Assignment 4: Learning to Rank

In this assignment you will implement a pointwise, pairwise and listwise Learning to Rank (LTR) losses to optimize three ranking models and evaluate their differences in their performance on the Yahoo Webscope LTR dataset, read more about the dataset here.

In [1]:

```
!wget http://gem.cs.ru.nl/IR-Course-2021-2022/dataset.py
!wget http://gem.cs.ru.nl/IR-Course-2021-2022/evaluate.py
!wget http://gem.cs.ru.nl/IR-Course-2021-2022/ranking.py
!wget http://gem.cs.ru.nl/IR-Course-2021-2022/LTR-dataset.npz
```

```
--2021-10-11 10:19:25--  http://gem.cs.ru.nl/IR-Course-2021-2022/dataset.py
Resolving gem.cs.ru.nl (gem.cs.ru.nl)... 131.174.31.31
Connecting to gem.cs.ru.nl (gem.cs.ru.nl)|131.174.31.31|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4688 (4.6K) [text/x-python]
Saving to: ‘dataset.py.1’

dataset.py.1        100%[===================>]   4.58K  --.-KB/s    in 0s

2021-10-11 10:19:26 (374 MB/s) - ‘dataset.py.1’ saved [4688/4688]

--2021-10-11 10:19:26--  http://gem.cs.ru.nl/IR-Course-2021-2022/evaluate.py
Resolving gem.cs.ru.nl (gem.cs.ru.nl)... 131.174.31.31
Connecting to gem.cs.ru.nl (gem.cs.ru.nl)|131.174.31.31|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4067 (4.0K) [text/x-python]
Saving to: ‘evaluate.py.1’

evaluate.py.1       100%[===================>]   3.97K  --.-KB/s    in 0s

2021-10-11 10:19:26 (439 MB/s) - ‘evaluate.py.1’ saved [4067/4067]

--2021-10-11 10:19:26--  http://gem.cs.ru.nl/IR-Course-2021-2022/ranking.py
Resolving gem.cs.ru.nl (gem.cs.ru.nl)... 131.174.31.31
Connecting to gem.cs.ru.nl (gem.cs.ru.nl)|131.174.31.31|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 833 [text/x-python]
Saving to: ‘ranking.py.1’

ranking.py.1        100%[===================>]     833  --.-KB/s    in 0s

2021-10-11 10:19:26 (112 MB/s) - ‘ranking.py.1’ saved [833/833]

--2021-10-11 10:19:26--  http://gem.cs.ru.nl/IR-Course-2021-2022/LTR-dataset.npz
Resolving gem.cs.ru.nl (gem.cs.ru.nl)... 131.174.31.31
Connecting to gem.cs.ru.nl (gem.cs.ru.nl)|131.174.31.31|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 634123360 (605M)
Saving to: ‘LTR-dataset.npz.1’

LTR-dataset.npz.1   100%[===================>] 604.75M  32.6MB/s    in 19s

2021-10-11 10:19:45 (31.9 MB/s) - ‘LTR-dataset.npz.1’ saved [634123360/634123360]
```

Dataset Setup

First we will import the numpy and tensorflow packages and some additional files that will help us with loading data, ranking and evaluation.

In [2]:

```
import numpy as np
import tensorflow as tf
import dataset
import ranking as rnk
import evaluate as evl
```

The dataset has been pre-processed for you, we will start by loading it with our custom dataset package.

In [3]:

```
data = dataset.load(path='LTR-dataset.npz')

print('Data has been loaded in, here are some statistics:')
print('Number of features: %d' % data.num_features)
print('Number of queries in training set: %d' % data.train.num_queries())
print('Number of documents in training set: %d' % data.train.num_docs())
print('Number of queries in validation set: %d' % data.validation.num_queries())
print('Number of documents in validation set: %d' % data.validation.num_docs())
```

```
Data has been loaded in, here are some statistics:
Number of features: 501
Number of queries in training set: 19943
Number of documents in training set: 473092
Number of queries in validation set: 2993
Number of documents in validation set: 71041
```

The dataset has a preselected set of documents per query that need to be ranked, so instead of ranking all documents per query, we only need to rank a smaller selection.

The dataset class keeps track of all query-document combinations for us. Let's take a closer look at how it works.

To start we will select a random query, in this case we use the query in the training set with the ID:

In [4]:

```
qid = 12
```

To start, we can look at how many documents are preselected for this training query and the relevance labels for each document w.r.t. the query:

In [5]:

```
query_n_docs = data.train.query_size(qid)
query_labels = data.train.query_labels(qid)

print('The %d documents have been preselected for query %d.' % (query_n_docs, qid))
print('These are the labels for the documents:')
print(query_labels)
```

```
The 16 documents have been preselected for query 12.
These are the labels for the documents:
[2 2 3 2 1 2 1 2 2 1 2 1 2 2 1 2]
```

The dataset has pre-computed many relevance signals as features, some of the features are TF-IDF, BM-25, PageRank, Text-Matching, Document Classification, etc. See the publication for a more extensive list.

