Text Document Search

Authors: Matthew Dudley, Mitchell Hodzen, Dalton Moore

Description:

A program that takes a user input string and searches available text files for the most similar documents. Documents are then displayed from most similar to least similar. The user may select a document from the list, which will open that document on their computer.

Process:

Each available document is read, the text is cleaned (remove punctuation and spacing, set all characters to lower case) and stop words are removed (stops words being words such as “and”, “the”, and “as”. These words don’t contribute to the overall theme of the document, so they are removed).

Each word in the documents are assigned an index position, and each document is modeled in a “document vector”. A document vector is a vector where each column (index position) represents a specified word, and the entry of that column represents the frequency of that word in the document. Because this vector will be very sparse (most columns will most likely be 0) we hold this information in a sparse vector: a data structure which holds the index position of non-zero entries and the value of those entries.

After the document vector is created, it is added to the inverted index. The inverted index has a list of all documents which contain a specified word, this is used to narrow down similarity checks (we will only check documents which have at least one word in common with the input).

Once the documents are modeled and the inverted index is populated, we get an input string from the user (what the user searched). We take that string and model it in the same way as the other documents, cleaning it and creating a document vector for it. We then check the inverted index and get a list of all documents that share at least one word in common with the input. Once we have all the documents which may be similar to the input, we calculate the cosine similarity of the input and each document in the list. That list is then sorted, with most similar documents (documents with a cosine similarity closer to 1) in front, and least similar documents (documents with a cosine similarity closer to 0) in the back. This list is then displayed to the user, where the user may select and read any text file displayed.

Github: https://github.com/md05062/CSCI3230-Project-2017

Contribution:

Matthew Dudley: Inverted index (index.cs), front end

Mitchell Hodzen: Document storage (SparseVector.cs, DocumentVector.cs, DocumentVectorGenerator.cs), Document similarity calculation (DocumentSimilarityCalculator.cs), back end interfacing (Backend.cs), front end

Dalton Moore: Document cleaning (StringCleaner.cs), front end

Analysis:

Matthew Dudley:

Index.cs Report:

Index.cs is used to create an index of all words and map them to a corresponding list of DocumentVectors it is contained in.

I chose to use a dictionary with words as keys and a list of DocumentVectors the words are in as values for the index.

The index as a dictionary will speed up information retrieval due to the hashing the Dictionary class uses and we can directly pull the list of documentVectors out by its key which takes **O(1)** in most cases.

Index.cs has a indexPopulate(DocumentVector vector) method that takes in a DocumentVector in order to populate the index. After taking in a DocumentVector I store the words in an array by using the GetDocumentTerms() method in the DocumentVector class. I then loop through the array of words to populate the index. If a word is already contained in the index I simply add the DocumentVector to the list, at the word key, as a value.

If the word is not already in the index, then it will get added as the key and the DocumentVector it came from will get added to the list as the value.

The indexPopulate method has a worst case runtime of **O(k)** where k is the number of words from the DocumentVector. Worst case the method will loop through k number of words and have to add all words as a key.

Index.cs also has a getDocuments(int key) method that returns the list of DocumentVectors by a specific key.

Retrieving a value by using its key is very fast, close to **O(1)**, because the Dictionary class is implemented as a hash table.

This method is used after the user inputs a string or word to search. The DocumentSimularity classes uses the index to retrieve the list of DocumentVectors to perform its test.

Mitchell Hodzen:

Sparse Vector Analysis:

A sparse vector represents a mathematical vector in which a majority of the entries are 0. That is, in any given sparse vector there are relatively few columns (compared to the total number of columns) that have non-zero entries in them.

To save space, and to make computations substantially faster, we can simply hold the index positions where a non-zero value is held and the value in that position.

To do this we use a Dictionary (Note that in .NET a dictionary is actually a map, so it will be referred to as a map for the rest of the analysis) where any given KEY is an index with a non-zero entry, and the VALUE for that key is the value at that index.

Because a majority of the indexes in a sparse vector are 0, this reduces memory overhead and makes calculations such as the norm and the dot product (used to find cosine similarity) much faster.

Any given sparse vector has the following fields:

Dictionary<int, int> internalMap: The internal map where we hold all non-zero indexes and values. The KEY is the index where a non-zero value is held and the VALUE for that key is the value at that index.

Methods:

AddElement(index, value): This method adds values into the vector. First, it checks to see if there is already a value at the given index position.

If there is a value already at that position, we add the given value to the value already there.

