://bigquery.cloud.google.com/dataset/nyc-tlc:yellow

I. Definition

Project Overview

Making a prediction based on unknown data can be a deceptively difficult problem. What would be a simple calculation in hindsight, when you have the benefit of all the variables, often proves to be much more complicated when several variables cannot be known in advance. This project is a fantastic example of this problem and attempts to solve it with a mountain of data and a state-of-the-art algorithm.
[1]

Specifically, this project creates a submission to the New York Taxi Fare Prediction Kaggle competition, using a selection from the massive New York City (NYC) Taxi & Limousine Commission (TLC) Yellow Cab dataset that is publicly available on Big Query. The goal is to accurately predict a rider's taxi fare using only factors that could be known when booking the ride, such as pickup and dropoff location but not unknowns like future traffic conditions or the route the driver is going to take. However, I'm not approaching this competition like most other Kagglers, but instead I'm putting myself in the shoes of an engineer with the NYC TLC, and am attempting to create a production-ready model deployed in the cloud. So a key aspect of this project is that the model developed is not just some prototype without regard to scalability, but actually something that is ready to be deployed into an app.

[1] Image credit AllPosters.com:
https://www.allposters.com/-sp/Flatiron-Building-Taxi-Cabs-Yellow-Manhattan-New-York-City-United-States-Posters_i9560315_.htm

Now you might be thinking that solving this problem could be a lot simpler by utilizing a resource such as the Google Maps API, since it's gotten good enough for pretty accurate traffic predictions and determining routes, which are the critical missing variables in determining a taxi's fare. But for the purposes of this exercise, I'm operating as if that is not an option and I have to create a model based on just a few years of taxi records; perhaps, management has made the decision that utilizing a 3rd party API would be too expensive. Meanwhile, there are plenty of other, similar problems with unknown variables that don't have such a robust solution already available.

Problem Statement

The goal is to create an almost production-ready model that predicts a taxi ride's fare based only on the information any rider would be able to provide to the driver at the time of booking. Success will entail:

- ❏ Building a model that could be deployed to a production environment with minimal extra work, specifically this will be a Wide and Deep[2] regressor, which is a relatively novel approach
- ❏ Model's error rate will be at least 2 times better than the baseline, so as to justify the cost of the extra complexity and show it's worth further refinement (better than a 4.061245 RMSE)
- ❏ Utilize the enormous 55+ Million rows of data spanning taxi rides from 2009 to 2015, with some deep exploration of the data and cleaning/fixing erroneous entries as needed
- ❏ Develop a process to transform the data based just on historical records without accessing 3rd party services, and without using data that a rider wouldn't know when requesting the ride
- ❏ Error will be measured by submitting predictions to the Kaggle competition, achieving at least one above average entry.

To do all this, I will create an estimator directly within TensorFlow (TF), which will be trained and deployed in the cloud, explore and preprocess the dataset before feeding it to the TF model for training. Then to test it I'll run a batch prediction job using Kaggle's test data and use those results as an indication of the overall performance and viability of the model.

Metrics

The Kaggle competition uses the Root Mean Square Error (RMSE) as the evaluation metric, which seems to be a perfect match as it is similar to the Mean Absolute Error (MAE, which gives an average of all the errors without considering their direction), but it amplifies the magnitude of large errors. This favors model consistency as large errors, even if infrequent, will greatly increase RMSE.

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

Similarly, when training the estimator in TensorFlow, performance will be gauged using Mean Square Error (MSE), which is just the RMSE before the square root is applied. MSE has similar characteristics to RMSE, but is not as human friendly, as the values are on on a different scale and thus not directly comparable. Meanwhile, an RMSE of 4.0 would give you an idea that most results are within $3-4 of the true price.

[2] **arXiv:1606.07792 [cs.LG]**

II. Analysis

Data Exploration and Visualization

The train dataset is provided for the Kaggle competition is a ~55 million row subset taken from the total 1,108,779,463 yellow-cab trips nyc-tlc dataset publicly hosted on Google BigQuery, and the test dataset contains 9914 additional rows selected from the same BigQuery dataset. Within the Kaggle subset, are columns for all the information you'd expect to receive from a rider:

- ❏ pickup and dropoff location (separate longitude and latitude columns with floats)
- ❏ date/time (timestamp)
- ❏ number of people (integer)

All other columns in the BigQuery set have been removed, such as `rate_code` (since it'd be a little too helpful in predicting the fare) and `payment_type` (since that shouldn't have much bearing on the prediction). Additionally, an ID column was created with a `key` value (string) consisting of the timestamp and an integer for uniqueness, and the train set contains the `fare_amount` as that's that target value to predict.

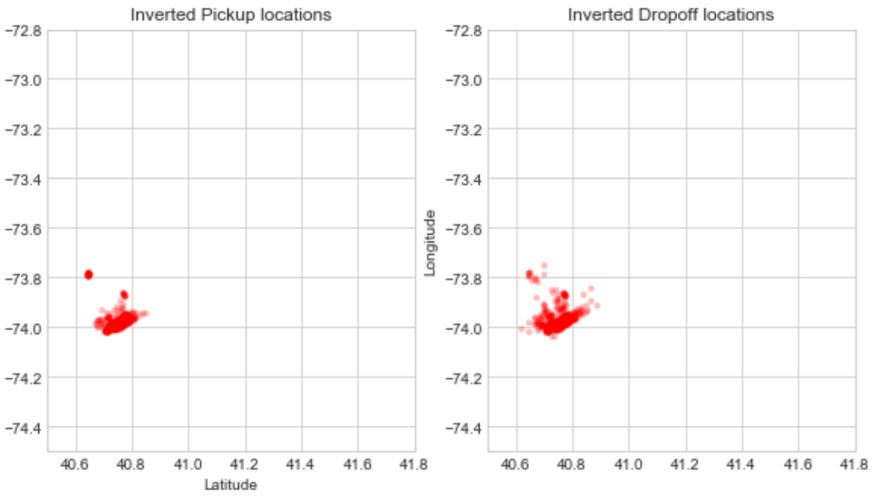
	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count
count	2.000000e+06	2.000000e+06	2.000000e+06	1.999986e+06	1.999986e+06	2.000000e+06
mean	1.134779e+01	-7.252321e+01	3.992963e+01	-7.252395e+01	3.992808e+01	1.684113e+00
std	9.852883e+00	1.286804e+01	7.983352e+00	1.277497e+01	1.032382e+01	1.314982e+00
min	-6.200000e+01	-3.377681e+03	-3.458665e+03	-3.383297e+03	-3.461541e+03	0.000000e+00
25%	6.000000e+00	-7.399208e+01	4.073491e+01	-7.399141e+01	4.073400e+01	1.000000e+00
50%	8.500000e+00	-7.398181e+01	4.075263e+01	-7.398016e+01	4.075312e+01	1.000000e+00
75%	1.250000e+01	-7.396713e+01	4.076710e+01	-7.396369e+01	4.076809e+01	2.000000e+00
max	1.273310e+03	2.856442e+03	2.621628e+03	3.414307e+03	3.345917e+03	2.080000e+02

A quick summary of the train dataset's statistics reveals some questionable numbers. Within the first 2M rows, there are fares as low as -$62, which could be due to refunds or just erroneous, but I don't suspect the test dataset contains refunds or otherwise negative fares. According to the TLC[3], they charge an initial $2.50 for every trip, so there shouldn't be any fares under that amount, and all records below that threshold were removed. Next, there are some trips with 0 passengers, which might be due to a delivery or something, but since the test data doesn't contain trips with 0 passengers, those offending records were also removed. Finally, from these latitudes and longitudes, you'd think that New York city spanned half the globe; I don't think the big apple is quite that big.

