LEARNING TO RANK ALGORITHM

BY AKSHIT KALANTRI

Introduction to Learning to Rank

Today, the online world is driven by search. Google has become synonymous to looking up information online, Amazon and other e-commerce marketplaces have sophisticated search results catered to the user's browsing and purchase history and even airline booking services have their algorithms offer the best deals with both prices and travel time considered. Now this magic behind the scenes is something that is a bit more unique than traditional ML, it is an algorithm termed 'Learning to Rank' (LTR).

Learning to Rank is a smart algorithm that ranks a list based on the gradient or the degree of its suitability. To do so, it does not rely on a numerical designation for each item in the list, but instead just to what degree the item satisfies the criteria. It is a unique approach to help tackle what can be computationally intensive rankings and accurately predict what the query demands of it.

Different Approaches to Learning to Rank

1. Pointwise Ranking- This is the simplest ranking method utilized for learning. It uses a linear regression approach to help the model determine the relevance factor of a given query and the document in question. Then, the model sorts all relevance scores in a descending order and thus finds the best possible results for each query.
   For a query q and a document d_i from a list of documents D, the score s is determined by comparing with the expected output y, i.e. $s \approx y$

2. Pairwise Ranking- The next method utilizes a similar approach, but it instead determines the relevance scores for each possible document-query pair, and then sorts them to determine the most relevant pairs. Popular implementations of this method include RankNet and subsequently LambdaRank as well.
   Here, pairs of the possible solutions are compared, $q,(d_i,d_j)$, using their scores and expected predictions.

3. Listwise Ranking- The newest method employs boosted decision trees to improve on the LambdaRank model and help determine the best ordering for each list by placing the options in different sequences.
   The most popular method is listwise ranking, as it compares the whole list of possible solutions, q,D, and thus produces the most accurate predictions.

Learning to Rank versus Traditional ML

Learning to Rank is not too dissimilar to the traditional Machine Learning approach but has a few key changes. Both the approaches rely on a large amount of data and learning to improve and also aim to cut down computational load. However, unlike an ML approach, LTR does not seek to numerically rank each item on the list, making it far more efficient in ordering long lists of data. It only aims to rank the items by their relevance to the search, hence giving the best outputs for use cases such as search engines.

Current Practical Applications of LTR

LTR has become widely popular in the recent years and is now deployed for various applications in the industry.

1. Search Engines:

   The most prominent use of LTR is to optimize search engines. These include websites in several domains such as e-commerce, media platforms and recommendation systems for any number of products and services. There is a drastic difference of interaction in websites which offer a sophisticated search functionality and those that do not. In fact, studies have found there to be a multi-fold improvement in conversion rate of users visiting and users actually making purchases, in sites that provide a search option.

2. Translation:

   An interesting and unique area of application of LTR is in translation, where a lot of models can help with streamlining and optimizing translation features. This can result in better query results, as translation can be a nuanced task, with the same words and phrases having differing interpretations.

3. Newsfeed:

   Social media apps rely heavily on user engagement and interaction with posts, making the newsfeed the most important part of website. Using learning to rank, the user can receive personalized recommendations and the newsfeed is populated with the most relevant posts first. Facebook and Twitter both implement state-of-the-art LTR in their social networks.

4. Amazon Delivery:

   A practical application outside of the digital world is LTR being used in Amazon delivery services to pinpoint the location of a delivery address by using data from previous deliveries, helping a future prospect of drone-based delivery.

5. Amazon OpenSearch:

   Amazon OpenSearch, hosted on AWS, offers a free open-source plugin to perform learning to rank based searches. It works on the foundation of XGBoost and Ranklib to help rank the queries based on relevance and create a strong model for a search engine. It relies on keywords being repeated to rank the options available

