Ankai Lou

CSE 5243 - Data Mining

Instructor: S. Parthasarthy

Report # 4 - Minwise Hashing

Group Members & Contributions

Ankai Lou

- Wrote entire codebase for preprocessing raw sgml input into processable tokens.
- Wrote entire codebase for generating feature vectors using td-idf and normalization.
- Added functionality for paring down original feature vector to 10% size.
- Added functionality for computing Baseline Jaccard Similarity Matrix
- Added functionality for generating hash functions and signatures for $k = 16, 32, 64, 128, 256$
- Added functionality for generating Minhash-estimate Jaccard Similarity Matrices for signatures

Problem Statement

The purpose of this experiment is to estimate the Jaccard similarity of a set of document feature vectors using minwise hashing. Documents in the dataset represent articles in the Reuters database. A minhash sketch will be generated for signature sizes of $k = 16, 32, 64, 128, 256$. The similarity matrix of the signature sets will be compared against the similarity matrix of the raw feature vectors for accuracy and runtime.

Proposed Solution & Rationale

To generate the original feature vector dataset:

- Remove stopwords and terms found in topic/place class labels.
- Lemmatize the remaining words using the built-in NLTK Lemmatizer.
- Stem the remaining word using the built-in NLTK Porter Stemmer.
- Remove sufficiently short stems (less than 5 characters).
- Compute the TF-IDF score for each document-word pair using the built-in NLTK TF-IDF module.
- Select the top 5 words from each document as features to be used in the feature vector.
- Remove duplicate terms from the feature set (performed implicitly in Python).

The top 5 words for each document hold valuable context for determining an article?s purpose and — any fewer would impact the dataset quality. For each feature vector, the values in each vector were remapped as such:

- If the tf-idf score for the document-feature is non-zero, write 1 to the slot — Else, write 0 to the slot.

The above modification from float vectors to boolean vectors was applied to the feature vectors specifically for the purpose of simplifying the minwise hashing process during the baseline computation and signature generation.

To generate the pared-down feature vector:

- Begin with the set of terms generated for the original feature vector.
- Select the top 10% of words in the feature set by averaging the TF-IDF score across all documents.
- Use the features selected by average TF-IDF score as the feature set for the pared feature vectors.

The pared down feature vector was also used in the minwise hashing process in order to observe the impact dimensionality of the dataset has on the performance benefits of using minwise hashing.

For the Baseline, compute the Jaccard Similarity for each distinct feature vector pair as such:

- Compute the number of `1`s in the element-wise `and` of two vector as `num`
  - e.g. `[ 1 0 1 ] & [1 1 0 ] = [ 1 0 0 ]` $\Rightarrow$ `1` is stored in `num`
- Compute the number of `1`s in the element-wise `or` of two vector as `den`
  - e.g. `[ 1 0 1 ] & [1 1 0 ] = [ 1 1 1 ]` $\Rightarrow$ `3` is stored in `den`
- Return the value `num / den` as the Jaccard Similarity of two raw feature vectors

For the minhash sketch for $k = 16, 32, 64, 128, 256$, first generate k distinct hash functions as such:

- Begin with an empty list `functions = []`
- Given some positive integer `k`, loop until `len(functions) = k`:
  - Take `range(0,num_features)`, where `num_features` is the size of a feature vector
  - Randomly shuffle up `range(0,num_features)` and store the new list as `temp`
  - If `temp` is not in `functions`, add it to `functions` — otherwise, continue looping.

Note that for $k = 32$, the 16 hash functions generated for $k = 16$ are used and so on for other iterations of k. This was implemented in order to minimize the amount of time required to generate hash functions as well as to maintain some level of consistency in the experiments for different values of k.