If there is not a value already at that position, we add that index position to the internal map and add the given value at that index position.

Because we do this using a map, checking if there is a value at a given key and adding a value to a given key takes (in most cases) constant time, so the runtime complexity of this method is **O(1)**.

GetValueAtIndex(index): This method returns whatever value is in the specified index of the vector.

If the vector has a value at the specified index, that value is returned.

If the vector does not have a value at the specified index, that value is understood to be 0, so 0 is returned.

Because we look up this value using a map, finding the value given the index (key) takes (in most cases) constant time, so the runtime complexity of this method is **O(1)**.

GetNonZeroIndexPositions(): This method returns an array where each element in the array represents an index position in the vector where the value is non-zero. To return these indexes, which is every key in the map, we use the Keys.ToArray method which takes constant time, so the runtime complexity of this method is **O(1)**.

In order to calculate the Cosine similarity of two vectors we need to first define two functions: Norm and Dot product

Note: These two functions are static as they are simply operations that can be performed on sparse vectors.

static Norm(vector): Calculates the norm of a given vector.

To do this we first get an array of each index position where a non-zero value is held by calling the GetNonZeroIndexPosition() function. This operation takes **O(1)** time.

Then, we loop through each index position and add on the square of the value at that position to the total. To loop up those values we use the GetValueAtIndex() function, which takes **O(1)** time. We do this for each index in the array, so in total this summation takes **O(k)** time, where k is the amount of index positions with non-zero values in the vector.

Finally, we take the square root of the total value and return it. This final value is the norm of the passed in vector. This operation takes **O(1)** time.

Overall this method takes linear time, so the runtime complexity of this method is **O(k)**, where k is the amount of indexes in the vector that have non-zero entries.

static DotProduct(v1, v2): Calculates the dot product of two given vectors.

To do this we first get an array of each index position in the first vector where a non-zero value is held by calling the GetNonZeroIndexPosition() function. This operation takes **O(1)** time.

Next, we loop through each index position in that array, multiply the values at that position in v1 and v2 together, and that that to the total. Finding values at an index position uses the function GetValueAtIndex(), which takes **O(1)** time. We do this for each value in the array, so in total this step takes **O(k)** time, where k is the amount of index positions with non-zero values in v1.

Note: Since index positions not held in the internal map are understood to be zero we only need to check one vector (preferably the smaller one), since we only care about values the two vectors have in common. Since all other values are understood to be zero, and zero multiplied by any value is zero, all other summations would go to 0 anyways and not add anything to the total.

Finally we return the total. This total is the final value for the dot product between the two sparse vectors v1 and v2.

Overall this method takes linear time, so the runtime complexity of this method is **O(k)**, where k is the amount of indexes in v1 that have non-zero entries.

Using the two above functions we can now find the Cosine similarity of two given vectors:

static CosineSimliarity(v1, v2): Calculates the Cosine similarity of two given vectors.

To do this we first calculate the dot product of the two vectors by calling SparseVector.DotProduct(v1, v2). This takes **O(k)** time where k is the max length amount of non-zero indexes in a sparse vector.

Next we divide that by the norm of both vectors multiplied together. To do this we call SparseVector.Norm() twice. This takes **O(k)** time each.

We then return this value. This value is the cosine similarity of two given vectors.

Overall this method takes **O(3k)** time, simplified down to **O(k)**, where k is the max amount of index positions containing non-zero entries in a sparse vector.

DocumentVector Analysis:

A document vector models a particular document. This model represents the document as a sparse vector, where every index position (column) in the vector represents a word which appears in at least one of the total number of documents, and the entry for that index represents the amount of times that word appears in this particular document. Because a document will more than likely not contain every word that is present in all documents, a majority of the columns will have a value of 0. this sparse vector is held in the internalVector field. The model also keeps track of the path to the document on the disk. This path is held in the documentLocation field.

Any given document vector has the following fields:

string documentLocation: A string which holds the path on disk of the document which this document vector represents.

SparseVector internalVector: A sparse vector which contains information about what words appear in the document and the frequency of those words.

Constructor:

DocumentVector(documentLocation, internalVector): The constructor sets the internal fields to the values passed in. This takes constant (**O(1)**) time.

Methods:

GetDocumentLocation(): Returns the documentLocation string. This takes constant (**O(1)**) time.

GetDocumentTerms(): Returns an array of integers where every integer represents a word that appears in the document. This method uses in internalVector.GetNonZeroIndexPositions function, which takes constant (**O(1)**) time. Therefore this function takes **O(1)** time.

