I. The Problem

Broadly, the problem is to infer, from written text, information about the demographic profile of the writer. For this specific project, I am limiting the text to be used to the text portion of Twitter statuses (tweets), and I am limiting the demographic information to be inferred to gender. In addition to extensions to other demographic attributes and to text from other media, the specific Twitter context suggests possible extensions to using non-text information (e.g. emojis included in a tweet, visual information from Twitter profiles, etc.), but I have made no deliberate attempts to extend the project beyond text.

My imaginary client is a consulting firm (Imaginary Marketing Associates, LLC, or IMA for short) that helps its clients with targeting their marketing efforts. IMA will use inferred demographic information to target marketing efforts for products expected to be of interest to particular demographic profiles.

For a couple of reasons, IMA will be interested in how the predicted demographic information associated with a tweet changes over time: first, because their clients will be reacting in real time; second, because they will be developing profiles of individuals that will depend on how their text is classified at different points in time. For example, if the typical characteristics of male and female tweets change over time, it would add confidence if the same Twitter profile produced tweets at different points in time that were classified as the same gender using models sensitive to different conditions that apply at those different points in time.

I have relied on streaming data to be obtained from Twitter via their API. I accessed these data via a developer account with Twitter, which enables me to access random streams of public tweets.

The approach has involved applying machine learning models to text embeddings. In a preliminary analysis, I used Google's Universal Sentence Encoder to produce embeddings and then applied a gradient boosted decision tree model to those embeddings to predict gender. (I can impute the "ground truth" for gender for some tweets from the display name of the user, using available gender imputation software, such as the Python gender-guesser package. For training and evaluation of the model, I have relied on tweets for which I can impute gender from the display name with reasonably high confidence.)

From a machine learning point of view, the initial problem (gender inference) is a supervised binary classification problem. Later extensions (applying it to different demographic information) could involve multi-class classification, or possibly regression.

II. Initial Data Preparation

Twitter's data stream consists of a continuous random stream of tweets in JSON format. The tweet itself is contained in the "`text`" field, with lots of metadata in the other fields. For example:

```
{"created_at":"Sun Jun 02 02:49:12 +0000
2019","id":1135015480010452993,"id_str":"1135015480010452993","text":"but any ,
back to the backwood dilema","source":"\u003ca
href=\"http:\/\/twitter.com\/download\/iphone\" rel=\"nofollow\"\u003eTwitter
for
iPhone\u003c\/a\u003e","truncated":false,"in_reply_to_status_id":null,"in_reply
_to_status_id_str":null,"in_reply_to_user_id":null,"in_reply_to_user_id_str":nu
ll,"in_reply_to_screen_name":null,"user":{"id":1636148010,"id_str":"1636148010"
,"name":"Ben Dover","screen_name":"flexthegod_","location":"the
trap","url":null,"description":"i cant hear you , im high. #LE\u00d8 RIP
Anthony \u2018Tony\u2019 Edward
Stark\ud83d\ude4f\ud83c\udffe","translator_type":"regular","protected":false,"v
erified":false,"followers_count":425,"friends_count":397,"listed_count":6,"favo
urites_count":3862,"statuses_count":15241,"created_at":"Wed Jul 31 19:05:54
+0000
2013","utc_offset":null,"time_zone":null,"geo_enabled":true,"lang":null,"contri
butors_enabled":false,"is_translator":false,"profile_background_color":"C0DEED"
,"profile_background_image_url":"http:\/\/abs.twimg.com\/images\/themes\/theme1
\/bg.png","profile_background_image_url_https":"https:\/\/abs.twimg.com\/images
\/themes\/theme1\/bg.png","profile_background_tile":true,"profile_link_color":"
0084B4","profile_sidebar_border_color":"000000","profile_sidebar_fill_color":"D
DEEF6","profile_text_color":"333333","profile_use_background_image":true,"profi
le_image_url":"http:\/\/pbs.twimg.com\/profile_images\/1129952119413497856\/KRL
hae1F_normal.jpg","profile_image_url_https":"https:\/\/pbs.twimg.com\/profile_i
mages\/1129952119413497856\/KRLhae1F_normal.jpg","profile_banner_url":"https:\/
\/pbs.twimg.com\/profile_banners\/1636148010\/1558226759","default_profile":fal
se,"default_profile_image":false,"following":null,"follow_request_sent":null,"n
otifications":null},"geo":null,"coordinates":null,"place":null,"contributors":n
ull,"is_quote_status":false,"quote_count":0,"reply_count":0,"retweet_count":0,"
favorite_count":0,"entities":{"hashtags":[],"urls":[],"user_mentions":[],"symbo
ls":[]},"favorited":false,"retweeted":false,"filter_level":"low","lang":"en","t
imestamp_ms":"1559443752659"}
```

