Bryan Gnipp

September 2016

CAPSTONE PROJECT

Machine Learning Engineer Nanodegree

Kaggle TalkingData User Demographic Prediction Competition

I. Project Overview

For my final project, I decided to participate in a machine learning competition hosted by Kaggle.

- Link to my Kaggle profile: https://www.kaggle.com/bgnipp.
- Link to the contest home page: https://www.kaggle.com/c/talkingdata-mobile-user-demographics.
- Description of the contest (taken from the link above):

  Nothing is more comforting than being greeted by your favorite drink just as you walk through the door of the corner café. While a thoughtful barista knows you take a macchiato every Wednesday morning at 8:15, it's much more difficult in a digital space for your preferred brands to personalize your experience.

  TalkingData, China's largest third-party mobile data platform, understands that everyday choices and behaviors paint a picture of who we are and what we value. Currently, TalkingData is seeking to leverage behavioral data from more than 70% of the 500 million mobile devices active daily in China to help its clients better understand and interact with their audiences.

  In this competition, Kagglers are challenged to build a model predicting users' demographic characteristics based on their app usage, geolocation, and mobile device properties. Doing so will help millions of developers and brand advertisers around the world pursue data-driven marketing efforts which are relevant to their users and catered to their preferences.

Competition participants are provided with mobile-use data for tens of thousands of TalkingData customers. For most customers, the type of device they are using is provided. For some (about 1/3) of the customers in the data sets, information on app 'events' are provided, which contains the name of the associated app, the category(s) of the associated app, the timestamp of the event, and the latitude/longitude at which the event occurred. As in all Kaggle competitions, participants train machine learning models with the provided labeled training data set, in order to generate predictions on the "test" data set (for which target labels are hidden until the competition is closed and scored).

Project Directory

1. Notebook1.ipynb contains the bulk of my work, it includes:
   a. Data exploration and preprocessing
   b. First level model training (xgboost, neural nets, and logistic regression)

    c. Time/location feature engineering
    d. Second level (stacked) model training (incorporating output from first models, and engineered features)
    e. Feature blending (arithmetic, geometric, and harmonic means)
2. Notebook2.ipynb contains the code supporting my highest scoring submissions. I found that my first model(s) (from Notebook1) was overly complex, Notebook2 simplifies applying lessons learned.
3. 'Leak_exploit_incomplete.py' provides incomplete code I wrote in an attempt to take advantage of a data leak that was disclosed in the final days of the competition. I did not have enough time to complete my implementation, unfortunately.
4. Readme.md provides info on the utilized packages
5. The 'input' directory contains all of the files provided to competition participants, unaltered
6. The 'output' directory (left empty to save space in my submission) is where my prediction/submissions are written
7. The 'preprocessed' directory contains CSVs of features I engineered. They took a long time to process, so the notebooks I submitted load the features from CSVs instead of processing them fresh (the feature processing code is provided, but is commented-out)

Problem Statement (and Metrics)

Target labels consist of different demographic groups (e.g. Females aged 24 - 26, Males aged 24 - 26, Females aged 27 - 28, etc.). For each element in the test set, a device_id is provided (which can in most cases be related to a device_id, and in some cases can be related to app events). A model must be built which receives this data as an input, and outputs a floating point number for each of the demographic groups which indicates the probability of membership. Log loss is the scoring metric for the competition (the winner of the competition is the participant who achieves the lowest log loss on the training set).