The dataset class can retrieve the feature vectors for all the documents of a query at once:

In [6]:

```
query_features = data.train.query_feat(qid)

print('The document features for query %d:' % qid)
print(query_features)
```

```
The document features for query 12:
[[1.         0.         1.         ... 0.         0.97737896 0.05315655]
```

```
[1.         0.         0.         ... 0.         0.9510745  0.19329825]
[1.         0.         0.97065877 ... 0.         1.         0.27863213]
...
[1.         0.         0.         ... 0.         0.6194661  0.53677223]
[1.         0.         0.         ... 0.         0.6194661  0.79946817]
[1.         0.         0.         ... 0.         0.         0.07183368]]
```

The label vector and the feature matrix are aligned in the first dimension, for example, we can get the label and the 501 features for a single document like this:

In [7]:

```
doc_id = 0

print('Document %d for query %d has the relevance label %d and the features:' % (doc_id,
qid, query_labels[doc_id]))
print(query_features[doc_id,:])
```

```
Document 0 for query 12 has the relevance label 2 and the features:
[1.         0.         1.         0.         0.90329384 1.
 0.         1.         0.95928169 1.         1.         0.90396066
 1.         0.         0.         0.         1.         0.
 0.39953607 0.94563438 1.         0.         0.92551186 0.12434507
 0.23218051 1.         0.89486675 1.         0.99601268 0.
 1.         0.         0.         0.         0.         0.
 1.         0.         1.         0.78885573 0.         0.33577561
 0.74456986 1.         0.         0.         0.98197495 0.96996409
 0.95190763 0.         0.52807419 0.         1.         0.92661817
 1.         0.         0.         0.         0.1390352  0.8998157
 0.         1.         0.         1.         1.         0.
 0.         1.         0.         0.24563449 0.         0.99559206
 0.61638237 1.         0.83498663 0.         0.96056869 0.23648585
 0.53835469 0.         1.         1.         0.         1.
 0.         0.         0.62672411 0.96934613 0.94563438 0.
 1.         0.56607304 0.         0.         0.         0.97711957
 1.         1.         0.99776056 0.78172178 0.66605695 0.97306764
 0.69327295 0.90769632 0.         0.         1.         1.
 0.         0.         1.         0.         1.         1.
 0.         0.07913779 0.         1.         1.         0.
 0.         0.         1.         1.         0.         0.
 0.         0.         0.96986775 0.25024912 1.         0.
 0.         0.         0.         1.         0.94961925 0.
 0.         1.         0.57132119 0.         0.         1.
 0.         0.84848709 0.4707879  0.81120368 1.         1.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.85442756 0.
 0.         1.         0.         1.         0.         0.
 0.20092322 0.15990543 0.85411569 0.         1.         0.
 1.         0.71021152 0.         0.40775753 0.         1.
 1.         0.         1.         0.83644548 1.         1.
 1.         0.53752993 1.         0.83137126 0.         0.
 1.         0.         1.         0.         1.         1.
 1.         1.         0.         0.         0.         0.
 0.61638237 0.         1.         0.         0.         1.
 0.         0.8910248  0.78409556 0.         1.         0.63198261
 0.36060694 0.         0.77601958 0.         0.88999477 0.
 0.         0.27959126 0.         0.         1.         0.
 0.         0.         0.         0.7945365  0.27961714 0.
 0.48532734 0.         1.         0.76134276 0.         0.
 0.84955632 0.94020222 0.37828575 0.86705101 0.         1.
 0.         1.         0.         1.         0.97469324 0.60551243
 0.         0.27195386 0.         1.         0.9938965  0.7210379
 0.         1.         0.         0.         0.43168281 0.
 0.         0.7626422  0.96991017 0.99246343 0.         0.36060694
 0.04009782 0.         0.         1.         1.         0.
 1.         0.         0.         0.         0.         0.
 0.88030146 0.63048792 0.         1.         0.94765115 0.99584079
 1.         0.         0.35279846 0.         1.         0.89039216
 1.         0.         0.         0.         0.         0.
 0.82124906 0.         0.88575408 0.99630849 0.9690716  0.
 0.         0.31276796 1.         1.         0.9985694  0.
 0.19792167 0.         1.         0.86343279 0.9312308  1.
```

```
0.19792167 0.         1.         0.86545279 0.9512306  1.
 0.         0.82932866 0.72232381 0.         0.         1.
 0.         0.         0.88999477 0.97853448 0.         0.66993831
 0.94852236 0.         0.         0.         0.         0.
 0.84102558 0.         0.86492186 1.         1.         1.
 1.         0.9514758  1.         1.         1.         1.
 0.         0.         0.4612897  1.         0.4612897  0.
 1.         0.80629605 0.98889557 0.         0.         0.67486017
 1.         0.         0.79903623 1.         0.         1.
 1.         1.         0.         1.         0.         0.27988587
 0.         0.         0.         0.         0.         0.
 0.         1.         0.99741824 0.         1.         0.
 1.         1.         1.         0.         1.         0.
 0.9753476  0.         0.         0.81088711 0.         1.
 0.         0.         1.         1.         0.         0.
 0.         0.         1.         0.         0.         1.
 1.         0.         0.80946441 0.         0.         1.
 1.         0.         1.         0.80629605 0.         0.
 0.         0.36507297 1.         0.         1.         1.
 0.69564837 0.7528325  0.14759495 1.         0.93622618 0.
 1.         0.         0.         0.9822453  0.         1.
 1.         0.91060281 0.         0.83644548 0.         0.
 0.         0.8699614  0.         1.         0.28004686 0.
 0.         0.         0.92990789 0.47059449 0.         0.24497688
 0.98611167 0.45899814 1.         1.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.5643054  0.         0.         0.         1.         1.
 0.         0.7503118  0.         0.         1.         0.
 1.         0.         0.         0.         1.         0.
 0.         0.         0.11782658 0.         0.         1.
 1.         0.         1.         1.         1.         0.
 0.         0.97737896 0.05315655]
```

