Capstone Project: TalkingData Mobile User Demographics

https://github.com/stephenlsiegel/kaggle-talkingdata

By Stephen Siegel

I. Definition

Project Overview

In this project I will be participating in the TalkingData Mobile User Demographics competition on Kaggle. The competition has ended, but submissions will continue to be scored.

TalkingData is a large Chinese mobile data platform[1]. They created a software development kit that mobile app developers can implement to record information about app usage. For this competition they have provided app download and usage data for thousands of mobile device users.

Problem Statement

The challenge of this competition is to develop a model to predict the gender and age range of mobile device users in a test set that has been provided by TalkingData. There are twelve discrete prediction categories:

F23-	M22-
F24-26	M23-26
F27-28	M27-28
F29-32	M29-31
F33-42	M32-38
F43+	M39+

where F/M represent the female and male genders and the numbers represent the age range. One thing to note is that the **age ranges are different** among the two genders. Instead of a binary prediction, we can allocate a probability to each outcome, e.g.

User ID	F 23-	F 24-26	F 27-28	F 29-32	F 33-42	F 43+	M 22-	M 23-26	M 27-28	M 29-31	M 32-38	M 39+
1	0.5	0.25	0.25	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0.5	0.35	0.15	0	0
3	0.833	0.833	0.833	0.833	0.833	0.833	0.833	0.833	0.833	0.833	0.833	0.833

This is a supervised learning problem. More specifically, it is a probabilistic classification problem, meaning that the model predicts a probability distribution over a set of discrete labels.

We could think about this problem in two ways:

(1) *standard multiclass classification problem*: we simply try to develop a model that outputs a probability distribution over the 12 classes listed above.
(2) *multi-label classification problem*: we further divide the problem into predicting two different labels, gender and age. This would make the problem "multi-label" in nature. It might be interesting to compare our prediction abilities on gender versus age to see if our model is much better at one or the other.

Metrics

The competition is evaluating submissions using **multi-class logarithmic loss**. In short, it's the average of the log of the predicted probability for the accurate group. The formula is:

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} log(p_{ij})$$

where (quoting from the competition site): "N is the number of devices in the test set, M is the number of class labels, log is the natural logarithm, y_{ij} is 1 if device *i* belongs to class *j* and 0 otherwise, and p_{ij} is the predicted probability that observation *i* belongs to class *j*."

If we dive into the formula further, we see that y_{ij} equals 0 when device *i* does not belong to class *j*. Thus nothing gets added to the log loss in these cases. When device *i* does belong to class *j*, we add $\log_e(p_{ij})$ to the log loss.

The natural log is a monotonically increasing function, so the value of $\log(p_{ij})$ will hit its min and max when p_{ij} equals *0* and *1* respectively. The natural log of *x* is only defined for *x > 0*, so for scoring purposes Kaggle approximates *0* with 10^{-15}. Likewise, Kaggle approximates *1* with $1-10^{-15}$.

In the poorest possible prediction, we give a probability of device *i* being in class *j* a *0*. This gets approximated to 10^{-15}. The $\log_e(10^{-15}) = -34.539$. The formula takes an average over all predictions and multiplies by *-1*, so **the worst possible prediction score is** *-(-34.539 * N) / N* = **34.539**.

In the best possible prediction, we give a probability of device *i* being in class *j* a *1*, which gets approximated to $1-10^{-15}$. The $\log_e(1-10^{-15})$ is so close to 0 that it's not worth typing out all the decimals. Therefore the **best possible prediction score is 0**.