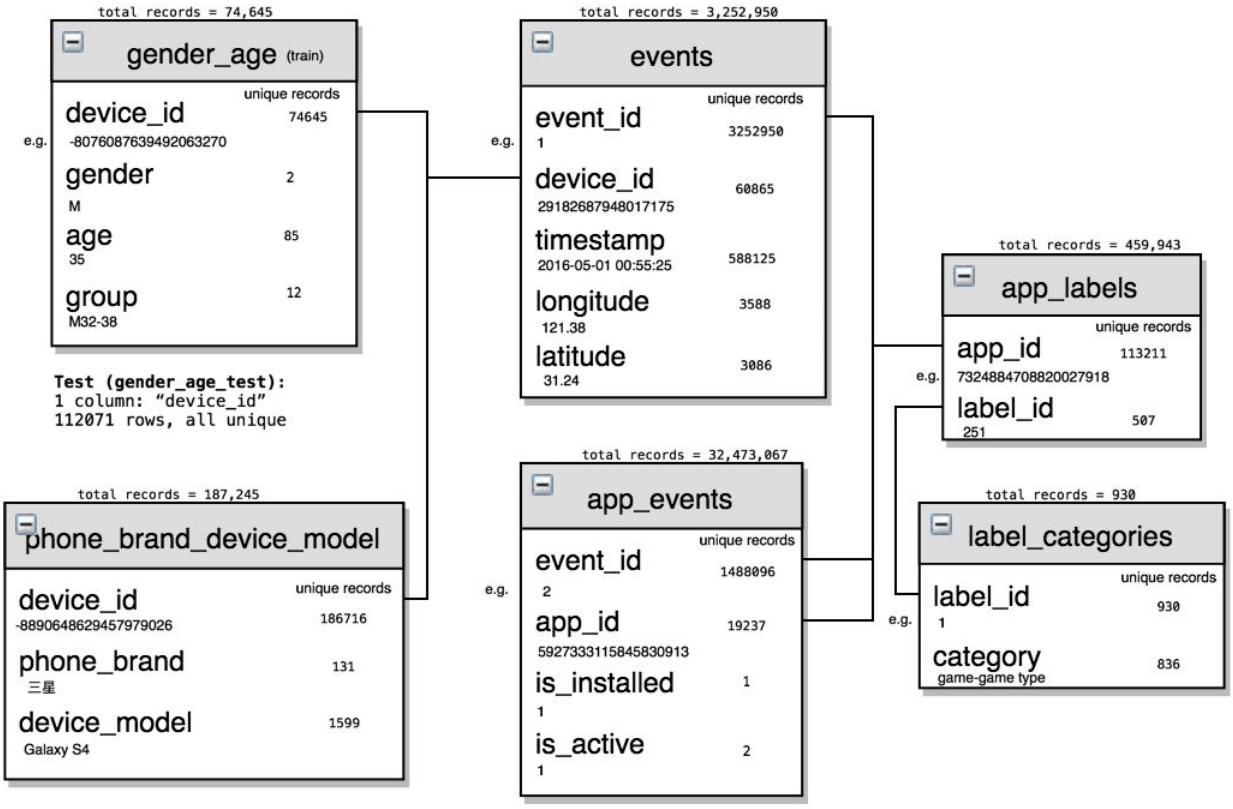
My high level approach to solving the problem is what most would expect in a Kaggle competition, it includes: 1) gaining an understanding of the data, 2) exploring the data to better understand the underlying signal, 3) build a predictive ML models, iterating based on results achieved (via cross validation and the scoring function provided by the Kaggle public leaderboard).

At the outset, I expected that the optimal solution involves: 1) engineering useful features, 2) determining which ML model(s) are best suited to the data, 3) optimizing model parameters, 4) bagging results from various models. Recently, the XGBoost (extreme gradient boosting) package has been dominant in Kaggle competitions involving columnar data. I expected that this model would likely be viable. Also, I expected that bagging would be useful, as it has been used to win many similar Kaggle competitions.

II. Analysis

Data Exploration

The first step was to gain an understanding of the data schema. Data was not provided in consolidated train/test sets, rather the data included what appear to be dumps of several relevant database tables. Competitors had to join the data as a pre-processing step. A high-level schema diagram was provided by the competition sponsor. I made several annotations to this diagram (I added examples of each field, unique record counts, and total record counts):



Feature definition and discussion:

- **Gender_age (training) table**
  - *Device_id:* This is a unique identifier for a mobile device (e.g. iphone, samsung phone, etc). The training data set includes device_ids and target labels. This must be joined with other tables to aggregate features. The test set consists solely of a vector of device_ids.
  - *Gender:* The gender of the device user. I did not use this feature (like most other Kagglers) as it is included in the 'group' feature (which is the target label we are trying to predict).
  - *Age:* The age of the device user. I did not use this feature (like most other Kagglers) as it is included in the 'group' feature (which is the target label we are trying to predict).
  - *Group:* The target label we are trying to predict. Consists of twelve age/gender demographic classes (see the graph in the *exploratory visualization* below for details).
- **Phone_brand_device_model table**
  - *Device_id:* Already defined, see above

