Corpora Build Project

Author: Carlos Guzman

The scripts in this project can be split up into three categories,

1. Parsing Scripts
2. List Creation Scripts
3. Subtraction Scripts.

The lists created can also be split into three field categories

1. QC lists: Associated with quantum computing, quantum sensing, and quantum information only but not quantum mechanics.
2. Q lists: Associated with quantum mechanics only.
3. C lists: Anything not associated with QC and Q.

I was able to get three different results when parsing the TeX files. The results are split up into:

1. Results_first_run: The lists that we created first time that had too much noise for our liking.
2. Results_second_run: Lists that have multiple words omitted
3. Results_third_run: Lists that do not have multiple words omitted and the final lists that were presented.

Parsing Scripts

Shell scripts

For all of these scripts make sure that the paths are adjusted properly. I was running these through my HPC account, but since I moved the project to the lab directory, only a few changes should be made at the beginning of the path.

corpora_build.sh

This script has 4 major functions. It will take 3 arguments from the command line, which are the node number to create a directory (i.e. 1 to create a folder called node1). This folder is used to organize the results. The next 2 arguments are to split the total amount of arXiv files to run through in this node. The total amount of arXiv files were 2375, so for each node I iterated through 100 arXiv files to speed up the processes and run multiple scripts. The script will then run 20 tasks in parallel. This number was chosen because it was the maximum allowed of task per node. The number of arXiv files to process per call can be adjusted. It will run the following functions one step at a time for each task:

- Main: This function will begin by calling the following 3 functions. Once those functions are completed. It will execute an s srun call to run_main_py.sh to begin parsing the TeX files found in each arXiv directory.
- create_directory: This function copies one tar file at a time and creates a temporary directory for the file in the test_data directory. It then calls on extract_inner_directories.
- Extract_inner_directories: This function extracts the contents in the arXiv tar file.
- change_encoding: This function searches for files with an encoding that is not utf-8. It then tries to change the encoding to utf-8.

run_main_py.sh
This script was created in order to run the python parsing scripts in parallel on Slurm with an srun command. It simply iterates through the TeX files found in the arXiv directory and call on main.py to begin the parsing process.

c_b.sh
This script is identical to corpora_build.sh. I created this run 10 tasks in parallel for the first 100 arXiv tar files. These first 100 arXiv tar files took the longest to extract and process because they contain more data than the others.

run_scirpts.sh
This script was created to make all the sbatch calls necessary to process all the arXiv files. It makes 10 c_b.sh calls to process the first 100 arXiv files split in 10s. Followed by making 23 corpora_build.sh calls to process the last arXiv files split in 100s.

Python scripts
These scripts are all called from corpora_build.sh or c_b.sh. They are used to parse each TeX file found and extract only information found within the section, subsection, subsubsection, paragraph, subparagraph, chapter, part LaTeX environments. This was done to help filter out any noise found in other LaTeX commands and to only extract meaningful information found in each paper. It will also skip through commands such as table, figure, equation, etc. in order to filter out unnecessary noise found associated with those commands.

nltk_setup.py
This script simply downloads NLTK's stopwords. It is only called once per corpora_build.sh or c_b.sh.

main.py
This script has 4 command line arguments:

1. TeX file location
2. arXiv temporary directory.
3. The 'node' number
4. The current TeX file number [0, Total number of TeX files in arXiv directory – 1]

Then it will call parse_arxiv_tex.py's function latex_parser to parse the current TeX file using the first command line argument. If any tokens are returned, then it will call on tex_word_processing.py using the last 3 command line arguments.

parse_arxiv_tex.py
This script will simply parse the TeX file and extract information found only in the section, paragraph, etc. environments mentioned before. It will also skip through commands such as table, figure, equation, etc. that contain unnecessary noise. It also relies extensively on regular expressions to filter out any commands and noise found in the text extracted. It will terminate early if the bibliography, appendix, or acknowledgements are encountered. This script will return the tokens, extracted using NLTK's word tokenize function, from the TeX file back to

main.py. If no tokens are extracted from the TeX file, then it will return an empty list and terminate main.py.

tex_word_processing.py

This script has 3 major functions:

- normalize_tokens: Removes stopwords using NLTK's stopwords set any word with a length less than 3 and greater than 30. This is done further filter noise.
- create_ngram_lists: This function uses NLTK's n-grams to create list of words either in unigrams, bigrams, or trigrams.
- categorize_words: This function first counts the number of occurrences the bigram 'quantum information', 'quantum comput', 'quantum sens' appears in the given TeX file. It then counts the number of occurrences the unigram 'quantum' appears in the TeX file. If no occurrence of the word 'quantum', then it prints each n-gram list to a text file in the C field category. If an occurrence of the word 'quantum' but no occurrence of 'quantum information', 'quantum sens', or 'quantum comput', then it will print each n-gram list to text file in the Q field category. Lists with an occurrence of 'quantum information', 'quantum sens', or 'quantum comput' are printed to a text file in the QC field category.

original_tex_word_processing.py

This script is identical to tex_word_processing.py except that it omits multiple occurrence of a word in the TeX file. For example, the word 'quantum' is counted once, and any other occurrence is ignored. This was the first script used during testing, but I later changed it to include all occurrences of each word.