Dataset Description

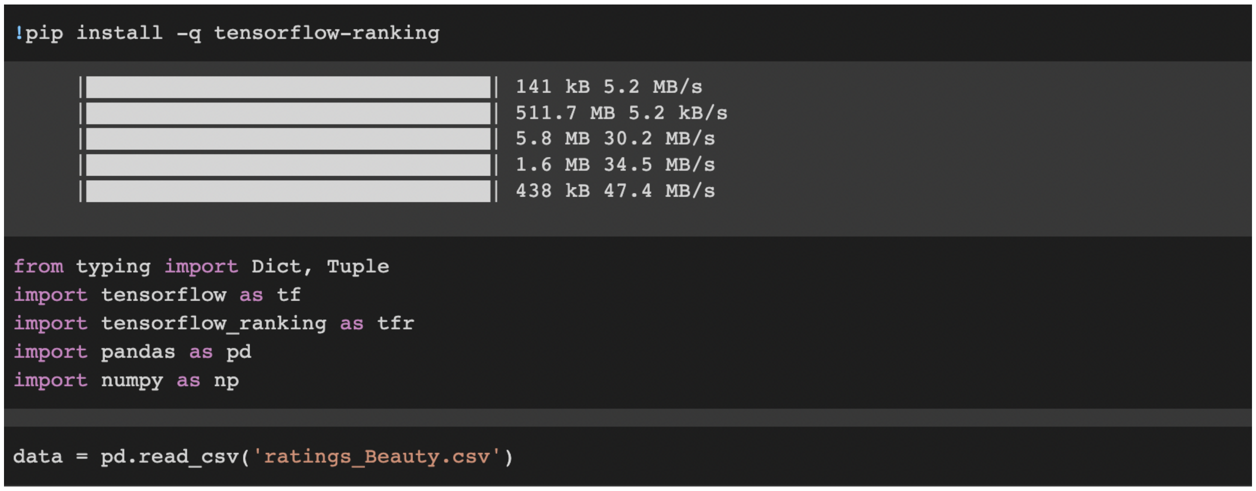
In this demo, the dataset being used is the MovieLens 100K, which is a standard benchmark dataset containing 1,00,000 ratings of 1700 movies from 1000 unique users. Each rating ranges from 1-5 and is the parameter that will be used as a label to help determine the ranking.

Dataset Preprocessing

Since the dataset being used here is a standardized tensorflow dataset, it is ready to be used. However, if a custom dataset needs to be used, these steps can help prepare it for an LTR algorithm.

Here, as an example, a CSV file consisting of reviews from Amazon Beauty is preprocessed and converted to a standard tensorflow dataset.

1. The standard libraries for our tensorflow model are imported, along with pandas and numpy. The CSV file is then read and stored in a data variable.



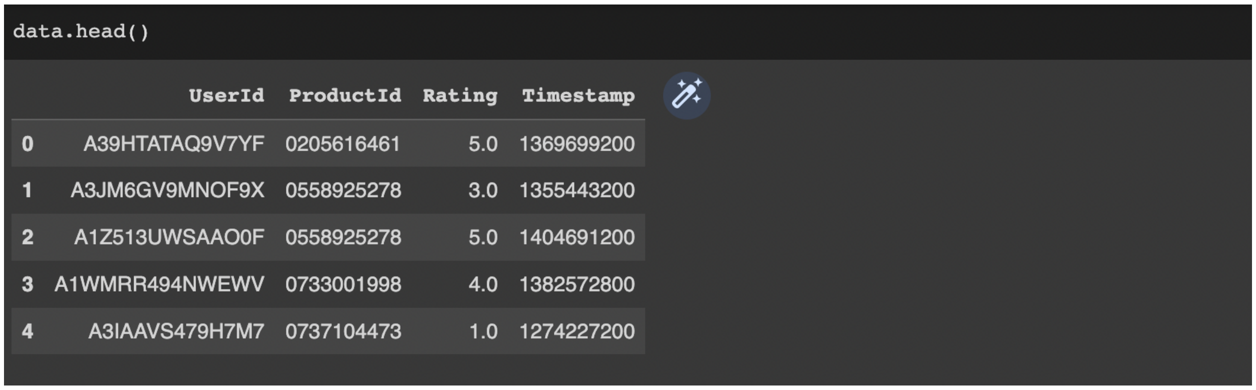
```
!pip install -q tensorflow-ranking
```

```
     |████████████████████████████████| 141 kB 5.2 MB/s
     |████████████████████████████████| 511.7 MB 5.2 kB/s
     |████████████████████████████████| 5.8 MB 30.2 MB/s
     |████████████████████████████████| 1.6 MB 34.5 MB/s
     |████████████████████████████████| 438 kB 47.4 MB/s
```

```python
from typing import Dict, Tuple
import tensorflow as tf
import tensorflow_ranking as tfr
import pandas as pd
import numpy as np
```

```python
data = pd.read_csv('ratings_Beauty.csv')
```