    - *Phone_brand:* The brand of the phone (e.g. Apple, Samsung). This was a useful feature (brand correlates with demographic group).
    - *Device_model:* The device model of the phone (e.g. iPhone 6, Galaxy 6). This was a useful feature (device model correlates with demographic group).
- **Events table**
    - *Event_id:* Unique identifier for an event (an instance of a user performing a significant action within an app)
    - *Device_id:* Already defined above
    - *Timestamp:* The date/time the event occurred. I found that this feature was not useful in my models.
    - *Longitude:* Location the event occurred. Some events don't have location data. I found that this feature was not useful in my models.
    - *Latitude:* See above
- **App_events table**
    - *Event id:* Already defined above
    - *App_id:* Unique identifier for a mobile app. I found this to be the most useful feature (this indicates different apps are used more by different demographic groups).
    - *Is_installed:* All values are identical (true), so this was discarded
    - *Is_active:* Binary value indicating whether the user was actively using the app at the time of the event. I did not use this feature (like most other competition participants).
- **App_labels table**
    - *App_id:* Already defined above
    - *Label_id:* Provides a unique numeric identifier for the 'app_label,' defined below
- **Label_categories**
    - *Label_id:* Already defined above
    - *Category:* The category of application (e.g. game-motorcycle, chinese comic, online malls, etc.). One application may map to many categories. I trained models with app_ids (categories excluded), and also trained models with both app_ids and categories. The latter was more effective (signal is present in this feature that is not already provided by app_id).

The training and test sets are keyed on 'device_id' (corresponds to an individual phone).

- Most of the devices (in both training and test sets) have no events, so we must rely on phone brand/model for these
- All is_active values in the app_events data/table are the same, so we can disregard it
- The timestamps span a period of about a week. Times are in UTC.
- Many latitude/longitude values are 0.0 (missing)
- The label_categories data/table simply provides a string identifier for each (numerically identified) label_id, so this is not useful for processing
- Refer to the feature engineering section below for further exploration/analysis

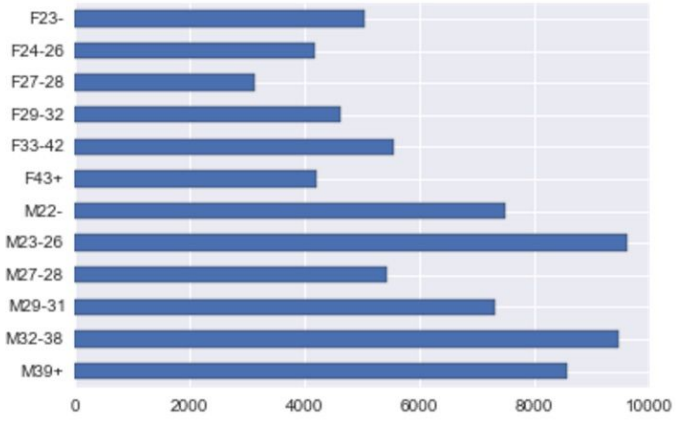
Refer to the provided .ipynb files for additional detail.

Exploratory Visualization

I plotted the target label distribution (demographic groups) with help from some code from a fellow competition participant:

Age/gender distribution (training set)

```
In [11]: gatrain.group.value_counts().sort_index(ascending=False).plot(kind='barh')
Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x10618c350>
```



The labels are somewhat imbalanced. However, as this imbalance was not extreme, I did not feel the need to take special steps to compensate for it. A statistically significant representation of all classes is present in the data. Additionally, my primary ML models (xgboost/gradient boosting and neural nets) were applied in a way so as to mitigate the impact of imbalanced classes.

Refer to the provided .ipynb files for additional details.

Algorithms

Logistic Regression

This algorithm trains quickly and can be well suited for data with less complex learning tasks. It is often a good idea to apply it before applying other more complex algorithms, to serve as a benchmark. It essentially 'draws a line' (or a plane/hyperplane in higher dimensions) through the training data to identify a pattern in the data, which can be exploited to make predictions on novel data.

Significant parameters:

- Regularization constant: Essentially, this "smooths out the curves" log the line/plane drawn. This can be used to mitigate overfitting (smoother curves generalize better as the are not fit to 'noise'). My code performs several iterations of LR model fitting, iterating over different regularization constant values, and plots the results. I used this to select the optimal value.