Based on the coordinates in the test set (and sanity checked using this tool[4] incase the test set was too limited), we'll consider every instance of a coordinate outside -74.5 to -72.8 longitude and 40.5 to 41.8 latitude to be an outlier. But before discarding these outliers, let's see if any data is

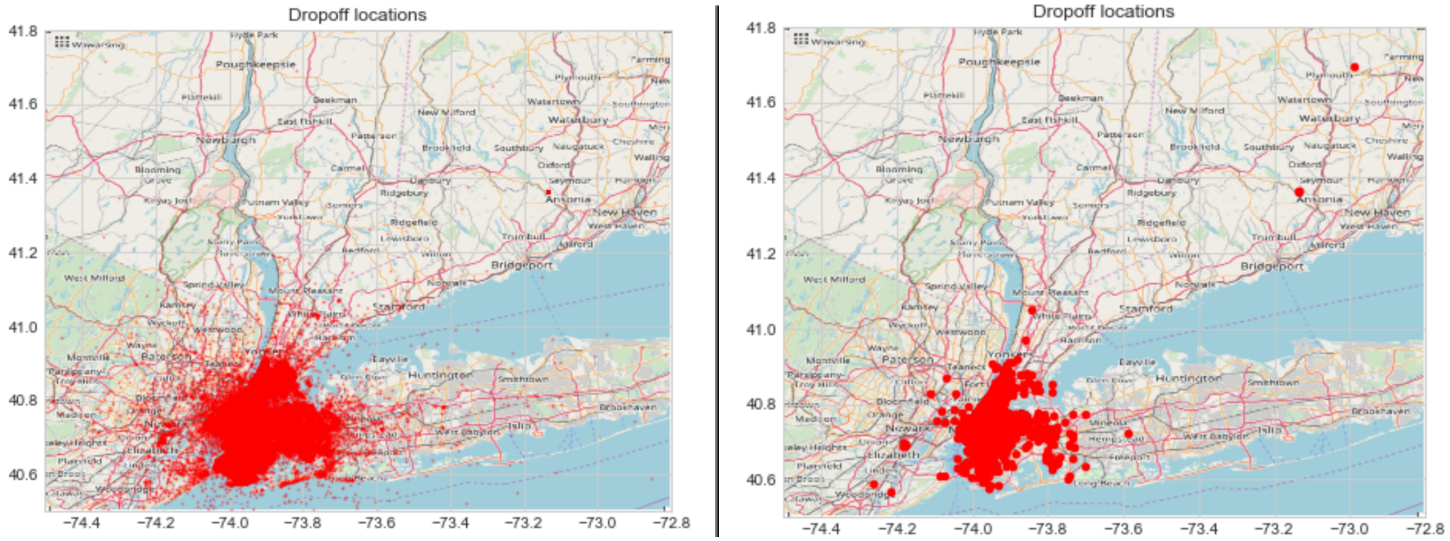
[3] http://www.nyc.gov/html/tlc/html/passenger/taxicab_rate.shtml
[4] http://www.mapdevelopers.com/geocode_bounding_box.php

salvageable. A lot of them just have 0s for coordinates or other meaningless values, but a few (955 out of the first 2M records) contained coordinates that completely (both pickup and dropoff) fell within an inverse of this bounding box, which is geographically located in Antarctica and probably doesn't have many taxi cabs. Considering that these data points otherwise seem legitimate and could simply be due to a misconfigured data logger or an import error into the main dataset, we'll reverse the ordering on these coordinates and re-incorporate them back into the training data.



Now that we have the more extreme outliers handled, let's look into another potential source of junk data: NYC and the surrounding areas have a lot of water, and while they do have water taxis, that data is not present here. And sure enough, look at all those points in the water.

Here's the train set: compared to the test data:



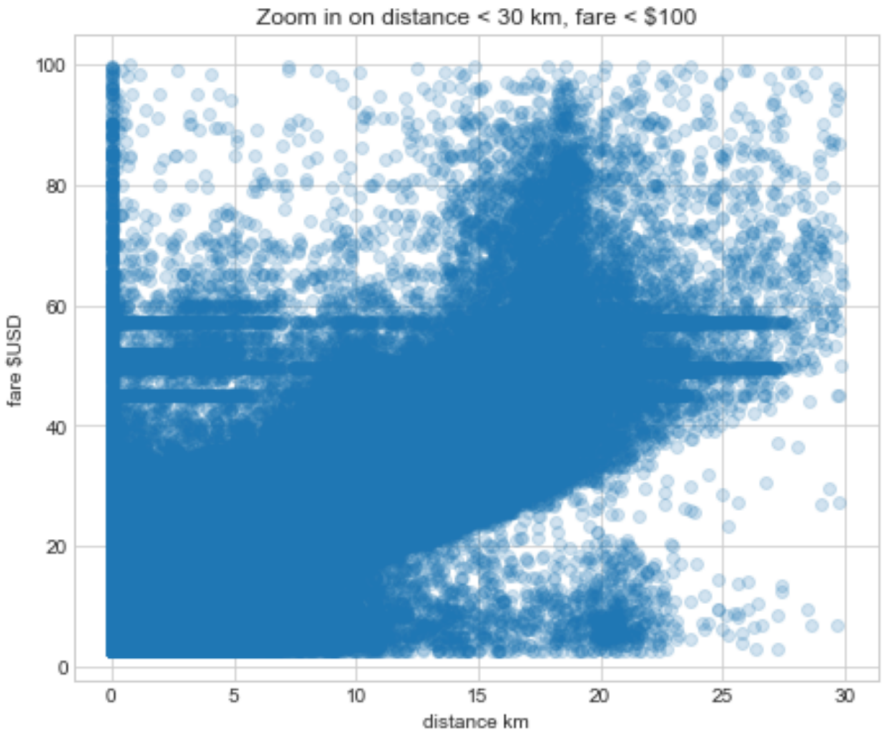
To fix this, a mask was created of the same section of the NYC map, with all visible water manually removed, then using the mask, all data-points within a water section were dropped from the dataset.

The next step is to calculate the straight line distance in kilometers between the pickup and dropoff points using the Haversine formula[5] to account for the spherical nature of the earth (deal with it flat-earthers), which is a bigger factor at longer distances, and add it as a column on the dataset. While straight-line is not nearly as useful as the actual route the taxi took, it wouldn't be feasible for us to predict that without using a 3rd party tool, which is not an option for this project. This should result in much greater route variation for shorter rides due to the extra distances added from turns

[5] https://rosettacode.org/wiki/Haversine_formula

accounting for a larger proportion of the total route distance, but the straight distance is still a very helpful indicator here.