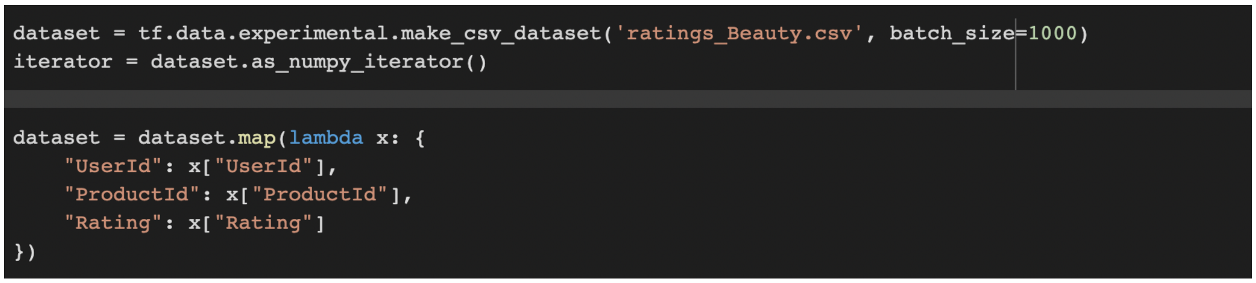
2. The first few entries of the data are displayed and verified.



```
data.head()
```

	UserId	ProductId	Rating	Timestamp
0	A39HTATAQ9V7YF	0205616461	5.0	1369699200
1	A3JM6GV9MNOF9X	0558925278	3.0	1355443200
2	A1Z513UWSAAO0F	0558925278	5.0	1404691200
3	A1WMRR494NWEWV	0733001998	4.0	1382572800
4	A3IAAVS479H7M7	0737104473	1.0	1274227200

3. The data is then converted to a tensorflow friendly dataset format using the make_csv_dataset functionality. An iterator is also created to help navigate the newly formed dataset. Now the dataset can be utilized in an LTR model in tensorflow. The relevant fields are then selected as User ID, Product ID and Rating, and the algorithm can be initiated.



```
dataset = tf.data.experimental.make_csv_dataset('ratings_Beauty.csv', batch_size=1000)
iterator = dataset.as_numpy_iterator()
```

```
dataset = dataset.map(lambda x: {
    "UserId": x["UserId"],
    "ProductId": x["ProductId"],
    "Rating": x["Rating"]
})
```

Metrics for an LTR model

Since LTR models do not have a numerical score to rank objects and instead use relevance to sort them from most to least relevant, there needs to be a unique way of determining the accuracy of a model. This is generally done using distance metric, which is calculated as the number of items between the position of the predicted result and the actual expected result. This does have a flaw, as the distance is just a number and does not account for the direction of the query, making it difficult to discern if the items are above their expected position or below the expected position. This is solved by using NDCG (Normalized Discounted Cumulative Gain), a metric that factors in distance and helps identify the direction by weighting those results above higher than those below. An NDCG value can range from 0 to 1, and higher values indicate better results.

Code Breakdown

- First, the tensorflow ranking and tensorflow datasets packages need to be installed.

```
!pip install -q tensorflow-ranking
!pip install -q --upgrade tensorflow-datasets
```

- The packages are then imported, along with the standard typing functions.

```
from typing import Dict, Tuple

import tensorflow as tf

import tensorflow_datasets as tfds
import tensorflow_ranking as tfr
```

- Then the data from the MovieLens 100K standard datasets is read and stored into two datasets- ratings and movies.

```
%%capture --no-display
# Ratings
ratings = tfds.load('movielens/100k-ratings', split="train")
# Features
movies = tfds.load('movielens/100k-movies', split="train")
```

