CMSC 12300 Project: Sentiment Analysis Using Parallel Computing Methods

Shyamsunder Sriram, Aanya Jhaveri, Katarina Keating. Samantha Stagg

Introduction

Our project, based on analysis of the Yelp dataset, was focused on review content and sentiment analysis. We used the dataset to help create a restaurant suggestion engine (with tf/idf vectors), along with assessing which words were frequently used in conjunction with others (word embeddings).

We tackled these tasks by first prepping our dataset for the work we wanted to do using parallel techniques such as MapReduce. After we prepped our dataset, we conducted our analysis and creation of tf/idf vectors using MPI.

In addition to handling a large raw dataset, our creation of a tf/idf vector per review actually ended up increasing the complexity of the project. In order to handle that, we had to use Google Cloud features such as buckets as well as compressed file storage techniques such as sparse matrices.

This report details how we approached the project, the algorithms we used, the parallel programming techniques we needed, and the challenges along the way.

Dataset Specifications

Total Size: 8.69 GB
Reviews JSON Size: 5.35 GB, 192,609 businesses
Business JSON Size: 138.3 MB, 6.6 million reviews

We needed both the business and reviews JSON files in order to recommend restaurants based on reviews each user wrote.

The business JSON file contained relevant attributes for this that included:

- “business_id”: used to cross-reference with the same field in the reviews JSON
- “name”: name to output as suggestions
- “stars”: filtered to be above 4-stars before outputting a suggestion
- “is_open”: filtered to “1” before outputting a suggestion (to only suggest open restaurants)

The review JSON file contained relevant attributes for this that included:

- “review_id”: used as a unique identifier for each review
- “user_id”: used to group reviews by user
- “business_id”: used to associate to business name etc
- “stars”: only above 4-star reviews used to suggest restaurants
- “text”: review contents analysis was run on

Hypotheses

Question #1: Can we successfully bag words and emphasize words with sentiment over unrelated words?
Approach: Build Tf-idf vectors for each review
Hypothesis: Tf-idf vectors will emphasize sentiment words over regular words

Question #2: Can we identify positive vs. negative words and words that are related to each other?
Approach: Create PMI word embeddings
Hypothesis: Sentiment words will be related to other sentiment words

Question #3: Can we make review recommendations more personalized based on text analysis?
Approach: Cosine similarity of Tf-idf vectors
Hypothesis: Reviews with similar words will yield similar results

Algorithms Used

Algorithm #1: Tf/idf

A tf/idf vectors is a vector representation of a block of text. An *n*-dimensional tf/idf vector contains a vocabulary of *n* words, with each entry corresponding to a frequency calculation for that specific word:

$$Tf(w) * IDF(w)$$

$$\text{where } Tf(w) = \frac{\#\ times\ word\ w\ appears\ in\ a\ review}{total\ \ of\ words\ in\ a\ review}$$

$$\text{and } Idf(w) = \ln \frac{Total\ \#\ of\ reviews}{Total\ \#\ of\ reviews\ that\ contain\ word\ w}$$

We started by first creating a vocabulary of the 20k most frequently used words in reviews that were not "stop words", and then created a 20k-dimensional vector for each review using the formula above.

Algorithm #2 Word Embeddings

Word embeddings are used to determine which words are most frequently used along with other words. We used a skip-gram method (check out the word2vec Wikipedia page for more information), to figure out words that are related to each other. The whole point

of adopting this algorithm was to gain some intuition as to whether sentiment words were related to each other. The way to approach this is to first keep track of word, context pairs for a given window size of all reviews

For a given context "Thai", and a defined window size 2, the corresponding word embedding pairs within a defined window size of 2 are (Thai, I), (Thai, think), (Thai, 55) and (Thai, is). If the window size is 1, then we the only pairs are (Thai, think) and (Thai, 55). We would keep track of such pairs by creating an n x n matrix (where n is the size of our vocabulary) and keep track of all the word pairs (increment corresponding indices (word_index1, word_index2 of the matrix by 1) for word pairs for each review. We used a "window" of size 5, i.e. we looked 5 words to either side of a particular word and created pairs of words within this window of each other.