Also notice that a good number of the data points in the train set are at or near 0 km distances, so we'll filter out the data into very short distances (less than 0.1 km) and longer distances. Looking at the statistics, nothing sticks out about these points that indicates a pattern useful to correlate them. While that data could be accurate, it's hard to imagine how these data-points could be useful under these circumstances. But also notice that the test dataset also contains data in this category, which is slightly concerning, since so far the test set has seemed to contain perfect data. However, this seems unavoidable, so we'll just remove them from the training data and hope the model prediction gets close enough based on the other inputs in the test data or in real world usage. Also, since the test data's validity is in question, we also check it for points in the water and find 2 out of the 9914, which isn't too bad.



Other data exploration was done, including charting the density of pickups/dropoffs on the map, a histogram of ride distance counts, calculating the average rate per kilometer ($3.40/km, which ended up being used for the baseline), and comparing high frequencies of certain fares with airport locations. While all of these were helpful for understanding the data on a deeper level, they don't represent any transformations that we should make to the data, but rather they signal factors that we hope the model will pick up on and work into its calculations without us having to hard code into our data. The timestamp was also split off into several columns: `hour`, `day_of_week`, `day_of_month`, `week`, `month` & `year`; but no filtering of the data was performed based on these factors since they all seem like valid values (and should be to have been a recognized timestamp). Similarly, while these time/date columns certainly equate to deviation in fares, due to different rates at night/day/rush-hour and rate increases over the years, these are the types of things that we want our estimator to correlate from the data and factor into its predictions.

	key	fare_amount	pickup_longitude	pickup_latitude	dropoff_longitude	dropoff_latitude	passenger_count	hour	day_of_week	day_of_month	week	month	year	distance_km
0	2009-06-15 17:26:21.0000001	4.5	-73.844315	40.721317	-73.841614	40.712276	1	17	0	15	25	6	9	1.031069
1	2010-01-05 16:52:16.0000002	16.9	-74.016045	40.711304	-73.979271	40.782005	1	16	1	5	1	1	10	8.449763
2	2011-08-18 00:35:00.00000049	5.7	-73.982735	40.761269	-73.991241	40.750561	2	0	3	18	33	8	11	1.389644
3	2012-04-21 04:30:42.0000001	7.7	-73.987129	40.733143	-73.991570	40.758091	1	4	5	21	16	4	12	2.799485
4	2010-03-09 07:51:00.000000135	5.3	-73.968094	40.768009	-73.956657	40.783764	1	7	1	9	10	3	10	1.998886

Algorithms and Techniques

Wide and Deep Learning is a relatively new technique developed by Google AI researchers that combines the generalizability of a Deep Neural Network (DNN) with the specific memorizational capability of a wide model like Linear Regression, basically by training and running both types of models in parallel. This allows the two models to utilize their respective strengths while covering for the other's weaknesses.

A wide algorithm is really good at picking up on similarities in data and memorizing how those aspects relate, so for example it might notice that a taxi ride from Central Park to Queens College around 4p on a Thursday is similar to other rides in the Train Dataset from Central Park to Queens College around 4p on Thursdays, thus predicting a similar price and probably being pretty accurate. However, a wide model might have trouble accurately guessing a ride from the Bronx to Stamford, CT due to few similar rides, if any.

Meanwhile, a deep algorithm is good at abstracting the factors and connecting how they relate to each other, so it might notice that the rides to Bridgeport or New Haven, CT (both past Stamford) tend to have pretty consistent fares with relation to their straight-line distance (possibly due to less traffic and mainly highway driving), and then apply that logic to all rides heading North-East out of the city, Stamford included. However, when it comes to the ride from Central Park to Queens College at 4p on Thursday, a DNN might get too distracted by various higher-level abstractions, such as general traffic levels (measured by fare/distance) around 4p on Thursdays, instead of focusing on the specifics that could yield a highly accurate answer in this case.

The Wide and Deep approach is perfect for this problem due to the enormous amount of data in this dataset, providing a lot for the wide model to work with, while the data is also so varied and highly dimensional, preventing the wide model from memorizing everything thoroughly; so then the deep model can pick up on the meta factors at play that influence each column of data, picking up where the wide model doesn't have anything specific to compare it to.

Additionally, the feature columns will be crafted in ways that not only for compatibility with the wide and deep TensorFlow estimator[6], but also to inform the algorithm about the structure and relationships, aiding in training and prediction. Since all the data fields are numeric, we could just import them all into the model as a `tf.numeric_column`, which could work but ask yourself "is November really 11 times more than January" or "what do you get if you add December and May"? A numeric column tells the model that the values are on a scale that's directly comparable, which isn't the case for anything but our `fare_amount` (the target value) or the `distance_km`. However, this is where we need to start when importing our numbers into TF.

Next, many of these categories fall into discrete buckets where each bucket can be represented by a number (or a name) but it doesn't make sense to do math using these numbers; so

[6] https://www.tensorflow.org/versions/r1.9/api_docs/python/tf/estimator/DNNLinearCombinedRegressor

January falls into bucket 1 and May is 5. From these buckets, we'll apply feature crosses[7] to some of our data, which tells the algorithm specifically how a couple of the columns relate to each other and creating a dimensional reduction from those columns. For example, we'll use `tf.crossed_column` on the longitude and latitude buckets to create discrete 1.1km squares across our map. This is super helpful for helping the model memorize the data in a meaningful way (which is okay since we have enough data to memorize that it becomes pretty generalizable). We'll also use the feature cross technique on:

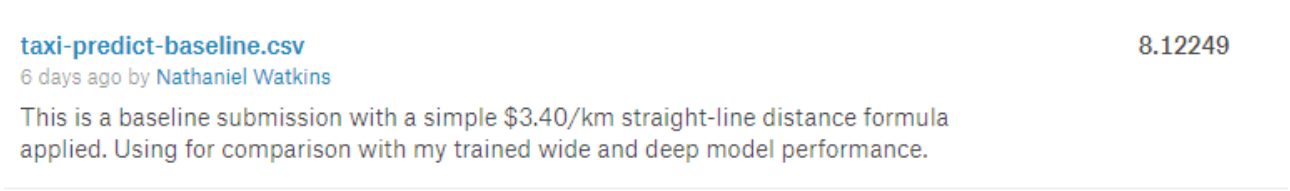
- The day of the week and time - ex. 4-5p on Thursdays bucket or 1-2a on Fridays bucket
- The day and month - ex. 1st Monday in May or 4th Friday in August
- The month and year - ex. January 2009 or December 2011

Benchmark

Since we're creating this model within the context of a (pretend) production setting, with the goal of delivering this model into an actual (fictional) product, we need to keep costs in mind throughout this process, not just the dollar costs to develop and deploy the model but also the computational cost to run the model with real users. Perhaps the model would remain deployed in the cloud where predictions would be sent to an API in real-time, which could end up costing the NYC TLC a significant amount to maintain, with over 9 million API calls a year. Or perhaps the model could be deployed into TensorFlow Lite[8] and run directly on users' devices, but then the model would make the install size much larger and likely take a noticeable, or even annoying, amount of time to make an estimation whenever the user requests a ride.