Having the features in matrix form allows us to quickly apply models to all documents at once.

Let's initialize a random linear model and apply it to all documents for the query.
A linear model is captured by a vector of weights and it can be applied via a dot product:

In [8]:

```
# initialize a random linear model
random_model = np.random.uniform(low=-1, high=1, size=data.num_features)

# score all documents for the query at once
query_scores = np.dot(query_features, random_model)

print('The predicted scores for the documents of query %d:' % qid)
print(query_scores)
```

```
The predicted scores for the documents of query 12:
[-3.46778441 -4.46054814 -8.49003054 -0.81630148 -9.53120973 -4.65187424
 -9.09207128 -4.82369679 -5.1688116  -1.86487585 -5.433439   -5.89330252
 -3.71481049 -6.2722189  -7.26054323 -6.56617617]
```

For this assignment, we recommend you use the ranking module to create rankings out of scores.
(The difference with argsort is that the ranking module randomly breaks score ties.)

The following creates a ranking for a single query:

In [9]:

```
(query_ranking, query_inverted_ranking) = rnk.rank_and_invert(query_scores)

print('The ranking for query %d:' % qid)
print(query_ranking)
```

```
The ranking for query 12:
[ 3  9  0 12  1  5  7  8 10 11 13 15 14  2  6  4]
```

To get an idea of the quality of the ranking, let's look at how it ranks the labels:

In [10]:

```
print('The ranked labels for query %d are:' % qid)
print(query_labels[query_ranking])
```

```
The ranked labels for query 12 are:
[2 1 2 2 2 2 2 2 1 2 2 1 3 1 1]
```

Furthermore, the ranking module also provides the inverted ranking, this provides a quick way to look up the rank of a document from its ID:

In [11]:

```
print('The inverted ranking for query %d:' % qid)
print(query_inverted_ranking)
print('Document %d for query %d is placed at rank %d.' % (doc_id, qid, query_inverted_ra
nking[doc_id]))
```

```
The inverted ranking for query 12:
[ 2  4 13  0 15  5 14  6  7  1  8  9  3 10 12 11]
Document 0 for query 12 is placed at rank 2.
```

Evaluation

Finally, for this assignment there is a light-weight implementation of evaluation metrics that works with the dataset class.

As input it takes the dataset partition (e.g. the validation set) and a vector of predicted scores:

In [12]:

```
# score every document in the validation set, results in a vector with a score for each d
ocument
all_vali_scores = np.dot(data.validation.feature_matrix, random_model)

print('Evaluation on the validation partition:')
results = evl.evaluate(data.validation, all_vali_scores, print_results=True)
```

```
Evaluation on the validation partition:
dcg: 15.1444 (11.45161)
dcg@03: 3.8307 (3.61337)
dcg@05: 5.4435 (4.49589)
dcg@10: 8.2662 (5.93075)
dcg@20: 11.2307 (7.52914)
ndcg: 0.6924 (0.16223)
ndcg@03: 0.3694 (0.29174)
ndcg@05: 0.4237 (0.28074)
ndcg@10: 0.5170 (0.26859)
ndcg@20: 0.6003 (0.23003)
precision@01: 0.1173 (0.32182)
precision@03: 0.1277 (0.21411)
precision@05: 0.1352 (0.18037)
precision@10: 0.1345 (0.14066)
precision@20: 0.1125 (0.09813)
recall@01: 0.0422 (0.15312)
recall@03: 0.1377 (0.27359)
recall@05: 0.2436 (0.35202)
recall@10: 0.4551 (0.42100)
recall@20: 0.6843 (0.37866)
relevant rank: 21.6389 (17.28658)
relevant rank per query: 86.7326 (140.49700)
```

