

# How to Implement Learning to Rank Model using Python

The step-by-step guide on how to implement the lambdarank algorithm using Python and LightGBM



Ransaka Ravihara · [Follow](#)

Published in Towards Data Science · 9 min read · Jan 18



73



1





Photo by [Andrik Langfield](#) on [Unsplash](#)

In my previous two articles, I discussed the basic concepts of Learning to Rank models and widely used evaluation metrics for evaluating LTR models. You can access those using the links listed below.

### **What Is Learning to Rank: A Beginner's Guide to Learning to Rank Methods**

A guide on how to approach LTR problems in Machine Learning

[towardsdatascience.com](https://towardsdatascience.com)

## How to evaluate Learning to Rank Models

A practical guide on how to evaluate LTR models in Machine Learning

towardsdatascience.com

In this article, we will build a lambdarank algorithm for anime recommendations. A research group first introduced LambdaRank at Microsoft, and now it's available on Microsoft's LightGBM library with an easy-to-use sklearn wrapper. Let's start.



Search

Write

Sign  
up

Sign  
In



As mentioned above, ranking is widely used in search engines. But it's not limited to search engines; we can adopt the concept and build a solution whenever applicable. Assuming that we want to develop a ranking model for a search engine, we should start with a dataset with queries, its associated documents(URLs), and the relevance score for each query document pair, as shown below.