There are two main conclusions from this metric: lower multi-class log loss scores are better, and the distribution of inaccurate predictions does not matter. For example, the following two predictions for the same user would have the same log loss score (assume the correct label is F29-32):

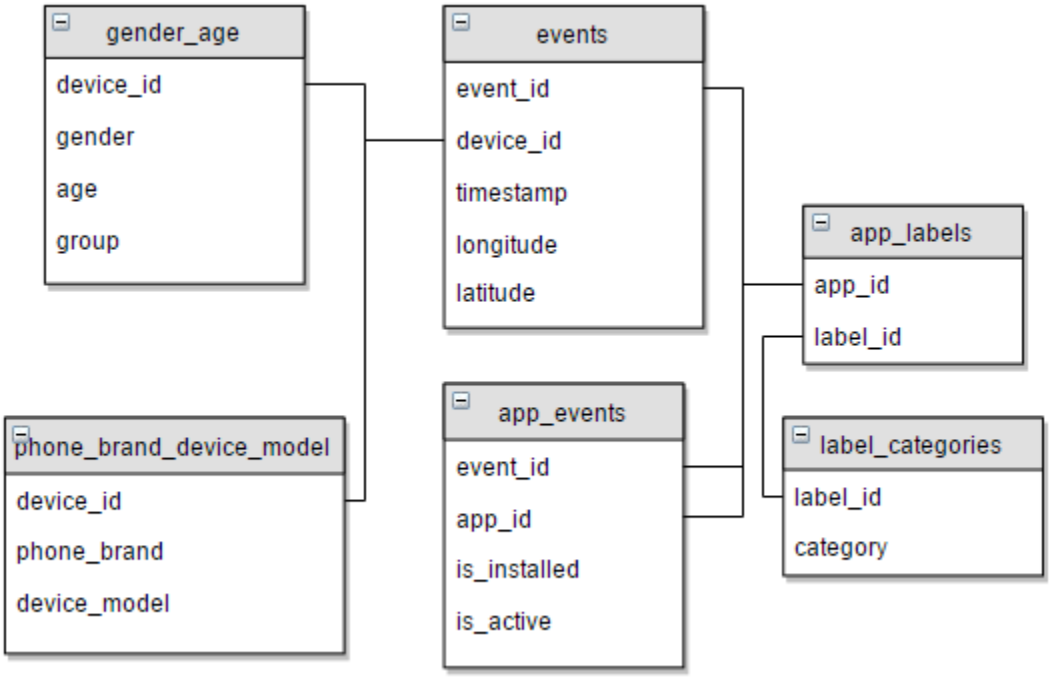
User ID	F 23-	F 24-26	F 27-28	F 29-32	F 33-42	F 43+	M 22-	M 23-26	M 27-28	M 29-31	M 32-38	M 39+

1	0	0	0.22	0.56	0.22	0	0	0	0	0	0	0
1	0.04	0.04	0.04	0.56	0.04	0.04	0.04	0.04	0.04	0.04	0.04	0.04

II. Analysis

Data Exploration

TalkingData has provided the following schema for the data:



- **gender_age** has been divided into a training (gender_age_train.csv) and testing (gender_age_test.csv) set. The training set is labeled with the gender, age, and group data. The testing set is not. The idea is to build a model on the training data and submit predictions on the testing data to Kaggle, which will score the predictions.
- **events** and **app_events** log when a user uses an app connected to the TalkingData SDK.
- **app_labels** and **label_categories** provide more information on the category of each app.
- **phone_brand_device_model** contains the phone brand and device model for each device ID in the gender_age table.

Initial Thoughts

From the schema, the data looks relatively clean and well-organized. The main thing that sticks out right now is we will need to do a significant amount of feature

engineering. Somehow we need to find a way to convert the massive log of events/app_events into features for each device.

Data Size

The **app_events** table is the largest in the data set at 989 MB. This was the only table that gave me a memory error when trying to load it into Python. To accommodate this, I was able to load the table and do some manipulations in R.

Table Information and Data Checks

Table Name	Columns	Rows
gender_age_train	4	74,645
gender_age_test	1	112,071
phone_brand_device_model	3	187,245
events	5	3,252,950
app_events	4	32,473,067
app_labels	2	459,943
label_categories	2	930

Gender Age Check

The first thing I checked is the **gender_age** tables. I confirmed the following:

- all device IDs in gender_age_train are unique and there are 74,645
- all device IDs in gender_age_test are unique and there are 112,071
- all device IDs across both sets are unique, and there are 186,716. This ties to 74,645 + 112,071.