- The features needed are then selected as user-id, movie-title and rating. Here, the rating is used as the label as that will be the feature used to rank the lists.

```

# Select the basic features
ratings = ratings.map(lambda x: {
    "movie_title": x["movie_title"],
    "user_id": x["user_id"],
    "user_rating": x["user_rating"]
})

```

- Once the features are extracted, they need to be converted into integer indices to help the algorithm process them and ultimately rank them. This is done by building a vocabulary for both user-ids and movie-titles, as a list of the unique values in each are needed to pair them using the labels.

```
movies = movies.map(lambda x: x["movie_title"])
users = ratings.map(lambda x: x["user_id"])

user_ids_vocabulary = tf.keras.layers.experimental.preprocessing.StringLookup(
    mask_token=None)
user_ids_vocabulary.adapt(users.batch(1000))

movie_titles_vocabulary = tf.keras.layers.experimental.preprocessing.StringLookup(
    mask_token=None)
movie_titles_vocabulary.adapt(movies.batch(1000))
```

- The unique user-id vocabulary can then be used to group the data into lists.

```
key_func = lambda x: user_ids_vocabulary(x["user_id"])
reduce_func = lambda key, dataset: dataset.batch(100)
ds_train = ratings.group_by_window(
    key_func=key_func, reduce_func=reduce_func, window_size=100)
```

- For example, the items in the first list are shown below.

```
for x in ds_train.take(1):
  for key, value in x.items():
    print(f"Shape of {key}: {value.shape}")
    print(f"Example values of {key}: {value[:5].numpy()}")
    print()
```

```
Shape of movie_title: (100,)
Example values of movie_title: [b'Man Who Would Be King, The (1975)' b'Silence of the Lambs, The (1991)'
 b'Next Karate Kid, The (1994)' b'2001: A Space Odyssey (1968)'
 b'Usual Suspects, The (1995)']

Shape of user_id: (100,)
Example values of user_id: [b'405' b'405' b'405' b'405' b'405']

Shape of user_rating: (100,)
Example values of user_rating: [1. 4. 1. 5. 5.]
```

- The features and labels are then batched to aid the data processing. A batch refers to the number of lists fed through the model at once.

```
def _features_and_labels(
    x: Dict[str, tf.Tensor]) -> Tuple[Dict[str, tf.Tensor], tf.Tensor]:
  labels = x.pop("user_rating")
  return x, labels


ds_train = ds_train.map(_features_and_labels)

ds_train = ds_train.apply(
    tf.data.experimental.dense_to_ragged_batch(batch_size=32))
```

- These batches are of size 32 for both user-ids and movie-titles, in lists of 100 except for when 100 items are not available in the list.

```
for x, label in ds_train.take(1):
  for key, value in x.items():
    print(f"Shape of {key}: {value.shape}")
    print(f"Example values of {key}: {value[:3, :3].numpy()}")
    print()
  print(f"Shape of label: {label.shape}")
  print(f"Example values of label: {label[:3, :3].numpy()}")
```

```
Shape of movie_title: (32, None)
Example values of movie_title: [[b'Man Who Would Be King, The (1975)'
  b'Silence of the Lambs, The (1991)' b'Next Karate Kid, The (1994)']
 [b'Flower of My Secret, The (Flor de mi secreto, La) (1995)'
  b'Little Princess, The (1939)' b'Time to Kill, A (1996)']
 [b'Kundun (1997)' b'Scream (1996)' b'Power 98 (1995)']]

Shape of user_id: (32, None)
Example values of user_id: [[b'405' b'405' b'405']
 [b'655' b'655' b'655']
 [b'13' b'13' b'13']]

Shape of label: (32, None)
Example values of label: [[1. 4. 1.]
 [3. 3. 3.]
 [5. 1. 1.]]
```

