While preparing to use the previous labs for this assignment, I came across some issues. I noticed that some of the feature vector words had a TF-IDF value of 0, which should only occur for words that are found in every document. These noise words should've been filtered out. After extensive investigation, it was discovered this was due to the method in which TF-IDF values were selected in lab1. Essentially there was a list of TF-IDF values that were being sorted in the wrong order (ascending vs descending), and the lowest values were being selected instead of the highest. Lab1 was modified to behave as expected and additional tests were ran. This modification resulted in a more sparse, but richer dataset. TF-IDF values near the overall mean were selected to be placed in the output feature vectors to combat sparsity while maintaining meaningfulness. Lab2 was used to determine impact on Classifier accuracy (as a case study into the effectiveness of the new classification). As the tables show, The new methodology is slightly more accurate, though feature vectors take about 2.5x longer to make (Classifier on-line, off-line cost is equivalent).

Bugged Implementation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | KNN (300 words) | Decision Tree (300 words) | KNN (800 words) | Decision Tree (800 words) |
| Avg # Words | 300 | 300 | 795 | 795 |
| Accuracy(%) | 40.38 | 32.33 | 39.76 | 33.63 |

Avg Num Words: Number words used in each Cross Validation Trial

Accuracy: (#correct classes / #correct+#incorrect)

Fixed Implementation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | KNN (300 words) | Decision Tree (300 words) | KNN (800 words) | Decision Tree (800 words) |
| Avg # Words | 300 | 300 | 792 | 792 |
| Accuracy(%) | 41.20 | 37.73 | 53.21 | 46.02 |

Avg Num Words: Number words used in each Cross Validation Trial

Accuracy: (#correct classes / #correct+#incorrect)

Different comparison methods were using to calculate True Jaccard similarity in this lab. At first, the word vectors were treated as a list of booleans indicating if a given word is within a given document. For a two document pair their lists were iterated through and if they contained a word, union size was incremented. If they contained the same word intersection size was incremented. Jaccard similarity was calculated as intersection size divided by union size. This was methodology was inefficient though: it was allowed to run for 5.5 hours and resulted in 50% of the document pair comparisons being completed.

The data was sampled down from ~22k documents to 5k documents (requiring about 1/20 the number of comparisons), and running the original Jaccard similarities still took over 40 minutes. To speed this up, the boolean lists were converted to bit vectors, and Jaccard similarity was calculated by dividing the AND of two bit vectors by their OR values. This was much faster, only taking ~40 seconds for 10k documents containing 300 word vectors. Using the full ~22k data samples resulted in running out of RAM (~7GB).

Varying methodologies were also used for document signature comparisons. Document signatures are the list of hash values the various hash functions return. Even though the signature's size is much smaller than the word vectors (16, 32, 64, 128, 256 vs 300), using list comparisons for signatures still takes much more time than the bit vector comparisons for the original word vectors. Given that signature usage is supposed to improve speed, this cannot be allowed. So the list vectors were converted into bit vectors.

If the signatures are treated as an unordered set, then bit vector conversion is easy. A boolean list of 0's is created and indexes matching signature values are set to 1. This list is then converted to a binary number. Signatures are not meant to be unordered sets though (repeat elements can occur for different hash functions, which is technically different signatures), which makes the bit vectors harder to form. One way might be to convert every value in a signature to binary, concatenate them the values together, and use those for comparisons, but it will not work. Signature values with similar binary representations (2 => 10 , 6 => 110) will appear have some intersection when they should not.

Instead, each decimal signature value can be converted to a bit vector representing its position in the hash, as hash functions have an upper limit: the max bucket value (in this lab it is the number of hash functions used for a given signature). This means each signature value would be represented by NUM\_HASH\_FUNCTIONS bits. So a minwise hash using 4 functions, and a document with signature {0,2,1,3} would result in signature 00 10 01 11. The total number of bits it take to represent a signature in this manner is (NUM\_HASH\_FUNCTIONS–1)\*(NUM\_HASH\_FUNCTIONS). For a 16 hash functions that is 64 bits. For 64 functions it is 384 bits, more than the original word vector and enough to overflow python when attempting to convert these numbers to floats during similarity division. So, there are limitations on this approach but it is still much faster than using list comparisons.

Below are the efficiency and efficacy results of running different Jaccard comparisons.

True Jaccard Similarity

|  |  |
| --- | --- |
| Time to run comparisons (s) | Mean Squared Error (MSE) |
| 36.845 | 0 |

Treating Signature as Unordered Set

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| # hash functions | 16 | 32 | 64 | 128 | 256 |
| Time to create signature (s) | 0.1585 | 0.2842 | 0.525 | 1.0347 | 2.0248 |
| Time to run comparisons (s) | 27.018 | 26.653 | 35.762 | 36.099 | 35.121 |
| MSE | 2070 | 2123 | 2130 | 2169 | 2046 |