XCS330 Problem Set 1

Due Sunday, February 4 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs330-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit only the `src/submission.py` file in the online student portal. For further details, see Writing Code and Running the Autograder below.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be foudn at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Writing Code and Running the Autograder

All your code should be entered into `src/submission.py`. When editing `src/submission.py`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files other than `src/submission.py`.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.

- `hidden`: These unit tests are the evaluated elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests. These evaluative hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

Local Execution - Hidden Tests

All hidden tests rely on files that are not provided to students. Therefore, the tests can only be run remotely. When a hidden test like `1a-1-hidden` is executed locally, it will produce the following result:

```
----- START 1a-1-hidden:  Test multiple instances of the same word in a sentence.
----- END 1a-1-hidden [took 0:00:00.011989 (max allowed 1 seconds), ???/3 points] (hidden test ungraded)
```

Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:

```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```


Remote Execution

Basic and hidden tests are treated the same by the remote autograder. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.

1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.

1a-0-basic) Basic test case. (2.0/2.0)

1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

1 Multitask Training for Recommender Systems

In this assignment, we will implement a multi-task movie recommender system based on the classic Matrix Factorization [1] and Neural Collaborative Filtering [2] algorithms. In particular, we will build a model based on the BellKor solution to the Netflix Grand Prize challenge and extend it to predict both likely user-movie interactions and potential scores. In this assignment you will implement a multi-task neural network architecture and explore the effect of parameter sharing and loss weighting on model performance.

The main goal of these exercises is to familiarize yourself with multi-task architectures, the training pipeline, and coding in PyTorch. These skills will be important in the course. **Note: This assignment is a warmup, and is shorter than future homeworks will be.**

Code Overview: The code consists of several files; however, you will only need to interact with two:

- `main.py`: To run experiments, execute this file by passing the corresponding parameters.
- `submission.py`: This file contains our multi-task prediction model **MultiTaskNet**, which you will need to finish implementing in PyTorch.

Dataset In this assignment, we will use movie reviews from the MovieLense dataset. The dataset consists of 100K reviews of 1700 movies generated by 1000 users. Although each user interaction contains several levels of meta-data, we'll only consider tuples of the type **(userID, itemID, rating)**, which contain an anonymized user ID, movie ID and the score assigned by the user to the movie from 1 to 5. We randomly split the dataset into a **train** dataset, which contains 95% of all ratings, and a **test** dataset, which contains the remaining 5%.

Problem Definition Given the dataset defined above, we would like to train a model $f(\text{userID}, \text{itemID})$ that predicts: 1) the probability p that the user would watch the movie and 2) the score r they would assign to it from 1 to 5. For some intuition on this setting, consider a user who only watches comedy and action movies. It would not make sense to recommend them a horror movie since they don't watch those. At the same time, we would want to recommend comedy or action movies that the user is likely to score highly.

Evaluation Once we have our trained model, we evaluate it on the test set.

Score Prediction We will evaluate the mean-squared error of movie score prediction on the held-out user ratings, i.e. $\frac{1}{N}\sum_{i=1}^{N} \|\hat{r}_i - r_i\|^2$, where $\hat{r}_i$ is the predicted score for user-movie pair $(\text{userID}_i, \text{itemID}_i)$. The summation is over all pairs in the test set. Better models achieve lower mean-squared errors.

Likelihood Prediction To evaluate the quality of the likelihood model, we use the mean reciprocal rank metric, which provides a higher score for highly ranking the movies the user has seen. The metric is computed as follows:

1. For each user, rank all movies based on the probability that the user would watch them
2. Remove movies we know the user has watched (those in the training set)
3. Compute the average reciprocal ranking of movies the user has watched from the held-out set

Matrix Factorization Consider an interaction matrix M, where $M_{ij} = 1$ if userID_i has rated movie with itemID_j and 0 otherwise. We will represent each user with a latent vector $\mathbf{u}_i \in \mathbb{R}^d$ and each item with a latent vector $\mathbf{q}_i \in \mathbb{R}^d$. We model the interaction probability $p_{ij} = \log P(M_{ij} = 1)$ in the following way:

$$p_{ij} = \mathbf{u}_i^T \mathbf{q}_j + a_i + b_j \tag{1}$$