From further mathematical derivation, we can derive the following formula for computing the matrix of PMI word embeddings, and apply it to our matrix.

$$M_{ij} = v_{w_i}^T v_{w_j} = \log\left(\frac{N^p(w_i, w_j) \cdot N(\mathcal{S}^p)}{N^p(w_i) \cdot N^p(w_j)}\right)$$

Where $N^p(w_i, w_j)$ represents the total number word pairs counts, $N(\mathcal{S}^p)$ represents the total number of pairs in all reviews, $N^p(w_i)$ represents the total number of pairs containing the word (w_i) and $N^p(w_j)$ contains the total number of pairs containing (w_j). The expression is taken with the log to the base e. In our implementation we added one to $N^p(w_i, w_j)$ in order to prevent 0 values.

After creating this matrix, we would take the k-SVD (note k << n) and get three matrices U E and V. Multiplying U * E^(½) We would then get an n x k matrix, where each word is represented by a k dimensional vector. Let's call this matrix M. Finally for a given word we would find the k-dim vector corresponding to that chosen word, and find the other k-dim vectors closest to that word and output those words as similar words.

Finally we could apply the same approach to find analogies of words with one small tweak. For a given input (word0, word1, word2) eg. (awesome, phenomenal, bad) such that word0 is related to word1, and the function outputs words related to word2.

Where the vector in question isn't just a row in matrix M, but instead expressed as mathematical operations of 3 rows in matrix M where V = V_word1 - V_word0 + V_word2, and then find the word vectors that have the closest euclidean distance away from V.

Algorithm #3 Recommendation Engine

After creating tf/idf vectors, we were able to use them to recommend businesses' to users based on those they liked and have positive reviews. We computed cosine similarity scores between positive reviews and took the highest scores' associated businesses as the recommendations.

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

$$\begin{array}{c} \\ \mathbf{doc_1} \\ \mathbf{doc_2} \\ \vdots \\ \mathbf{doc_m} \end{array} \begin{array}{c} \begin{array}{cccc} \mathbf{term_1} & \mathbf{term_2} & \cdots & \mathbf{term_n} \end{array} \\ \begin{bmatrix} tfidf_{11} & tfidf_{12} & \cdots & tfidf_{1n} \\ tfidf_{21} & & & tfidf_{2n} \\ \vdots & & & \vdots \\ tfidf_{m1} & tfidf_{m1} & \cdots & tfidf_{mn} \end{bmatrix} \end{array} \theta \begin{array}{c} \mathbf{doc_{new}} \\ \begin{bmatrix} tfidf_1 \\ tfidf_2 \\ \vdots \\ tfidf_n \end{bmatrix} \end{array} = \begin{array}{c} \\ \mathbf{doc_1} \\ \mathbf{doc_2} \\ \vdots \\ \mathbf{doc_m} \end{array} \begin{array}{c} \mathbf{doc_{new}} \\ \begin{bmatrix} sim_1 \\ sim_2 \\ \vdots \\ sim_n \end{bmatrix} \end{array}$$

Tfidf matrix | Tfidf vector of liked restaurant | Indices of most similar restaurants

Big Data Approaches

Tasks and which approach was used

- Creating vocabulary - MapReduce
- Counting #reviews each word appears in - MapReduce
- Assigning index key to businesses and reviews - MapReduce
- Computing tf/idf vectors - MPI
- Computing cosine similarity scores - MPI

MapReduce:

In order to prep the dataset for tf/idf computations and word embeddings, we needed to first convert the unique IDs provided by yelp into indices we could easily keep track of. Additionally, we needed to do frequency counting for tf/idf and the vocabulary, so we employed MapReduce to handle these tasks.