Models in Tensorflow

For this assignment you will optimize three models with identical architectures. The following function will initialize a neural network with two hidden layers (30 units each) that outputs an unbounded score prediction:

In [13]:

```
def init_model(learning_rate=0.1, hidden_layer_units=[30, 30]):
    optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate)
    layers = [tf.keras.layers.Dense(x, activation='sigmoid', dtype=tf.float64)
              for x in hidden_layer_units]
    layers.append(tf.keras.layers.Dense(1, activation=None, dtype=tf.float64))
    nn_model = tf.keras.Sequential(layers)
    return nn_model, optimizer
```

Predictions can be obtained from the model by calling it as a function with a feature matrix as the input.

Let's initialize a model and see the scores it predicts:

In [14]:

```
random_nn_model, _ = init_model()

print('Predicted scores by a random neural network for query %d:' % qid)
print(random_nn_model(query_features).numpy()[:,0])
```

```
Predicted scores by a random neural network for query 12:
[-0.00361282 -0.0581304  -0.04898529 -0.05391738 -0.07033061 -0.02633059
 -0.05917993 -0.0527478  -0.07254753 -0.09656442 -0.05517618 -0.02558689
 -0.04136196 -0.081109   -0.0712976  -0.07019205]
```

Note that the predicted scores are tensorflow *tensors* which can be converted to numpy arrays with the *.numpy(* function.

The evaluation model also has a function for easily evaluating a tensorflow model:

In [15]:

```
print('Evaluation on the validation partition:')
results = evl.evaluate_tf_model(data.validation, random_nn_model, print_results=True)
```

```
Evaluation on the validation partition:
dcg: 16.5976 (12.71995)
dcg@03: 5.5845 (5.18258)
dcg@05: 7.4429 (5.99711)
dcg@10: 10.3750 (7.42099)
dcg@20: 13.2449 (9.15409)
ndcg: 0.7405 (0.15704)
ndcg@03: 0.4709 (0.29550)
ndcg@05: 0.5185 (0.26907)
ndcg@10: 0.5989 (0.24777)
ndcg@20: 0.6669 (0.20996)
precision@01: 0.2329 (0.42269)
precision@03: 0.2331 (0.26859)
precision@05: 0.2130 (0.20716)
precision@10: 0.1781 (0.15099)
precision@20: 0.1335 (0.10748)
recall@01: 0.0840 (0.20842)
recall@03: 0.2569 (0.34609)
recall@05: 0.3780 (0.38602)
recall@10: 0.5834 (0.39611)
recall@20: 0.7795 (0.31780)
relevant rank: 17.5411 (15.95159)
relevant rank per query: 70.3076 (120.00249)
```

4.1 Pointwise Learning to Rank

There first loss to implement is the pointwise Mean-Squared-Error (MSE) regression loss:

$$\mathcal{L} = \frac{1}{|Q|} \sum_{i=1}^{|Q|}$$

$$\sum_{q \in Q} \frac{1}{|D_q|}$$
$$\sum_{d \in D_q} (\hat{y}_d - y_d)^2$$

.

Here Q denotes the set of queries, D_q is the set of preselected documents for query q and $\hat{y}_d$ is the predicted score for document d whereas y_d is the true label.

Below we provide you with a scaffolding that optimizes a ranking model with a nonsense loss, your task is to replace this loss with the MSE pointwise loss.

The following tutorial may be helpful with the Tensorflow aspects for this assignment, this tutorial further explains how gradients can computed with Tensorflow.

A correct implementation will reach a performance of 0.750 NDCG@10 or higher.

