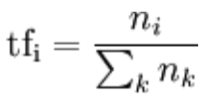
Project 2 Report

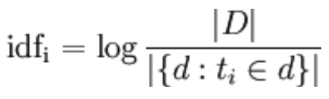
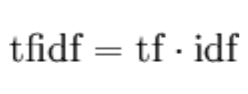
For this Project we had to compute the term-term similarities for a given query term against a large collection of documents. To accomplish this computation we were instructed to use PySpark and it’s mapreduce like structure to deliver results more quickly. Also this problem was to be broken into sub problems, including calculating the TF (Term Frequency), IDF (Inverse Document Frequency) and the term-term similarity or cosine similarity.

To start it is important to know how the core features of Pyspark and many other MapReduce like systems work. The basics rely on functional programming, meaning your code must be constructed in such a way that everything is a function. MapReduce algorithms work by first applying a Mapping function to every element. This mapping typically transforms the data into a format which allows for parallel computing on a distributed system. The next step is to potentially apply a combiner step, which helps organizes data within each node before reducing. The last step is to reduce the data by computing values within each node and then aggregate the results into a single master node. This ability to process data in a distributed and reliable manner, allows for fast computation on very large datasets. The easiest way to remember the core ideology of functional programming is to think in terms of “bringing the code to the data”.

Our implementation starts out by taking in the raw data and transforming it into a more usable form. The initial data is structured such that each line of the input file represents a document with the first word being the document number, and the rest being the contents. Each Term within a line is separated by a space. Our program reads in each line, filters out words which don’t contain the tags “gene\_” or “disease\_” and finally flat maps each line into multiple pairs. The resulting pairs are of the form (Term, (Document, Total Document Term Count)), 1). Once in this form we then reduce each line by key using summation to get the total number of times a given term appears in a document. We now have enough information to complete our first computation which is TF (Term Frequency), the official formula is depicted below.



Next we apply a mapping function which reorders all of the pairs with the term as the key and everything else as the value. This makes pairs of the following form ((Term), (Document, Total count of given term within given document, Total terms within given document)). We then apply an aggregation function to get the count for each term effectively giving us the total number of documents each term appears in within the corpus. This number is then broadcasted to each node for later use. Next we apply a mapping function which reorders the documents to have keys which are term-document pairs and insert the newly computed count into the values each key. The previous transformation gives us lines in the form of ((Term, Document),(Total count of given term within given document, Total terms within given document, Number of documents term appears in throughout corpus)). The next step we take is to apply another aggregation function which counts the total number of unique documents in the corpus and broadcasts it to all the nodes, again for later use. Now we have all the information we need to compute first the IDF (Inverse document frequency) and the TF-IDF which is just Term-Frequency times the Inverse Document Frequency. The official formulas for both of these are depicted below.

To calculate the formulas above we start by simply applying a mapping function to each line which leaves the keys the same and uses the given values to compute the TF-IDF. The results are lines in the form of ((Term, Document), (Number of times a given Term appears in a given Document / Total number of Terms in a given Document) \* Log(Count of the total number of unique documents in the corpus / Total number of documents where a given Term appears).

Now that we have the TF-IDFs for every Term-document pair, we can compute the term-term similarity for a given term and the rest of the corpus. To start, we create a Dictionary containing all of the TF-IDFs for each Document which pertains to the given query. Next we then broadcast this to each node for easy access. To do this we simply apply a filter that only returns lines which have the query term as it’s term then set the document as the key and the TF-IDF as the value. We also create a list which contains all the terms which are related to the query term by having shared at least one document which is listed within the new dictionary we just created. To do this we filter down to only lines which have a document that is also a key in our dictionary and then map each line to just the name of the term. We then collect the results and filter out lines so we are left with only distinct terms. Just like in the previous step we also broadcast this list to each node for easy access. Next we create a dictionary containing the square root of the summation of the squared TF-IDFs which are keyed by a given term. To do this we filter down to just lines which have a term in the related terms list generated from the previous step. We then map each line into a pair that contains the term and it’s squared TF-IDF. Using the results we reduce each line by key using summation then apply a mapping function to the output which creates pairs in the form of (Term, Square-root of summation of all related squared TF-IDFs). These pair are then used to create our last dictionary.

We now have everything we need to compute the final list of term-term similarities sorted by decreasing values. To start we filter down to lines which don’t contain the query term but do have a document in common with the query term. Next we map these lines into the form of ((query term, term), (TF-IDF of the query-term for a given document \* TF-IDF of the given term for a given document)). We then reduce each line by key using summation. Nearing the end, we map the results of each line into the form of ((query term, term), ((Summation of given TF-IDFs)/((square root of the summation of the squared TF-IDFs for the query term)\*(square root of the summation of the squared TF-IDFs for the given term)))). Last we sort the results in descending order based on the new calculated value for each line, giving us the a sorted list of term-term similarities based on a given query term.

In terms of improvement there are a few points in which I think we can improve on. First and most obvious is the broadcasting of the potentially large dictionary. This is could be dangerous if the amount of related terms is too large to be broadcasted or significantly slows down the process. However, at this time we were unable to think of a solution in terms of map and reduce like functions. Another improvement might be to add a couple of combiners at certain steps during the calculation of the TF-IDFs. While combiners aren’t required to achieve the final result, they may potentially speed up the overall time needed to compute the outputs.