Therefore, we need to determine a benchmark to compare our model's performance against just doing a simple mathematical calculation. In the data exploration phase we came to an average cost per kilometer calculation of $3.40, which we'll use as a benchmark to compare against. Then again, simply beating the benchmark by 1% almost definitely wouldn't yield enough benefit to justify the cost of running a complicated Machine Learning model. Where that line is drawn is primarily a business decision, but for our purposes, we'll say that the model needs to be 2 times better than the benchmark to justify its cost and further investment in refining the model; since our measurement is RMSE, we'll divide the benchmark error by 2 for our break even line. In other cases, this line could be 5, 6, or even 10 times improvement over the baseline.

Using this calculation a `fare_amount` column was added to the test dataset, and a benchmark prediction file saved then submitted to Kaggle, yielding a RMSE of 8.12249 and our threshold to determine model viability will be 4.061245.



[7] https://developers.google.com/machine-learning/crash-course/feature-crosses/video-lecture
[8] https://www.tensorflow.org/mobile/tflite/

III. Methodology

Data Preprocessing

Much of the data preprocessing was performed in conjunction with the data exploration and discussed in detail above, but that was only done on a small subset (the first 2 million rows) of the total dataset due to memory constraints (the full 55M row CSV is over 5.5GB, but it'd be much larger as a Pandas DataFrame held in memory). To ease the memory burden a numeric downcast function was created to reduce the variable sizes of the values stored in the DataFrame, which was rather successful resulting in about a 40% size reduction. Unfortunately, TensorFlow threw an unexpected dtype error when attempting to train on the data that had been downcast, but thankfully the error went away when the dataset was prepared without using the downcast step.

In the end it wasn't feasible to load the entire train dataset into memory to process all at once and the preprocessing needed to be systematized into something that is nearly ready for deploying into a production model. So every step that transformed the data in some way was converted into a function or a series of functions. This enabled the creation of a single master `prepare_train_valid` function that uses Pandas' `chunksize` option to load chunks of the file individually then calls each data conditioning step on the chunk before saving the chunk to a processed training file. This also organized the various steps of preprocessing so that if/when it came time to create the production model, the functions needed to format the user input data would be easy to find and copy.

The final function in `prepare_train_valid` is `split_by_hash`, which divides the entire train dataset into a train and validation dataset in a repeatable manner without introducing bias.

```
    df["hashed_result"] = df.key.apply(hash) % 100

    train = df[df.hashed_result < train_percent]
    validate = df[df.hashed_result >= train_percent]
```

Instead of simply splitting based on the result of a random number generator, which would split the results differently each time it's run and wouldn't be scientifically rigorous, a hash was generated from the `key` column and the modulo of 100 saved in a new column. Then the ratio of the split could be determined with the `train_percent` variable; for example, a `train_percent = 73` would result in roughly 73% of the data getting saved into the train dataset, and roughly 27% saved in the validation file. This method also avoids bias as much as possible, since if we were to split based on one of the columns that we're using for training, we might inadvertently under/over-represent certain types of data in one set, which could influence the model's bias in training. For instance, if we split based on `passenger_count > 2` we might not get comparable types of rides in the validation set, and thus the model couldn't tune itself properly. Worse, if we were to split based on a location column, certain neighborhoods might get predicted for inaccurately low fares, and as a result cab drivers might tend to avoid those neighborhoods. Plus, we might not get the split ratio that we were hoping for.[9]

[9] https://www.oreilly.com/learning/repeatable-sampling-of-data-sets-in-bigquery-for-machine-learning

Implementation

Creating the model in a production friendly format, involved working with TensorFlow directly and deploying the model on Google Cloud Platform (GCP)'s ML Engine. Creating models within SciKit-Learn or Keras is easy, but doesn't easily scale up for deployment in a production setting.

A project was created in GCP, and a Cloud Storage Bucket was created for the project with the train, test and validate data loaded in. Then a Datalab VM was created to run the TensorFlow commands from the "Train, Deploy, Predict with MLengine" notebook, but running that aspect locally would have yielded the same results. The notebook mainly functioned as a one-stop-shop to issue a series of bash commands including:

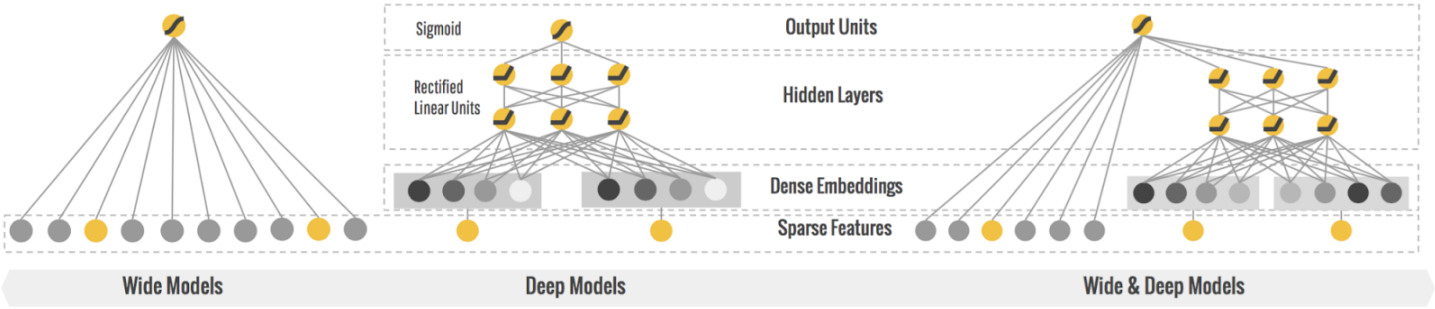
1. Setting environment variables
2. Setting up permissions
3. Wrangling files into place
4. Running Tensorboard
5. Creating the training job in ML Engine
6. Deploying the trained model
7. Submitting online/real-time predictions
8. Submitting batch prediction jobs

This notebook was also used to create the baseline prediction file.

The TensorFlow graph was constructed within the "model.py" file, which dictates everything TF does. First, we specify all the columns in the input files, and which one is the target column. Then we tell TF how to read the CSV depending on the mode, shuffling the data if it's training or stopping after the first epoch if it's validating.

