IR Competition Project Documentation

Cody Webster

Contents

Overview 3

Project Demonstration 3

Project Results 3

- **DirichletPrior** 3
- **JelinekMercer** 4
- **OkapiBM25** 4
- **BM25+** 4
- **BERT** 5
- **SciBERT *difficulty running this model limited options** 6
- **Tensorflow Ranking custom model** 7

Assumptions 7

Installation Instructions 7

- **What to install if running locally** 7
- **What python packages to install** 8

How to Run 8

- **Prepare Data (Required if local)** 8
- **(Method 1) Run BM25+ or another ranker (reproduces best results)** 9
  - Option 1 (Google Colab) 9
  - Option 2 (run locally) 9
- **(Method 2) Run Tensorflow Ranking** 9
  - Option 1 (Google Colab) 9
  - Option 2 (run locally) 9
- **(Method 3) Run a BERT model** 9
  - Option 1 (Google Colab) 9
  - Option 2 (run locally) 10

Script API Documentation 10

- **General** 10
- **Metapy (BM25, …)** 12
- **Tensorflow Ranking** 13

BERT 16

Resources 20

Citations 20

Installation Guidance 20

Stopwords 20

BM25 21

Tensorflow Ranking 21

Tensorflow Ranking (how to serve a model) 21

Tensorflow Ranking (how to predict) 21

BERT Checkpoint Conversion 21

BERT 21

SciBERT 21

Docker 21

Overview

The purpose of this project is to participate in the IR Competition. For this project I developed and extensively tested three main ways to rank documents. The first is ranking documents using a ranking function either packaged with metapy or through a custom definition implemented through metapy. Initially, the best results that I achieved with this method were through a custom implementation of BM25+. Through a longshot attempt on the last day, I actually beat all of my other rankings using metapy's BM25 implementation. The second is ranking documents using a custom model that is trained using the python library tensorflow_ranking. The third option is to use a pretrained model and it's associated vocabulary and then finetune it using tensorflow_ranking. Overall the best results that I have achieved have come from running the OkapiBM25 algorithm implemented with metapy and using parameters that were produced from a brute force optimization.

Project Demonstration

I have uploaded "cs410_project_demo.mp4" into the git repository.
I have also uploaded the file to Illinois Media: https://mediaspace.illinois.edu/media/t/1_07py0q5f

Project Results

Unfortunately, despite my best attempt I was not able to beat the initial baseline performance score as defined by the class professor or TAs. Nobody else was able to beat the initial baseline either so it is likely that baseline was an unreasonable standard for us to achieve with our limited knowledge and experience. In this section I will briefly detail some of the variations and experiments that I attempted. It is not possible for me capture every variation that I tried but I will cover as best as I can. I do not have a consistent record of the score for every attempt so I will not be including those scores.

On the last day of the competition, the class administrators lowered the score of the baseline. I decided to give it another shot because I really had nothing to lose. I was able to beat the new baseline using the standard BM25 algorithm implemented via metapy. The only difference between what I did on the last day and what I had done on previous days was that I allowed one of the parameters to vary more than I initially had. I will detail that more within the BM25 section.

DirichletPrior

This algorithm was implemented using metapy. The value of the parameter was optimized on the train dataset using a brute forcing approach of testing every reasonable parameter and selecting the parameter that achieved the best NCDG@20 score on the training dataset. None of the attempts for this algorithm achieved results that rivaled the BM25 algorithm so it was quickly determined to be a non-ideal approach.

Initial attempts with this algorithm were made using the body text only. Later attempts also used the title and abstract/intro. Further experimentation with the KLDivergencePRF implementation in metapy and variations on its parameters was also performed but performance lagged behind other approaches.

JelinekMercer

This algorithm was implemented using metapy. The value of the parameter was optimized on the train dataset using a brute forcing approach of testing every reasonable parameter and selecting the parameter that achieved the best NCDG@20 score on the training dataset. None of the attempts for this algorithm achieved results that rivaled the BM25 algorithm so it was quickly determined to be a non-ideal approach.

This algorithm was only ever attempted with the body text and results were deemed not good enough to justify further experimentation.

OkapiBM25

This algorithm was implemented using metapy. The value of the three parameters was optimized on the training dataset using a brute force approach of testing every reasonable combination of parameters and selecting the set that achieved the greatest. Due to the number of variations that I tried with this algorithm, I had to optimize the parameters a number of times and I developed a multi-processing script that was capable of doing this at a much quicker pace. Initially I only attempted to run the algorithm on the body text for each paper but the performance for that was poor. With the body text I attempted to optimize a different NDCG values, 10, 20, and 50 respectively. None of them produced significant improvements in the overall performance.