That's the record of a single tweet. The text itself is limited by design to 280 characters, and long tweets even have their text truncated in the data stream, but with all the metadata included, the tweet takes up almost a whole page.

Initially I wasn't sure which fields would be of interest, and I decided to collect whole records, but to make the data more manageable I eliminated tweets that weren't relevant to the project. Specifically, I eliminated deletes (records merely indicating deletion of a previous tweet) and retweets (repetition of a previous tweet by a different user), both indicated by the presence of a specific field in the record; I included only tweets that had "`lang`", "`user`", and "`text`" fields, and for which the language indicated was English; I eliminated tweets with no "`name`" subfield or no "`id`" subfield under "`user`"; and I eliminated tweets for which the gender indicated by the "`name`" subfield was inconclusive. Overall, this process eliminated about 95% of the original records.

I used a package called `gender-guesser` to categorize tweets as male or female based on the display name (the "`name`" subfield of the "`user`" field).

I collected files of up to 3000 tweets each. (Sometimes the stream is broken, so the number of tweets that can be collected consecutively is not reliable. However, the majority had the full 3000.) The code for collecting a file of tweets is in `get_tweets.ipynb`. A quick initial look at some of the tweets is in `process_tweets.ipynb`.

I decided to index the tweets by timestamp and user ID, to facilitate validation in a way that avoids leakage. (This information was in the "`created_at`" field and the "`id`" subfield of the "`user`" field.) Since I had already eliminated irrelevant records in the data collection process, I had only to select relevant fields from the remaining records. Since the project was defined as predicting gender from text, the only relevant fields (aside from the user ID and timestamp to be used as record identifiers) were the "`text`" field (the tweet itself) and the "`name`" subfield of "`user`" (used to ascertain the gender label).

To clean the text, I used a function copied from Timothy Renner on Github to eliminate non-text elements (links, hashtags, mentions, media, and symbols). The code to read the tweet files, aggregate them, extract the relevant data, clean the text, and supply gender labels is in `aggregate_tweets.ipynb` and `utils.py`.

After getting an initial corpus for the primary analysis, I continued to collect files of tweets for use in a subsequent online learning procedure.

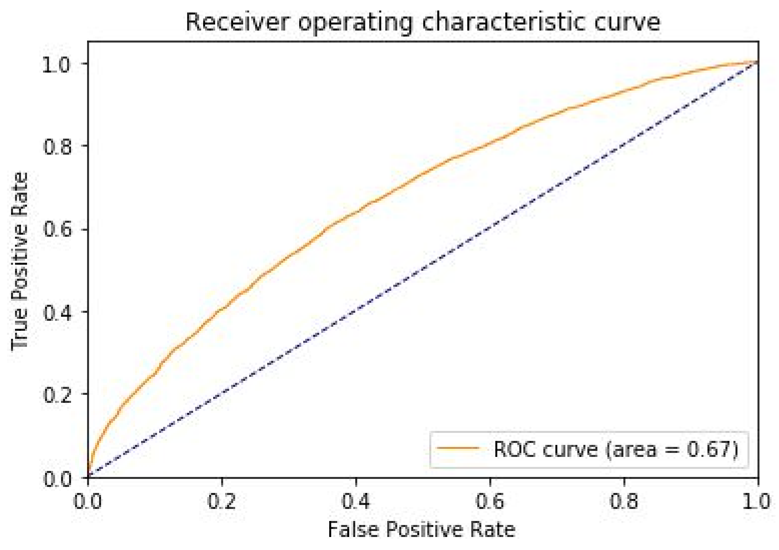
To minimize leakage, I decided to split the corpus into train, validation, and test sets in a way that entirely avoided overlap by time period or by user ID and where the time sequence from train to validation to test was monotonically forward. Also, to avoid class imbalance issues and to facilitate interpretation of the results, I decided to downsample the modal gender (male) in all three sets to create sets with equally balanced labels. The code to split and downsample the data is in `utils.py` and `split_initial_tweet_corpus.ipynb`.