After that we instruct TF how to create feature columns from the data, starting with making a `numeric_column` for everything. Next, we transform all but `distance_km` into `bucketized_columns`, specifying the number of buckets and where the splits are using a list of split points based on the each column. For example, the longitude and latitude feature columns split every 0.01 degrees. And then the previously discussed `crossed_columns` combine these bucketized columns to create a grid of NYC and the surrounding area with 1.1km squares. Finally, the `feature_columns` had to be converted into `embedding_columns` due to compatibility with the deep portion of the algorithm. We created a list of columns to feed to the wide model and a separate list for the deep model, disregarding all the `numeric_columns` except for `distance_km` because they were replaced.

```
pickup_longitude = tf.feature_column.bucketized_column(
    source_column = feats[0],
    boundaries = list(np.linspace(-74.5, -72.8, 170)))
pickup_latitude = tf.feature_column.bucketized_column(
    source_column = feats[1],
    boundaries = list(np.linspace(40.5, 41.8, 130)))
```



Lastly, model.py attempts to pop the `key` though to prediction time then specifies the estimator as a `DNNLinearCombinedRegressor` and all of its necessary specs. The point of popping the `key` through the algorithm is so that when running a batch prediction, the key stays linked to the prediction and one can be certain that the predictions line up with the correct data. However, this attempt to keep the key associated with each prediction didn't work so well, as the TF model would spit out an error if fed data with a `key`, and of course, data fed without a `key` was returned without any `key`s. This isn't a very well documented issue, with a few workarounds but it was unclear how to implement them (the `key_model_fn_gen` was the first workaround I tried) and no way to know if the workaround worked until after training finished (requiring retraining, since the TF graph would have been changed) and a prediction job was submitted.

Unfortunately, this wasn't the last of the issues that I faced, but they all proved to be learning experiences. I'd started with a toy model trained and deployed on a small portion of the dataset, just to experiment with TF and the cloud, but ran into the `key` issue described above; trying all the workarounds I could find, nothing seemed to work so I moved on and hoped it'd magically work in my final model. Then trying to load the whole 5.5GB CSV into Pandas took a lot of time before I discovered the `chunksize` option. Perhaps the most frustrating was submitting a training job, waiting 10 minutes for a VM to spin up, just for a cryptic error message to be returned. Many of the errors were related to dtypes and feature columns, but worst of all at least a few were due to my own typos.

Once, I'd squashed all those bugs and had it training successfully, I let it train for about 12 hours overnight, just to discover that at that rate it'd take about 135 days to train, since I first tried using a "basic" tier ML Engine job. Next, I tried the "TPU" (Tensor Processing Unit) tier, Google's new purpose built Machine Learning hardware that's supposed to be faster than anything out there, but after 8 hours of that, I realized that it'd still take another 22 days to finish training. That wouldn't work, so I then tried the "standard" tier, hoping not to have to pay the big bucks for the "premium" tier; then I realized the beauty of TensorFlow as it did 8 hours worth of (single-machine) TPU work within about 20 minutes due to the several worker machines that all work in parallel on the "standard" tier.

About 24 hours later, the training was complete, but oddly enough ML Engine kept running after it should have wrapped up the job. Unfortunately, the VM kept running and charging my account, but I didn't notice until about 48 hours after it should have finished. I contacted support, but they weren't going to be able to provide specific help for a few business days. Not wanting to jeopardize my progress, I let it keep running and charging me money, but I successfully attempted to deploy the model and was able to complete a batch prediction job, unfortunately still without a way to preserve

the `key` through to the predictions. Fortunately, I was able to save the basics needed for this project, but I got locked out before I could save the log files. Currently, I'm still awaiting a resolution from support and cannot access my model.

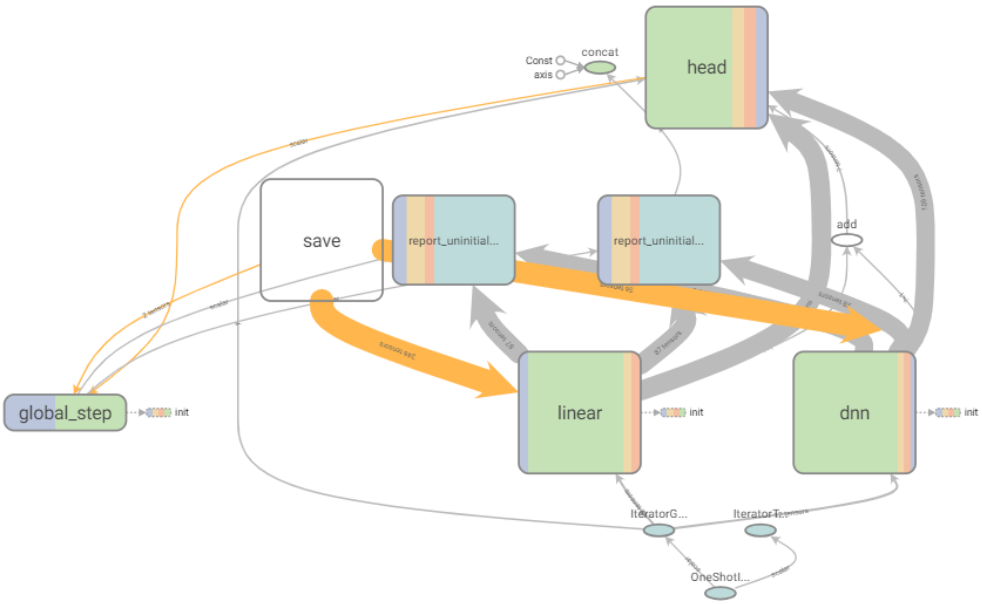
Refinement

Originally the plan was to start with a more basic model to use as a jumping-off point, then to perform feature selection, run better data preprocessing, then train a more optimized final model, which is why this model is referred to as v2. However, with the time spent trying to debug the `key` issue and the deadline fast approaching, I ended up skipping the intermediate step of training the basic model (v1) and just applied a deeper, smarter data cleaning/preprocessing to train an optimized model as my first and final model. This current model exceeds my original expectations, with an RMSE of 3.26726 and has been preemptively improved by anticipating factors like feature crosses and carefully combing through the data to remove misleading entries while preserving any that seem to provide value.

Before beginning this project in earnest, I created a prototype linear regressor model with a small sample of the data, which didn't have any data preprocessing and all the feature columns remained `numeric_column`s. The validation RMSE was somewhere around 12; the goal was to experiment with TensorFlow and see if this project was something I'd like to work on, so none of the code was saved, nor were any screenshots or logs. Though, if you compare that model to this model, you'll notice huge improvements due to the refinements made all along the way as described above.