Next, I attempted to use the title, abstract, and introduction along with the body text and that produce marginal improvements in the score. Another boost came once I removed the body text from the dataset and only trained on the title, abstract, and introduction. I also attempted to use metapy’s implementation of Rocchio feedback. I optimized the parameters for Rocchio in a similar manner to the normal OkapiBM25 algorithm. Ultimately the Rocchio feedback produced worse results then the standalone ranker so I excluded it from further test with this algorithm. The performance was initially still inadequate so I pursued further methods.

One the last day, after I noticed that they had lowered the baseline, I attempted some new variations of this algorithm. I did not run the other algorithms because they all take longer to run and I did not have time to implement anything new. The new variation that I tried on the last day was to let the k3 parameter vary. I had previously not done this due to the faulty assumption that it would only hurt my results because that was the experience that I had during MP2.2. Almost immediately while running the pooled optimization, I was able to see that the performance was superior my previous results. I selected a number of the different parameter combinations to try from the generated set. On the third try I was able to beat the baseline using k1=2.0, b=0.75, and k3=4450. I am immensely frustrated that I have wasted so much effort trying to find other ways to rank documents and the answer was so simple but I am happy to have beaten the baseline.

BM25+

This algorithm was implemented with metapy’s api using the definition of BM25+ from (Trotman et al, 2014). Since it was implemented in Python and C++ like the native metapy implementations, it ran slower. I optimized it in the same way that I optimized the OkapiBM25 algorithm. This implementation ultimately outperformed the native OkapiBM25 algorithm, but it did not beat the baseline. Better results were obtained when using this in conjunction with the BERT implementation.

I tried numerous variations on this algorithm in an attempt to increase my score on the test dataset. Similar to the other algorithms, I initially tried to only use the body text for my analysis. This did not produce the results that I desired. I have briefly listed some of the other variations that I tried:

- attempted with body text only
- attempted with title, abstract, intro, and body text
- attempted with title, abstract, and intro
- attempted with title only
- attempted with Rocchio pseudo feedback, varying the parameters to Rocchio
- attempted to remove all urls in the corpus
- attempted to replace all variants of the words coronavirus, covid-19, 2019-ncov, and sars-ncov-2 with coronavirus
- attempted to run with query text, question text, or narrative text
- attempted to run with all variations of query combined
- attempted different variations of metapy analyzer chains
- attempted to remove docs with duplicate s2_id but different uids, docs were functionally duplicates
- attempted to use a date cutoff for document
- attempted to pre-tokenize the queries and documents using BERT's tokenizer
- attempted to use multiple datasets with various weightings and combine into one result
- used different variations of stopwords that were gathered online (listed in resources)

Ultimately, I was not able to beat the baseline using this custom algorithm, so I continued my exploration of other possibilities.

BERT

I had completed my Technical Review over the BERT model and had learned through that about its superior performance when classifying documents. I chose this as the algorithm to attempt to rank with. It was a significant struggle to get BERT to work for document ranking. There are very few tutorials online for how to achieve this and I spent a significant amount of time trying to get it to work. I ultimately was able to find a tutorial provided by Peter Jansen from the University of Arizona (link in resources). This tutorial gave an example of using the tensorflow_ranking module with BERT to achieve document ranking.

Unfortunately, the tutorial relied on Docker to serve a fine-tuned model. I had to do a significant amount of researching about how to get Docker to work with a tensorflow model on Windows. In order to get it to work, I had to upgrade my Windows 10 OS to the Development version so that it would support the latest Windows Subsytem for Linux (WSL). I needed the latest WSL because that was what nVidia required for their latest CUDA drivers for GPUs. I needed the latest CUDA drivers because that was the only way to get Docker to run on my GPU using WSL. I also had to install Bazel for Windows because the training script was meant to be compiled to run.

This lengthy setup process is obviously unsuitable for the rapid development that I needed and because it is not realistic for reviewers to reproduce this setup on their own machines. Docker is also not supported in Google Colab so I needed to arrive at a repeatable and easily setup solution for the graders.

I was able to remove the need for Docker and the need to serve the model at all by exploring the documentation for tensorflow. This exploration give me insight into how to directly load the model and use it to predict document scores. I implemented this methodology into the scripts and was thus able to create a version that can be easily ran within Google Colab for easy review.