In short, device_id is a primary key for the combined gender_age table.

Phone Brand Device Model Check

The next check is on **phone_brand_device_model**. At first glance, we see that it has 529 more rows than the combined gender_age table. The hope is that the device IDs will be unique and there will be 529 extras, but we must check for duplicates.

There are only 186,716 unique device IDs in **phone_brand_device_model**. This is reassuring because it hopefully means we have at least one record for every device in the **gender_age** table. However, it also means there are some duplicate device IDs, and we will have to determine how to pick one of the duplicates. It's not surprising that there are some duplicate values as these users likely changed phones during the data collection period.

There are 529 duplicate device IDs in **phone_brand_device_model**. Of these, 523 are "good duplicates", meaning that the entire row is duplicated so the phone_brand and device_model does not change. The remaining 6 duplicates do change, and there really

is no good reason to pick one over another. We don't know about the timing of the device change, so we have three options: keep the first, keep the last, or throw out these devices from the training set entirely. I've decided to keep the first.

Other Tables Checks

We expect event_id to be a primary key for the **events** table, and it is. All 3,252,950 event IDs in the table are unique.

App_events is a log of the apps that are installed and whether they are active during a particular event ID. It doesn't have a primary key, but the combination of event_id and app_id should be unique. If it is not unique, this would mean we have multiple records for the same app under the same event ID. I checked this in R (02-app-events-pk-check.R), and all 32,473,067 concatenations of event_id and app_id are unique.

The same should be true for **app_labels** over the app_id and label_id fields. An app_id can have multiple labels. There happen to be 491 duplicates, but the duplicates can be removed by only keeping one of them.

In the **label_categories** table, all 930 label IDs are unique. However, there are some duplicate categories. This is useful to remember as we'll want to encode on the category and not the label ID.

Distribution of Group in Labeled Data

Group	F 23-	F 24-26	F 27-28	F 29-32	F 33-42	F 43+	M 22-	M 23-26	M 27-28	M 29-31	M 32-38	M 39+
Dist	6.8%	5.6%	4.2%	6.2%	7.5%	5.6%	10.0%	12.9%	7.3%	7.8%	12.7%	11.5%

From the table above we see that men account for 64% of the devices while women account for 36%. This distribution across groups also acts as a great benchmark, a topic which I will expand on further in the benchmark discussion below.

Exploratory Visualization

In the visualization to the right I've mapped a sample of 1,300 events based on their longitude and latitude. We see clusters around the three cities in blue, which are (north to south) Beijing, Shanghai, and Hong Kong.

Algorithms and Techniques

The task of applying probabilities over all classes is called probabilistic classification[2]. There are multiple algorithms which can perform this as explained in detail in the sklearn documentation[3].

It is well established that logistic regression performs well on probabilistic classification problems, and it can be set to directly minimize log loss. Naïve Bayes, Random Forests, and Support Vector algorithms can also return probabilistic predictions, however all three usually require some calibration to perform well. I will be using a logistic regression algorithm.

Benchmarks

There are three main benchmarks I want to use: equal probabilities, distribution-weighted probabilities, and the top score on Kaggle.

Equal Probabilities Benchmark

In the equal probabilities benchmark, we set the probability of a user belonging to a specific group as 1 / # of groups = 1 / 12 = 0.08333. A model that does not perform better than this benchmark is very bad: it means that the information provided in the data is not being used in a way that makes our prediction better than naively saying all groups are equal.