IV. Results

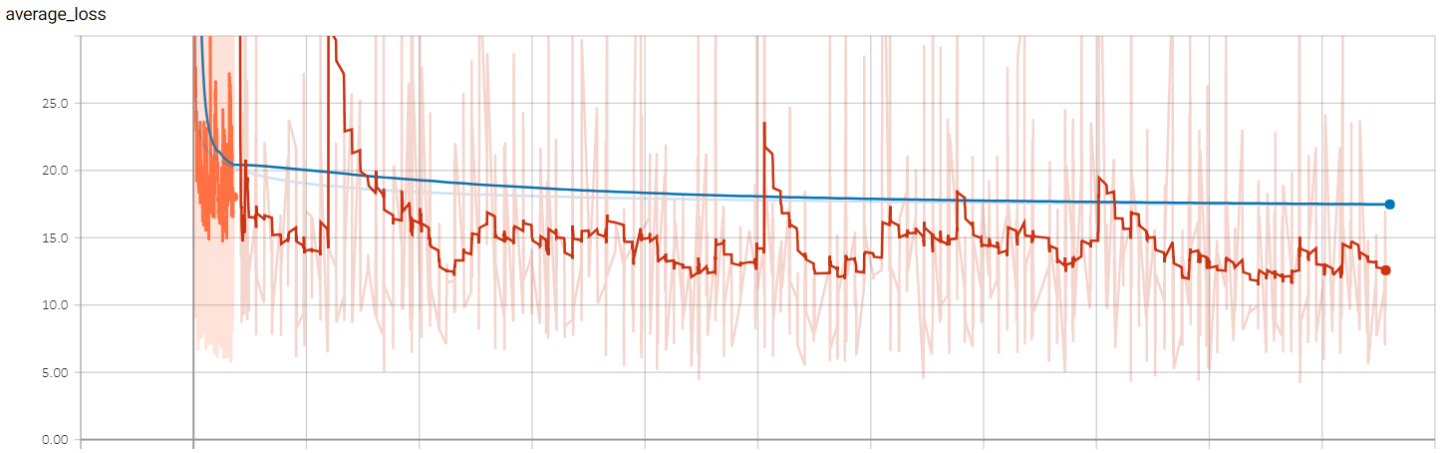
Model Evaluation and Validation



The wide and deep TensorFlow model was chosen based on research into various regression algorithms and seemed to be the most cutting-edge that was also aimed at being ready to deploy. While other state-of-the-art algorithms, such as XGBoost[10], might have an edge on sheer accuracy, the core `tf.estimator.DNNLinearCombinedRegressor` has been optimized for deployability and other real-world considerations (though that's not to say it's plug & play).

[10] https://xgboost.readthedocs.io/en/latest/

During training and validation, the model reached near peak performance pretty quickly, but didn't deviate away despite clear signs of a lot of noise still present in the dataset (note: this graph is average loss, so take the square root of this to find the RMSE). This goes to show how robust this



model is, as it was able to generalize pretty quickly then adapt to many changes in the training data. With whole a training dataset of around 55M rows (validation being an even sampling of about 20% of that) of real world data, this is almost guaranteed to run the gamut from minor perturbations to outright outliers. In fact, there still were outliers with fares as high as $500 or numbers of passengers as high as 9, so we know this dataset doesn't need us to make any manipulations to it for sensitivity analysis, as it's practically built right into the data. The final (and default) parameters of the estimator proved sufficient as the model performed well not just on the huge validation data, but also on the completely held out test data.

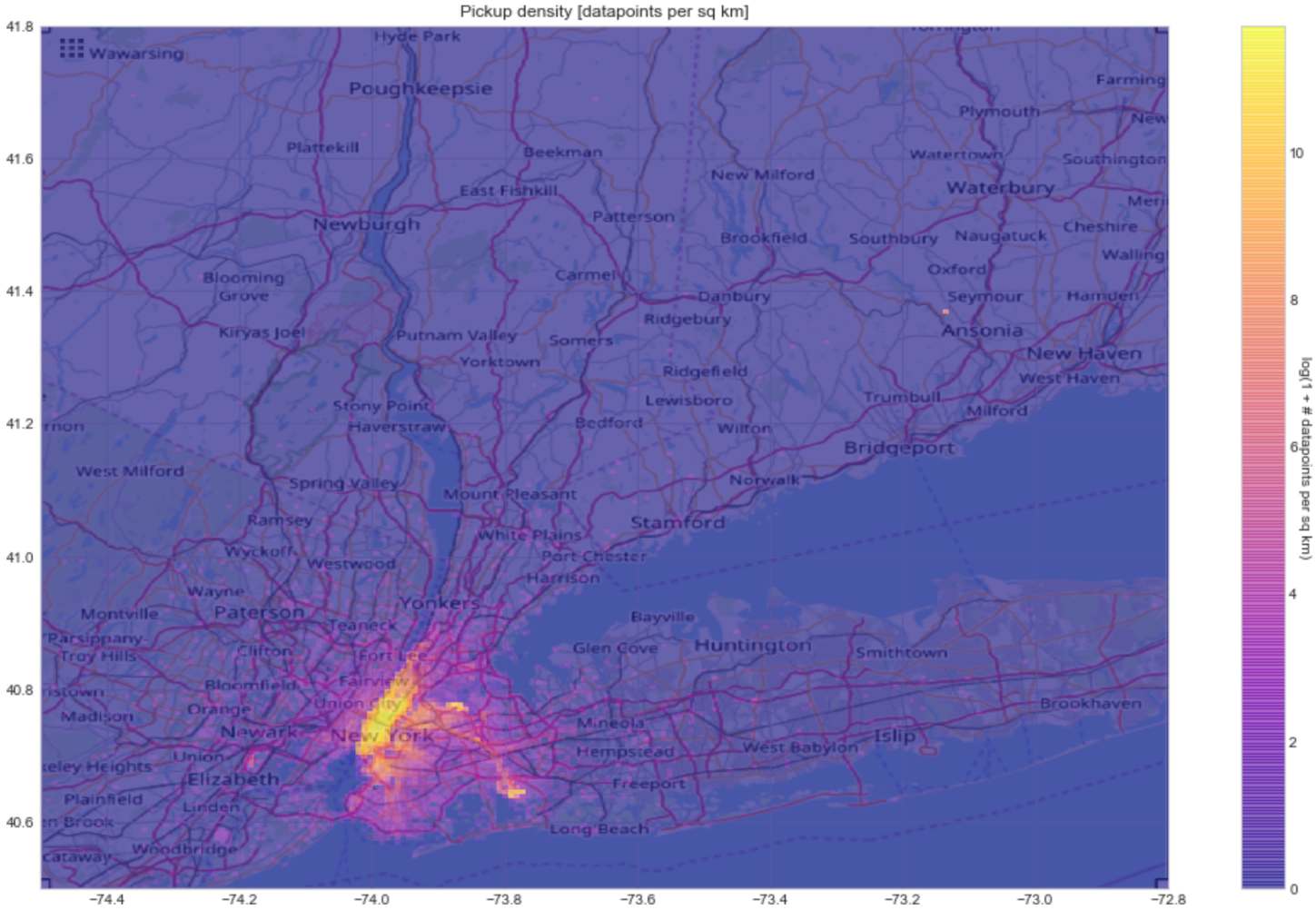
Justification

The Wide and Deep Algorithm, as predicted, performed well on this dataset; with an RMSE of 3.26726, it surpassed the goal of less than half the baseline score. This final model did bear a high cost (not just figuratively, but also in actual cloud fees), but that was to be expected as the task in this scenario was to create a sort of Minimum Viable Product (MVP) fare estimator that achieves a score at least twice as good as the simple math baseline, in order to gain approval to continue developing and improving this model along the pathway towards a feature/product release. This exercise has shown that there is definitely potential in this approach, and since there are still plenty of tricks up my sleeve (such as feature selection or parameter tuning), there's definitely room to see big improvements in this model.