Once I had an effective way of running the model, I was able to experiment with various ways to utilized it. I attempted to run three variations of the pre-trained BERT model, all provided by Google Research, BERT-Mini, BERT-Small, and BERT-Base. The BERT-Base model was too large to effectively train without utilizing TPUs. I choose to do my testing with the BERT-Mini model because it allowed me the most flexibility with my training parameters. Some key parameters used are list size and batch size. The size of the model along with those two parameters determine what hardware is required to run the model. If you try to run on inadequate hardware than you will easily run out of memory and be unable to train the model. I attempted a number of variations in ways to score or setup the data. Ultimately, I was not able to beat either baseline when using BERT but I was able to improve upon my initial results with BM25+. I have detailed some of the variations that I tried below.

- varied the list size
- varied the batch size
- updated vocabulary to include most common coronavirus variants
- attempted to run with query, question, or narrative
- attempted to vary the max token sequence size 256, 512, 1024
- relevance = score of first doc segment
- relevance = max score of all doc segments
- relevance = mean score of all doc segments
- ignore body text and only use title+abstract+intro
- varied the number of training steps
- used both NCDG approximated loss and softmax loss
- re-rank top 1000, 2000, 5000, and 10000 results of BM25+

SciBERT *difficulty running this model limited options

This model was initially very promising but the issues with getting it to run ultimately made it insufficient for what I needed. This model was create by Iz Beltagy et al. for scientific research. This model is essentially just a BERT-Base model that was pretrained on a scientific corpus instead of the generic one. The vocabulary consists of more scientific terms as a result and should theoretically be able to perform better at ranking the scientific documents in the CORD-19 corpus. I updated a few of the unused vocabulary items to the command variants of coronavirus and included the drug remdesivir as well. Unfortunately, due to the size of the model I was very limited in the list size and batch size options that I could try. I firmly believe that with a larger list size I would have been able to adequately finetune this model and it would have performed better than the BERT model. My inability to get the model to run on the TPUs available through Google Colab prevented me from realizing this goal.

Tensorflow Ranking custom model

This model was created by basing it off of the example in the tensorflow ranking repository. I left the structure of the context features and the example features the same. The context features contain the query tokens and the example features contain the document tokens and when training, the relevance judgement of the file. I attempted to change the script to use the argparse module instead of the flags from the absl module but when that was attempted the script stopped producing the results of the training data to stdout. I determine that there are likely scripts deeper within the tensorflow ranking module that are using these flags to determine various facets of the training process, so I restore the flags.

The model consists of 3 hidden layers at a size of 64, 32, and 16 respectively. During testing the dropout rate of the model was adjust to 0.65 from the default of 0.80. This adjustment produced better results but further drops in this dropout rate risked overfitting the model to the training data. The batch size was set to 1 and the list size maximized so that it would train on more docs for a single query at once. Based on empirical results that I observed during experimentation, this produced better results than increasing batch size and lowering the list size. The maximum limit for the list_size was 100 depending on the hardware available in Colab and if it exceeded that value then it would run out of memory when training.

One other significant variable was the max sequence length to use for each example. I tested with both 512 and 1024. The results did not appear to differ significantly between the two but this variable is important because it defines how much of the document can be captured per example. For this model I only used the title, abstract, and introduction because previous experimentation on other models and rankers showed that the body text was not helpful. Ultimately the results produced with this method were worse the results obtained from the BERT models.

Assumptions

- If you are running this locally then you are running this code on Windows 10 machine that has ample RAM and a CUDA capable gpu. Every script can run on Linux but I have not generated scripts within this repo to accommodate that.
- All file paths are relative to the base directory of the repository

Installation Instructions

What to install if running locally

Acquire the datasets:

Files should be downloaded and unzipped here: .\competition\datasets

The test files can be obtained from https://drive.google.com/file/d/1FCW8fmcneow5yyDgApkPIGM-r2x6OFkm/view?usp=sharing

The train files can be obtained from https://drive.google.com/file/d/1E_Y-MkNvoOYoCZUZZa8JJ3ExiHYTfKTo/view?usp=sharing

Python

Instructions: https://docs.conda.io/projects/conda/en/latest/user-guide/install/

Latest Nvidia Drivers (required to run BERT or Tensorflow Ranking on GPU)

Instructions: https://docs.nvidia.com/cuda/wsl-user-guide/index.html

Instructions require installation of other software and packages you must follow all of them

BERT Model (required to run BERT)

Due to the size of the models, they are not directly included in the repository

Google BERT Models (recommended):

Google offers a variety of different sizes for the models to run. I ran using a BERT Mini but could possibly use a BERT Small as well. Anything larger likely requires a TPU through Google Colab to run.

https://github.com/google-research/bert

SciBERT Model (not recommended, requires TPU):

https://github.com/allenai/scibert/

What python packages to install

To run the full capability of all scripts defined in this project it is required that you setup two environments. All but one script can run in the main environment but the BERT checkpoint convertor relies on the dev version of a module that conflicts with requirements of the other modules:

Main Python Environment:

This is the main environment and should be used when running most of the scripts. You can install the latest stable versions of these modules to run the project.