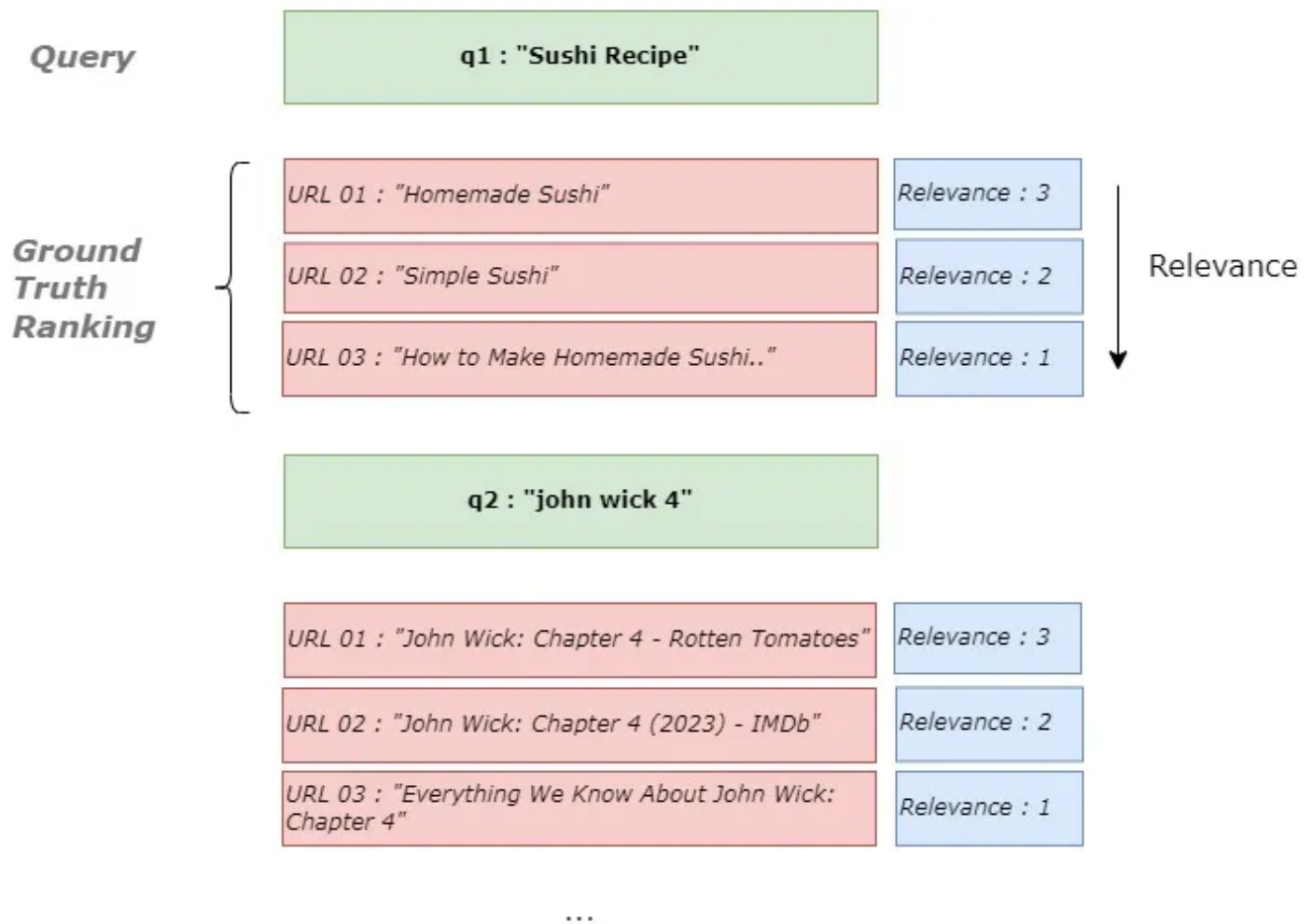


Image by Author

Finally, we can derive features based on each query and document pair.

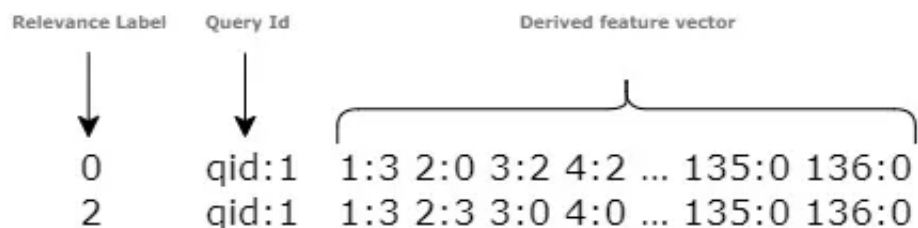


Image by Author

This is the dataset format for well-known Learning to Rank research papers and datasets. Well, that's about search engines. Let's discuss how we can adapt these concepts for traditional product recommendation tasks. There are various ways to recommend products to users. You can show recommendations when the user purchases an item, or while a user is browsing the page, etc. But for simplicity, let's narrow it down to a specific scenario.

This article will build an anime recommendation model for users' homepage customization. When the user logs into the account, we need to show animes based on the relevance scores predicted by the ranker model. Here I will use Anime Recommendation LTR Dataset, which is available under a public license. You can access it via [this kaggle link](#).

Let's read the dataset and quickly check its columns.

```
import zipfile
import pandas as pd

zipped_data = zipfile.ZipFile("anime-recommendation-ltr-dataset.zip")

anime_info_df = pd.read_csv(zipped_data.open('anime_info.csv'))
relevance_scores = pd.read_csv(zipped_data.open('relevance_scores.csv'))
user_info = pd.read_csv(zipped_data.open('user_info.csv'))

anime_info_df.columns.tolist()
```

```
# ['anime_id',  
#  'Genres',  
#  'is_tv',  
#  'year_aired',  
#  'is_adult',  
#  'above_five_star_users',  
#  'above_five_star_ratings',  
#  'above_five_star_ratio']  
  
user_info.columns.tolist()  
  
# ['user_id',  
#  'review_count',  
#  'avg_score',  
#  'score_stddev',  
#  'above_five_star_count',  
#  'above_five_star_ratio']  
  
relevance_scores.columns.tolist()  
#['anime_id', 'anime_name', 'user_id', 'relevance_score']
```

Alright, let me explain the methodology I am going to use here. Unlike search engine data, we don't have query and document pairs in this use case. So we will treat users as queries and the animes they are interested in as documents. One search query can be associated with multiple documents; users can interact with many animes. You got the idea, right :)

The target label we predict here is *relevance\_score*, stored in the *relevance\_scores* dataset. When we build a ranking model, it will learn a function to rank those anime into an optimum order where the highest relevant anime comes first for each user.