In [16]:

```
pointwise_model, pointwise_optimizer = init_model()
results = evl.evaluate_tf_model(data.validation, pointwise_model)
print('Epoch 0 - NDCG@10: %0.03f' % results['ndcg@10'][0])

pointwise_losses = []
pointwise_dcg = []
for epoch in range(15):
    loss_list = []
    for qid in np.random.permutation(data.train.num_queries()):
        query_labels = data.train.query_labels(qid)
        query_features = data.train.query_feat(qid)
        with tf.GradientTape() as tape:
            query_tf_scores = pointwise_model(query_features)

            # Calculate MSE loss; reduce_mean is over documents
            loss = tf.reduce_mean(tf.square(query_tf_scores[:,0] - query_labels))

            loss_list.append(loss.numpy())

        gradients = tape.gradient(loss, pointwise_model.trainable_variables)
        pointwise_optimizer.apply_gradients(zip(gradients, pointwise_model.trainable_vari
ables))

    average_loss = np.mean(loss_list)
    results = evl.evaluate_tf_model(data.validation, pointwise_model)
    print('Epoch %d - Loss %0.03f - NDCG@10: %0.03f' % (epoch+1, average_loss, results['
ndcg@10'][0]))

    pointwise_losses.append(average_loss)
    pointwise_dcg.append(results['ndcg@10'][0])

print('Final results:')
results = evl.evaluate_tf_model(data.validation, pointwise_model, print_results=True)
```

```
Epoch 0 - NDCG@10: 0.653
Epoch 1 - Loss 0.727 - NDCG@10: 0.742
Epoch 2 - Loss 0.691 - NDCG@10: 0.746
Epoch 3 - Loss 0.681 - NDCG@10: 0.748
Epoch 4 - Loss 0.672 - NDCG@10: 0.750
Epoch 5 - Loss 0.668 - NDCG@10: 0.750
Epoch 6 - Loss 0.661 - NDCG@10: 0.753
Epoch 7 - Loss 0.658 - NDCG@10: 0.750
Epoch 8 - Loss 0.654 - NDCG@10: 0.751
Epoch 9 - Loss 0.651 - NDCG@10: 0.753
Epoch 10 - Loss 0.648 - NDCG@10: 0.753
Epoch 11 - Loss 0.644 - NDCG@10: 0.751
Epoch 12 - Loss 0.643 - NDCG@10: 0.752
Epoch 13 - Loss 0.640 - NDCG@10: 0.751
Epoch 14 - Loss 0.638 - NDCG@10: 0.753
Epoch 15 - Loss 0.637 - NDCG@10: 0.753
Final results:
dcg: 19.4085 (14.99675)
dcg@03: 9.1179 (7.64289)
```

```
dcg@05: 11.1229 (8.62302)
dcg@10: 14.0955 (10.17326)
dcg@20: 16.7695 (12.05438)
ndcg: 0.8449 (0.14301)
ndcg@03: 0.6786 (0.28212)
ndcg@05: 0.7021 (0.24847)
ndcg@10: 0.7530 (0.21433)
ndcg@20: 0.7980 (0.18005)
precision@01: 0.6316 (0.48236)
precision@03: 0.4392 (0.31371)
precision@05: 0.3496 (0.25086)
precision@10: 0.2464 (0.18251)
precision@20: 0.1586 (0.12789)
recall@01: 0.2697 (0.32482)
recall@03: 0.4715 (0.37470)
recall@05: 0.5782 (0.37157)
recall@10: 0.7438 (0.32815)
recall@20: 0.8806 (0.23646)
relevant rank: 12.3217 (13.74442)
relevant rank per query: 49.3876 (94.93966)
```

4.2 Pairwise Learning to Rank

Implement the pairwise loss from the [LambdaLoss paper]:

$$\mathcal{L} = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{|\{y_{d_i} > y_{d_j} \in D_q\}|} \sum_{y_{d_i} > y_{d_j} \in D_q} \log(1 + e^{\hat{y}_{d_j} - \hat{y}_{d_i}}).$$

The $y_{d_i} > y_{d_j}$ indicates that the label of document i is greater than j, i.e. d_i is more relevant than d_j, and $\sum_{y_{d_i} > y_{d_j} \in D_q}$ sums over all document pairs with unequal labels from the preselected documents for query q.

Note: You are allowed to make changes outside of the demarcated area.

Hint: broadcasting can speed up your implementation.

A correct implementation will reach a performance of 0.750 NDCG@10 or higher.