where a_i is a user-specific bias term and b_j is a movie-specific bias term. At each training step we sample a batch of triples $(\text{userID}_i, \text{itemID}_j^+, \text{itemID}_{j'}^-)$ with size B, such that $M_{i,j} = 1$, while $\text{itemID}_{j'}^-$ is randomly sampled (indicating no user preference). Note the $+$ and $-$ refers to the positive and negative sample. The positive example means the correctly paired itemID for the user, and the negative sample is the randomly sampled one. Let

$$\begin{aligned} p_{ij}^+ &= \mathbf{u}_i^T \mathbf{q}_j + a_i + b_j \\ p_{ij'}^- &= \mathbf{u}_i^T \mathbf{q}_{j'} + a_i + b_{j'} \end{aligned} \tag{2}$$

and optimize the Bayesian Personalised Ranking (BPR) [3] pairwise loss function:

$$\mathcal{L}_F(\mathbf{p}^+, \mathbf{p}^-) = \frac{1}{B}\sum_{i=1}^{B} 1 - \sigma(p_{ij}^+ - p_{ij'}^-) \tag{3}$$

where σ is the sigmoid function.

Regression Model: For training the regression model, we consider only batches of tuples $(\text{userID}_i, \text{itemID}_j^+, r_{ij})$, such that $M_{i,j} = 1$ and r_{ij} is the numerical rating userID_i assigned to itemID_j^+. Using the same latent vector representations as before, we will concatenate $[\mathbf{u}_i, \mathbf{q}_j, \mathbf{u}_i * \mathbf{q}_j]$ (where $*$ denotes element-wise multiplication) together and pass it through a neural network with a single hidden layer:

$$\hat{r}_{ij} = f_\theta([\mathbf{u}_i, \mathbf{q}_j, \mathbf{u}_i * \mathbf{q}_j]) \tag{4}$$

We train the model using the mean-squared error loss:

$$\mathcal{L}_R(\hat{\mathbf{r}}, \mathbf{r}) = \frac{1}{B}\sum_{i=1}^{B} \|\hat{r}_{ij} - r_{ij}\|^2 \tag{5}$$

1. **Implement MultiTaskNet Model**

   The first part of the assignment is to implement the above model in `submission.py`. First you need to define each component when the model is initialized.

   (a) **[3 points (Coding)]** Consider the matrix $\mathbf{U} = [\mathbf{u}_1 |, \ldots, | \mathbf{u}_{N_{\text{users}}}] \in \mathbb{R}^{N_{\text{users}} \times d}$, $\mathbf{Q} = [\mathbf{q}_1 |, \ldots, | \mathbf{q}_{N_{\text{items}}}] \in \mathbb{R}^{N_{\text{items}} \times d}$, $\mathbf{A} = [a_1, \ldots, a_{N_{\text{users}}}] \in \mathbb{R}^{N_{\text{users}} \times 1}$, $\mathbf{B} = [b_1, \ldots, b_{N_{\text{items}}}] \in \mathbb{R}^{N_{\text{items}} \times 1}$. Implement $\mathbf{U}$ and $\mathbf{Q}$ as `ScaledEmbedding` layers with parameter $d =$ `embedding_dim` and $\mathbf{A}$ and $\mathbf{B}$ as `ZeroEmbedding` layers with parameter $d = 1$ (defined in `submission.py`). These are instances of PyTorch Embedding layers with a different weight initialization, which facilitates better convergence. When `embedding_sharing=False` we will set separate latent vector representations used in the distinct factorization and regression tasks $\mathbf{U_{reg}}, \mathbf{U_{fact}}, \mathbf{Q_{reg}}, \mathbf{Q_{fact}}$ of similar type and dimensions of $\mathbf{U}, \mathbf{Q}$. **Note:** Order does matter here! Please declare the layers in the order they are returned.

   Please complete the following functions in `submission.py`:

   i. `init_shared_user_and_item_embeddings`

   ii. `init_separate_user_and_item_embeddings`

   iii. `init_user_and_item_bias`

   (b) **[2 points (Coding)]** Next implement $f_\theta([\mathbf{u}_i, \mathbf{q}_j, \mathbf{u}_i * \mathbf{q}_j])$ as an MLP network. The class `MultiTaskNet` has `layer_sizes` argument, which is a list of the input shapes of each dense layer. Notice that by default `embedding_dim`=32, while the input size of the first layer is 96, since we concatenate $[\mathbf{u}_i, \mathbf{q}_j, \mathbf{u}_i * \mathbf{q}_j]$ before processing it through the network. Each layer (except the final layer) should be followed by a ReLU activation. The final layer should output the final user-item predicted score in and have an output size of 1. Please complete the `init_mlp_layers` function in `submission.py`.

2. **[9 points (Coding)] Implement Forward**

   In the second part of the problem you need to implement the `forward_with_embedding_sharing` and `forward_without_embedding_sharing` methods of the `MultitaskNet` module.

   Both forward methods above receive a batch of $(\text{userID}_i, \text{itemID}_j)$ of user-item pairs. The model should output a probability p_{ij} of shape $(batch\_size,)$ that user i would watch movie j, given by Eq. 1 and a predicted score $\hat{r}_{ij}$ of shape $(batch\_size,)$ the user i would assign to movie j, given by Eq. 4.

   Be careful with output tensor shapes!

3. **Experiments**

To execute experiments, we will run the `run_all.sh` script, which will automatically log training MSE loss, BPR loss and test set MSE loss and MRR scores to TensorBoard. Once you're done with your implementation run the following 4 experiments:

Here the `--factorization_weight` and `--regression_weight` arguments correspond to λ_F and λ_R respectively.

As you run the commands below, ensure to run `tensorboard --logdir=run` in another terminal to view training progress across experiments on your localhost:6006. Note that you can also run `--device` to specify running on gpu or cpu and add `--debug` to view console output training progress.

I Evaluate a model with shared representations and task weights $\lambda_F = 0.99, \lambda_R = 0.01$. You can run this experiment by running:

```
python main.py --factorization_weight 0.99 --regression_weight 0.01
--logdir run/shared=True_LF=0.99_LR=0.01
```

II Evaluate a model with shared representations and task weights $\lambda_F = 0.5, \lambda_R = 0.5$. You can run this experiment by running:

```
python main.py --factorization_weight 0.5 --regression_weight 0.5
--logdir run/shared=True_LF=0.5_LR=0.5
```

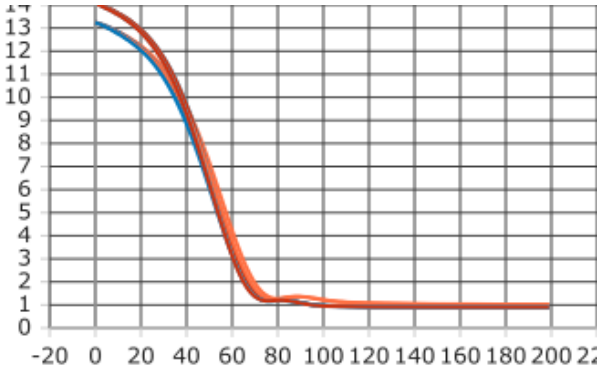
III Evaluate a model with **separate** representations and task weights $\lambda_F = 0.5, \lambda_R = 0.5$. You can run this experiment by running:

```
python main.py --no-shared_embeddings --factorization_weight 0.5
--regression_weight 0.5 --logdir run/shared=False_LF=0.5_LR=0.5
```

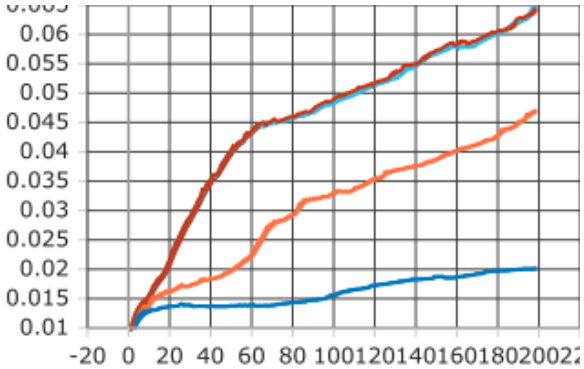
IV Evaluate a model with **separate** representations and task weights $\lambda_F = 0.99, \lambda_R = 0.01$. You can run this experiment by running:

```
python main.py --no-shared_embeddings --factorization_weight 0.99
--regression_weight 0.01 --logdir run/shared=False_LF=0.99_LR=0.01
```

After running all experiments above, you should generate graphs that look like

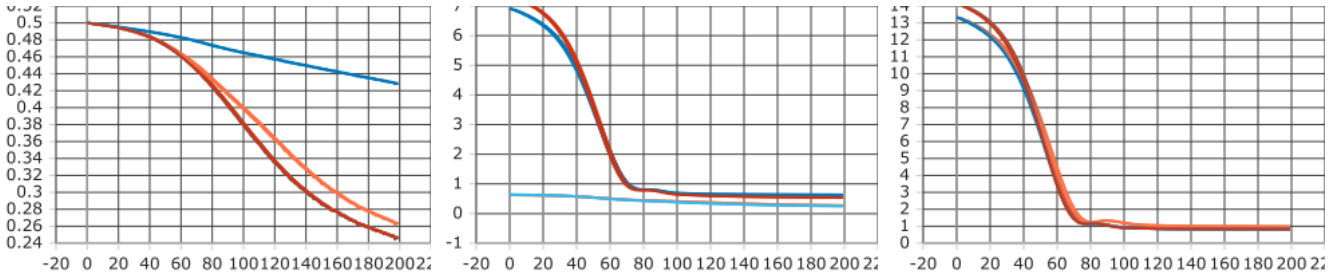


MSE



Mean Reciprocal Rank

Figure 1: Evaluation Results



Factorization Loss Joint Loss MSE

Figure 2: Training Results

Orange: `shared=True_LF=0.99_LR=0.01`
Dark Blue: `shared=True_LF=0.5_LR=0.5`
Red: `shared=False_LF=0.5_LR=0.5`
Light Blue: `shared=False_LF=0.99_LR=0.01`

(a) **[2 points (Written)]** Consider the case with $\lambda_F = 0.99$ and $\lambda_R = 0.01$. Based on the train/test loss curves, does parameter sharing outperform having separate models?

(b) **[2 points (Written)]** Now consider the case with $\lambda_F = 0.5$ and $\lambda_R = 0.5$. Based on the train/test loss curves, does parameter sharing outperform having separate models?

(c) **[2 points (Written)]** In the **shared model setting** compare results for $\lambda_F = 0.99$ and $\lambda_R = 0.01$ and $\lambda_F = 0.5$ and $\lambda_R = 0.5$, can you explain the difference in performance?

Coding Deliverables

For this assignment, please submit the following files to gradescope to recieve points for coding questions:

- `submission.pdf`
- `src/submission.py`
- `src/experiments_1.npy`
- `src/experiments_2.npy`
- `src/experiments_3.npy`
- `src/experiments_4.npy`

References

[1] Koren Yehuda, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42, 2009.

[2] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pages 173–182, 2017.

[3] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. Bpr: Bayesian personalized ranking from implicit feedback. *Conference on Uncertainty in Artificial Intelligence*, 2009.

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the `README.md` for this assignment includes instructions to regenerate this handout with your typeset LaTeX solutions.

3.a

3.b

3.c