Next, we need to create a dataset by joining the above three datasets. I will

also create new features based on anime and user features. Let's create a dataset.

```
from sklearn.preprocessing import MultiLabelBinarizer

popular_genres = ['Comedy',
                  'Action',
                  'Fantasy',
                  'Adventure',
                  'Kids',
                  'Drama',
                  'Sci-Fi',
                  'Music',
                  'Shounen',
                  'Slice of Life']

def create_genre_flags(df, popular_genres):
    df = df.dropna(subset=['Genres'])
    df['Genres'] = df['Genres'].apply(lambda x: ",".join(s.strip() for s in x.split(',')))
    # use MultiLabelBinarizer to create a one-hot encoded dataframe of the genres
    mlb = MultiLabelBinarizer()
    genre_df = pd.DataFrame(mlb.fit_transform(df['Genres'].str.split(',')),
                           columns=mlb.classes_,
                           index=df.index)

    # create a new dataframe with the movie id and genre columns
    new_df = pd.concat([df['anime_id'], genre_df[popular_genres]], axis=1)
    new_df.columns = ['anime_id'] + popular_genres
    return new_df

anime_genre_info_df = create_genre_flags(anime_info_df, popular_genres)
anime_info_df_final = anime_info_df.merge(anime_genre_info_df, on='anime_id')
anime_info_df_final.columns = [col if col != 'anime_id' else f"ANIME_FEATURE {col}" for col in anime_info_df_final.columns]
user_info.columns = [col if col != 'user_id' else f"USER_FEATURE {col}" for col in user_info.columns]

train_interim = relevance_scores.merge(anime_info_df_final)
train = train_interim.merge(user_info, how='inner')
```

Let's quickly check a few statistics of our dataset. In total, there are 4.8Mn user-anime interactions, 15K users, and 16K animes in the dataset.

Here is the user and anime interaction distribution.

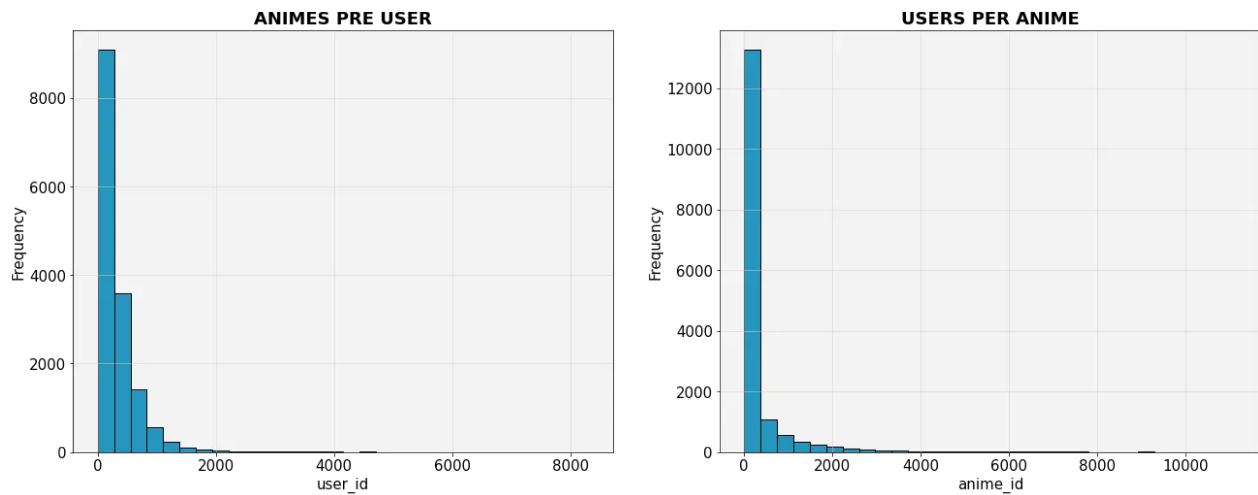


Image by Author

The below chart shows how the relevance score is distributed.