Python Version: 3.7

Pip Packages:

tensorflow_ranking
metapy
pytoml

BERT Conversion Environment:

This is required because there are conflicts in the dependencies of the modules required to convert a model's checkpoints and the modules used to run it.

Python Version: 3.7

Pip Packages:

tf-models-nightly

How to Run

WARNING: running some of these scripts will require a large amount of RAM if you use the full dataset

* Your python environment should be activated before running any scripts

Prepare Data (Required if local)

Generate the json file that contains the information for each dataset. This is required in order to run any of the different methodologies.

File Path: .\competition\create_bert_data.bat

(Method 1) Run BM25+ or another ranker (reproduces best results)

Option 1 (Google Colab)

Run each cell individually from top to bottom to fully. If you view the file within Github there is a link at the top of the file to open in Google Colab. This is the only Google Colab file that does not require a Google Colab Pro account. The free account should be capable of running this notebook.
File Path: .\colab_cranfield_metapy.ipynb

Option 2 (run locally)

Generate the Cranfield Datasets that will be required to run the ranker
File Path: .\competition\cranfield_metapy\cm_create_data.bat

Run the ranker of your choice (default arguments already in the file)
File Path: .\competition\cranfield_metapy\cm_rank_docs.bat

(Method 2) Run Tensorflow Ranking

Option 1 (Google Colab)

Run each cell individually from top to bottom to fully. Training data for a fine-tuned model is provided so there is not need to run training but the capability is provided. If you view the file within Github there is a link at the top of the file to open in Google Colab. T This file will likely require a Google Colab Pro account for the script to work. If not using a Colab Pro account it will run out of memory.
File Path: .\colab_tfr.ipynb

Option 2 (run locally)

Convert the train data into two tensorflow example list with context (elwc) files
File Path: .\competition\tfr_custom\tfr_create_train_elwc.bat

Train the model on the elwc files
File Path: .\competition\tfr_custom\tfr_train_model.bat

Re-rank the output of a previous run (requires a file in the format of a predictions file, no limit on docs per query but will truncate output to 1000
File Path: .\competition\tfr_custom\tfr_predict.bat

(Method 3) Run a BERT model

Option 1 (Google Colab)

Run each cell individually from top to bottom to fully. Training data for a fine-tuned model is provided so there is not need to run training but the capability is provided. If you view the file within Github there is a link at the top of the file to open in Google Colab. This file will likely require a Google Colab Pro account for the script to work. If not using a Colab Pro account it will run out of memory.
File Path: .\colab_bert.ipynb

Option 2 (run locally)

Convert the train data into two tensorflow example list with context (elwc) files
File Path: .\competition\bert\bert_create_train_elwc.bat

Train the model on the elwc files
File Path: .\competition\bert\bert_train_model.bat

Re-rank the output of a previous run (requires a file in the format of a predictions file, no limit on docs per query but will truncate output to 1000
File Path: .\competition\bert\bert_predict.bat

Script API Documentation

**Unless otherwise specified paths are relative to the root of the repository

General

File: ./competition/check_covid_variants.py
Purpose: Check for coronavirus variants in the corpus
Source: developed by project team
API:

variant_file:
the file that the coronavirus variants will be output to. File can already exist. Existing variants will be loaded.

known_variants:
known variants of the coronavirus that exist in the corpus and will be used to determine other variants

doc_keys:
the keys to use when search the document dictionary for variants

run_type:
the dataset that will be searched for variants

input_dir:
the directory that contains the json representation of the datasets to be processed

Detailed Description:

This is a python script that was implemented in python 3.7. This script relies on the output of create_bert_data.py in order to run. It loads the json for one or both datasets and then processes the text to find variants of some predefined words in the corpus and queries. The text is processed using python multiprocessing module in order to speed up execution of the script.

File: ./competition/checkpoint_converter.py
Purpose: Converter BERT checkpoint files from Tensorflow v1 to v2+
Source: Tensorflow Model Garden Repository
API:

bert_config_file:
Bert configuration file to define core bert layers.

checkpoint_to_convert:

Initial checkpoint from a pretrained BERT model core (that is, only the BertModel, with no task heads.)

converted_checkpoint_path:

Name for the created object-based V2 checkpoint.

checkpoint_model_name:

The name of the model when saving the checkpoint, i.e., the checkpoint will be saved using: tf.train.Checkpoint(FLAGS.checkpoint_model_name=model).

converted_model:

Whether to convert the checkpoint to a `BertEncoder` model or a `BertPretrainerV2` model (with mlm but without classification heads).