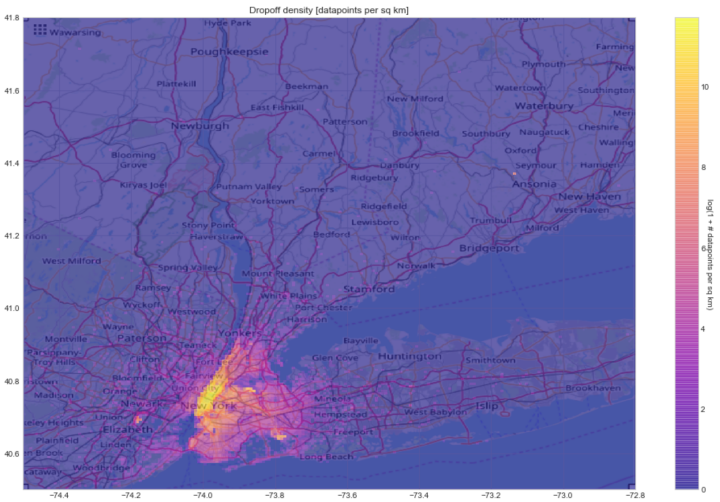
While this specific problem has been clearly solved, this final solution is perhaps even significant enough that it could be useful for real world users. Certainly, some people would certainly be annoyed if their ride was estimated at $42 but it ended up costing $45, though many probably wouldn't be too bothered, especially if the UX made it clear that they were getting a prediction and that real world conditions would affect the final price.

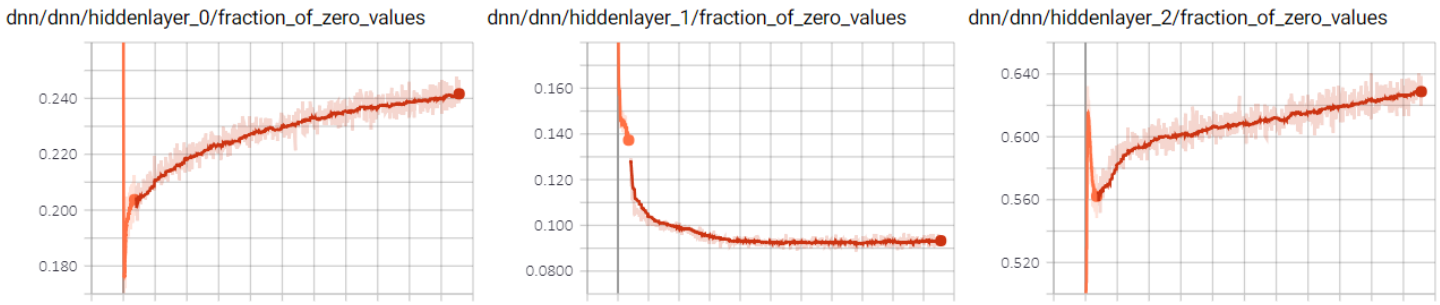
V. Conclusion

Free-Form Visualization

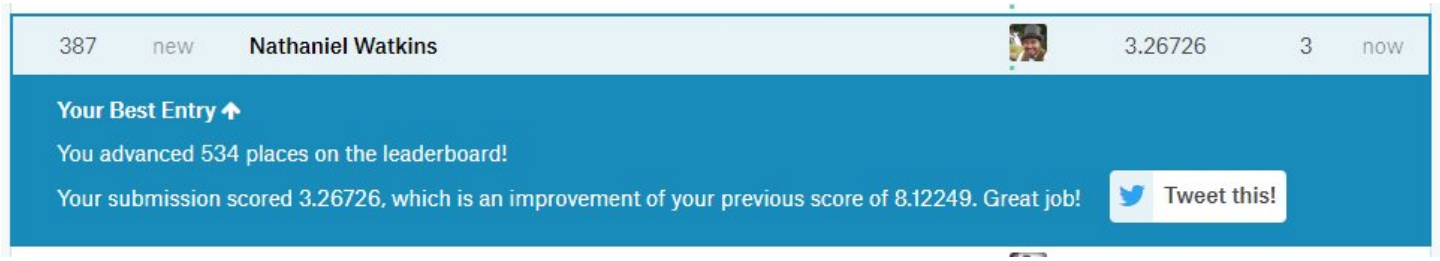


This hotspot map displays just how concentrated most of the data is, which makes sense since the hot areas pretty closely match up with NYC city limits, but keep in mind that the TLC serves a much wider region, just with much less demand in the other parts of the metro area. Also, notice the scale on the side, which is log(1 + # of datapoints), meaning that it takes around 100-1000 rides within a given square KM to start making any kind of visible impact on this map, with the noticeable areas being on the order of 10,000-1M rides per KM^2. Furthermore, note that this was pulled from just the first 2 million records in the train data, and that these bounds were chosen due to the test dataset containing datapoints as far reaching as the borders of this map. In reality, the entire 55M train dataset contained at least a few datapoints from just about all valid areas of this map.





The fractions of zero values in the hidden layers seem to indicate that the model could definitely use some feature selection, if not just for better predictions, but also to reduce the dimensionality of the input space and lower the computational cost.



My final model's predictions yielded a nice 3.26726 RMSE, which advanced my position up the Kaggle leaderboard significantly and places me around the top third of all 1079 competitors! Of course, I'd like to be among the very best, but all things considered, I'd say this is a decent entry for my very first Kaggle competition.

Reflection

Going into this project, I had a few goals for problems I'd like to try to solve, all centered around the concept of getting comfortable with creating a production ready model:

- ✓ Work directly with TensorFlow, getting proficient in how to build/modify a graph
- ✓ Explore a very large dataset (large enough that it wouldn't all fit into memory on a reasonably powerful local machine) utilizing and training with the whole thing
- ✓ Train and deploy the model in the cloud, allowing for online and batch predictions
- ✓ Enter my first Kaggle Competition, earning a respectable position on the leaderboard
- ✓ Surpass the baseline performance by at least a factor of 2

By these metrics, my project was a smashing success and I learned a lot along the way, despite the complications and roadblocks.

The problem was primarily solved through deep data exploration, cleaning out obviously erroneous entries, and careful crafting of the model's inputs to frame them in a structure that conveys the meaning to the estimator, but the brute force of a pack of computers mulling over a massive amount of data was definitely helpful. Interestingly, I found that the large scale of the data could only go so far in getting to a good solution, as you can see from the loss plateauing around 20% of the way

through training, indicating that similar results might have been achievable with a fifth of the data. And the initial model that I trained didn't seem to be able to make heads or tails of the data when it was provided it in an unconditioned manner, without optimizing the feature columns. As they say: "garbage in, garbage out." Most of the difficulties revolved around some surprising bugs and generally the headaches that come with learning a new platform. The final result does exceed my expectations, and works within the made up scenario as a fantastic proof-of-concept to justify further investment in refining and developing the model for a production release.