List Creation Scripts

These scripts are used to combine all the field n-gram lists created from the parsing scripts into one list in each field category with the proper n-gram. For example, all the unigram C lists combined into one unigram C lists (C1). All of the python scripts except for Make_Corpus.py will also create a text file with the mean value and standard deviation that

For these scripts I will provide the shell script I used to submit to Slurm and the python script. The shell scripts use the following sbatch options

- partition=himem: These scripts require more memory which can be provided using the high memory partition than the general partition.
- 1 task
- 2 CPUs per task
- 250G per CPU per task: this is required or else it will crash, and core dump files will be created.

You can try with 1 CPU and increasing the memory, but these options worked for me and I just stuck with it. I believe I had over 80% CPU efficiency also.

run_make.sh and **Make_Corpus.py**

These scripts take two arguments:

1. The list field category lowercase (qc, q , or c)
2. The n-gram number (1, 2, 3)

The python script simply finds all lists corresponding to the field category. Then it iterates through them and add the strings and they're occurrence to a dictionary. The dictionary is updated through each iteration. Once every list is read, it sorts the dictionary from highest to lowest count. Then computes its probability by dividing the occurrence of each string by the total number of string occurrences in the list. The final list with probabilities is printed into a text file.

run_mean_cutoff.sh and **mean_cutoff.py**
run_mean_std_cutoff.sh and **mean_std_cutoff.py**

These scripts take one argument:

1. List file name to analyze
   (QC1.txt, QC2.txt, QC3.txt, Q1.txt, Q2.txt, Q3.txt, C1.txt, C2.txt, or C3.txt)

For these scripts I used the mean and the mean plus the standard deviation as cutoffs for each list to filter noise. I used these parameters because strings with the most importance should have a higher probability and the probability distribution is skewed left. I felt the mean was a good cutoff to in order to reduce noise without removing too much meaningful strings. By adding the standard deviation to the mean, more noise was filtered out. A cutoff can be adjusted by changing the cutoff variable to a different value.

The python script mean_cutoff.py simply reads each string in the list and its probability, computes the mean using NumPy's mean function, and then it prints any string whose probability is greater than or equal to the mean. It also prints the probability.

The python script mean_std_cutoff.py simply reads each string in the list and its probability, computes the mean and the standard deviation using NumPy's mean and std functions, and then it prints any string whose probability is greater than or equal to the mean plus the standard deviation. It also prints the probability.

run_filter_rand_str.sh and **filter_rand_str.py**

These scripts take two arguments:

1. List file name to analyze
   (QC1.txt, QC2.txt, QC3.txt, Q1.txt, Q2.txt, Q3.txt, C1.txt, C2.txt, or C3.txt)
2. The cutoff to be used
   (none, mean, mean_std)

This script begins by reading each string the list and probability and adds it to a dictionary. Then it trains a model to detect random string of characters. This is done with the intuition of taking known words in the English language and using the character n-grams of that word. If a string presented to the model has all the character n-gram that the model knows, then it can be considered not a random string of characters. I used trigram of characters because I had set the cutoff of strings with at least a length of 3. A dictionary is used as the data structure that holds the first letter in the trigram as the key and a list of the last two characters in the trigram as the value. For example, for the trigram 'and', the dictionary is

dict = {'a': ['nd']}

This is done with the following steps:

1. Download NLTK's words corpus and iterate through each word.
2. Iterate through each word character by character.
3. Add the first character (string) in each trigram of the word to a dictionary as the key (string). If it does not exist.
4. Append the last two characters in the trigram (string) to the value (list) for its respective key.
5. Repeat until each trigram in the word is read.
6. Repeat until each word is read.

Once the model is trained, simply just iterate through each string in our list and each trigram in that string. If there is a trigram that is not known to the model, then stop the process and it can be considered a 'random' string of characters. The script can also cutoff list further by calculating the mean or the mean plus the standard deviation and using those values as a threshold. Once again, these values can be adjusted by changing the value of the threshold variable.

This script creates three text files:

1. Strings that are considered noise.
2. Strings that are below the cutoff.
3. Strings that are above the cutoff.

Subtraction Scripts

These scripts are basically split up into two different types, removing the frequencies in each list to prepare for subtractions and scripts to actually do the subtractions. These scripts use Unix commands and can actually be done on the command line. I just copied these scripts where each corresponding list was located and submitted these scripts to Slurm. The scripts for c and q lists took longer because the lists were significantly bigger than the qc lists.

The shell scripts use the following sbatch options

- partition=himem: These scripts require more memory which can be provided using the high memory partition than the general partition.
- 1 task
- 1 CPUs per task

Using these option I was able to generate at least 90% CPU efficiency with each script.

c_remove_freq.sh, **q_remove_freq.sh**, and **qc_remove_freq.sh**

These scripts are used to remove the probabilities from each list. Since it is using Unix Commands just make sure that the paths to the files in these scripts are correct. These scripts use the Unix command awk to remove the column containing the probabilities.

subs1.sh, **subs2.sh**, and **subs3.sh**

These scripts are used to do the subtractions for each n-gram lists. The number in the name of the script corresponds to the n-gram lists. The subtractions of importance are:

- QC_C
- QC_Q_C
- Q_C

For these scripts make sure you create a directory call subs so that the resulting subtraction list is created there. You can remove that part in the script, but I kept it there to keep lists organized.

Once again make sure that the paths and names in each script is correct before running. The only thing that should change is the removal of ‘cxg078/Documents’ and replacing it with ‘lab/’.