Detailed Description:

This script was not written by me but was lightly modified for the purpose of this project. This is a python script that was tested in python 3.7. This script is meant to convert a BERT module that you downloaded into a version that is capable of being ran using tensorflow version 2.0+.

File: ./competition/create_bert_data.py

Purpose: Combine and compile the information from the dataset into and easily loadable json for use by other scripts

Source: developed by project team

API:

vocab_file:

The file containing the vocabulary to be used when tokenizing a text

variant_file:

A file containing variants of the word coronavirus or its equivalents

variant_default:

The value that will be used to replace all variants in the corpus

run_type:

The dataset that will be searched for variants

tokenize:

If use, this will store a tokenized representation of the script in the output json. Must be used with `vocab_file`

input_dir:

The directory that contains all of the source files for the dataset

output_dir:

The directory where the json representations of the documents will be stored

Detailed Description:

This is a python script that was tested in python 3.7. The inputs to the script will determine the exact behavior at run time but lets discuss the more complete case of the training dataset. When the training dataset is specified it will start by loading the queries from there original xml format into a python dictionary. If the script is ran with the tokenize option then each of the three variants of the query (query, question, and narrative) will be tokenized according to the vocabulary defined in the specified vocab file. This json is then dumped into a target directory. For the training dataset it will also load the query relevance judgements. A list of all of the documents and their associated uid, title, abstract, publication date, and list of files containing the text representation is pulled from the metadata.csv file. The list of documents to further process is pruned done to the same as the list of relevance judgements. For each document to process the script iterates through the list of files that contains the text representation until it finds a suitable candidate or exhausts all options. The text from the representation file is loaded in the document dictionary object. If the tokenize option is used, the documents will then be tokenized through the use of a multi-processed pool. The resulting output is then written to the disk.

Metapy (BM25, …)

File: ./competition/cranfield_metapy/create_cranfield.py
Purpose: Use the json representation of the dataset to create a cranfield dataset for use with metapy
Source: developed by project team
API:

variant_file:
A file containing variants of the word coronavirus or its equivalents

variant_default:
The value that will be used to replace all variants in the corpus

run_type:
The dataset that will be searched for variants

query_keys:
The keys that will be used when creating the cranfield query file. If multiple keys are specified, text will be combined.

doc_keys:
The keys that will be used when creating the cranfield document files. Multiple keys can be specified and multiple keys can be combined into a single document. If creating separate datasets, use a ';' to separate keys. If combining keys for a dataset use a ':' to separate keys:

cranfield_dir:
The directory to use as a base for generating the cranfield data.

input_dir:
The directory where the json representations of the documents are stored

Detailed Description:

This is a python script that was implemented in python 3.7. This script relies on the output of create_bert_data.py in order to run. This uses the loaded json representation of the dataset to produce a use specified cranfield dataset for use with metapy. Each cranfield dataset can be uniquely defined by the user to consist of a one-to-one relationship with keys in the document dictionary or it can consist of multiple keys combined. This allows for flexibility in how the expansive each iteration of ranking testing is.

File: ./competition/cranfield_metapy/search_eval.py
Purpose: Rank the documents in the corpus and create the predictions file
Source: original version from MP 2.2 and heavily modified to fit use case
API:

config_template:
The template file that will be used for creating the configs for each run of the ranker

run_type:

The dataset that will be searched for variants

dat_keys:

The keys to use that indicate the name of the cranfield dataset(s). This corresponds to the first key used for every section of 'doc_keys' parameter and the 'create_cranfield.py' script.

doc_weights:

The weights to use when combining the rankings of multiple datasets.

ranker:

The ranker to use for ranking the documents. Valid rankers can be found in the script.

params:

The value(s) for the ranker parameters. Multiple values should be separated by `;`.

cranfield_dir:

The directory that is the base for the cranfield dataset(s)

predict_dir:

The directory to contain the predictions file.

remove_idx:

Delete an existing inverted index and create a new one. If no index exists it will not fail

Detailed Description:

This is a python script that was implemented in python 3.7. This script relies on the output of create_cranfield.py in order to run. This script allows the user to specify what ranker to run from a predefined list of rankers. The user can also run rankings over multiple datasets and specify the weight that each ranking will contribute to the final prediction rankings.