XGBoost (gradient boosted trees and LR)

XGboost is an implementation of the 'gradient boosting' technique. Essentially, gradient boosting techniques train many separate sub-models. After each training iteration, the model is cross validated; weighting is then added to observations (within the training set) which were predicted incorrectly. Subsequent models are trained which perform better on observations which were previously misclassified. This process continues until ensemble of models is arrived at which minimizes the loss function.

I employed both linear and tree based gradient boosting via the xgboost library. I found that linear boosting worked better than tree boosting (which is usually not the case for Kaggle problems). I used parameters posted by others on the Kaggle forums as a starting point. I iterated on a few of the parameters but found they didn't make too much of a difference.

Significant parameters:

- Objective: This specifies the loss function (which the algorithm minimizes). Here we specify multi:softprob, because we are predicting probability across multiple classes.
- Booster: There are two options here, 'tree' (uses simple decision trees as base models) and 'linear' (uses simple LR). Tree usually performs better on tabular data, and is used for more frequently than linear. Interestingly, linear performed better on my primary (non blended) models, and I'm not totally sure why this was the case.
- Max_Depth: Specifies the maximum depth for the decision tree. Trees with lower depth are less prone to overfitting. This important isn't as important with gradient boosted trees (compared to a single non-gradient boosted tree), because the iterative boosting process compensates for over/underfitting of individual trees. I tried running a few different values, and it did not make a significant difference.
- Eval_metric: Specifies the metric to be used to evaluate models. I selected logloss, as this is the scoring metric for the competition.
- ETA: Specifies the learning rate, i.e. how 'large of a step away' each model should take to correct errors present in prior models. Lower values result in higher training times, higher values pose risk of increased variance. After some experimentation, I settled on a default value.

Neural Nets (Keras/Theano implementation)

The neural nets I trained performed better than the logistic regression and gradient boosted models. This is unusual for tabular data. In the final days of the competition, a data leak was disclosed; there is some signal

in the ordering of the test and train sets (due to poor shuffling performed by the competition organizers). It is possible that the neural nets are picking up on this, and this explains why they are outperforming gradient boosting (as xgboost usually outperforms neural nets for tabular data).

My neural net architecture is quite simple. As the data is strangely shaped, tabular data (neural nets are not usually used on tabular data), I could not identify any useful/relevant precedents. I had to use experimentation and communication with the Kaggle community to determine what architecture to use. I settled on two architectures (refer to Notebook2.ipynb for the code). The performance of both models was similar; my idea was to blend the output to try to increase performance (as increased diversity of models in ensemble approaches is often beneficial, especially when applying averaging techniques such as taking the harmonic mean).

My first neural net uses is a simple Keras sequential model, with a 50 node dense layer, which is fed into a 60% dropout layer, which is then fed into a 12 layer dense layer (as there are twelve target classes).

My second neural net is also a Keras sequential model, with a 150 node dense layer, fed into a 40% dropout layer, fed into a 50 node dense layer, fed into a 20% dropout layer, fed into a final 12 node dense layer.

For each model, I trained for 15 epochs.

As stated above, my process for arriving at these architectures wasn't terribly robust. Models take a long time to train, which makes rapid iteration problematic. I used models shared by fellow Kagglers a starting point, and performed some experimentation, none of which improved performance. I ended up settling on model architectures that were very similar to what other Kagglers were using in their kernels.

Explanation/definition of parameters

- Sequential model: layers feed to one another in serial
- Dropout: Randomly drops the specified percentage of the training data at each iteration, helps make the model more robust and less prone to overfitting

Stacking Techniques

Output chaining

In Notebook1.ipynb, I trained six models:

1. Logistic regression, with binary (one-hot) data for apps (i.e. for a given device id, an app_id related event appears at least once, that app id is marked with a '1'. If it does not, it is marked with a 0)
2. Xgboost (linear) with the same type of data as #1 above
3. Neural net, with the same type of data as #1 above
4. Logistic Regression, with integer data for apps (i.e. for a given device id, if an app_id related event appears, it is marked with the number of appearances)
5. Xgboost (linear) with the same type of data as #4 above
6. Neural net with the same type of data as #4 above

I then fed the output of these values (the predicted probabilities for each of the twelve classes) to an xgboost model and a neural net (in parallel). Submissions (to the Kaggle leaderboard) were made for the output of each model, as well as a blend of the output from each model (see below for details on this), but the results were lackluster.