The log loss is easy to calculate in this case because p_{ij} is always the same.

$$logloss = -\frac{1}{N} * N * \log\left(\frac{1}{12}\right) = -\log\left(\frac{1}{12}\right) = \mathbf{2.48491}.$$

Distribution of Group in Labeled Data Benchmark

Recall above that we calculated the distribution of each group in the labeled data. A good benchmark would be to use this distribution instead of an equal probabilities distribution. It's still a pretty naïve prediction because it treats all devices the same but it

at least attempts to use some information in the data. When I submitted to Kaggle based on the actual distribution of the group, it returned a log loss score of **2.42721**, which is better than the equal distribution benchmark. From the log loss score we can reverse engineer the average probability prediction for the correct label, which in this case is 0.088283 (slightly better than 0.08333 in the equal distribution case).

Top Score on Kaggle

Since this was a Kaggle competition, we have access to see how other teams' models performed. The winning score on the private leaderboard is **2.13153**. This is a good reference point, as these competitions tend to be very competitive. The top three places received $12,500, $7,500, and $5,000 respectively, so we should be very happy if our model comes close to those scores.

Benchmark Summary

Benchmark	Log Loss	Avg Probability
Kaggle – 1st place	2.13153	0.118656
Kaggle – 2nd place	2.14223	0.117393
Kaggle – 3rd place	2.14949	0.116544
Dist of labeled data	2.42721	0.088283
Equal probability	2.48491	0.083333

III. Methodology

Data Preprocessing

My goal is to create a sparse matrix on app IDs and label categories, with much inspiration coming from this kernel[4]. The final sparse matrix will look something like the following:

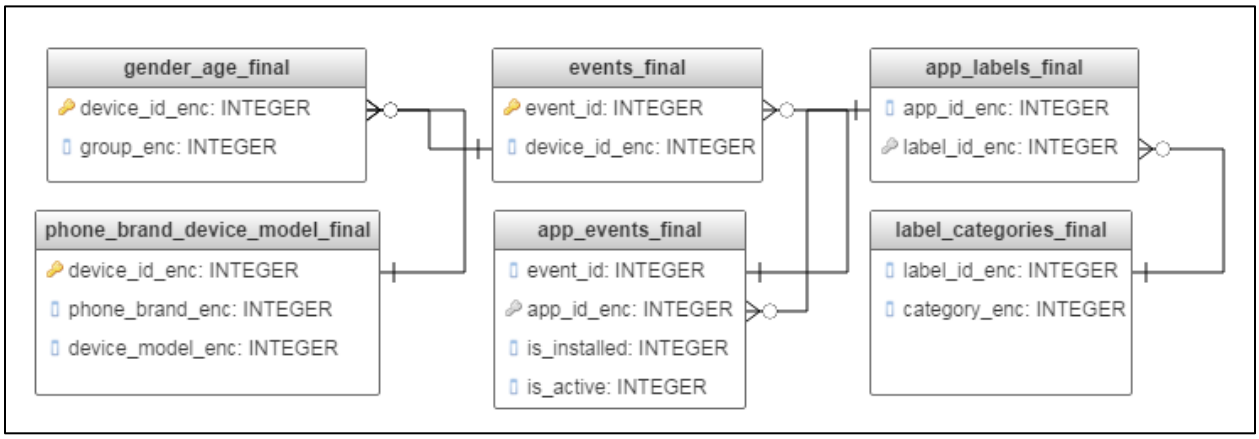
Dev ID	PB_0	PB_1	…	PB_N_1	DM_0	…	DM_N_2	AppID_0	…	AppID_N_3	LabID_0	…	LabID_N_4
1	0	1	…	0	0	…	0	0	…	0	1	…	0
2	0	0	…	0	0	…	0	0	…	0	0	…	0
…	…	…	…	…	…	…	…	…	…	…	…	…	…
M	0	0	…	0	0	…	0	0	…	0	0	…	0

Where N_1, N_2, N_3, and N_4 correspond to the number of phone brands, device models, app IDs, and label IDs. A cell will have a value of 1 if that device corresponds to that phone brand or device model, or if that device used the app or any app with that label.