For each document, we will generate the signature using the k hash functions previously generated as such:

- For the signature of document `d`, start with the empty list `signature`
- For each hash function `h` in `functions`:
  - Iterate sequentially through the integers `i` in `h`
  - If the `i`th element of the feature vector for `d` is `1`, write `i` to `signature` and stop iterating `h`
  - Else if the `i`th element of the feature vector for `d` is `0`, do not write and continue iterating `h`
- Return the list `signature` as the signature for `d` — note: `len(signature) == k`

At this point, there should be a collection of signature lists — one for each document. From here, compute the Jaccard Similarity of the document signatures as an estimate for the feature vector similarity:

- Compute the size of the intersection of two document signatures:
  - e.g. `[ 2 3 4 ] & [1 2 6 ] = [ 2 ]` $\Rightarrow$ `1` is stored in `num`
- Compute the size of the union of two document signatures:
  - e.g. `[ 2 3 4 ] & [1 2 6 ] = [ 1 2 4 6 ]` $\Rightarrow$ `4` is stored in `den`
- Return the value `num / den` as the Jaccard Similarity of two raw feature vectors

These similarity estimates are stored as a matrix and compared against the baseline similarity matrix for error. The error is computed as the mean-squared error — i.e., the square root of the sum of squared element-wise differences between the baseline matrix and k-minhash matrix. This is repeated for $k = 16, 32, 64, 128, 256$ for a total of 5 trials for the standard and pared feature vector — 10 trials altogether. The generation of the baseline similarity matrix and each of the k-minhash estimate matrices is timed and reported.

Resources

This assignment was implemented and tested using Python 2.7.5 and used the following libraries/modules:

- os, sys, time
- string
- math
- random
- operator
- BeautifulSoup4
- NLTK
  - nltk.stem.porter.*
  - nltk.corpus.stopwords
  - nltk.corpus.wordnet
  - nltk.stem.wordnet.WordNetLemmatizer

No external libraries or modules were used in the implementation of the baseline or minwise-hash calculation. Everything down to the Jaccard Similarity calculation and hash generation was implemented from scratch.

Experiment

Note that the hash functions are generated randomly each time the code is run, so different runs of the `lab5.py` code may yield different results. The general pattern of the data for varying values of k should be consistent though. The following trials were performed through the code:

(1) Baseline: computation of similarity matrix of the raw feature vectors for the standard dataset.
(2) For `k = 16,32,64,128,256`: generation of minhash sketch and signatures of size `k`
(3) For `k = 16,32,64,128,256`: compute the similarity matrix of the signatures of size `k`
(4) For `k = 16,32,64,128,256`: compute the error of the `k`-minhash estimate against the baseline
(5) Repeat: redo steps (1) through (4) using the pared feature vector

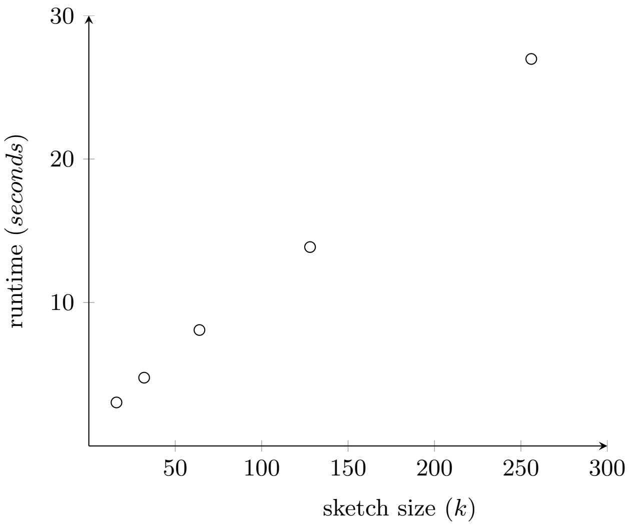
Note that each feature vector dataset has 6 trials each — baseline and for $k = 16, 32, 64, 128, 256$

Results & Interpretation

For the experiment set on the standard feature vector:
For the runtime (efficiency measure) of the similarity matrix and minhash sketch computation:

- The baseline similarity matrix is computed in **116.424607038 seconds**
- The k-minhash sketch and estimate for $k = 16$ is computed in **3.0309510231 seconds**
- The k-minhash sketch and estimate for $k = 32$ is computed in **4.75810003281 seconds**
- The k-minhash sketch and estimate for $k = 64$ is computed in **8.08107209206 seconds**
- The k-minhash sketch and estimate for $k = 128$ is computed in **13.8629560471 seconds**
- The k-minhash sketch and estimate for $k = 256$ is computed in **26.9747610092 seconds**

The sketch size vs runtime plot is provided on the next page:

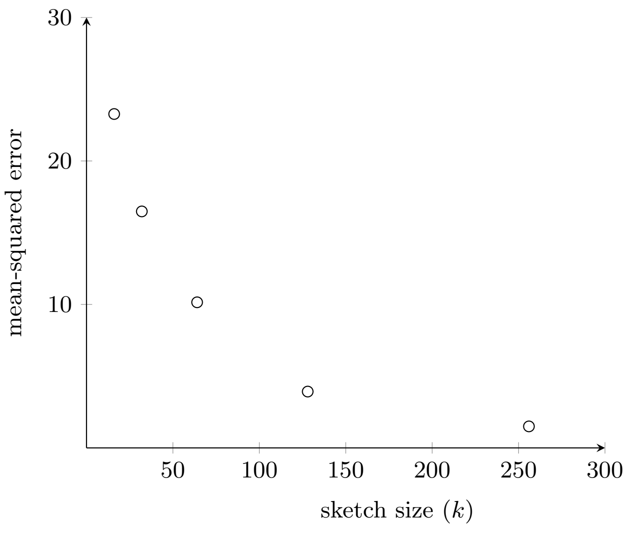


As observed from the plot, the runtime for the minhash sketch computation scales linearly with the sketch size. This is because signature generation scales linearly with the number of hash functions and calculation of similarity two signatures also scales based on signature size — all of which are dependent on k.

For the error (efficacy) of the k-minhash estimates against the baseline matrix:

- The k-minhash sketch and estimate for $k = 16$ has a mean-squared error of **23.2726300064**
- The k-minhash sketch and estimate for $k = 32$ has a mean-squared error of **16.4827202243**
- The k-minhash sketch and estimate for $k = 64$ has a mean-squared error of **10.1477212486**
- The k-minhash sketch and estimate for $k = 128$ has a mean-squared error of **3.92535674193**
- The k-minhash sketch and estimate for $k = 256$ has a mean-squared error of **1.49806417807**

The sketch size vs error rate plot is provided below:



As observed from the plot, as dimensionality of the minhash sketch size increases, the mean-squared error from the baseline matrix decreases. Sketch size and error are inverse proportional, so error and runtime are inversely proportional. This is because more attributes in the signatures leads to more detailed synopses of document contents. It also seems that increasing sketch size leads to greater improvement in error at smaller values of k — e.g. there is a point of diminishing returns for increasing the sketch size k beyond some point.

The results for the pared feature vector are uninteresting and omitted from the report for space considerations. The error for the minhash estimates becomes 0.0 after $k = 16$ and the baseline takes 11.4757399559 seconds to compute so the runtime improvements are negligible (around $3 - 6$ seconds). This suggests that to effectively take advantage of the runtime improvements and minimal error rates via minwise hashing, the dimensionality and size of the data should be large — as small datasets get very little performance improvement.

Interpretation of Output

The implication of these results is that minwise hashing can be used to produce estimates for Jaccard similarity in a document set at substantially shorter runtimes. In its base case at $k = 16$, the runtime estimating similarity for 1000 feature vectors with about 20000 features each is 3.0309510231 seconds vs 116.424607038 seconds to compute the baseline — an improvement by two orders of magnitude. Even the worse case runtime for $k = 256$ finishes in 26.9747610092 seconds — an improvement by one order of magnitude.

From here, the accuracy of the similarity estimates for the k-minhash sketches improves logarithmically as k increases — while runtime only increase linearly. This means there is little reason not to increase the sketch size when k is small — as the performance improvement will be higher — e.g. going from $k = 16$ to $k = 32$ led to a 7 point drop in the mean-squared error. After $k = 128$, the accuracy of the k-minhash sketch is within 5 points of the baseline — which is sufficiently strong of an estimate at 10% of the computation time.

On the other hand, there is a point of diminishing returns as the increase from $k = 128$ to $k = 256$ only yields a 1.5 point drop in mean-squared error. Therefore, the sketch size should not be increased to be arbitrarily large — as eventually the performance improvement will not justify the runtime increases.

Another conclusion is that minwise hashing should only be applied to problems with massive datasets. This is because the results of performing minwise hashing on the pared feature vector set does not lead to runtime improvements that justify the additional logic to compute the document signatures.