III. Initial Analysis

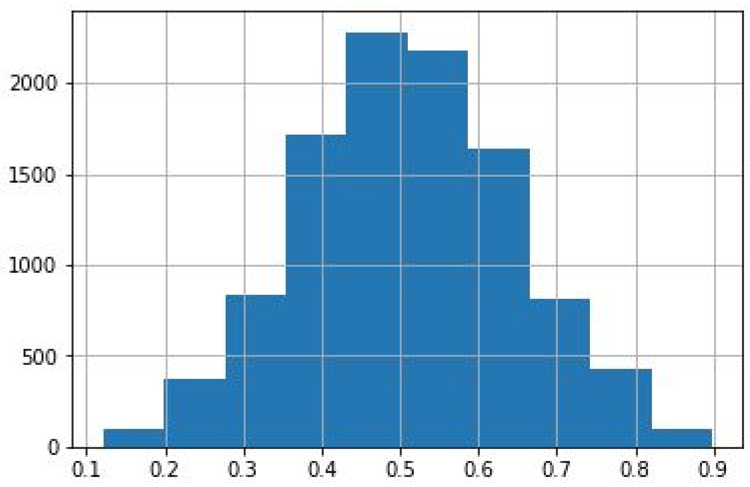
I began by using Google's Universal Sentence Encoder to create sentence embeddings. Though my initial attempts used the standard version (linked above), which uses a deep averaging network to create sentence representations, I soon settled on using the "Large" version, which applies a transformer approach. The Large version (henceforth "USE-L") gets better results and turns out to be only a little bit more resource intensive. For each tweet ("sentence"), the output of the encoder is a 512-dimensional embedding vector. The encoder can be called with its pre-trained embedding weights to produce such an output, or it can be used as part of a deep learning model with user-specified layers on top, and the weights can be fine-tuned for the specific application. The fine-tuned model can also then be used to produce embeddings as features for input to a different model.

I tried several different types of prediction models (multi-layer perceptrons, gradient boosted decision trees, and logistic regression with various kinds of preprocessing) using the embeddings from USE-L. In general the results did not differ greatly from one another in overall quality. In the initial analysis, the best results seemed to be from a regularized logistic regression of quadratic features generated from principal components of the embeddings. This is the model I have used for most of the analysis. As a demonstration, I made a Kaggle kernel that applies this model using the pre-trained embeddings. As a baseline, the test set accuracy is around 61.8 percent. (Hyperparameters—namely the number of principal components and the regularization parameter—were tuned using the validation set.)

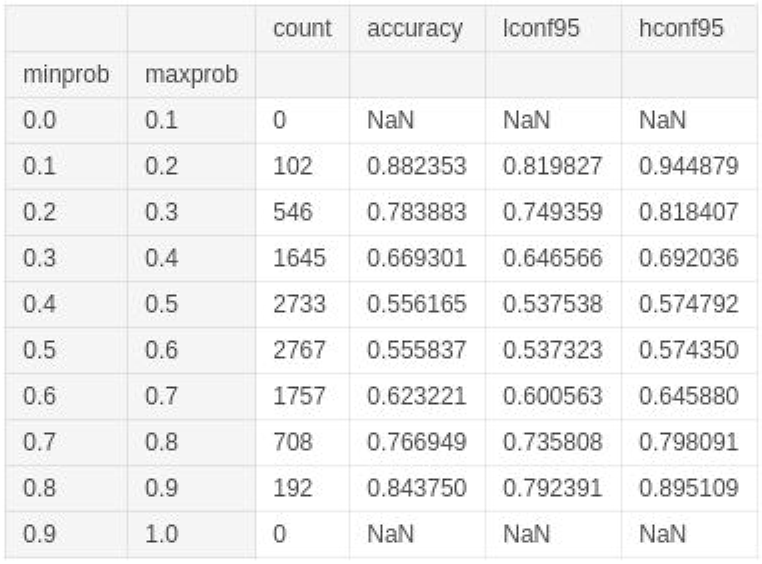
Here's what the ROC curve for the test set looks like:



Here is the distribution of predictions:

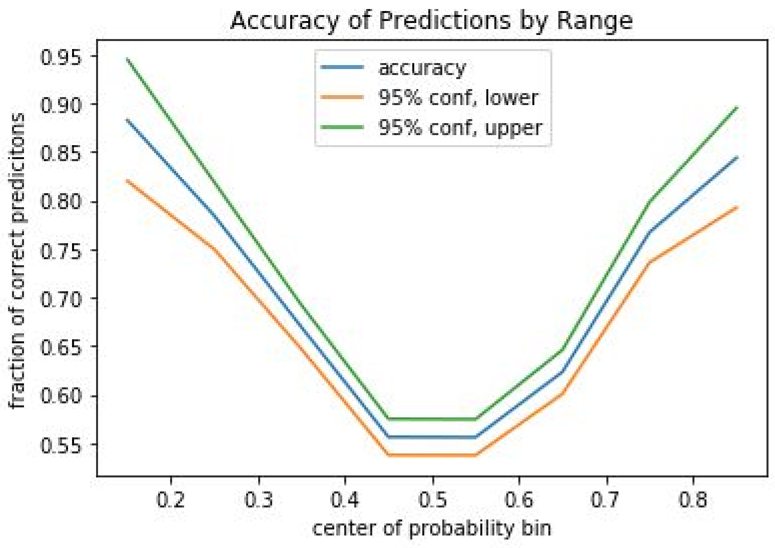


And here is how well the predictions performed, along with confidence intervals for each range:



		count	accuracy	lconf95	hconf95
minprob	maxprob				
0.0	0.1	0	NaN	NaN	NaN
0.1	0.2	102	0.882353	0.819827	0.944879
0.2	0.3	546	0.783883	0.749359	0.818407
0.3	0.4	1645	0.669301	0.646566	0.692036
0.4	0.5	2733	0.556165	0.537538	0.574792
0.5	0.6	2767	0.555837	0.537323	0.574350
0.6	0.7	1757	0.623221	0.600563	0.645880
0.7	0.8	708	0.766949	0.735808	0.798091
0.8	0.9	192	0.843750	0.792391	0.895109
0.9	1.0	0	NaN	NaN	NaN

Visually,



Essentially, for most of the tweets, the model realizes it doesn't have enough information to make a reliable prediction, but for some of the tweets, it makes appropriately aggressive predictions. It's encouraging that the actual accuracy within each prediction range corresponds (with high precision, indicated by the confidence bands) to the predicted probability within that range. The model seems to be doing a good job of predicting probabilities, in a way that generalizes well, even if the majority of those probabilities do not contain much information.

The next step was to fine-tune the embeddings. This process required adding an additional layer or layers on top of the embeddings to produce a complete deep learning model that could be optimized.   The procedure was then *first* to fit the model with the embedding weights frozen and *then* to make the embedding weights trainable and fit the model again starting from the optimized weights for the upper layer(s). The structure that seemed to work best was a single 512-unit hidden layer on top of the embeddings with a large dropout (drop=0.8) going into the final prediction. Given the large number of parameters (over 200 million) and the small amount of data (34146 training samples), I found that the model always overfit on the second unfrozen epoch, even with such extensive dropout and even with very slow learning. The final model therefore trains for a single epoch to fine-tune the embeddings after training for several epochs to generate initial weights for the upper layer.

The fine tuning model (`twitgen_use_large_best.ipynb`) has a test set accuracy of about 62.8 percent. The performance can change from one run to another, but in general it is comparable to the best performance I have obtained using the fine-tuned embeddings as features in more comprehensive models. However, the process of tuning the embeddings is costly, so for purposes of online learning it makes sense to use the already fine-tuned embedding weights rather than re-fitting the whole model. It's an open question whether the diversity allowed by including features generated by other embedding models will make the online learning process more robust.

Using the fine-tuned embeddings in my preferred logistic model (using quadratic features of principal components, see `lr_poly_corpus_tweets.ipynb`) produces a test set accuracy of only 62.0 percent, better than with the pretrained embeddings but not as good as the full deep learning model. However, the advantages of the logistic model are that it allows me to include additional features conveniently and it is somewhat easier to interpret.