GetDocumentSimilarity(d2): Finds the similarity between the document and another document, d2. To do this, this function returns the Cosine similarity of the internalVectors of the two document vectors. This function takes **O(k)** time where k is the max amount of index positions containing non-zero entries in a sparse vector. Note that k also represents the number of unique terms in the document.

DocumentVectorGenerator Analysis:

The document vector generator generates a document vector given the path to a document and the text in that document. It also holds a map which contains all words in every document and the index those words are assigned in a map, as well as a HashSet of stop words.

Note: Stop words are words such as "And" "The" and "as" which don't have much meaning in the actual document. If these words are removed then the overall meaning of the document is not changed substantially. Removing these stop words can speed up similarity calculations and make them more accurate.

Each word found while reading a document is assigned a specific integer. That integer now represents that word, and is also used as index positions in a sparse vector (since each column represents a word).

DocumentVectorGenerator is a static class, therefore has static methods and fields

The fields for the DocumentVectorGenerator are:

Dictionary<string, int> termMap: A dictionary (map) of terms where the key is the term and the value is the integer that term is mapped to. From this point forward in the program that term is now synonymous with that integer, and is represented by that integer (used as the index position in a sparse vector)

int termIndex: the index position of the next word to add to the term map.

HashSet<string> stopWords: A hash set of stop words. Used to quickly check if any given word is a stop word.

Methods:

static PopulateStopWordsSet(location): This method reads a set of stop words from a file and adds them to the stopWords hash set.

All of the following is put in a try-catch block to account for the case where the stop words document cannot be found.

First the document is read and assigned to a string. Next, that string is split on newline (since each stop word is separated by a new line). Each individual word in the stop words document is added to an array. This uses the string method Split, which has a runtime complexity of O(m), where m is the max number of characters in a given document.

Then, for each word in the stop words array is added to the stop words hash set. Adding a new entry into the hash set uses the hashset.Add function which takes constant (**O(1)**) time. This loop goes through each word in the document, so this step overall takes **O(n)** time, where n is the max number of words in a document.

If for any reason an error is encountered the error text is output to the console and the hash set is not populated.

Overall, this method has a time complexity of **O(m+n)** where m is the max amount of characters in a document, and n is the max amount of words in a document.

static GetTermIndex(term): Returns the index of a word if it is already in the termMap. If the word is not already in the term map, add it to the term map and return its index.

Checking if the term is in the termMap uses the dictionary method ContainsKey(), which takes constant (**O(1)**) time. After checking if the current term is in the termMap one of two things can happen:

Case 1: The term is already a key in the term map. If this is the case, we return the value associated with that key. Getting the value associated with a key from a map takes (in most cases) constant (**O(1)**) time.

Case 2: The term is not already a key in the term map. If this is the case, we first add the term to the map and find the value for this term using the termIndex field. We then return the new value for this key. Adding a new key and value to the termMap takes (in most cases) constant (**O(1)**) time.

In either case, this method takes constant time, and has a time complexity of **O(1)**.

static GenerateDocumentVector(input, documentLocation): Creates a document vector given a document location and the text of that document.

First, using the StringCleaner.clean method, the text of the document is "cleaned" (spaces removed, punctuation removed, capitalization change to lower case). This method takes **O(m)** time where m is the max number of characters in a given document.

After the text is cleaned, we split the text using the String.split method, which puts each word into an array. This operation takes **O(m)** time as well. We then instantiate a new SparseVector. This will be the vector which models the current document.

Next we loop through each word in the now clean document and do the following:

If the word is a stop word we throw it out. If it is not a stop word we continue. Checking if the word is a stop word by checking if that word is in the stop words hash set. This check take (in most cases) constant (**O(1)**) time.

If the word is not a stop word we do the following:

We find the word index. We do this by calling the DocumentVectorGenerator.GetTermIndex method. This method takes **O(1)** time.

We add that word to the previously intantiated sparse vector. We do this by calling the sparse vector's AddElement method, which takes **O(1)** time.

Since we check each word in the document, in total this will take **O(n)** time, where n is the is the max number of words in a document (as previously stated).

Finally, we create a new DocumentVector using the newly populated sparse vector and the passed-in document location. This document vector is now returned.

Overall, this method takes **O(n+2m)** time, simplified to **O(n+m)** time, where n is the max number of words in a document, and m is the max number of characters in a document.

static GenerateInputVector(input): Creates a document vector which will represent the user input. Note that user input is usually much, much smaller than the documents generated by the method above (since other documents are, in our case, books).