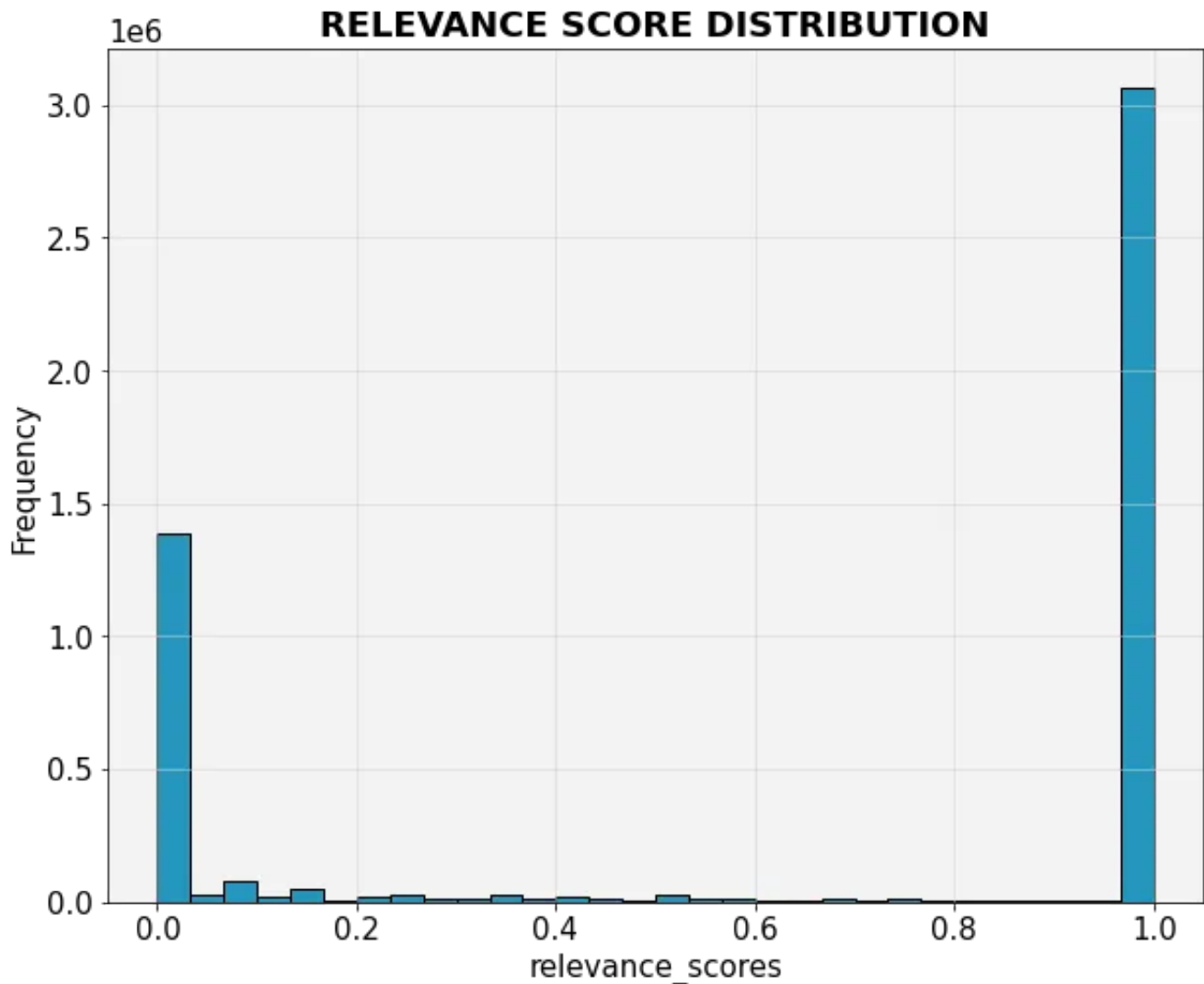


Image by Author

Now we have created the dataset for model training. But one of the main confusing points of the Learning to Rank model is the *group* parameter. Because the *group* is a strange parameter for us since it is not commonly used in other machine-learning algorithms. The idea of a *group* parameter is partitioning the dataset for each query and document pair. It enables the model to learn the relative importance of different features within each group, which can improve the model's overall performance. Having ten user-anime pairs means we have ten groups in our LTR model. Each group

can be different in size. But the sum of groups should be the number of samples we have. In simple words, we have to provide group boundaries using *group* parameters. Then the model can identify each user-anime instance separately.

For example, if you have a 6-anime dataset with `group = [3, 2, 1]`, which means that you have three groups, where the first three records are in the first group, records 4–5 are in the second group, and record 6 is in the third group.

Image by Author

**So it's vital to sort the dataset by *user\_id(query\_id)* before creating the group parameter.**

```
na_counts = (train.isna().sum() * 100/len(train))
train_processed = train.drop(na_counts[na_counts > 50].index,axis=1)
train_processed.sort_values(by='user_id',inplace=True)
train_processed.set_index("user_id",inplace=True)

features = ['ANIME_FEATURE IS_TV',
            'ANIME_FEATURE YEAR_AIRED', 'ANIME_FEATURE IS_ADULT',
            'ANIME_FEATURE ABOVE_FIVE_STAR_USERS',
            'ANIME_FEATURE ABOVE_FIVE_STAR RATINGS',
            'ANIME_FEATURE ABOVE_FIVE_STAR_RATIO', 'ANIME_FEATURE COMEDY',
            'ANIME_FEATURE ACTION', 'ANIME_FEATURE FANTASY',
            'ANIME_FEATURE ADVENTURE', 'ANIME_FEATURE KIDS', 'ANIME_FEATURE DRAMA',
            'ANIME_FEATURE SCI-FI', 'ANIME_FEATURE MUSIC', 'ANIME_FEATURE SHOUNEN',