I made some attempt at interpretation in another notebook (`analyze_lr_poly_results.ipynb`, which uses an earlier version of the model). Sentence embeddings do not submit easily to human interpretation, and they are not reliable from one run of the model to another, but there may be some insight that can be gained from looking at the principal components. Under the heading of "Feature Importance", I found the components that had the greatest individual impact in producing predictions near 1 or near 0 ("most masculine components" and "most feminine components") and then looked at the tweets that scored most highly on those components. Subjectively, in this particular analysis, some important "masculine" components seem to represent "argumentativeness", "inspirational talk", and "male celebrities", while some important "feminine" components seem to represent "negative interpersonal emotions", "geography", "tearfulness", and "parenthood and making a living." Subsequent runs, however, show that your mileage may vary quite a bit.

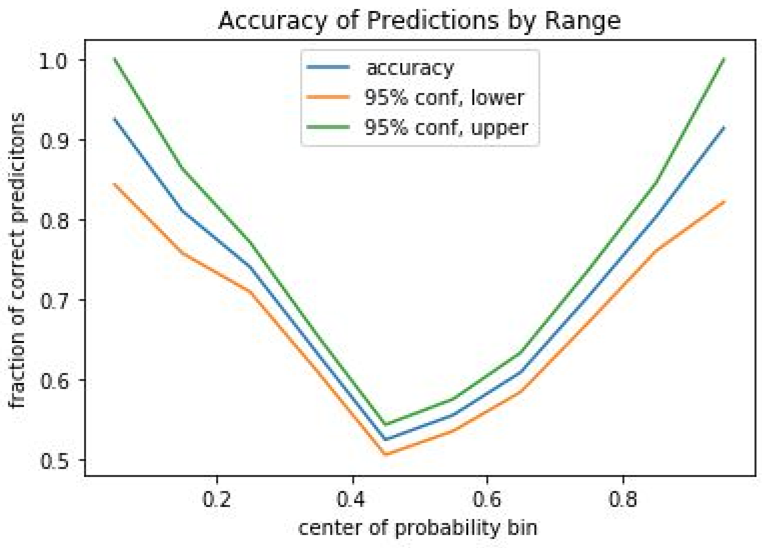
IV. Full Static Model

The final model contains two other elements: an LSTM network built on word-level embeddings initialized with Glove vectors pre-trained on Twitter, and a simple pooling network built on similarly initialized embeddings. The former is meant to capture aspects of word sequence that may not be fully captured in the transformer-based sentence embeddings, and the latter is meant—via fine tuning of the embeddings—to reflect differences in individual word usage that may not be fully captured by sentence-level (or rather "tweet-level") features.

Each of these additional submodels is used to produce a set of activations which are used as additional input features in the PCA-quadratic-logistic model described above. Since both the scales of the activations and their relative predictive power differ among the 3 underlying deep-learning models, the features from each of these models are scaled separately before taking principal components. The scaling factors, like other hyperparameters, were determined by subjectively optimizing validation set performance.

The tuned LSTM model is contained in `twitgen_glovinit_best_dl_model.ipynb`. It uses a separated alternating bidirectional two-layer-deep LSTM structure to feed into two parallel fully connected layers, along with a residual connection around those fully connected layers feeding into the final prediction layer. There is dropout at every level, including multiple kinds of dropout in the recurrent layers, and a very high rate of dropout (0.75) in the top layer.

I also make a Kaggle kernel of the standalone LSTM model, which includes test set performance. The overall test set accuracy is about 60.7 percent, not quite as good as the USE-L model. As the following chart indicates, however, the generalization is good across the range of predicted probabilities:



Adding features from the LSTM model to the PCA-quadratic-logistic model (and re-tuning the number of principal components and the regularization penalty) produces a small but noticeable improvement in test set performance (see `lr_poly_corpus_tweets_try_add_lstm.ipynb`). Accuracy rises from 62.0 percent to 62.4 percent.