Output blending

By blending, I mean combining the output (predicted probabilities for target classes) from many models into a single submission by taking the mean. I implemented three types of means for blending:

1. *Arithmetic mean:* This is the most common type of mean (sum of all inputs divided by the number of all inputs).
2. *Harmonic Mean:* Definition available here: https://en.wikipedia.org/wiki/Harmonic_mean. My best submission was a harmonic mean of the output of six neural nets.
3. *Geometric Mean:* Definition available here: https://en.wikipedia.org/wiki/Geometric_mean. Unfortunately, I implemented geometric mean blending late in the competition and was not able to make a Kaggle leaderboard submission with it (because Kaggle limits total submission to five per day).

Refer to the provided .ipynb files for additional details.

Benchmark

Kaggle competitions provide a leaderboard where submissions are scored and ranked against other competitors. Most competitors use this as their primary benchmark (cross validation scores are also considered). I used both of these, primarily relying on leaderboard scores as an indicator of the quality of my models. There is some risk of overfitting to the leaderboard (as submissions are scored against a different set of hidden data after the competition ends). I did my best to mitigate the risk of overfitting to the leaderboard by attempting to keep my models sufficiently simple/general so as to mitigate the risk of overfitting.

For the first few runs of my models contained in Notebook1.ipynb documented results in the google sheet tracker:
https://docs.google.com/spreadsheets/d/1nA9Tmq4fGEN0941bnhjD_FHIily4ndXO89RrzGoBvLU/edit#gid=843687245.

There are some gaps in this tracker, because:

- I abandoned some models due to poor performance in early runs
- I was constrained by Kaggle's limit of five leaderboard submissions per day

The Udacity capstone project guidelines require the specification of a benchmark which makes failure a possibility. I essentially used my logistic regression models as my benchmark in this sense. These models were very simple, however they performed significantly better than random guessing (as indicated by cross validation and the kaggle leaderboard). Very little effort was required to construct the model. Logistic regression usually performs worse on complex/irregular data such as the data provided for this competition. I found this to be the case; my subsequent models performed significantly better than my logistic regression models. My best model (a harmonic mean bag of NNs) scored 2.24764 on the Kaggle leaderboard, vs. my LR benchmark's leaderboard score of 2.34543 (lower is better).

III. Methodology

Data Preprocessing

A significant amount of preprocessing was necessary.

1. Data was provided in several different files (resembling relation database table dumps), so several joins were required.
2. Due to the large number of phone brands (131), device models (1599), app_id's (19,237), and app label_ids (507), sparse matrix encoding was necessary to enable efficient memory use. A sparse matrix was created for training with the shape 74645, 21527. In what is referred to as the 'unsized' matrix in Notebook1.ipynb, the features listed above are one-hot encoded (either they are present for a given observation/device_id or they are not. In the 'sized' matrix, the number of occurrences is marked.
3. I computed several features based on event location and time (though I found that these did not improve the performance of my models and dropped them in Notebook2.ipynb). Refer to Notebook1.ipynb for the code used to compute these features.
    a. Location features
        i. Mean lat/long
        ii. Range of lat/long (max - min)
        iii. Std dev of lat/long
    b. Time features
        i. Distribution of events across days of the week
        ii. Distribution of events across hours of the day (split into 4 hour buckets)
        iii. Binary distribution of events: halves of the day and week
        iv. Day and hour medians, .25 quantile, and .75 quantile
        v. Standard deviation of days (measure of dispersion of events across days)
        vi. Standard deviation of hours (measure of dispersion of events across hours)

Refer to the provided .ipynb files for additional detail.

Complications that occurred during the coding process

1. My first model (which includes phone brands/devices, app names, and app labels), has about 20,000 columns/features (a unique column/feature for each device type, app, and app_label). Using regular Pandas dataFrames uses lots of memory and slows computation. I had to figure out how to encode the data into sparse matrices, and make these matrices work with the different models I used. Keras (neural net) and XGBoost both required different preparatory steps. I found some help in the Kaggle forums, but had to troubleshoot some issues myself (particularly for Keras).
2. Computing my time and location features was very expensive/time consuming. I could not afford to re-compute features each time my .ipynb files were run. I solved this by pre-computing the features, commenting out the feature-computation code in the .ipynb's, and loading the features from a csv file.
3. I found that my simpler baseline model performed better than my more complex models with added time/location features and app/label event "sizes" (number of occurrences). Under the assumption that there must be at least some signal in these additional features, I tried several implementing several different permutations of them with many different models and parameters. None of these approaches succeeded in improving my leaderboard score.