In [18]:

```
pairwise_model, pairwise_optimizer = init_model()
results = evl.evaluate_tf_model(data.validation, pairwise_model)
print('Epoch 0 - NDCG@10: %0.03f' % results['ndcg@10'][0])

pairwise_losses = []
pairwise_dcg = []
for epoch in range(15):
    loss_list = []
    for qid in np.random.permutation(data.train.num_queries()):
        n_docs = data.train.query_size(qid)

        # Moved this outside the if-function because we need them there
        query_labels = data.train.query_labels(qid)
        query_features = data.train.query_feat(qid)

        # Skip over documents where all elements are the same
        if False in tf.equal(query_labels, query_labels[0]):

            with tf.GradientTape() as tape:
                query_tf_scores = pairwise_model(query_features)
```

```
                ### YOUR LOSS CODE STARTS HERE
                l = len(query_labels)
                # Get all possible pairs
                x = tf.range(0, l)
                y = tf.zeros((l, l), dtype=tf.dtypes.int32)
                a = tf.reshape(x[None,:] + y, (l*l, 1))
                b = tf.reshape(x[:,None] + y, (l*l, 1))
                pc = tf.reshape(tf.stack([a, b], axis=1), (len(a), 2))

                # Filter out (a, a) pairs
                y = tf.gather(query_labels, pc)
                pairs = tf.gather(pc, tf.where(y[:,0] > y[:,1])[:,0])

                score1 = tf.transpose(tf.gather(query_tf_scores, pairs[:,0]))
                score2 = tf.transpose(tf.gather(query_tf_scores, pairs[:,1]))
                loss = tf.reduce_mean(tf.math.log(1 + tf.math.exp(score2 - score1)))
                #loss = 0
                #for i, j in pairs:
                #    loss += tf.math.log(1 + tf.math.exp(query_tf_scores[j] - query_tf_s
cores[i]))
                #loss = tf.reduce_mean(loss)

                ### YOUR LOSS CODE ENDS HERE
                loss_list.append(loss.numpy())

                gradients = tape.gradient(loss, pairwise_model.trainable_variables)
                pairwise_optimizer.apply_gradients(zip(gradients, pairwise_model.trainab
le_variables))

    average_loss = np.mean(loss_list)
    results = evl.evaluate_tf_model(data.validation, pairwise_model)
    print('Epoch %d - Loss %0.03f - NDCG@10: %0.03f' % (epoch+1, average_loss, results['
ndcg@10'][0]))

    pairwise_losses.append(average_loss)
    pairwise_dcg.append(results['ndcg@10'][0])

print('Final results:')
results = evl.evaluate_tf_model(data.validation, pairwise_model, print_results=True)
```

```
Epoch 0 - NDCG@10: 0.581
Epoch 1 - Loss 0.592 - NDCG@10: 0.748
Epoch 2 - Loss 0.578 - NDCG@10: 0.745
Epoch 3 - Loss 0.572 - NDCG@10: 0.749
Epoch 4 - Loss 0.568 - NDCG@10: 0.751
Epoch 5 - Loss 0.564 - NDCG@10: 0.748
Epoch 6 - Loss 0.562 - NDCG@10: 0.751
Epoch 7 - Loss 0.559 - NDCG@10: 0.751
Epoch 8 - Loss 0.557 - NDCG@10: 0.753
Epoch 9 - Loss 0.554 - NDCG@10: 0.753
Epoch 10 - Loss 0.552 - NDCG@10: 0.752
Epoch 11 - Loss 0.550 - NDCG@10: 0.751
Epoch 12 - Loss 0.548 - NDCG@10: 0.752
Epoch 13 - Loss 0.546 - NDCG@10: 0.753
Epoch 14 - Loss 0.544 - NDCG@10: 0.755
Epoch 15 - Loss 0.542 - NDCG@10: 0.753
Final results:
dcg: 19.3789 (14.97087)
dcg@03: 9.0761 (7.58552)
dcg@05: 11.0919 (8.53630)
dcg@10: 14.0620 (10.08665)
dcg@20: 16.7507 (12.02060)
ndcg: 0.8445 (0.14174)
ndcg@03: 0.6771 (0.28153)
ndcg@05: 0.7021 (0.24523)
ndcg@10: 0.7529 (0.21141)
ndcg@20: 0.7976 (0.17898)
precision@01: 0.6188 (0.48568)
precision@03: 0.4390 (0.31180)
precision@05: 0.3510 (0.25002)
precision@10: 0.2467 (0.18191)
precision@20: 0.1592 (0.12736)
```

```
recall@01: 0.2622 (0.32147)
recall@03: 0.4707 (0.37628)
recall@05: 0.5773 (0.36925)
recall@10: 0.7490 (0.32821)
recall@20: 0.8835 (0.23193)
relevant rank: 12.2552 (13.69640)
relevant rank per query: 49.1208 (95.97356)
```

4.3 Listwise Learning to Rank

Implement the listwise loss from the [original listwise LTR paper](), this paper models the probability of a ranking as a Plackett-Luce model (similar to a soft-max function). As the loss function we will use the negative log-likelihood of the optimal ranking. Let R_q denote the optimal ranking for query q, $R_q[i]$ as the ith document in the optimal ranking and $\hat{y}_{R_q[i]}$ as its predicted label, the loss is then:

$$\mathcal{L} = -\frac{1}{|Q|}\sum_{q \in Q} \sum_{i=1}^{|R_q|} \log \left(\frac{\exp\left(\hat{y}_{R_q[i]}\right)}{\sum_{j=i}^{|R_q|} \exp\left(\hat{y}_{R_q[j]}\right)} \right)$$

.

You can use the rank module to obtain the optimal ranking, the following code provides the optimal ranking for you.

A correct implementation will reach a performance of 0.750 NDCG@10 or higher.