The preprocessing stage is going to include three parts: data cleaning, label encoding, and sparse matrix creation.

Sneak Peek: Final Data Schema

This is my target schema for the data after all preprocessing:



- **gender_age_final** has two fields, the encoded device ID (to make the device IDs smaller) and the encoded group.
- **phone_brand_device_model** now uses encoded versions of its three fields.
- **events_final** has event ID and encoded device ID.
- **app_events_final** has an encoded version of app ID to save size.
- **app_labels_final** now uses encoded versions of app ID and label ID.
- **label_categories** now uses encoded versions of label ID and category ID.

Two questions naturally arise: what is encoding, and why bother with it? In this case, I'm going to encode each field by taking all the unique values in that field and assigning them a unique number between 0 and the count of unique values in that field (minus 1). For example (from **gender_age_train_final and gender_age_test_final**):

device_id	device_id_enc
-1000025442746372936	0
-1000030473234264316	1
-1000146476441213272	2
-1000156738840795 72	3
-1000369272589010951	4
-1000572055892391496	5

The benefit of encoding is that we save significantly on data size. We convert fields that have 15+ characters per row (e.g. device_id, app_id) into 6 characters or less depending on the number of unique values. For example, the **app_events** table given by the competition is 989 MB. After encoding the app_id field, I shrunk the table down to 695 MB while still retaining 100% of the information.

It will also be easier to create the sparse matrix from encoded values. Instead of having columns like AppID_15641251253548, they will be simplified to look like AppID_14.

Data Cleaning and Encoding

The sklearn preprocessing package has a handy function called *LabelEncoder()* that takes an array of all the possible values (duplicates are acceptable) and creates a mapping that encodes each unique value to a unique number between zero and the count number of unique values minus 1. For each field that we want to encode, we must make sure that we pass the encoder every possible value. The mappings between original and encoded value can be stored and reused.

Gender Age

I want to encode the device_id field. We know that *gender_age* is divided between a training and a testing set, so we need to combine the device_id values across both sets and pass that combination to the encoder. I also encode group by passing it to another encoder object. Lastly I remove the gender and age fields because they are both included in the group field. There are no issues here, and I write the **ga_train_final** and **ga_test_final** files to csv.

Phone Brand Device Model

Recall from the data exploration phase that **phone_brand_device_model** has duplicates that need to be fixed. I used the *.drop_duplicates* method for Pandas dataframes to keep the first record for each device ID. Also, I encoded the device ID (using the same encoder from gender_age), the phone brand, and the device model.

Events

In the events table, I first filter out the records that have a device ID that doesn't correspond to a device ID in the training or testing set. We won't be using these records to train a model, and they will all throw errors if we try to pass them through the device ID encoder. I then encode the device ID and eliminate the timestamp, longitude, and latitude fields because I won't be using them.

App Events, App Labels, and Label Categories

All I need to do to **app_events** is encode the app ID field. I do this in the R script 01-appevents-applabels-labelcategories.R. I do this manually, first creating a vector of all unique app IDs, and then assigning each to a unique number between 1 and 19,237 (the number of unique app IDs). I create a table called **app_id_encoder** that includes each app ID and its encoded value. I then join the encoder (essentially a lookup table) to **app_events**, remove app ID and keep the encoded app ID field.

The **app_labels** and **label_categories** provide more details about each app, but we only need details for the apps that are actually in **app_events**. Of the 113,211 app IDs in **app_labels**, only 19,237 exist in **app_events**. I filter out the unused app IDs from **app_labels**, and then I filter out the unused label IDs from **label_categories**.

In **app_labels_final** I encode the app IDs and the label IDs. I use the label encoder to encode label ID in **label_categories_final**, and I separately encode the categories.

More Manipulations

As I said at the beginning of this section, the goal of data preprocessing is to create a sparse matrix of binary values. I do that in the Python notebook *CapstoneSparseMatrixImplementation*, but first I make some manipulations in R.