First, the input is cleaned and split in the same way as in the GenerateDocumentVector method. We know from above this process takes **O(2m)** time, where m is the max number of characters in a document (for specifics see the entry for GenerateDocumentVector above).

Next, for each word we do the following:

We check to see if the word is not a stop word (not in the stop word hashset) AND if the word is already in the termMap (if the word is not in the term map there is no point in adding it to the vector, since it will not be compared against any previous document). This operation checks both the stop word hash set and the termmap to see if the term exists in either of them. Checking if a key exists in either data structure takes (in most cases) constant (**O(1)**) time.

Next, if the word is not a stop word and is in the termMap, we add it to the sparseVector which will represent the input, the same way we do when generating documents above. We know from above this process takes constant (**O(1)**) time (for specifics see the entry for GenerateDocumentVector above).

Since we check each word in the document, in total this will take **O(n)** time, where n is the max number of words in a document. Finally, we create a new DocumentVector using the newly populated sparseVector. Since this is user input there is no document path so that field is left null. That document vector is now returned.

Overall, this method takes **O(n+2m)** time, simplified to **O(n+m)** time, where n is the max number of words in a document, and m is the max number of characters in a document.

DocumentSimilarityCalculator Analysis:

The document similarity calculator calculates the cosine similarity of some input document any similar document (I.E any document which contains at least one word from the input). This is returned as an unsorted map where the key is the DocumentVector and the value is the similarity of that document vector to the input vector.

DocumentSimilarityCalculator is a static class, therefore has static methods and fields

The fields for the DocumentSimilarityCalculator are:

Index index: An instance of an inverted index of the documents to compare the input to. We use this index to reduce the number of documents we need to check the input against (If the document has no words in common with the input then the similarity will be 0 anyways, so we don't check it).

Methods:

static SetIndex(index): Simply sets the index instance, takes constant (**O(1)**) time.

static GetDocumentSimilarityMap(inputVector): Returns a map where the key is a documentvector in the index and the value is the cosine similarity of that document to the given input.

The first half of the method searches the index for every document which has some term in common with the input vector.

First, we create a list of the total number of documents to check against. Then, we create a hashset of those documents so we only count each instance once. Then, we populate an array of terms which represents each term that appears in the input document vector. We do this by calling input.GetDocumentTerms() which takes constant (**O(1)**) time.

Next, we loop through each term in the term array and get a list of documents in which that term is present. We add all of those documents to the total list of documents to check.

We find what documents belong to what terms using index.GetDocuments. This operation takes constant (**O(1)**) time.

We append these documents to the current document to check list. To do this we use the List.AddRange method, which takes **O(j)** time, where j is the max number of documents which contain a specified word

We do these operations once for each term in the input document vector. Therefore, this operation takes **O(jk)** time, where k is the max amount of unique words in an input vector (the amount of index positions with non-zero entries in the input vector's internal sparse vector), j is the max number of documents which contain a specified word. Note that in the average case k will be very small because the input string is generally only a few words long.

Next, we loop through each document to check and add them to a hashset. This removes duplicates. To do this we use the hashset.Add method, which takes (in most case) constant (**O(1)**) time. We do this once for every document in the documentToCheck list, so this part takes **O(jk)**, with j and k define above. This is because the maximum number of documents in the list will be the maximum number of documents at each index position multiplied by the maximum amount of unique terms in the input.

Adding these two steps, we find the total runtime complexity of the first half of the method takes **O(2jk)**, simplified to **O(jk)**, with j and k defined earlier in the entry.

The second half of the method calculates the similarity of each document in the set of documents to check to the input document vector. The documents are held as the keys in a map, while their respective values contain the similarity to the input.

For each document in the hash set to check, we do the following operations:

Find the document we are checking by using the hashset.ElementAt method, which returns the current document vector. This operation takes constant (**O(1)**) time.

Calculate the cosine similarity of the input document vector and the current document vector we are checking. This operation uses the inputVector.GetDocumentSimilarity method, which takes **O(k)** time, where k is the number of unique terms in the document. Note that k here is, in practice, the number of unique terms in the input document, which will always be relatively small, as input generally has a very small number of terms.

We do these operations once for each documentVector in the documents to check hash set. Because of this, the runtime complexity of this step is **O(gk)**, where k is the amount of unique words in a document vector (in practice, the input vector), and g is the number of documents in the index that share at least one term with the input document vector.

Adding the two half together, we find that the total runtime complexity of this method **is O(jk + gk)**, simplified to **O(k(j + g))**, where k is the max number of unique terms in a document vector, j is the max number of documents which contain a specified word, and g is the number of documents in the index that share at least one term with the input document vector.

Backend Analysis:

The backend class file is how we allow the front end to interface with the backend, and how we tie everything together.

The Backend class has two things:

A constructor, which reads the documents from the resources folder, creates document vectors out of them, and populates the inverted index with those documents

A GetDocumentSimilarityList method which returns a sorted list of key value pairs, where the key is the document and the value is the similarity to the given input. This list is sorted by most similar to least similar.

Constructor:

DocumentVector(location): The constructor reads all text documents from a resource folder, generates document vectors for those documents, and populates the inverted index with those documents. First, we populate the list of stop words given a stop words text file. See DocumentVectorGenerator for more information. We do this by using the DocumentVectorGenerator. Overall this operation takes **O(m + n)** where m is the max amount of characters in a document, and n is the max amount of words in a document. Then, we get the file locations of each document in the resources folder.

We then loop through each file location and do the following:

First, we extract the text from the file. We do this using File.ReadAllText.

Next, we generate a document vector for this document using the DocumentVectorGenerator.GenerateDocumentVector function, which returns the document vector created from the given document. This method takes **O(m + n)** time, where m is the max amount of characters in a document, and n is the max amount of words in a document.

Finally, we populate the index given the new document vector. For each unique word in the document vector we add the document to that place in the index. See the index analysis for more information. We do this using the index.indexPopulate method, which takes **O(k)** time, where k is the max amount of unique words in a given document.

If there is any error reading the given file, we print a message and don't read it.

Each time we do this it takes in total **O(m + n + k)** time, where m, n, and k are defined above.

Since we do this for each document in the resources folder, in total this section takes **O(w(m + n +k))** where m, n, and k are defined above, and w is the max amount of individual documents that can be found in the resources folder.

Finally, we set the index for the documentSimilarityCalculator. This takes constant (**O(1)**) time.

Overall, the entire method takes **O(w(m + n + k) + m + n)** simplified to **O(w(m + n + k))**

w = the maximum amount of documents in the resource folder

m = the maximum amount of characters in a given document

n = the maximum amount of words in a given document

k = the maximum amount of unique words in a given document.

Methods:

GetSortedSimilarityList(input): Returns a sorted list of key value pairs. The key is the document vector and the value is the similarity to the given input. The list is sorted from most similar document to least similar document.

First, we generate a document vector for the input. We do this by using the DocumentVectorGenerator.GenerateInputVector function. This returns a document vector which models the given input. This operation takes **O(m + n)** time, where m is the max number of characters in a document, and n is the max number of words in a document.

Next, we generate a similarity map where each key is a different DocumentVector and each value is the cosine similarity of that document to the input. We do this by using the DocumentSimilarityCalculator.GetDocumentSimilarityMap function. This operation takes **O(k(j + g))**, where k is the max number of unique terms in a document vector, j is the max number of documents which contain a specified word, and g is the number of documents in the index that share at least one term with the input document vector.

Next, we turn that map into a List of key value pairs. We do this using the Dictionary.ToList method, which takes constant (**O(1)**) time.

Finally, we sort and return this list. We sort the list using the List.sort method to compare the values of each key-value pair. This type of sorting uses the quicksort algorithm, which has an average runtime of **O(g log (g))**, where g was defined above.

In total, this method has a runtime complexity of **O(m + n + k(j + g) + g log(g))** where:

m = the maximum amount of characters in a given document

n = the maximum amount of words in a given document

k = the maximum amount of unique words in a given document

j = the maximum amount of documents which contain a specified word

g = the maximum amount of documents which share at least one word with the input

Dalton Moore:

StringCleaner Analysis:

Line 15 sends Regex.Replace a string to parse through, a pattern of white spacing to look for, and a string to replace each occurrence with. Takes **O(m)** time where m is the number of characters in the first string.

Line 16 sends Regex.Replace a string to parse through, a pattern of white spacing and words to look for, and a string to replace each occurrence that aren't a space or a word. Takes **O(m)** time where m is the number of characters to search through.

Both lines go character by character.

Line 17 takes the string and puts all of its characters to lowercase.

References:

Project Gutenberg; free ebooks. Used for the text documents we search: https://www.gutenberg.org/