Refinement

My refinement process consisted of the following:

1. Run initial models (logistic regression and xgboost) with baseline parameters, on base features (device_model, device_brand, app_id, app_label). Use CV and leaderboard scores as a baseline.

2. Implement a more complex model, a Keras neural net. This achieved a significant boost in performance.
3. Engineer additional features based on event time and location. These did not improve the performance of single models.
4. Run output of base feature models into second-level model, adding the the time and location features (but only as input into the secondary model). This did not improve performance.
5. Iterate on parameters of single and aggregate models. Parameters and results tracked here: https://docs.google.com/spreadsheets/d/1nA9Tmq4fGEN0941bnhjD_FHIily4ndXO89RrzGoBvLU/edit#gid=843687245 . As documented in the tracker, this lead to some performance gains.
6. Make significant changes to approach (this is when Notebook1.ipynb was abandoned and Notebook2.ipynb was created). All models other than neural nets were abandoned, as neural nets were consistently achieving the best performance. The two best performing neural net architectures (keras parameters) were selected, and three iterations were run for each (with different seeds). This resulted in six models.
7. Continuing the refinement described above as #6, blend the output of the six models by taking either the arithmetic, harmonic, or geometric means. I found that the harmonic mean achieved the highest score. My best scoring submission was achieved by taking this approach with the harmonic mean.

Much of my methodology is discussed in the above sections, please refer to them for details on model selection, parameterization, data wrangling, feature engineering, and feature selection.

IV. Results

Model Evaluation, Validation, and Justification

The above documentation (culminating at refinement step seven) describes the process of arriving at the final model. This addresses how the model was evaluated, validated, and justified. Additional detail is available in the included ipython notebooks.

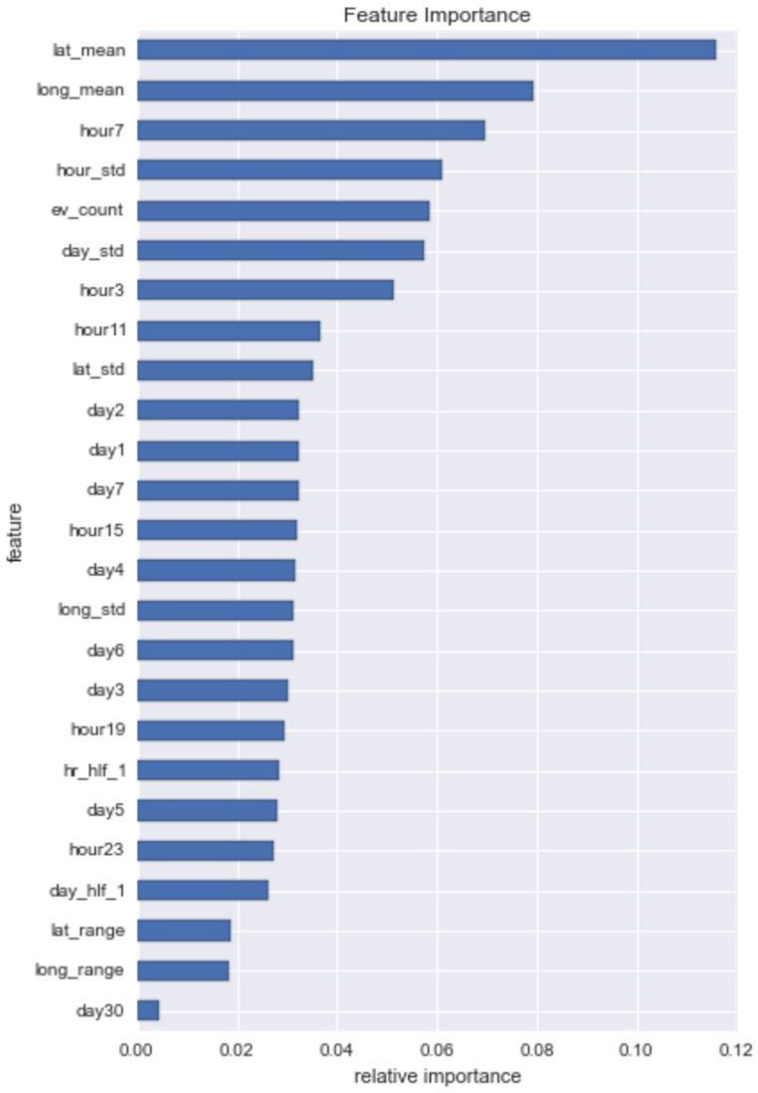
I finished in the top 23% (381st out of 1689). Refer to my Kaggle profile to validate: https://www.kaggle.com/bgnipp . My best score was 2.34543 (log loss).