More Data Manipulation

In the R script *02-all-app-data.R*, I make and output all the necessary manipulations to **app_events** to get the data in the exact format I will use. The initial goal is to create a log of all app_events and include the device ID (from **events**) and app label category (from **label_categories** via **app_labels**).

I join **events** to **app_events** to get device ID. Then I join the result to **app_labels** and that to **label_categories**. The final table looks like this:

device_id_enc	app_id_enc	is_installed	is_active	label_id_enc	category_enc
139459	1	1	0	1	256
81333	1	1	1	1	256
160723	1	1	1	1	256
…	…	…	…	…	…

This allows me to create the following four tables:

app_id_sparse

device_id_enc	app_id_enc
139459	1
81333	1
160723	1

This is a list unique of device_id_enc and app_id_enc. Each record in this list is a device ID and an app ID which we have a record for in **app_events** where that device ID has that app ID installed. It has 2,302,969 records.

app_id_active_sparse

device_id_enc	app_id_enc
81333	1
160723	1
177826	1

This table looks similar to app_id_sparse, but it is filtered to only include records from **app_events** where is_active = 1. This table is a unique list of all device IDs and app IDs where there is a record for that device ID and app ID in app_events such that is_active=1.

category_sparse / category_sparse_active

I follow the same process for *category_sparse* and *category_sparse_active*. Both tables have 2 columns, device ID and category (both encoded). *Category_sparse* includes all records, *category_sparse_active* limits to records where is_active=1.

Sparse Matrix Creation

Now I head back into Python to create the sparse matrices. To create the matrices I use the *csr_matrix* function in the scipy.sparse package. These create "compressed sparse matrices" which take three array inputs of the same length:

- a list of all the non-zero values (in our case all non-zero values will be 1)
- a list of the row for each non-zero value
- a list of the column for each non-zero value.

For example, if we passed the following: [(1,1,1), (1,2,3), (3,5,6)] to the compressed sparse matrix, the resulting "dense" matrix would look like:

Index	1	2	3	4	5	6
1	0	0	**1**	0	0	0
2	0	0	0	0	**1**	0
3	0	0	0	0	0	**1**

The compressed sparse matrices allow me to save a lot of memory space by avoiding having to store all the zeros.

I create a total of six sparse matrices: one for each of the following:

- phone brand
- device model
- app ID
- app ID (is_active only)
- category
- category (is_active only)

I then stack the sparse matrices together to create the final feature set.

Implementation

First I split the labeled training data into a new training (80%) and testing (20%) set. I implement the *LogisticRegression* class from sklearn's linear_model package. Initially I specify two arguments: multi_class = 'multinomial' and solver = 'lbfgs'. Multinomial specifies that we want to minimize the log loss over a probability distribution, and lbfgs is a specification of the algorithm.

I train the algorithm on the training data, and the prediction on the test data returns an average log loss of 2.48997. This is slightly worse than the equal probabilities distribution, but I haven't yet refined the model.

Refinement

The main parameter I want to explore in the *LogisticRegression* class is C, the "inverse of regularization strength". Regularization is a way of preventing overfitting by applying a penalty to increasing the probability prediction. In short if the regularization is larger, we will generally see probability predictions that are closer to the distribution and "less aggressive". Because C is the inverse of regularization strength, a lower C value corresponds to a stronger regularization.

C value	Log Loss
2.0	2.52466
1.0	2.48997
0.5	2.41648
0.1	2.33788
0.05	2.30302
0.01	2.28251
0.005	2.28975
0.001	2.32869

I tested the following C values and got the resulting log loss on the testing set shown to the right. The default value is 1.0, so we see improvement as we increase the regularization strength up to 0.01, then worse log loss scores at 0.005 and below. When I tune even further, I get the best log loss score at 0.0149.