            'ANIME_FEATURE SLICE OF LIFE', 'USER_FEATURE REVIEW_COUNT',
            'USER_FEATURE AVG_SCORE', 'USER_FEATURE SCORE_STDDEV',
            'USER_FEATURE ABOVE_FIVE_STAR_COUNT',
            'USER_FEATURE ABOVE_FIVE_STAR_RATIO']
target = 'relavence_score'

test_size = int(1e5)
X,y = train_processed[features],train_processed[target].apply(lambda x:int(x *
test_idx_start = len(X)-test_size

xtrain,xtest,ytrain,ytest = X.iloc[0:test_idx_start],X.iloc[test_idx_start:],y.
```

In the above code snippet, I have done the following steps,

1. Dropped all the columns which have more than 50% null values.
2. Sorted dataset based on *user\_id*. Otherwise, a *group* parameter will have inconsistent groups everywhere.
3. After sorting the dataset, I declared the last 100,000 rows as validation data and the rest as training data.
4. Since the LightGBM model expects integers as target values, I have

scaled the target between 1–10 and converted it to an integer. ( If you want, try converting it to 0 or 1 based on the threshold).

Let's define the group parameters and quickly fit the model as follows.

```
get_group_size = lambda df: df.reset_index().groupby("user_id")["user_id"].count

train_groups = get_group_size(xtrain)
test_groups = get_group_size(xtest)

print(sum(train_groups) , sum(test_groups))
#(4764372, 100000)

model = LGBMRanker(objective="lambdarank")
model.fit(xtrain,ytrain,group=train_groups,eval_set=[(xtest,ytest)],eval_group=

#....
# [97] valid_0's ndcg@1: 0.900624 valid_0's ndcg@2: 0.900015 valid_0's ndcg@3:
# [98] valid_0's ndcg@1: 0.900624 valid_0's ndcg@2: 0.900015 valid_0's ndcg@3:
# [99] valid_0's ndcg@1: 0.900624 valid_0's ndcg@2: 0.901216 valid_0's ndcg@3:
# [100] valid_0's ndcg@1: 0.900624 valid_0's ndcg@2: 0.901216 valid_0's ndcg@3:
```

Alright, now we finished the training. It's time to make some predictions about specific customers.

