410 Project Documentation

Overview

At a high level, the code in the repository finds similar papers to a query paper where the title and abstract of the paper is provided. It will find papers that are within the same arxiv classification and then rank the papers based on TF-IDF scores. The major components of the code include the classification model, the ranking via TF-IDF, the evaluations for the classification model, and the Flask API. From our evaluations, the classification model classifies with a micro-precision, micro-recall, and micro-f1 score of 0.76. Applications of this software include paper recommendation systems to help researchers find similar papers during literature review process.

Implementation Details

build_dataset.py

My build_dataset.py script fetches research paper titles and abstracts from the arxiv API. The HTML response is then parsed in the "add_query" function using beautiful soup to find the summary and title fields. This script produces a dataset.json file after the papers for each topic are retrieved.

split_dataset.py

The split_dataset.py script splits the dataset.json file into train.json and test.json files for training and evaluations respectively. To do this, each topic is iterated through and the papers for each topic are shuffled randomly. Then, the first 10 papers are placed into the test set while the rest of the papers are left for the train set.

dataset.py

The dataset.py file creates a cache file for the BERT embeddings of the train set if the file does not exist. Otherwise, it loads the embeddings from the cache file. It also inherits the torch.utils.data.Dataset class so that it can be used with Pytorch's Dataloader. To cache the

embeddings, the train.json file is loaded and the title and abstracts for each paper are concatenated. This concatenated String is then fed into a BERT tokenizer that transforms this String into a series of subword tokens. These tokens are then fed into BERT, which computes the pooled embeddings which are then cached along with the numerical index of the label. The __len__ function returns the length of the training dataset and the __getitem__ function returns an entry (BERT embedding and label index) of the training dataset.

model.py

The model.py file defines the Neural Network classifier used to classify the research papers into categories. It initializes 2 linear layers in the constructors as well as a relu layer and a dropout layer. In the forward function, it takes in the cached BERT embeddings and feeds it through the Neural Network classifier.

train_model.py

The train_model.py file initializes an instance of the training dataset and feeds it into a Pytorch dataloader, which shuffles and batches the data. The model is also loaded and fed into the train function, which iterates over a specified number of epochs. For each epoch, the entirety of the dataset is iterated over and the embeddings from the dataset are fed into the Neural Network classifier to produce a set of logits. The loss is then computed using cross entropy loss between the computed logits and the labels from the training dataset. The backward call on the loss object performs backpropagation to compute the gradients and the Adam optimizer then updates the model parameters. Once all the epochs are finished, the trained parameters of the model are then saved to the "./model.pt" file.

eval_model.py

The eval_model.py file loads the evaluation dataset along with the saved model parameters. It iterates over the evaluation dataset, storing the labels and the computed predictions (which can be found by a simple argmax over the logits) for evaluation. At the end of the "eval" function, the classification_report function from scikit-learn is utilized to compute the micro and macro precision, recall, and f1 score, which is saved to the "metrics.txt" file.

ranking.py

The ranking.py file loads the model and predicts the arxiv class of the query document. It concatenates the query title and query abstract, tokenizes and computes the BERT embeddings of the query, and runs the trained classifier to produce the arxiv class predictions. Papers from the dataset the predicted class are selected and used to train TF-IDF. The rankings are then produced by comparing the TF-IDF similarity score between the query and the documents in the dataset.

api.py

The api.py file defines a Flask API with a single endpoint that runs locally on port 8000. This endpoint takes in a "title", "summary", and "k" query strings and use these parameters to call the ranking function. The final rankings are returned as a json file.

Usage Details

Dependency Installation

First install the dependencies, which include scikit-learn, pytorch, flask, transformers, and beautiful soup. A requirements.txt file is provided which can help in installation. To do this, run "pip install -r requirements.txt".

Creating the Dataset

Run build_dataset.py file if no dataset.json file exists. To add more topics, update the "topics" list in this script. If a dataset.json file exists or after running build_dataset.py, run split_dataset.py file to create train.json and test.json.

Train the Classifier

Run train_model.py, which should create a "cache.pickle" file on the first run and should create a model.pt file.

Evaluate the Classifier

Run eval_model.py, which outputs a metrics.txt file that includes all of the evaluation scores.

Run Rankings on Queries

The ranking function takes in the query title, query abstract, and k as input where k is the number of similar papers to return. Run ranking.py to test the ranking function and api.py to start the ranking API.

Contributions

I am working alone.