File: ./competition/cranfield_metapy/search_eval_pool.py
Purpose: Used to find the optimal parameters for ranking algorithms. Not for production use.
Source: original version from MP 2.2 and heavily modified to fit use case
API: N/A
Detailed Description:

This is a python script that was implemented in python 3.7. This script relies on the output of create_cranfield.py in order to run. This script allows the user to attempt to optimize the parameters for a predefined list of rankers. The optimization takes place using a multi-processed pool so that it can run numerous iterations over a predefined range of values. Each iteration of the optimization loop is evaulated based on normalized cumulative gain at 20 documents.

Tensorflow Ranking

File: ./competition/tfr_custom/tfr_convert_json_to_elwc.py
Source: modified version of file located here https://github.com/cognitiveailab/ranking
Purpose: To convert json data to an elwc formatted tfrecord file for use with model training
API:

vocab_file:

The file containing the vocabulary to be used when tokenizing a text

sequence_length:

The max length of any individual query or document query_file:

The json file that contains all of the queries

qrel_files:

The file containing the training relevance judgements

query_key:

The type of query that will be used as context for ranking, i.e. (query, question, narrative)

doc_file:

The json file that contains all of the documents

output_train_file:

The tfrecord file to be used for training

output_eval_file:

The tfrecord file to be used for evaluation

list_size:

The maximum number of documents to score per query

do_lower_case:

ensure all query and document strings are lowercase

Detailed Description:

This script was not originally written by me but has been highly modified for the purpose of this project. This is a python script that was tested in python 3.7. This script relies on the output of create_bert_data.py in order to run. This script loads in the queries, documents, and relevance judgments for the train dataset and converts them to an example list with context files for use in training the model. The maximum number of documents associated with each query is defined by the list size parameter. If the number of documents for a query exceeds this parameter value then the document list is chunked into multiple elwc representations before being output to the files. The elwc objects are formatted for the model's specific requirements. Each query and document is limited to a maxim number of tokens as defined by sequence_length and if a document or query is longer than this value then it is truncated.

File: ./competition/tfr_custom/tfr_train.py

Source: lightly version of file located here https://github.com/tensorflow/ranking/blob/master/tensorflow_ranking/examples/tf_ranking_tfrecord.py

Purpose: To train a tensorflow model on a dataset with predefined relevance judgements.

API:

data_format:

Data format defined in data.py.

train_path:

Input file path used for training.

eval_path:

Input file path used for eval.

vocab_path:

Vocabulary path for query and document tokens.

model_dir:

Output directory for models.

batch_size:

The batch size for train.

num_train_steps:

Number of steps for train.

learning_rate:

Learning rate for optimizer.

dropout_rate:

The dropout rate before output layer.

hidden_layer_dims:

Sizes for hidden layers.

list_size:

List size used for training. Use None for dynamic list size.

group_size:

Group size used in score function.

loss:

The RankingLossKey for the loss function.

weights_feature_name:

The name of the feature where unbiased learning-to-rank weights are stored.

listwise_inference:

If true, exports accept `data_format` while serving.

use_document_interactions:

If true, uses cross-document interactions to generate scores.

embedding_dim:

max size of any query or document

Detailed Description:

This script was not originally written by me but has been lightly modified for the purpose of this project. This is a python script that was tested in python 3.7. This script relies on the output of tfr_convert_json_to_elwc.py in order to run. This script loads in the elwc files generated by tfr_convert_json_to_elwc.py and trains the model on the data provided in the elwc files. The training will run for the "num_train_steps" defined by the user. The outputs of this script are stored in "model_dir" and you can load the output model for use in prediction of a documents relevance.

File: ./competition/tfr_custom/tfr_predict.py

Source: modified version of file located here https://github.com/cognitiveailab/ranking

Purpose: To score the documents in the test dataset against the queries in the test dataset and output a predictions file

API:

vocab_file:

The file containing the vocabulary to be used when tokenizing a text

sequence_length:

The max length of any individual query or document

query_file:

The json file that contains all of the queries qrel_files:

The file containing the training relevance judgements

query_key:

The type of query that will be used as context for ranking, i.e. (query, question, narrative)

doc_file:

The json file that contains all of the documents

output_file:

The file that will contain the scores

model_path:

The path to the saved model for use in predictions

docs_at_once:

The maximum number of documents to score at once

rerank_file:

The input file consisting of previous rankings that will be reranked with the tensorflow model

do_lower_case:

ensure all query and document strings are lowercase

Detailed Description:

This script was not originally written by me but has been highly modified for the purpose of this project. This is a python script that was tested in python 3.7. This script relies on the output of create_bert_data.py in order to run. This script loads in the queries and documents for the specified dataset. It also loads in the relevance judgements of a previous ranking and a trained model. The list of documents to rank is limited to only the files specified in the previous ranking. Each document query combination is converted into an example list with context(elwc) object and then it is passed to the loaded model. All of the documents ranked for each query is combined into a singular list that is sorted by the score and only the top 1000 documents are output to a predictions file. The predictions file output is the same as the file of the relevance judgements.