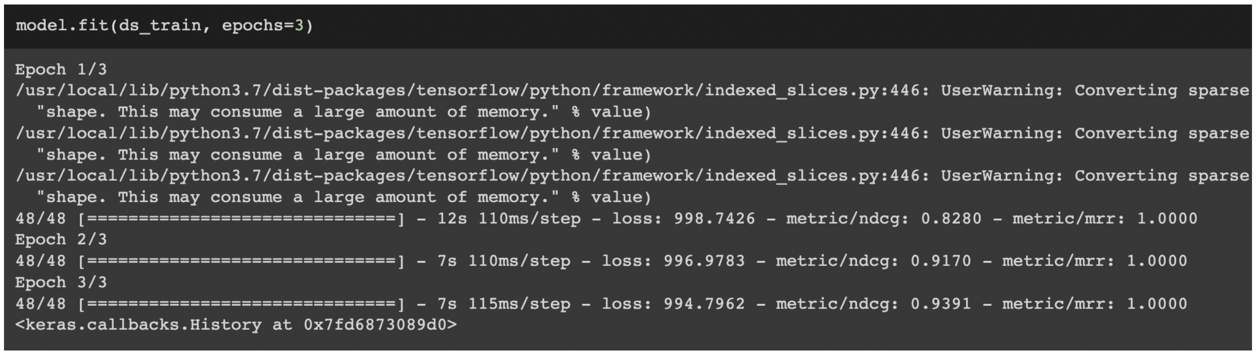
- A standard Keras model is then to be constructed, along with the call method. A dot product of the user and movie integers indices is returned for each iteration, to determine the gradient of ranking.

```
class MovieLensRankingModel(tf.keras.Model):

  def __init__(self, user_vocab, movie_vocab):
    super().__init__()

    # Set up user and movie vocabulary and embedding.
    self.user_vocab = user_vocab
    self.movie_vocab = movie_vocab
    self.user_embed = tf.keras.layers.Embedding(user_vocab.vocabulary_size(),
                                                64)
    self.movie_embed = tf.keras.layers.Embedding(movie_vocab.vocabulary_size(),
                                                 64)

  def call(self, features: Dict[str, tf.Tensor]) -> tf.Tensor:
    # Define how the ranking scores are computed:
    # Take the dot-product of the user embeddings with the movie embeddings.

    user_embeddings = self.user_embed(self.user_vocab(features["user_id"]))
    movie_embeddings = self.movie_embed(
        self.movie_vocab(features["movie_title"]))

    return tf.reduce_sum(user_embeddings * movie_embeddings, axis=2)
```

- The model is then finished, using SoftMax loss and standard LTR evaluation metrics such as NDCG (Net Discounted Cumulative Gain) and MRR (Mean Reciprocal Rank).

```
model = MovieLensRankingModel(user_ids_vocabulary, movie_titles_vocabulary)
optimizer = tf.keras.optimizers.Adagrad(0.5)
loss = tfr.keras.losses.get(
    loss=tfr.keras.losses.RankingLossKey.SOFTMAX_LOSS, ragged=True)
eval_metrics = [
    tfr.keras.metrics.get(key="ndcg", name="metric/ndcg", ragged=True),
    tfr.keras.metrics.get(key="mrr", name="metric/mrr", ragged=True)
]
model.compile(optimizer=optimizer, loss=loss, metrics=eval_metrics)
```