In [22]:

```
# This loss has a larger average value, therefore a smaller learning_rate is probably a g
ood idea.
listwise_model, listwise_optimizer = init_model(learning_rate=0.01)
results = evl.evaluate_tf_model(data.validation, listwise_model)
print('Epoch 0 - NDCG@10: %0.03f' % results['ndcg@10'][0])

listwise_losses = []
listwise_dcg = []
for epoch in range(15):
    loss_list = []
    for qid in np.random.permutation(data.train.num_queries()):
        n_docs = data.train.query_size(qid)
        if n_docs > 1:
            query_labels = 2.**data.train.query_labels(qid)-1.
            query_features = data.train.query_feat(qid)
            optimal_ranking = rnk.rank_and_invert(query_labels)[0]
            with tf.GradientTape() as tape:
                query_tf_scores = listwise_model(query_features)

                ### YOUR LOSS CODE STARTS HERE
                upper = tf.math.exp(tf.gather(query_tf_scores, optimal_ranking))
                lower = tf.cumsum(tf.math.exp(tf.gather(query_tf_scores, optimal_ranking
)), reverse=True)
                loss = tf.reduce_sum(tf.math.log(upper / lower))*-1
                ### YOUR LOSS CODE ENDS HERE

                loss_list.append(loss.numpy())

            gradients = tape.gradient(loss, listwise_model.trainable_variables)
            listwise_optimizer.apply_gradients(zip(gradients, listwise_model.trainable_v
```

```
ariables))

    average_loss = np.mean(loss_list)
    results = evl.evaluate_tf_model(data.validation, listwise_model)
    print('Epoch %d - Loss %0.05f - NDCG@10: %0.03f' % (epoch+1, average_loss, results['
ndcg@10'][0]))

    listwise_losses.append(average_loss)
    listwise_dcg.append(results['ndcg@10'][0])

print('Final results:')
results = evl.evaluate_tf_model(data.validation, listwise_model, print_results=True)
```

```
Epoch 0 - NDCG@10: 0.511
Epoch 1 - Loss 59.53994 - NDCG@10: 0.741
Epoch 2 - Loss 59.39560 - NDCG@10: 0.744
Epoch 3 - Loss 59.32582 - NDCG@10: 0.747
Epoch 4 - Loss 59.29510 - NDCG@10: 0.750
Epoch 5 - Loss 59.25933 - NDCG@10: 0.752
Epoch 6 - Loss 59.21332 - NDCG@10: 0.748
Epoch 7 - Loss 59.17156 - NDCG@10: 0.748
Epoch 8 - Loss 59.16173 - NDCG@10: 0.750
Epoch 9 - Loss 59.13823 - NDCG@10: 0.747
Epoch 10 - Loss 59.12497 - NDCG@10: 0.750
Epoch 11 - Loss 59.11875 - NDCG@10: 0.751
Epoch 12 - Loss 59.09096 - NDCG@10: 0.758
Epoch 13 - Loss 59.06871 - NDCG@10: 0.756
Epoch 14 - Loss 59.05575 - NDCG@10: 0.758
Epoch 15 - Loss 59.03426 - NDCG@10: 0.757
Final results:
dcg: 19.4517 (15.06760)
dcg@03: 9.1304 (7.64713)
dcg@05: 11.1816 (8.67199)
dcg@10: 14.1668 (10.22365)
dcg@20: 16.8566 (12.16374)
ndcg: 0.8472 (0.14184)
ndcg@03: 0.6827 (0.28021)
ndcg@05: 0.7072 (0.24680)
ndcg@10: 0.7572 (0.21172)
ndcg@20: 0.8016 (0.17845)
precision@01: 0.6223 (0.48481)
precision@03: 0.4404 (0.31681)
precision@05: 0.3558 (0.25703)
precision@10: 0.2487 (0.18388)
precision@20: 0.1606 (0.13152)
recall@01: 0.2629 (0.32205)
recall@03: 0.4719 (0.37841)
recall@05: 0.5842 (0.37214)
recall@10: 0.7495 (0.32922)
recall@20: 0.8846 (0.23252)
relevant rank: 12.0186 (13.49368)
relevant rank per query: 48.1728 (92.51739)
```

4.4 Submit your runs for the test set

The following code will create a submission file which you will hand in for evaluation.

Replace the placeholders with your student number and name.

In [23]:

```
student_number = 'STUDENTNUMBER'
student_first_name = 'FIRSTNAME'
student_last_name = 'LASTNAME'
output_path = '%s_%s_%s_LTR_assignment.json' % (student_number, student_first_name, stud
ent_last_name)
dataset.output_model_scores(output_path, data, pointwise_model, pairwise_model, listwise_
model)
```

4.5 Comparison

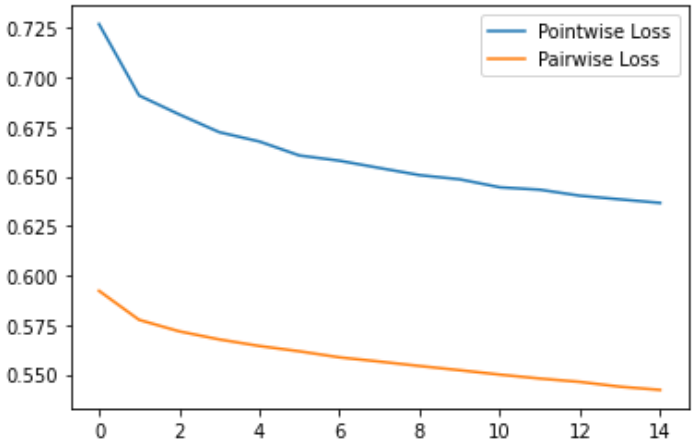
Plot the loss and NDCG@10 progression over the number of epochs, we have provided code that will do this for you below.