The tuned pooling model is contained in `twitgen_glovinit_best_pooling.ipynb`. Except for the prediction layer (which is only used while tuning the model, since the features ultimately used are the activations that feed into that layer), the only fitted parameters are the embedding weights for the individual words. Thus this is intentionally a model of word usage, not a model of tweet construction. The network uses both global average pooling and global max pooling and concatenates the two, in an attempt to capture both typical word usage and idiosyncratic word usage. (One might ask both, for example, “Does this tweet use any word that is rarely used by women?” and “Are most of the words in this tweet words that women are somewhat less likely to use?”)

Adding features from the pooling model to the PCA-quadratic-logistic model (and re-tuning) did not materially change test set performance. (On some runs the the performance was slightly better with the new features. On the final run it was roughly comparable, better by some criteria and worse by others.) Nonetheless, I felt that having word-level features was substantively important and would likely improve the robustness of the model, so in the absence of evidence for deteriorated performance, I chose to keep the new features. (This is a question that should probably be revisited before putting the model into production.) Evaluation of the final model is contained in `lr_poly_with_lstm_and_pooled.ipynb`.

V. Online Learning

Twitter conversations change over time, and it's likely that the typical usage by men and women changes over time. A static model fit on a particular time period, though it may generalize well to the immediately subsequent time period, will likely not generalize so well to the more distant future. It is therefore desirable that a model be able to adapt over time. Re-fitting a whole model every day or every week could be expensive, and this expense could be mitigated by a model that adapts gradually to new data without requiring a complete new fit.

My prediction model has two parts—the underlying deep learning models that generate the features and the PCA-quadratic-logistic model that makes the predictions. It seems likely that the underlying features change more slowly than does the way those feature map onto predictions. For example, certain topics may change their relative interest for men and women over time even when the topics themselves remain essentially the same. The procedure I have followed here is to keep the definitions of the underlying features static (as defined by the network weights derived from the original training set, and as projected using PCA weights from the original training set) while allowing the logistic regression coefficients to evolve over time. At some point, as the underlying features become less relevant, the underlying models will need to be re-fit, but this should only need to be done occasionally, whereas the logistic regression coefficients can evolve continuously with each new batch of data.

To implement the evolving logistic fit, I decided to use stochastic gradient descent as implemented in Scikit-learn. The first step in testing this approach was to create a more extended data set on which to simulate evolution of the logistic fit. The original data set (train+validation+test) included data from the 12-day period May 21, 2019 through June 1, 2019. I collected additional data over the subsequent 35 days and combined the two datasets to produce a base for simulation. As with the original data, I balanced the classes by downsampling the modal class.

Once I had gathered the full data set, I generated and saved embeddings (and/or activations) for those data from each of the three underlying models, using stored weights from the fits on the original training set. I merged those embeddings to produce a set of features to which to apply the online learning procedure. The merging and the subsequent analysis (described below) are in the file `twitgen_online_learning.ipynb`.

My test procedure began by designating the first 60% of the records for burn in, the subsequent 20% for validation, and the final 20% for testing. My fitting procedure was to divide the whole data set into equally sized minibatches, do a partial fit on each minibatch, and use each partial fit to make predictions for the subsequent minibatch. Each partial fit consisted of a fixed number of gradient steps, with the coefficients having been initialized using the result from the previous partial fit. The size of the minibatches and number of gradient steps per partial fit were treated as hyperparameters to be optimized. There were two other hyperparameters: a scaling factor applied to all features, and the *alpha* parameter from Scikit-learn's SGDClassifier module, which determines both the regularization penalty and the learning rate.

To optimize the hyperparameters, I used a package called parfit, which takes advantage of parallel processing to optimize parameters and visualize performance over the parameter space. Parfit is designed to be used with Scikit-learn, and it requires a class that implements a model using the same interface as Scikit-learn. Normally, a model class in Scikit-learn will have a *fit* method that saves the fitted model in some form to be used later with the *predict* method. For the present analysis, I implemented what I might call a pseudo-model class, where the *fit* method performs a series of partial fits (as described in the above paragraph) and simply saves the (out-of-sample) predictions, to be returned directly by a subsequent call to *predict*. (Saving the fitted model would be difficult, since the model is evolving, and different versions of the model need to be applied to different sections of the data.)

My (partly subjective) optimization procedure consisted of 5 steps:

1. Calculate validation set accuracy 4 times, for 4 separate random seeds, over a grid that varies all 4 hyperparameters.