There is one more critical point to notice here. That is, how the model predictions are calculated. I was confused because when calling the `.predict` method, it does not expect additional parameters such as *group*. So I found below information from GitHub issues. According to the creator of LightGBM mentioned in [this issue](#), LightGBM's `lambdarank` uses a

pointwise approach to generate predictions. Meaning we do not want to provide additional parameters such as *group*. We can call the `.predict()` method with the correct feature vector.

However, since we are developing the LTR model, it's essential to have some candidate products and rank those according to the predicted relevance score. To generate candidate animes, I will use a straightforward method here. That is, select the animes that are not exposed to a given user. Select an  $N$  random subset from unexposed animes. Generate features based on user and selected  $N$  anime subset. Finally, use generated feature vector to get the relevance score and sort animes based on the relevance score. But in real-world use cases, we should use some meaningful methodologies. As an example, you can generate candidates as follows,

- Select the user's favorite  $N$  number of genres.
- For each genre in the above-selected genres, pick the highest-rated  $m$  animes. Now you have  $M * N$  animes to rank for that user. Just create the user base and anime-based features. And finally, call the `.predict()` method with the created feature vector.

```
user_2_anime_df = relavence_scores.groupby("user_id").agg({"anime_id":lambda x:  
user_2_anime_map = dict(zip(user_2_anime_df.index,user_2_anime_df['anime_id']))  
  
#create candidate pool, this will be a all the animes in the database  
candidate_pool = anime_info_df_final['anime_id'].unique().tolist()  
  
#anime_id to it's name mapping  
anime_id_2_name = relavence_scores.drop_duplicates(subset=["anime_id","Name"])[
```

```

anime_id_2_name_map = dict(zip(anime_id_2_name['anime_id'],anime_id_2_name['Name']))

def candidate_generation(user_id:int,candidate_pool:list,user_2_anime_map:dict,
    """
    Note: this a totally random generation, only for demo purpose
    Generates a list of N anime candidates for a given user based on their prev

    Parameters:
        user_id (int): The user's ID.
        candidate_pool (list): A list of all possible anime candidates.
        user_2_anime_map (dict): A dictionary that maps users to their liked an
        N (int): The number of anime candidates to generate.

    Returns:
        already_interacted (list): List of animes which user already liked
        candidates (list): A list of N anime candidates for the user.
    """

    #get the already liked animes
    already_interacted = user_2_anime_map[user_id]

    #candidates will be rest of animes which are not exposed to user
    candidates = list(set(candidate_pool) - set(already_interacted))

    return already_interacted,np.random.choice(candidates,size=N)

def generate_predictions(user_id,user_2_anime_map,candidate_pool,feature_columns,
    """
    Generates predictions for anime recommendations for a given user.

    Parameters:
        user_id (int): The user's ID.
        user_2_anime_map (dict): A dictionary that maps users to their liked an
        candidate_pool (list): A list of all possible anime candidates.
        feature_columns (list): A list of feature columns to use for generating
        anime_id_2_name_map (dict): A dictionary that maps anime IDs to their n
        ranker (object): A trained model object that is used to generate predic
        N (int): The number of anime predictions to generate.

    Returns:
        predictions (DataFrame): A dataframe containing the top N anime recomme
    """
    already_liked,candidates = candidate_generation(user_id,candidate_pool,user

    #Create dataframe for candidates