IV. Results

Model Evaluation and Validation

Score on Kaggle

When I submit the predictions to Kaggle with the C=0.0149, I get a log loss score of 2.26966. This is better than some of our benchmarks, and I will discuss this in more detail soon.

Model Parameters

Our model has 41,124 features, each of which gets assigned a coefficient in a linear model for each of 12 possible groups. This means there are 41124*12 = 493,488 potential parameters to evaluate. Most of these are very close to 0. In the case of group F23-, only 1.7% of coefficients have an absolute value greater than 0.05.

We do see a range of intercept values, with the minimum of -0.68 and a maximum of 0.56.

Justification

Benchmark	Log Loss	Avg Probability
Kaggle – 1st place	2.13153	0.118656
Final Model	**2.26966**	**0.103347**
Dist of labeled data	2.42721	0.088283
Equal probability	2.48491	0.083333

The main justification for this model is whether we predict group with sufficiently high enough accuracy. TalkingData didn't provide any specific benchmarks, but we can compare our model's results to the benchmarks from earlier.

Our model's log loss is a 6.49% improvement over the distribution benchmark and an 8.66% improvement over the equal probability benchmark. The percentage increases

are even higher for the average probabilities (17.06% and 24.02%) It's not incredibly close to the top score on Kaggle, although achieving that would have been unexpected.

I don't work at TalkingData, so I don't have any insight over the utility of increasing our average probability to 10.33% from the distribution benchmark of 8.33%. On its face, it looks like this is very marginal improvement. Even the top score on Kaggle, 11.86%, seems like a relatively small improvement.