BERT

File: ./competition/bert/bert_predict.py

Source: modified version of file located here https://github.com/cognitiveailab/ranking

Purpose: To convert json data to an elwc formatted tfrecord file for use with model training

API:

vocab_file:

The file containing the vocabulary to be used when tokenizing a text

sequence_length:

The max length of any individual query or document

query_file:

The json file that contains all of the queries

qrel_files:

The file containing the training relevance judgements

query_key:

The type of query that will be used as context for ranking, i.e. (query, question, narrative)

doc_file:

The json file that contains all of the documents

output_train_file:

The tfrecord file to be used for training

output_eval_file:

The tfrecord file to be used for evaluation

list_size:

The maximum number of documents to score per query

do_lower_case:

ensure all query and document strings are lowercase

Detailed Description:

This script was not originally written by me but has been highly modified for the purpose of this project. This is a python script that was tested in python 3.7. This script relies on the output of create_bert_data.py in order to run. This script loads in the queries, documents, and relevance judgments for the train dataset and converts them to an example list with context files for use in training the model. The maximum number of documents associated with each query is defined by the list size parameter. If the number of documents for a query exceeds this parameter value then the document list is chunked into multiple elwc representations before being output to the files. The elwc objects are formatted for the model's specific requirements. Each query and document is limited to a maxim number of tokens as defined by sequence_length and if a document or query is longer than this value then it is truncated.

File: ./competition/bert/bert_train.py

Source: lightly version of file located here https://github.com/tensorflow/ranking/blob/master/tensorflow_ranking/extension/examples/tfrbert_example.py

Purpose: To train a tensorflow model on a dataset with predefined relevance judgements.

API:

local_training:

If true, run training locally.

train_input_pattern:

Input file path pattern used for training.

eval_input_pattern:

Input file path pattern used for eval.

learning_rate:

Learning rate for the optimizer.

train_batch_size:

Number of input records used per batch for training.

eval_batch_size:

Number of input records used per batch for eval.

checkpoint_secs:

Saves a model checkpoint every checkpoint_secs seconds.

num_checkpoints:

Saves at most num_checkpoints checkpoints in workspace.

num_train_steps:

Number of training iterations. Default means continuous training.

num_eval_steps:

Number of evaluation iterations.

loss:

The RankingLossKey deciding the loss function used in training.

list_size:

List size used for training.

convert_labels_to_binary:

If true, relevance labels are set to either 0 or 1.

model_dir:

Output directory for models.

dropout_rate:

The dropout rate.

bert_config_file:

The config json file corresponding to the pre-trained BERT model. This specifies the model architecture. Please download the model from the link: https://github.com/google-research/bert

bert_init_ckpt:

Initial checkpoint from a pre-trained BERT model. Please download from the link: https://github.com/google-research/bert

bert_max_seq_length:

The maximum input sequence length (#words) after WordPiece tokenization. Sequences longer than this will be truncated, and sequences shorter than this will be padded.

bert_num_warmup_steps:

This is used for adjust learning rate. If global_step < num_warmup_steps, the learning rate will be `global_step/num_warmup_steps * init_lr`. This is implemented in the bert/optimization.py file.

Detailed Description:

This script was not originally written by me but has been lightly modified for the purpose of this project. This is a python script that was tested in python 3.7. This script relies on the output of bert_convert_json_to_elwc.py in order to run. This script loads in the elwc files generated by bert_convert_json_to_elwc.py and trains the model on the data provided in the elwc files. The training will run for the "num_train_steps" defined by the user. The outputs of this script are stored in "model_dir" and you can load the output model for use in prediction of a documents relevance.

File: ./competition/bert/bert_convert_json_to_elwc.py
Source: modified version of file located here https://github.com/cognitiveailab/ranking
Purpose: To score the documents in the test dataset against the queries in the test dataset and output a predictions file
API:

vocab_file:

The file containing the vocabulary to be used when tokenizing a text

sequence_length:

The max length of any individual query or document

query_file:

The json file that contains all of the queries

qrel_files:

The file containing the training relevance judgements

query_key:

The type of query that will be used as context for ranking, i.e. (query, question, narrative)

doc_file:

The json file that contains all of the documents

output_file:

The file that will contain the scores

model_path:

The path to the saved model for use in predictions

docs_at_once:

The maximum number of documents to score at once

rerank_file:

The input file consisting of previous rankings that will be reranked with the tensorflow model

do_lower_case:

ensure all query and document strings are lowercase

Detailed Description:

This script was not originally written by me but has been highly modified for the purpose of this project. This is a python script that was tested in python 3.7. This script relies on the output of create_bert_data.py in order to run. This script loads in the queries and documents for the specified dataset. It also loads in the relevance judgements of a previous ranking and a trained model. The list of documents to rank is limited to only the files specified in the previous ranking. Each document query combination is converted into an example list with context(elwc) object and then it is passed to the loaded model. All of the documents ranked for each query is combined into a singular list that is sorted by the score and only the top 1000 documents are output to a predictions file. The predictions file output is the same as the file of the relevance judgements.

Resources

Citations

Rama Kumar Pasumarthi and Sebastian Bruch and Xuanhui Wang and Cheng Li and Michael Bendersky and Marc Najork and Jan Pfeifer and Nadav Golbandi and Rohan Anil and Stephan Wolf. 2019. TF-Ranking: Scalable TensorFlow Library for Learning-to-Rank. Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 2970-2978

Turc, Iulia and Chang, Ming-Wei and Lee, Kenton and Toutanova, Kristina. 2019. Well-Read Students Learn Better: On the Importance of Pre-training Compact Models. arXiv preprint arXiv:1908.08962v2

G. V. Cormack, C. L. A. Clarke, and Stefan Buttcher. 2009. Reciprocal Rank Fusion outperforms Condorcet and individual Rank Learning Methods

Andrew Trotman, Antti Puurula, Blake Burgess. 2014. Improvements to BM25 and Language Models Examined

Zhuyun Dai and Jamie Callan. 2019. Deeper Text Understanding for IR with Contextual Neural Language Modeling. In Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '19), July 21–25, 2019, Paris, France. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3331184.3331303

Iz Beltagy, Kyle Lo, Arman Cohan. 2019. SCIBERT: A Pretrained Language Model for Scientific Text. Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pages 3615-3620

Installation Guidance

tensorflow gpu requirements:

https://www.tensorflow.org/install/gpu

CUDA:

https://developer.nvidia.com/cuda-10.1-download-archive-update2?target_os=Linux&target_arch=x86_64&target_distro=Ubuntu&target_version=1804&target_type=deblocal

cuDNN:

https://developer.nvidia.com/rdp/cudnn-archive

install .deb:

https://www.quora.com/Is-it-possible-to-install-a-deb-package-in-Windows#:~:text=Potentially%20yes%2C%20as%20long%20as,deb

Stopwords

https://www.ranks.nl/stopwords
https://countwordsfree.com/stopwords

BM25

https://github.com/vespa-engine/cord-19/blob/master/cord-19-queries.md

https://docs.vespa.ai/documentation/reference/bm25.html

Tensorflow Ranking

http://cognitiveai.org/2020/09/08/using-tensorflow-ranking-bert-tfr-bert-an-end-to-end-example/

https://github.com/cognitiveailab/ranking

https://github.com/tensorflow/ranking

https://colab.research.google.com/github/tensorflow/ranking/blob/master/tensorflow_ranking/examples/handling_sparse_features.ipynb

Tensorflow Ranking (how to serve a model)

https://www.tensorflow.org/tfx/serving/serving_basic

https://www.tensorflow.org/tfx/tutorials/serving/rest_simple#serve_your_model_with_tensorflow_serving

https://github.com/tensorflow/docs/blob/master/site/en/r1/guide/saved_model.md#cli-to-inspect-and-execute-savedmodel

https://github.com/cognitiveailab/ranking/blob/master/tensorflow_ranking/extension/examples/tfrbert_client_predict_from_json.py

Tensorflow Ranking (how to predict)

https://github.com/tensorflow/ranking/issues/48

https://stackoverflow.com/questions/59528975/tf-estimator-predict-slow-with-tensorflow-ranking-module

BERT Checkpoint Conversion

https://github.com/tensorflow/models/tree/master/official/nlp/bert

BERT

https://github.com/google-research/bert

https://ai.googleblog.com/2018/12/tf-ranking-scalable-tensorflow-library.html

https://arxiv.org/pdf/1812.00073.pdf

SciBERT

https://www.aclweb.org/anthology/D19-1371/

https://huggingface.co/gsarti/scibert-nli

https://github.com/allenai/scibert/

Docker

https://superuser.com/questions/1382472/how-do-i-find-and-enable-the-virtualization-setting-on-windows-10

https://docs.docker.com/docker-for-windows/troubleshoot/#virtualization-must-be-enabled https://docs.docker.com/docker-for-windows/install/