```

```
candidates_df = pd.DataFrame(data=pd.Series(candidates,name='anime_id'))

# Merge with feature dataframe
features = anime_info_df_final.merge(candidates_df)

#Add user id as a feature
features['user_id'] = user_id

# Merge with user information
features = features.merge(user_info)

# If number of already liked animes is less than number of candidates
# Extend the already liked list with -1
already_liked = list(already_liked)
if len(already_liked) < len(candidates):
    append_list = np.full(fill_value=-1,shape=(len(candidates)-len(already_liked)))
    already_liked.extend(list(append_list))

#Create dataframe for predictions
predictions = pd.DataFrame(index=candidates)
#Add anime names
predictions['name'] = np.array([anime_id_2_name_map.get(id_) for id_ in candidates])
#Generate predictions
predictions['score'] = ranker.predict(features[feature_columns])
predictions = predictions.sort_values(by='score',ascending=False).head(N)

predictions[f'already_liked - sample[{N}]'] = [anime_id_2_name_map.get(id_) for id_ in candidates]
return predictions

#let's generate the predictions
generate_predictions(123,user_2_anime_map,candidate_pool,feature_columns=feature_columns)
```

Additionally, we can use shap to explain the model's predictions.

```
import shap

def generate_shap_plots(ranker, X_train, feature_names, N=3):
    """
    Generates SHAP plots for a pre-trained LightGBM model.

    Parameters:
        ranker (lightgbm.Booster): A trained LightGBM model
        X_train (np.ndarray): The training data used to fit the model
        feature_names (List): list of feature names
        N (int): The number of plots to generate

    Returns:
        None
    """
    explainer = shap.Explainer(ranker, X_train, feature_names=feature_names)
    shap_values = explainer(X_train.iloc[:N])

    # Create a figure with 2 subplots
    # fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15,5))

    # Plot the summary plot on the first subplot
    plt.subplot(1, 2, 1)
    shap.summary_plot(shap_values, feature_names=feature_names, plot_type='bar')
```



```
# Plot the feature importance plot on the second subplot
plt.subplot(1, 2, 2)
shap.summary_plot(shap_values, feature_names=feature_names, plot_type='dot')

plt.show()
```

```
generate_shap_plots(model,xtrain,features,N=10000)
```