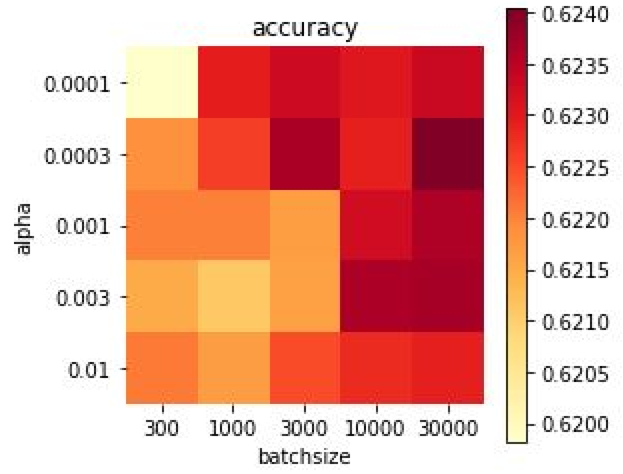
2. Choose the parameters that appear optimal, and fine tune 2 of the 4, again using 4 separate random seeds, over a 2-dimensional grid where the validation set accuracies for each seed can be represented as a heatmap.

3. Choose regions of the heatmaps that seem reliably good, fine tune the same parameters over that range (possibly extended if it was near the edge) using the same approach, and choose the point on the resulting heatmaps that seems most reliably good.

4. Fix the first 2 parameters at the chosen point, and repeat step 2 for the other 2 parameters.

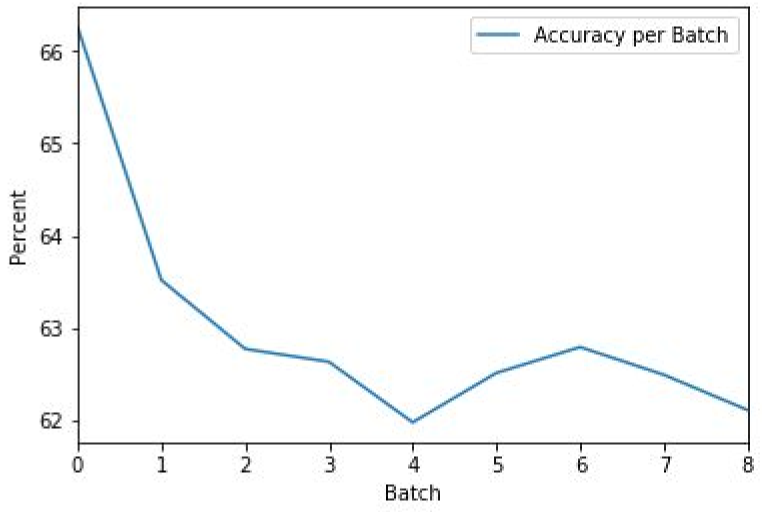
5. Fine tune those other 2 parameters again, as in step 3.

My procedure ultimately resulted in a very large batch size (20000 records, which is more than 10% of the entire data set) and very aggressive partial fits (20 gradient steps per minibatch). In practice, the online learning turned out more like a moving window fit than a slowly evolving fit. It seems that timely data is important for making an optimal prediction. On the other hand, the differences in performance relative to smaller batch sizes were not terribly dramatic, as the following representative heatmap shows:



After choosing hyperparameters, I examined the performance of the chosen model on the test set, for each of the 4 random seeds. (For coding convenience, I re-fit the model, but in principle the test set predictions would already have been made, as they simply represent allowing the same partial-fit-and-predict iterations to run past the end of the validation data.) Regardless of the seed, test set accuracy was around 62.3 percent, and test set ROCAUC was around 0.676. These results are comparable to the static model's performance on the original test set. It thus appears that the features created by the underlying model were fairly stable over the online learning time period (a total of 47 days). This result is encouraging, as it means the underlying models would not have to be re-fit very often.

I also examined the performance of the online learning model over time. The model of course performs better over the first two batches, because they represent the same data that were used to train and validate the underlying models that generate the features. After that performance drops off but seems to remain stable. One could have imagined a world in which performance would deteriorate, because the underlying features become stale. One could also have imagined a world in which performance would improve, because the first few batches didn't provide enough data to allow adequate generalization. As it is, we seem to have found a happy medium.


Accuracy per Batch
Percent
Batch