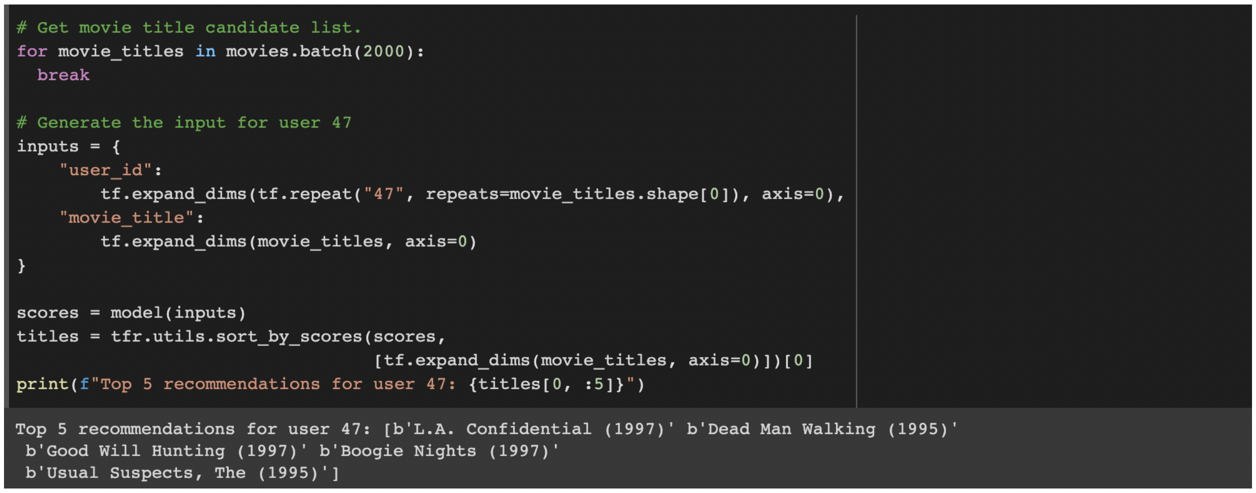
- The model is then fitted over 3 epochs and the metrics are obtained. As previously covered, NDCG is the key metric used in a listwise ranking model and values obtained here are close to 1, indicating that the model has strong predictions.



```
model.fit(ds_train, epochs=3)
```

```
Epoch 1/3
/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/indexed_slices.py:446: UserWarning: Converting sparse
  "shape. This may consume a large amount of memory." % value)
/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/indexed_slices.py:446: UserWarning: Converting sparse
  "shape. This may consume a large amount of memory." % value)
/usr/local/lib/python3.7/dist-packages/tensorflow/python/framework/indexed_slices.py:446: UserWarning: Converting sparse
  "shape. This may consume a large amount of memory." % value)
48/48 [==============================] - 12s 110ms/step - loss: 998.7426 - metric/ndcg: 0.8280 - metric/mrr: 1.0000
Epoch 2/3
48/48 [==============================] - 7s 110ms/step - loss: 996.9783 - metric/ndcg: 0.9170 - metric/mrr: 1.0000
Epoch 3/3
48/48 [==============================] - 7s 115ms/step - loss: 994.7962 - metric/ndcg: 0.9391 - metric/mrr: 1.0000
<keras.callbacks.History at 0x7fd6873089d0>
```

- The model is finally tested for predictions and attempts to return movie recommendations based on the ranking which it learns from both the user's ratings and other similar users' ratings.



```
# Get movie title candidate list.
for movie_titles in movies.batch(2000):
  break

# Generate the input for user 47
inputs = {
    "user_id":
        tf.expand_dims(tf.repeat("47", repeats=movie_titles.shape[0]), axis=0),
    "movie_title":
        tf.expand_dims(movie_titles, axis=0)
}

scores = model(inputs)
titles = tfr.utils.sort_by_scores(scores,
                                  [tf.expand_dims(movie_titles, axis=0)])[0]
print(f"Top 5 recommendations for user 47: {titles[0, :5]}")
```

```
Top 5 recommendations for user 47: [b'L.A. Confidential (1997)' b'Dead Man Walking (1995)'
 b'Good Will Hunting (1997)' b'Boogie Nights (1997)'
 b'Usual Suspects, The (1995)']
```

Conclusion

As it can be seen, Learning to Rank can be a swift and efficient technique to optimize searches and a whole host of processes across various domains. Its applications are plentiful and on the rise, and its evaluation metrics such as NDCG are a smarter alternative to its contemporaries, making it one to keep an eye on.

References

1. https://towardsdatascience.com/learning-to-rank-a-complete-guide-to-ranking-using-machine-learning-4c9688d370d4
2. https://practicaldatascience.co.uk/machine-learning/a-quick-guide-to-learning-to-rank-models
3. Dong, Xishuang & Chen, Xiaodong & Guan, Yi & Yu, Zhiming & Li, Sheng. (2009). An Overview of Learning to Rank for Information Retrieval. 2009 WRI World Congress on Computer Science and Information Engineering, CSIE 2009. 3. 600–606. 10.1109/CSIE.2009.1090.
4. https://www.tensorflow.org/ranking/overview
5. https://grouplens.org/datasets/movielens/100k/
6. https://lucidworks.com/post/abcs-learning-to-rank/