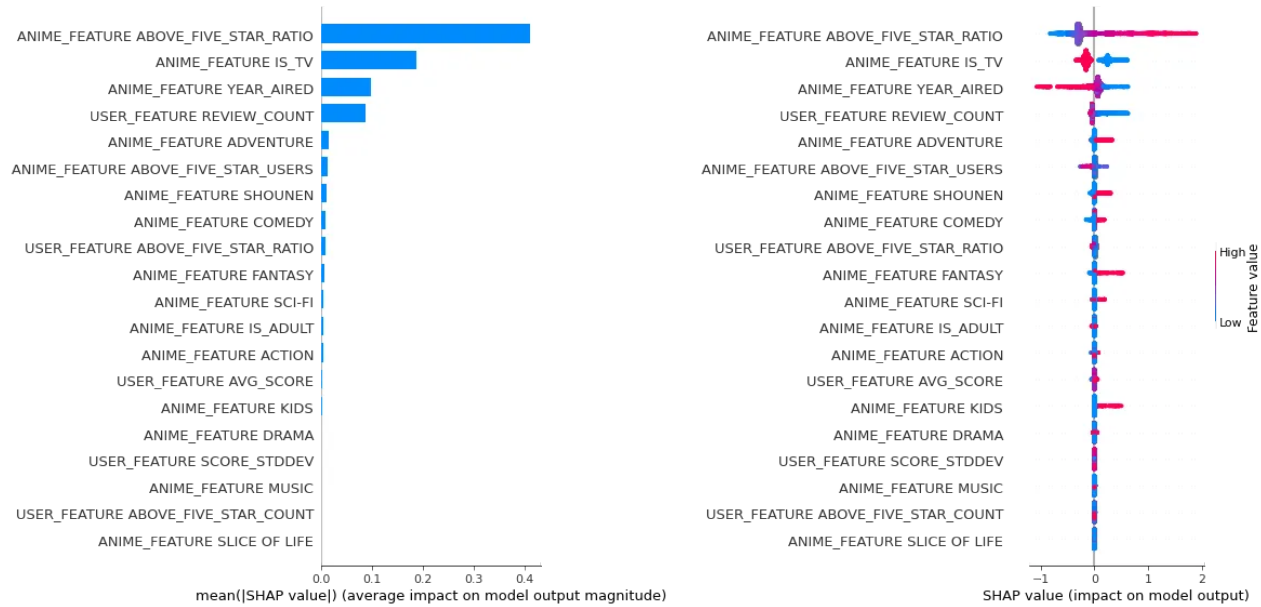


Image by Author

## Conclusion

This article aims to give a good starting point to address readers' specific use cases. Because when I started building an LTR model for my project, there wasn't a good guide for beginners. However, training the LTR model may not be necessary for some use cases. You can use straightforward methods like regression, multiclass/ label classification, or clustering. So don't over-engineer your solutions; pick the tool wisely ; )

Thank you for reading! The notebook for this article is available in my [GitHub repo](#).

## References :

- [LightGBM documentation](#)
- [Microsoft Learning to Rank Datasets](#)

[Machine Learning](#)[Data Science](#)[Recommendation System](#)[Learning To Rank](#)



## Written by Ransaka Ravihara

196 Followers · Writer for Towards Data Science

Data Scientist

Follow



### More from Ransaka Ravihara and Towards Data Science



Ransaka Ravihara in Towards Data Science

## Gaussian Mixture Model Clearly Explained

The only guide you need to learn everything about GMM

9 min read · Jan 10



49



4



Damian Gil in Towards Data Science

## Mastering Customer Segmentation with LLM

Unlock advanced customer segmentation techniques using LLMs, and improve your...

23 min read · Sep 26



3K



25



 Khouloud El Alami in Towards Data Science

## Don't Start Your Data Science Journey Without These 5 Must-D...

A complete guide to everything I wish I'd done before starting my Data Science...

18 min read · Sep 24



2.3K

 22



 Ransaka Ravihara in Towards Data Science

## What Is Learning to Rank: A Beginner's Guide to Learning to...

A guide on how to approach LTR problems in Machine Learning

7 min read · Jan 16



157






---

See all from Ransaka Ravihara

See all from Towards Data Science

---

## Recommended from Medium

 Chandramouli Guna

## Unlocking Advanced Ranking Algorithms: LambdaMART with...

Introduction:

10 min read · Jun 21

 Vyacheslav Efimov in Towards Data Science

## Introduction to Ranking Algorithms

Learn about main ranking algorithms for sorting search results

12 min read · Aug 15



---

### Lists



#### Predictive Modeling w/ Python

20 stories · 481 saves



#### Natural Language Processing

698 stories · 309 saves



#### Practical Guides to Machine Learning

10 stories · 553 saves



#### New\_Reading\_List

174 stories · 147 saves

---

 Ashish Ranjan Karn

## Thompson Sampling—Python Implementation

Thompson Sampling is a popular probabilistic algorithm used in decision-...


5 min read · Jul 20

 Benjamin Witte... in Stanford CS224W GraphML ...

## Spotify Track Neural Recommender System

By Eva Batelaan, Thomas Brink, and Benjamin Wittenbrink

22 min read · May 16

 Christophe Atten in DataDrivenInvestor

## Effective Feature Engineering for Random Forest, XGBoost, and...

Enhance Model Performance and Uncover Insights with Data Optimization

★ · 9 min read · May 22

 Pratyush Kh... in Artificial Intelligence in Plain En...

## How to Optimize Your Strategy using Multi-Arm Bandits and...

A Comprehensive Guide to Multi-Arm Bandits: Epsilon-Greedy, Upper Confidence...

6 min read · May 22



[See more recommendations](#)

---

[Help](#) [Status](#) [About](#) [Careers](#) [Blog](#) [Privacy](#) [Terms](#) [Text to speech](#) [Teams](#)