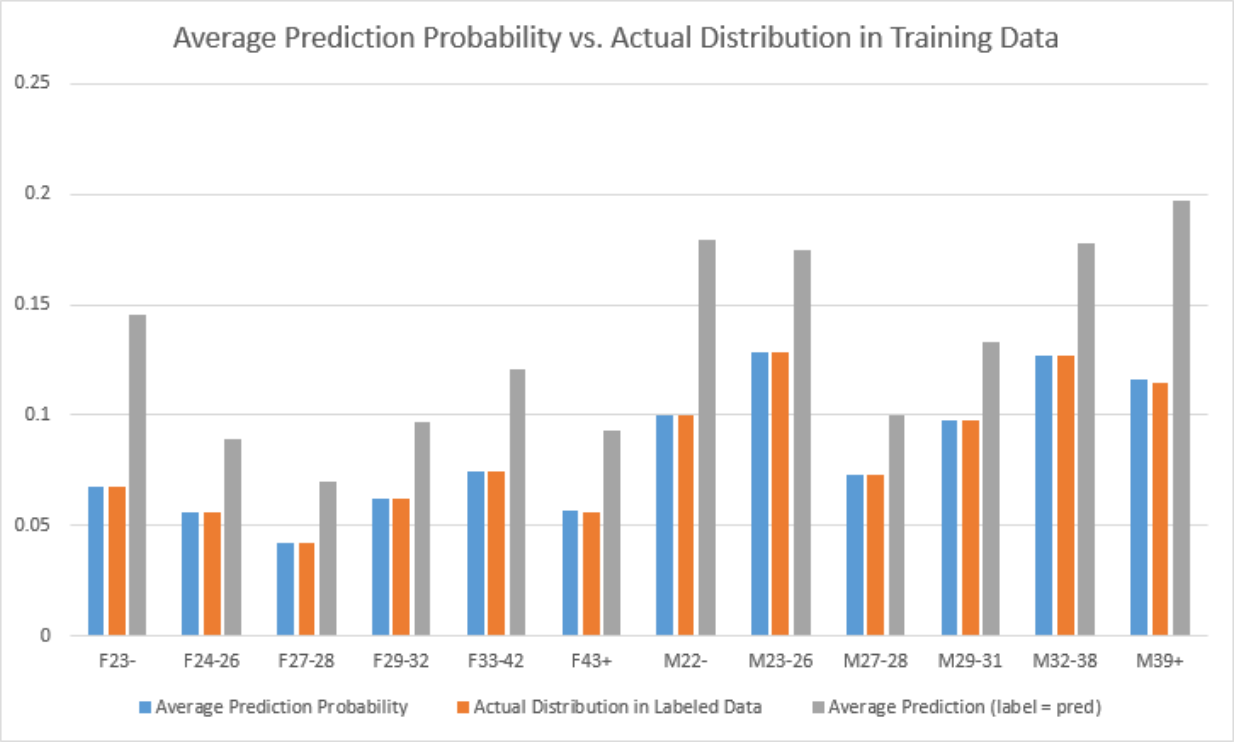
V. Conclusion

Free-Form Visualization

In the chart below I look at the average prediction probability for each group, its corresponding distribution in the labeled data, and the average prediction probability for the accurate label in the training data. There are two main points to note.

First, our average prediction probability is very close to the actual distribution across all classes. This is a good sign because it means we are not predicting any of the classes with significantly different probability than their distribution.

Second, the gray bars are the average prediction score for each label when the device corresponds to that label (from the training data). In short the goal of this project is to maximize the gray bars to as close to 1.0 as possible. It's also a good sign that the gray bars are higher than the blue and orange bars: it means our predictions are better than the distribution of the data.



Reflection

Because the goal on Kaggle was to minimize log loss of prediction on the testing set, this problem was almost entirely about prediction ability and supervised learning. Presumably we could have clustered devices by various locations and done more granular location analysis, but this was not asked for by the contest. Essentially this project said, “Here’s a ton of data, predict gender and age range.”

The data exploration and checking I did at the beginning should be common to almost all data projects. We need to verify that we have valid data as a necessary condition for a valid analysis and prediction. I also did a fair amount of data manipulation, driven largely by the need for feature engineering.

The sparse matrices I created are one of the simplest forms of feature engineering. There is no art to it, just the creation of a new column for each new binary flag (like a new app ID or phone brand). Ultimately this basic method of feature engineering allows us to make a significant improvement in predicting group.

Two integral parts of this project were new to me. I hadn’t before encountered sparse matrices or predicting a probability distribution across a set of classifications. The sparse matrices took some time to understand and implement, while the logistic regression I used for probabilistic classification was similar to other sklearn algorithms I’ve used in the past.

Improvement

The main thing I'd like to have is more information from TalkingData. Of the 186,716 devices in **gender_age**, we only have app data for 58,503. That means about two-thirds of devices are predicted solely on phone brand and device model. The only way to fix this is more data.

The only two pieces of data I did not use in the prediction were the timestamp and location information from the events table. This would require more feature preprocessing, but I think it would be interesting to see if features like most common location had some predictive value. Again, we only have data for one-third of the devices so that may only get us so far.

Lastly, Kaggle unfortunately announced late in the competition that there were some issues with "leakage" in the data provided[5]. In short, the same user was sometimes included in the training and testing set provided by Kaggle. There were some ways to figure out which devices were part of the leak, and this may have contributed to better prediction scores on Kaggle in a non-scientific way (like matching up rows in the training and testing sets). All that means is the top scores on Kaggle should be taken with a grain of salt.

VI. Appendix (16)

[1] https://www.talkingdata.com/about-us.jsp?languagetype=en_us

[2] Read more about probabilistic classification here: https://en.wikipedia.org/wiki/Probabilistic_classification

[3] http://scikit-learn.org/stable/modules/calibration.html

[4] https://www.kaggle.com/dvasyukova/talkingdata-mobile-user-demographics/a-linear-model-on-apps-and-labels. Kaggle kernels are a method for sharing data science projects. I took much inspiration from this kernel. All work in this report is mine.

[5] https://www.kaggle.com/c/talkingdata-mobile-user-demographics/forums/t/23403/some-thoughts-on-the-leak/134450