Machine Learning Engineer Nanodegree

Final Report

Ben Walsh
February 14th, 2021

Table of Contents

Introduction

Music recommendation algorithms have powered the growing dominance of audio streaming applications, despite constantly changing content and the inherently subjective nature of any art. While established artists and engaged users can provide sufficient historical data to enable a machine learning solution, there are challenges for an algorithm to predict the preferences for a new user or a new artist. Personal curation by a human is not scalable for a massive user base, so in order for online applications to efficiently scale while growing engagement with users, an automated algorithm is critical.

Project Overview

Problem Statement

Given a song-user pair, the algorithm predicts whether a user will like it or not, as evidenced by a recurring listening event, i.e. listening to the song a certain number of times within a certain time window. The target value for a song-user pair is 1 or 0, representing whether a recurring listening event occured. Overall algorithm performance is evaluated on a test set of many user-song pairs, resulting in a single aggregate percent correct.

Datasets

The datasets originate from a Kaggle competition: WSDM - KKBox's Music Recommendation Challenge

The primary training data contains:

- msno: user id
- song_id: song id
- source_system_tab: the name of the tab where the event was triggered
- source_screen_name: name of the layout a user sees
- source_type: an entry point a user first plays music on mobile apps
- target: the target variable. target=1 means there are recurring listening event(s) triggered within a month after the user's very first observable listening event, target=0 otherwise

Additional information on users are also available, which can be linked with msno (user id):

- msno
- city
- bd: age
- gender
- registered_via: registration method
- registration_init_time: format %Y%m%d
- expiration_date: format %Y%m%d

Additional information on songs are also available, which can be linked with the song_id:

- song_id
- song_length: in ms
- genre_ids: genre category. Some songs have multiple genres
- artist_name
- composer
- lyricist
- language

Benchmark Model

As is typical for Kaggle competitions, there is a leaderboard displaying the top performing submissions and notebooks which can be upvoted for relevancy and usefulness.

The top score on an unknown test set is 0.747, with the top 10% scoring at 0.692 and a median submission of 0.671. While it is impossible to know how much experience the top submissions had and how much time they put into their solution, the initial goal was to generate a solution that would score in the top half, or above 0.671 on the test set. Specifically, the solution is compared against benchmark performance by evaluating on a test set of many user-song pairs, resulting in a single aggregate percent correct.

Since the highest performing submissions have unknown approaches, the upvoted notebooks will be a proxy to benchmark models. This notebook has been upvoted and has a respectable score of 0.68138 (most are not public), which is above the median. The algorithm in the notebook is from LightGBM, a gradient boosting framework that uses tree based learning algorithms. Hyper-parameters in this solution to help compare to another tree-based approach are num_leaves: 108 and max_depth: 10.

Evaluation Metrics

The solution is compared against the benchmark models by evaluating the aggregate accuracy on a test set which the model has not trained on.

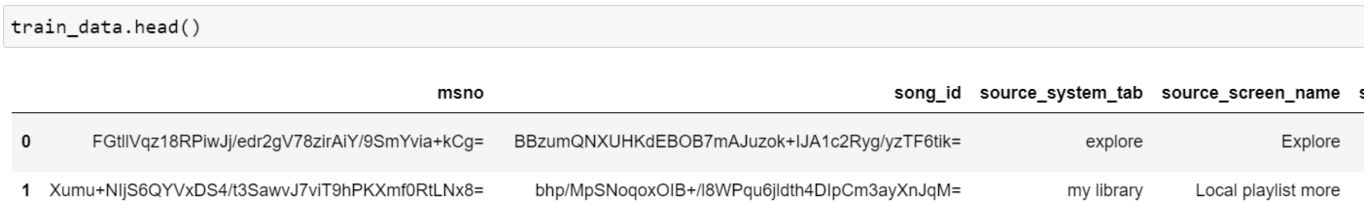
Data Pre-Processing

The full solution to the music recommendation challenge encompassed data exploration, data cleaning, feature engineering, feature selection, and algorithm implementation.

Data Exploration

The training data, member data, and song data were all received as separate files. Before merging into a single feature input, each dataset was explored.

Training Data

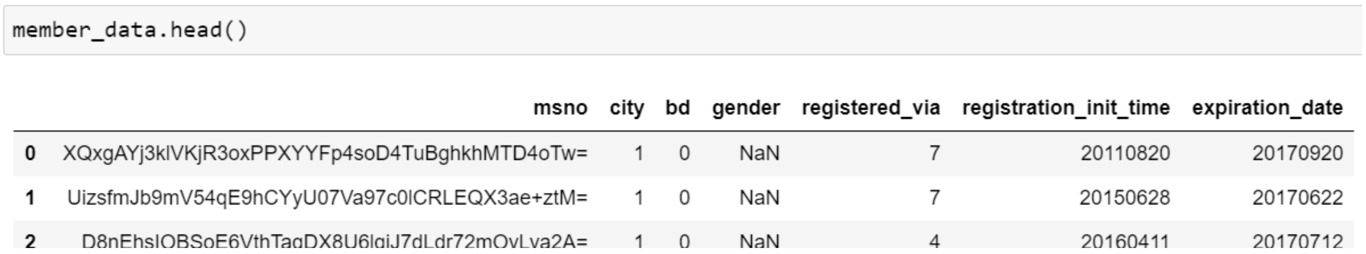


```
train_data.head()
```

	msno	song_id	source_system_tab	source_screen_name
0	FGtllVqz18RPiwJj/edr2gV78zirAiY/9SmYvia+kCg=	BBzumQNXUHKdEBOB7mAJuzok+lJA1c2Ryg/yzTF6tik=	explore	Explore
1	Xumu+NIjS6QYVxDS4/t3SawvJ7viT9hPKXmf0RtLNx8=	bhp/MpSNoqoxOIB+/l8WPqu6jldth4DlpCm3ayXnJqM=	my library	Local playlist more

The raw training data exhibits ID columns `msno` and `song_id` that were used to link with the member and song data, respectively, then dropped from the feature list since the underlying values are not informative. The remaining columns `source_system_tab`, etc. contain string values, which motivated either dropping or one-hot encoding the variables into interpretable numerical inputs.

Member Data



```
member_data.head()
```

	msno	city	bd	gender	registered_via	registration_init_time	expiration_date
0	XQxgAYj3klVKjR3oxPPXYYFp4soD4TuBghkhMTD4oTw=	1	0	NaN	7	20110820	20170920
1	UizsfmJb9mV54qE9hCYyU07Va97c0lCRLEQX3ae+ztM=	1	0	NaN	7	20150628	20170622
2	D8nEhsIOBSoE6VthTaqDX8U6lqjJ7dLdr72mOyLya2A=	1	0	NaN	4	20160411	20170712

The raw member data contains ID column `msno`, which is used to link with the raw training data. The `city` and `registered_via` values are numerical but categorical, which motivated one-hot encoding. The `bd` and `gender` values displayed unexpected values, with `gender` showing over 50% of data as `NaN`, and `bd` (age) values summarized below:

```
member_data['bd'].describe()
```

```
count    34403.000000
mean        12.280935
std         18.170251
min        -43.000000
25%          0.000000
50%          0.000000
75%         25.000000
max       1051.000000
Name: bd, dtype: float64
```

Seeing a negative minimum value and a maximum value of over 1000 motivated cleaning, where the outlier values were replaced with non-outlier mean.

The time and date-based variables are correctly expressed as meaningful numerical values.

Song Data

```
song_data.head()
```

	song_id	song_length	genre_ids	artist_name	composer	lyricist	language
0	CXoTN1eb7AI+DntdU1vbcwGRV4SCIDxZu+YD8JP8r4E=	247640	465	張信哲 (Jeff Chang)	董貞	何啟弘	3.0
1	o0kFgae9QtnYgRkVPqLJwa05zIhRlUjfF7O1tDw0ZDU=	197328	444	BLACKPINK	TEDDY| FUTURE BOUNCE| Bekuh BOOM	TEDDY	31.0
2	DwVvVurfpuz+XPuFvucclVQEyPqcpUkHR0ne1RQzPs0=	231781	465	SUPER JUNIOR	NaN	NaN	31.0
3	dKMBWoZyScdxSkihKG+Vf47nc18N9q4m58+b4e7dSSE=	273554	465	S.H.E	湯小康	徐世珍	3.0
4	W3bqWd3T+VeHFzHAUfARgW9AvVRaF4N5Yzm4Mr6Eo/o=	140329	726	貴族精選	Traditional	Traditional	52.0

The raw song data contains ID column `song_id`, which is used to link with the raw training data. The numerical columns `genre_ids` and `language` are IDs, which motivated one-hot encoding into interpretable numerical inputs.

(show metrics or .describe() of artist_name, composer, lyricist)

The columns `artist_name`, `composer`, and `lyricist` contain string values with many unique classes. Since related intuitively useful information is encompassed in `language` and `genre_ids`, this motivated dropping these features.

The `song_length` is correctly expressed as a meaningful numerical value.

Data Cleaning

In order for the raw input data to be suitable for machine learning algorithms, categorical data, missing data, and outliers were all addressed, as detailed below.

Categorical Data

Several variables are encoded as numerical IDs and are categorical in nature. For machine learning algorithms to properly predict the ID, the values are one-hot encoded to new columns corresponding to each class. The following processing is applied:

```
def dataframe_one_hot_encode(dataframe, feature):
    # Generate encoding
    oh_encoding = pd.get_dummies(dataframe[feature])

    # Concatenate new columns to dataframe
    dataframe = pd.concat([dataframe, oh_encoding], axis=1)

    # Drop original feature
    dataframe = dataframe.drop(feature, axis=1)

    return dataframe
```

As identified in Data Exploration - Training Data, `source_system_tab` is a categorical variable that is encoded in a string ID, and therefore was one-hot encoded.

As identified in Data Exploration - Song Data, song `language` is a categorical variable that is encoded in a numerical ID, and therefore was one-hot encoded.

As identified in Data Exploration - Member Data, member `city` and `registered_via` are categorical variables that are encoded in numerical IDs, and therefore were one-hot encoded.

Missing Data

As identified in Data Exploration - Member Data, member `gender` has the majority of values missing. Gender is likely to be informative, so instead of dropping the entire feature, the missing values are converted to a `UNK` keyword. Then the three categories for gender are one-hot encoded into three separate features.

Outliers

As identified in Data Exploration - Member Data, member `bd` (age) has positive and negative outliers. The outlier values, defined as outside of 0 and 120 (years), are replaced with the mean value. The calculated mean of the non-outliers is approximately 28 years old.

Feature Engineering

During the proposal, an anticipated engineered feature was `genre_group`, the result of an unsupervised learning method such as k-means clustering to categorize similar genres. The `genre_id` is intuitively useful, however one-hot encoding each genre as distinct classes misses the intuition that certain genres are much more similar to each other than others. However in practice, the input data contained no other information to create the clusters within genres, for instance descriptions of the genre or underlying audio features. To reduce the number of classes, the first digit of the ID was taken, assuming that the order is important. For instance, genres with IDs 101, 102, and 103, would all be categorized as 1.

Feature Selection

Only meaningful variables are input into an algorithm, removing variables such as IDs.

To finalize the feature input data, first the separate tables for training information with labels, song information, and user information were joined. Song information was joined on `song_id`, and member information was subsequently joined with on `msno`.

Only variables with interpretable numerical information are input as features to train on. Therefore `msno` and `song_id` were removed after joining, since the IDs are arbitrary and do not contain useful information for music recommendation.

Additional features that were removed were `source_screen_name` and `source_type` from the training data, and `artist_name`, `composer` and `lyricist` from the song data. The excluded training features were not expected to be informative. The excluded song features, while they would be informative for a human expert, do not inherently provide information without additional context such as genre or language, which were already included as features. With so many unique values, one-hot encoding would not be feasible.

Algorithm Implementation

Baseline Model

To establish a baseline for binary classification, a logistic regression model was created. The performance on a test set was 50.4%, or essentially the same as random chance.

Final Model

The final solution uses XGBoost, a popular optimized tree-based ensemble learning algorithm. After hyper-parameter optimization, the model parameters and performance are as follows:

Parameter	**Value**
test_accuracy	0.637
max_depth	12
n_estimators	50

For reference, the top score on an unknown test set is 0.747, with the top 10% scoring at 0.692 and a median submission of 0.671.

This notebook has been upvoted and has a respectable score of 0.68138 (most are not public), which is above the median. The algorithm in the notebook is from LightGBM, a gradient boosting framework that uses tree-based learning algorithms. Hyper-parameters in this solution are num_leaves: 108 and max_depth: 10.

Optimization

To aid model optimization, a simple model registry was created. The registry logs metadata for each model, creating a history to track what combination of hyper-parameters resulted in the best performance. In code:

```
hyper_params = { \
'objective': model.get_params()['objective'],
'colsample_bytree': model.get_params()['colsample_bytree'],
'learning_rate': model.get_params()['learning_rate'],
'max_depth': model.get_params()['max_depth'],
'alpha': model.get_params()['alpha'],
'n_estimators': model.get_params()['n_estimators']}

model_registry_info = { \
model_index : \
{\
    'time': timestamp_str,
    'data-features': list(X_train.columns.values),
    'model-params': xgb_hyper_params,
    'train-acc': train_acc,
    'test-acc': test_acc }
```

}

The above example creates a log with a timestamp, input features, model hyper-parameters, and accuracy.

The primary hyper-parameters explored were:

- **max_depth**: The maximum depth of any single tree
- **n_estimators**: Number of gradient boosted trees, or size of ensemble
- **alpha**: The L1 regularization term, penalizing large feature weights

Evaluation Metrics

The solution is evaluated against the benchmark models by comparing the aggregate accuracy on a test set which the model has not trained on. In code:

```
test_predictions = model.predict(X_test)
num_correct = (test_predictions == test_truth).sum()
test_acc = num_correct / len(test_prediction)
```

Discussion

Refinement

By establishing a model registry early in development, changes in model hyper-parameters and input data were observed and logged. Many of the early improvements were realized from cleaning input data. Certain cleaning steps actually resulted in slight decreases in performance, but those are considered "lucky" that a non-sensical trend that the model learned happened to result in better performance. In these cases, cleaned features were kept since it is more intuitive and more likely to perform when deployed, even though it "hurt" performance.

Additional performance gains were realized from hyper-parameter tuning. Generally, increasing max_depth and n_estimators had a positive effect on accuracy, with tradeoffs in training time and overfitting. Decreasing alpha had a positive effect on accuracy, with a tradeoff in overfitting since regularization had a smaller effect. In all cases, comparing accuracy between the training and test set mitigated the risk of overfitting.

While automatic hyper-parameter tuning methods such as Grid Search Optimization were initially considered, steady progress from manual tuning coupled with computing memory constraints discouraged further exploration.

Parameters

Compared to the upvoted LightGBM, the overall structure is of similar size. Although the algorithms are not directly comparable, `n_estimators` of 50 for XGBoost with a `max_depth` of 12 is of the same magnitude as the LightGBM solution's `num_leaves` of 108 and `max_depth` of 12. The performance is modestly lower, suggesting that futher data cleaning or hyper-parameter tuning may yield further performance benefit.

Conclusion

Overall, although the initial goal of matching a popular upvoted notebook was not achieved, performance is significantly greater than a baseline model. This capstone is considered a success, as model performance against an unseen problem space was achieved through successful implementation of data cleaning, feature selection, algorithm implementation, and hyper-parameter tuning.