Estimated Local Runtime: 6+ hours
Local MapReduce Runtime: 3+ hours
Google Cloud MapReduce Runtime: 20 minutes (5 clusters)

MPI:

For the computation-heavy processes that could be paralleled and output customized, we used MPI to compute tf/idf vectors and handle matrix multiplications as needed. In order to have efficient run times, we created 2, 8cpu 30GB processors that did our data analysis. The run times for computing tf/idf vectors through MPI were significantly faster ranging from 3 seconds for a dataset of 21000 entries to around 2 minutes with a larger dataset around 600000 entries).

We also computed word embeddings for a mini 30000 dataset for the top 10000 words, and this took 15 seconds to compute. Overall MPI was significantly faster than map reduce techniques.

Challenges

We faced quite a few challenges with MapReduce, MPI, and connecting both to Google Cloud.

Our first main challenge was setting up Google Cloud to run on a different machine to our VMs. As the dataset was large, some of our group members had to download it on their personal laptops and then set up Google Cloud on them.

After this was done, we faced a lot of authentication errors that were solved mostly by a lot of googling how to set all of the permissions needed to run dataproc jobs with MapReduce.

Finally, at times our computations on Google Cloud were not time intensive, but the outputs generated took an extraordinarily long time to stream back to a laptop (over 9 hours which would time out if the computer went to sleep). This actually helped us to understand which tasks were not MapReduce friendly--we attempted to use it for making tf/idf vectors as there was a lot of counting involved, but the output couldn't be written to a JSON as a numpy array, creating storage problems as it was written as a string instead.

This prompted us to change our computation-side programs to be reliant on MPI instead of MapReduce.

With MPI, we had issues again connecting to Google Cloud (it was actually down for a few hours the weekend before 10th week). With MapReduce, we also had issues with using a runner file on the cloud, as it worked differently to that on a local machine. Secondly there was a limitation of size when we called comm.gather, with larger matrices leading to memory errors. This was not as much of a problem for tfidf. Although it was a 20000 by #num_entries (which was much more than 20000), matrix, this was a sparse matrix that could be compressed using the csr_matrix function from scipy.sparse. However, with the word_embeddings any matrix above 10000 entries faced a memory error, and this matrix could not be converted into a sparse matrix using the csr_matrix method since most of the elements in the matrix were populated.

Results

In the end, we were able to create a recommendation engine and compute related words to words in our vocabulary using our tf/idf and matrix methodology. We needed to use parallel programming in order to achieve these goals due to our large dataset as well as the large size of tf/idf vectors for all of the reviews.

Question #1: Can we successfully bag words and emphasize words with sentiment over unrelated words?

For question 1, the tfidf vectors did a successful job in emphasizing sentiment words. Using a dataset of all yelp reviews in South Carolina in the dataset, we compared all the reviews that got 5 stars and 1 stars and summed up the tfidf vectors and made a word cloud, such that the size of the word is proportional to the tfidf vector value. As you can see below the words of sentiment have a greater size.

Furthermore, the tfidf vector values do recognize positive and negative reviews to a reasonable extent. We labeled the data such that 4+ stars was a positive review and anything else was a negative review. Training logistic regression on a 30000 size dataset we got an 83% accuracy. However, we got only a 79% accuracy rate when we set only 5 star reviews as positive reviews. This indicates that many of the 4 star reviews had positive sentiment words.

Question #2: Can we identify positive vs. negative words and words that are related to each other?

This was partly achieved through the tfidf word cloud but the word embeddings did a remarkable in determining how sentiment words are related. We got related words for sentiment words and for the word ('mexican', we got other cuisine names like and 'indian', 'korean' which makes complete sense.

We also got pretty good results for our analogy for sentimental words, but for other words the analogies didn't make sense. However we got this result with a limited dataset so with a larger window, a larger dataset and more entries we could get a word embedding.

Question #3: Can we make review recommendations more personalized based on text analysis?

We tried our review recommendations for all places in Champaign, IL and we computed this using cosine similarity. Since we recommended only places only 4 stars and above, certain places kept reappearing recommendation after recommendation but there was also a variation of places that varied based on different user ids. This indicates that the cosine similarity technique was successful in providing personalized recommendations to each unique user.

Furthermore, we created the files when substituted properly could perform the same recommendation for other cities in the dataset.