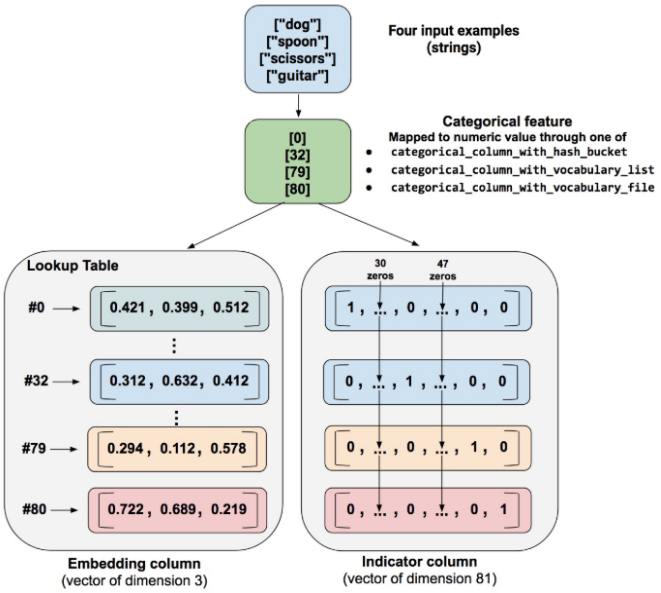
Improvement

The next steps for this project on its path towards final deployment, would entail cleaning up the data a bit further, optimizing the features, tuning the parameters with a grid search, reducing the model complexity, and maybe even trying a different model.

Firstly, I noticed that I had made a glaring oversight regarding `passenger_count`. While I caught the seemingly odd 0 passenger rides, I didn't notice that there was at least one entry with over 6 passengers within the first 2 million records. The test set only has a max of 6 passengers. While 1-2 rows shouldn't have any effect, if there are more in the rest of the dataset, they definitely could skew the training as it relates to the 6+ passenger bucket; notice how the distance and fare numbers are much higher than average. This makes sense as 6 passengers would be the max capacity for a normal taxi van, so any larger groups would need an extra large van or bigger type of vehicle. These probably aren't common, so they'd be more costly to operate. While I don't think the right approach would be to delete these records, it'd be good to structure the feature column with an extra bucket for 7+ passengers to catch these edge cases, and even be prepared to predict for them if/when they came up in real-world usage.

passenger_count	distance_km	fare_amount
1	3.283213	11.203874
2	3.507306	11.800907
3	3.380833	11.520043
4	3.430901	11.721943
5	3.335211	11.218318
6	3.415208	12.168925
9	13.046386	104.000000

Looking at the graphs of the weights, it's abundantly clear that we have too many vectors and many useless features. The next step needs to include feature selection to purge unhelpful columns for a more focused model that runs faster. Similarly, this graph needs to be optimized, as the number of vectors is in the thousands. This is due to the many bucketized columns, so after the number of columns is reduced, the remaining `bucketized_column`s should all be converted to `embedding_column`s[11]. Furthermore, maybe even smaller vector dimension values should be chosen; hash collisions would be

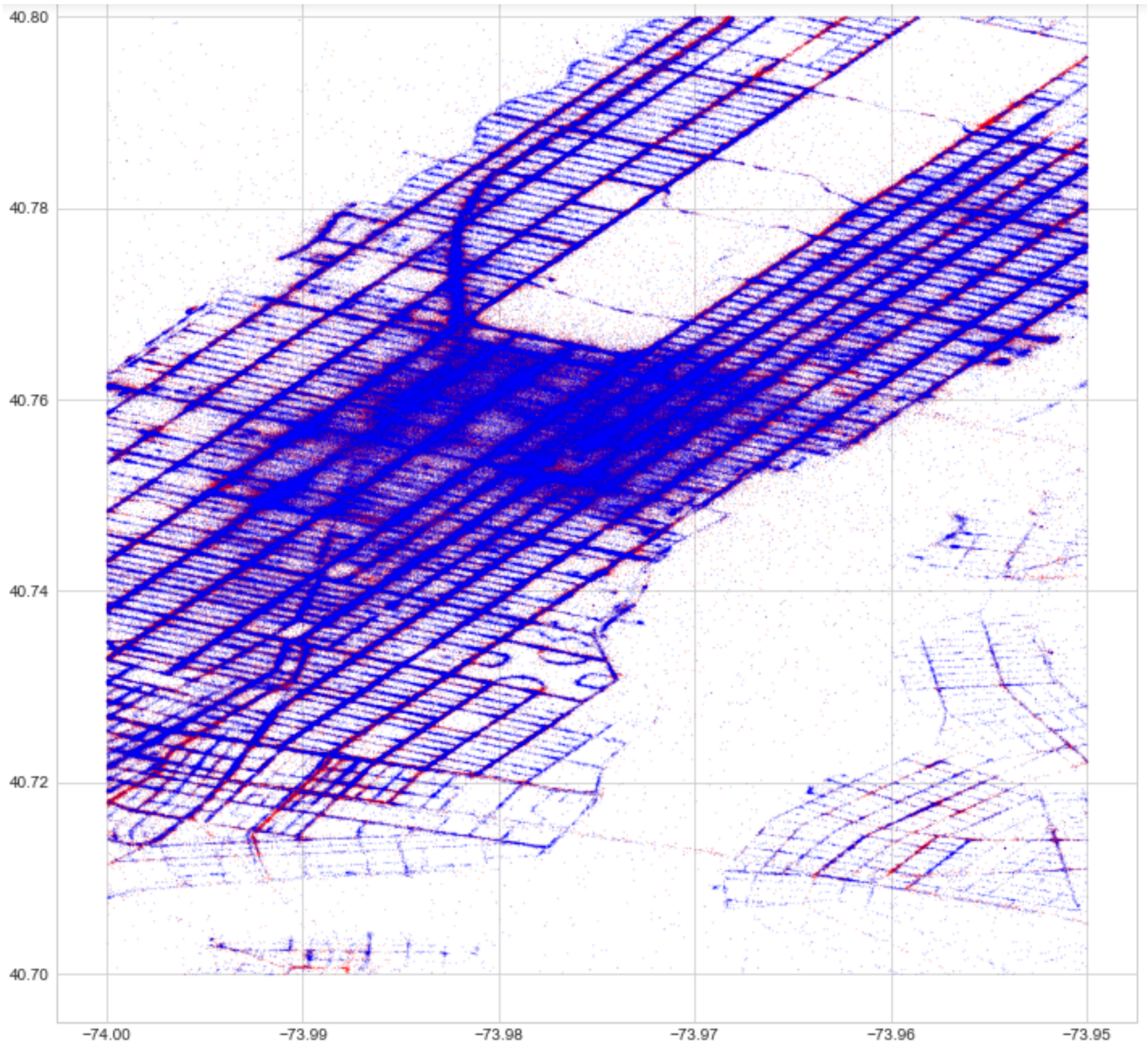


[11] https://www.tensorflow.org/versions/r1.9/guide/feature_columns#indicator_and_embedding_columns

more likely, but that turns out not to influence the results as poorly as one might suspect.

Both of those last two optimizations should greatly improve computational costs by reducing model complexity, but perhaps we can achieve much better results/cost by using a different algorithm, such as the `tf.estimator.BoostedTreesRegressor`. A training/validation/test run with that estimator should be compared to see if it runs significantly faster and/or more accurately. Then the superior algorithm should be used with a grid search to optimize the parameters from the defaults in search of the best performance possible.

I'll leave you with this map of Manhattan drawn entirely with datapoints from just the first 2M rows. Already, that's enough to draw a map from the points alone due to the sheer number of datapoints and diversity of locations!



Note, to be clear this is just a chart of longitude and latitude points on a plain white grid.