I think that I did reasonably well, given that this was my first Kaggle competition. I certainly learned a lot!

V. Conclusion

Free-Form Visualization

After engineering features based on time and location, I trained and xgboost model to rank features in order of their utility in predicting target labels. This knowledge enables me to exclude features (from models that do not automatically ignore less relevant features) in order to mitigate overfitting. Plot of the results:



I

Refer to .ipynb files for the code used to generate this plot, and details on the features.

Reflection

I learned a ton from this project. **Some highlights**:

- My proficiency with Pandas increased significantly. This competition required lots of pre-processing, in large part due to the fragmented structure of the provided data as well as the extremely large number of natural features. Even in cases where I borrowed code from fellow Kagglers, I ran every line of code in an interactive ipython session in order to gain a thorough understanding. I also used to Pandas to engineer several features and to implement three blending methods.
- I did extensive reading on many xgboost and neural net (keras) parameters. I found the theory behind neural nets fascinating. I found that very simple neural net architectures seemed to be best suited to the data (given the relative simplicity of the data compared to tasks neural nets are more traditionally applied to such as computer vision and natural language processing). I look forward to exploring and applying more complex neural net architectures as part of future projects.
- In the final days of the competition, I learned a lot about data leakage (in the context of a machine learning competition) and how predictions can be made based on features engineered on the basis of row ordering. Though this technique does not address the underlying business problem, the techniques and theory are fascinating (and may be applicable to certain business problems where order is significant). Unfortunately, I was not able to complete an implementation of leak-based features due to the closing of the competition.

Some of the biggest challenges:

- Getting Keras to work with the data I was providing it. This was not very straight-forward! I may check out TensorFlow next (though I understand that it can be challenging as well).
- The competition seemed a bit unique in that most engineered features did not improve model accuracy (this was the experience of many on the Kaggle message board). Therefore, this wasn't the best exercise in feature engineering.
- The 'data-leak' (discussed a few times above) changed the nature of the competition in the final days. Though some interesting lessons were learned from this, there were many negative aspects as well (distancing from the business problem, as well as the fact that those who could not quickly rebuild their models in the final days fell in rank)

Discussion of end-to-end solution:

My approach consisted of exploring the data, reviewing other Kagglers' approach to the problem, performing extensive experimentation (trying different features and models, as discussed above), and finally arriving at a final solution as the competition ended (after working on the problem over the course of two months).

Improvement

As I write this, I'm eagerly waiting for Kaggle to post the code and write-ups for the winning models. I will learn a lot from these.

As discussed above, the best way to improve my performance in this competition would be to implement features to exploit the data leak disclosed in the final days of the competition. Many participants achieved huge leaps in leaderboard via this relatively low-effort method.

I surmise that there are second-level features that could be engineered on top of my primary features (device_model, device_brand, app_id, and app_labels). Some of the less relevant items could be possibly be dropped, and compound/grouped features could be derived. Theoretically, a neural net (and to a lesser extent, gradient boosting) could pick up this signal without the additional features, but I believe it's likely that the additional features could help.

It's also very likely that changes to my model parameters (particularly, neural net architecture i.e. layering, layer sizing, dropout, activation functions, etc.) could significantly improve performance. I will continue to read research on neural nets to better understand how to implement appropriate architectures. Additional iteration/experimentation will also be helpful. It's also likely that further learning and experimentation with bagging and blending techniques would lead to enhanced performance.