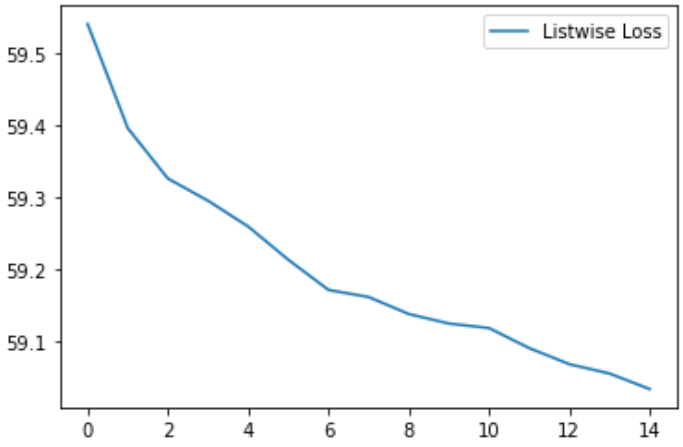
Discuss your the differences you see and how this relates to the LTR theory.

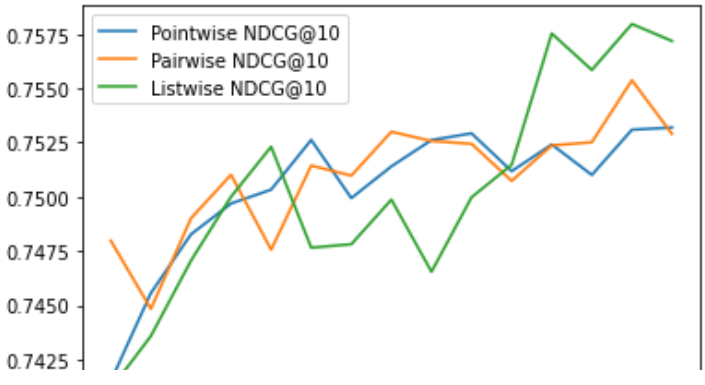
Keep you answer limited to 300 words.

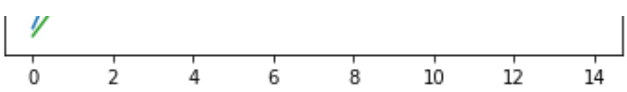
In [24]:

```
import matplotlib.pyplot as plt

plt.plot(range(len(pointwise_losses)), pointwise_losses, label='Pointwise Loss')
plt.plot(range(len(pairwise_losses)), pairwise_losses, label='Pairwise Loss')
plt.legend()
plt.show()

plt.plot(range(len(listwise_losses)), listwise_losses, label='Listwise Loss')
plt.legend()
plt.show()

plt.plot(range(len(pointwise_dcg)), pointwise_dcg, label='Pointwise NDCG@10')
plt.plot(range(len(pairwise_dcg)), pairwise_dcg, label='Pairwise NDCG@10')
plt.plot(range(len(listwise_dcg)), listwise_dcg, label='Listwise NDCG@10')
plt.legend()
plt.show()
```









The loss for listwise is generally a lot higher, and goes down further/quicker than the other two. This would imply that for listwise it might be better to add further iterations to make sure the network is converged (however this would increase the running time significantly). Regarding pointwise and pairwise, the losses are super similar albeit vertically shifted somewhat. The pointwise loss also goes down a bit quicker at the start, but this could easily be an outlier or something.

The pairwise NDCGs don't follow as nice of a line as the losses, but at least these values are comparable between all methods. We see that for the final iteration the listwise performs better than the other two, and the pairwise actually ends under the pointwise method. However this is likely not significant due to the large fluctuations between iterations.

Handing In

Hand in the following three files:

1) the filled in notebook, named:

> **STUDENTNUMBER_FIRSTNAME_LASTNAME_LTR_notebook.ipynb**

2) convert your notebook to PDF format and name it:

> **STUDENTNUMBER_FIRSTNAME_LASTNAME_LTR_notebook.pdf**

3) the json file with your runs on the test set, it should be named:

> **STUDENTNUMBER_FIRSTNAME_LASTNAME_LTR_assignment.json**

In [ ]: