NLCT : Towards Developing a Natural Language Processing Toolkit Library in C Language

Abhijith Venugopal
Dept of Computer Science
PES University, Bangalore
01FB16ECS008

Abhishek Narayanan
Dept of Computer Science
PES University, Bangalore
01FB16ECS016

Rachana Aithal K R
Dept of Computer Science
PES University, Bangalore
01FB16ECS483

***Abstract*— This project is aimed at developing a Natural Language Processing Toolkit Library for C Language from scratch with the attempt of providing robust and efficient utilities for performing various extensive Natural Language Processing tasks, comparable to existing state-of-the-art libraries such as NLTK framework in python. To our best knowledge, such a framework is not available in C language to be used as open source utilities in any of the online forums such as Github, etc. In order to achieve the task at hand, we have explored various extensively used NLP algorithms and implemented them from scratch in C and combined them into this library which can conveniently be imported and used to perform complicated NLP tasks with just an API call. In our project, we classify NLP broadly into three categories, namely Text Pre-processing, Feature Extraction and Text Similarity and implement various algorithms elaborated later to tackle each of these problems in NLP.**

I. INTRODUCTION

In computer science, a library is a collection of non-volatile resources used by computer programs, often for software development. These may include configuration data, documentation, help data, message templates, pre-written code and subroutines, classes, values or type specifications. A library is also a collection of implementations of behavior, written in terms of a language, that has a well-defined interface by which the behavior is invoked. For instance, people who want to write a higher level program can use a library to make system calls instead of implementing those system calls over and over again. Therefore a library for Natural Language Processing in C would be very convenient for building applications like Search Engines, Information extractors, Question answering agents and so on. As C language does not provide a built in NLP library, creating such a library would help to make creation of various applications in C a much more convenient task. We have put together a library incorporating common NLP processes such as file cleaning, stop words removal, tokenizing, vectorising, text similarity and stemming. These functions are necessary for pre-processing of large files for NLP applications such as sentiment analysis and other language applications.

For the implementation of these operations, different data structures and algorithmic principles were used. Steps were taken to ensure that unnecessary operations do not take place and memory is not wasted needlessly. Therefore some amount of optimization was done to ensure that these functions are implemented in such a way so as not to make them processor and memory intensive.

The NLCT Library, which we have developed has been largely classified into three main components, namely text pre-processing, feature extraction and text similarity. Text preprocessing and Normalization is composed of a sequence of steps for processing and cleaning of raw text data as elaborated further. The first step would be File cleaning, which involves going through the file and removing special characters and unnecessary symbols from the document which do not contribute to the meaning of the document. Secondly, we tokenize the cleaned raw text for convenience in further processing, which involves breaking up the long sentences into tokens which makes accessing relevant words much easier. Next, often Stop word removal is performed which filters the document of stop words such as 'a', 'any' and so on, which are irrelevant for any NLP applications such as search and other queries as they convey little significant meaning which may be neglected. We also implemented the Porter Stemming Algorithm, which is used to normalize text and reduce words with the same stem to a common root word, since those words essentially convey similar meaning often, but used in a different context. Some of these steps are optional and are based on the user's requirement for the task at hand and hence the individual API calls for each of the above mentioned step, provide developers flexibility of choice of preprocessing methods to use.

The second module for feature extraction is another important utility for Machine Learning purposes such as classification or clustering. Since machines (which usually are programmed to operate only on numerical or binary numbers) are not capable of understanding or processing raw text data, text has to be numerically represented as vectors. Vectorising involves various techniques, one of which is the bag of words model of numerically representing sentences.

The third module for Text similarity involves comparing different pairs of sentences to get a quantitative measure to determine how "close" or similar the two sentences are.

In order to evaluate our implementation of the algorithms, we compare the execution time involved in each of the above mentioned steps with existing benchmark and state-of-the-art libraries such as NLTK in python. We also establish a pipeline involving the three modules to find pair-wise similarity between a set of documents and evaluate the results obtained based on the comparison made.

II. METHODOLOGY USED

The various utilities available in our library have been broadly classified into three modules as elaborated in further subsections.

A. Text Pre-processing and Normalization

1) File reading and cleaning: The user file is read and then cleaned by passing it through a filter function which removes unnecessary symbols and special characters, which do not contribute to the semantics of the document. Care is also taken to remove additional spaces, which may be wrongly taken as a separate word, if left in our client file. The modified file is then updated to the same name as the original file, so that the cleaned file will be used for further operations.

2) Tokenization: Tokenization in NLP is the process of taking the entire text or document and fragmenting it into individual words or units, based on a delimiter. Parameters have been defined for maximum number of tokens(MAXTOKENS) and buffer size(MAXLINE), which can be altered according to the user's necessity. A copy of the original string is maintained and used for processing as a safety measure. A set of delimiters have been specified based on which the document is tokenised. Functions for tokenizing a single line of text as well as a full file have been implemented, so that the user can use one of them as per their requirement. The result of tokenization is stored using a two-dimensional array, consisting of pointers, pointing to the words.

3) Stop words removal: Filtering out useless data is one of the chief forms of pre-processing text for Natural Language Processing. In the field of natural language processing, such useless words, are referred to as stop words, which are commonly used words (such as the, a, an, in), which NLP application frameworks like search engines are often programmed to ignore, since they convey little significant meaning which may be neglected.
Often, we would not want such words consuming space in our database, or consume valuable processing time. Hence, we find a significant need for the development of a robust and quick algorithm to filter top words.
In order to achieve this purpose, we use the Trie data structure, which is an efficient data structure information retrieval, using which search complexities can be brought to the order of the length of the key to be searched.In contrast to a binary search tree, which when well balanced will need time proportional to M * log N, (where M is maximum string length and N is number of keys in tree), using Trie, we can search the key in O(M) time, though there is some penalty on storage requirements.
In order to filter out stop words, we build a dataset of stop words similar to that used by state-of-the-art libraries like NLTK in python and insert all of these words in a Trie data structure. The file to be cleaned is then read line wise and each line is tokenized using the previously developed tokenizer utility. Each of the tokens thus generated are searched for in the Trie of stop words and stored into a new file, provided that the search is unsuccessful.

4) Stemming: For grammatical reasons, documents use different forms of a word, such as organize, organizes, and organizing. Additionally, there are families of derivationally related words with similar meanings, such as democracy, democratic, and democratization. Processing on documents with several forms of the same word may become computationally expensive and giver results of low accuracy. Stemming is a process that chops of the ends of words to get the word to a base format. Therefore stemming can be done to reduce inflectional forms and sometimes derivationally related forms of a word to a common base form. The stemming model used in our library is Porter's Algorithm. It is based on the idea that the suffixes in the English language (approximately 1200) are mostly made up of a combination of smaller and simpler suffixes. Porter stemming consists of five phases of word reduction which are performed in a sequential manner. Within each phase there are various conventions to select rules, such as selecting the rule from each rule group that applies to the longest suffix. If a rule is accepted, the suffix is removed accordingly, and the next step is performed. The resultant stem at the end of the fifth step is returned. The rule looks like the following:
condition(suffix)== new suffix
Therefore if the word has ”SSES”, the rule for ”SSES” will be applied instead of the rule for ”S” or ”ES”. Some of the rules also use the concept of ”measure” of a word, which loosely checks the number of syllables to see whether a word is long enough that it is reasonable to regard the matching portion of a rule as a suffix rather than as part of the stem of a word. Hence the rule should change ”replacement” to ”replac” but not ”cement” to ”c”. The stemming conversion is based on the principle that the words that are being brought to the same form are semantically related. The stem need not be an existing word in the dictionary but all its variants should map to this form after the stemming has been completed. Therefore the functions needed for the five phases have been implemented and the Stem function takes a word and stems it.

B. Feature Extraction

1) Bag-of-Words Model: The bag-of-words model generates a simplified vector representation of text. In this model, a text (such as a sentence or a document) is represented as the bag (multiset) of its words, disregarding grammar and even word order but keeping multiplicity.The bag-of-words model is commonly used in methods of document classification where the (frequency of) occurrence of each word is used as a feature for training a classifier.
In order to implement this, the preliminary step involved is to find k most frequent distinct words in a document to generate the vocabulary needed to form the 'bag'.
A simple solution is to use Hashing. Hash all words one by one in a hash table. If a word is already present, then increment its count. Finally, traverse through the hash table and return the k words with maximum counts. We have

used Trie and Min Heap to get the k most frequent words efficiently. The idea is to use Trie for searching existing words adding new words efficiently. Trie also stores count of occurrences of words. A Min Heap of size k is used to keep track of k most frequent words at any point of time.
Once the bag has been obtained, we read an input test sentence from a file or from the user and search for each of the words present in the vocabulary generated in the sentence and place the frequency of that word in a corresponding feature array at the suitable index. We have used suffix arrays for searching and counting occurences for this task. When run on the training file itself, the model is capable of generating a feature matrix for training, ready to be passed to any machine learning classifier.

2) TF-IDF Model: Term-frequency-inverse document frequency (TF-IDF) is another way to judge the topic of an article by the words it contains. With TF-IDF, words are given weight TF-IDF measures relevance, not frequency. That is, wordc ounts are replaced with TF-IDF scores across the whole dataset.

First, TF-IDF measures the number of times that words appear in a given document (term frequency). But because words such as "and" or "the" appear frequently in all documents, those are systematically discounted. Thats the inverse-document frequency part. The more documents a word appears in, the less valuable that word is as a signal. Thats intended to leave only the frequent and distinctive words as markers. Each words TF-IDF relevance is a normalized data format that also adds up to one.

$$w_{i,j} = tf_{i,j} * (1 + log\frac{N}{df_i}) \tag{1}$$

Where $\mathrm{tf}_{\mathrm{i,j}}$ is the number of occurrences of i in j, $\mathrm{df}_{\mathrm{i,j}}$ refers to the number of documents containing i and N is the total number of documents. We have used the feature matrix output of the Bag of Words model to compute term frequencies and inverse document frequencies and compute the tf-idf weights.

3) Representation of Feature Matrices in Memory: Feature matrices generated by bag of words or TF-IDF model are often high dimensional and sparse, consuming lot of memory due to presence of several zeros in them. In order to optimize memory requirements, we store the feature matrices using the Compressed Sparse Row representation of sparse matrices as elaborated further. We represent a matrix M (m * n), by three 1-D arrays or vectors called as A, IA, JA. Let NNZ denote the number of non-zero elements in M and note that 0-based indexing is used.

- The A vector is of size NNZ and it stores the values of the non-zero elements of the matrix. The values appear in the order of traversing the matrix row-by-row
- The IA vector is of size m+1 stores the cumulative number of non-zero elements upto (not including) the i-th row. It is defined by the recursive relation :
  - IA[0] = 0
  - IA[i] = IA[i-1] + no of non-zero elements in the (i-1) th row of the Matrix
- The JA vector stores the column index of each element in the A vector. Thus it is of size NNZ as well.

C. Text Similarity

Lexical similarity between two strings is essential in multiple NLP applications like clustering, redundancy removal and information retrieval. We provide multiple variations of calculating similarity.

- *Cosine Similarity*: Cosine similarity takes in two vectors of equal lengths that represent the text and give the similarity between them. This is used when we have the vector representation of the text with the word frequencies.
- *Jaccard Similarity*: The Jaccard similarity measures similarity between finite sample sets and is defined as the cardinality of the intersection of sets divided by the cardinality of the union of the sample sets. This can take the output of bag of words as the input.
- *Levenshtein Distance*: The Levenshtein distance is the minimum edit distance with 3 edit options: insert, remove or replace. It uses dynamic programming. It can be optimized to work with only an array with the size of one of the words instead of a 2-dimensional array of size of the product of both the words. The 2-dimensional array constructed can be used to derive the edits required to change one string into another.

III. RESULTS

In this project, we have implemented from scratch, the necessary state-of-the-art NLP algorithms used commonly by developers and researchers around the world. To our best knowledge, this library is novel and there is no existing implementation of any such library in C language, as compared to other programming languages such as C++ or python.
Another reason to choose C language for our implementation is that C is a Mid-level language, i.e. it is closer to the way a processor work than more modern language, so it will be fast and efficient even with a poor compiler. However, python, being interpreted and high level language, might prove slower for certain applications, though the language has its own pros in other areas.

A. EVALUATION

1) Execution Time consumed: In order to evaluate our library, we measure the execution time consumed for each of the above mentioned operations and compare the same with python's state-of-the-art libraries such as nltk or scikit-learn, in which such functionalities are already available.
The results obtained have been summarized in Table I and it is clearly inferred from this table that our implementation has been found to be more efficient in

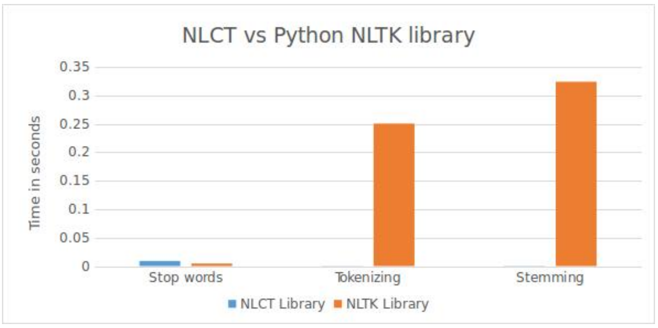

Fig. 1. Comparison of execution time for pre-processing techniques, Not that Cleaning and stop word removal has been found to be more efficient using our approach

terms of execution time than these tested existing libraries. The comparison done has been clearly portrayed in figures I.

2) Pipeline Evaluation: In order to test the results of our models and API, we create and establish two pipelines, one written in C language and invoking the library API's of our own NLCT implementation, and the other using python's inbuilt libraries in which a similar functionality as we implemented is present. For instance, nltk library possesses API's for text pre-processing while vectorization/ feature extraction or text similarity metrics are conveniently usable in the scikit-learn library.
The target of this pipeline is to achieve computation of text similarity between two documents of moderate size of about 190kB which is the sample dataset used for our evaluation metric. We execute the pipelines and compare the results and execution time consumed by both languages to deliver the end result.

EVALUATION OF EXECUTION TIME		
Algorithm	C NLCT library	python nltk/sklearn
File reading tokenization	0.000345	0.250589
Stop words filtering	0.008962	0.004481
Stemming	0.000509	0.324141
Bag of Words vectorizer	0.008962	0.043227
Cosine similarity	0.000008	0.000212
Pipeline for document similarity	0.394864	1.859305

The comparison of the execution time has been clearly portrayed in figure III. We observe the results obtain and draw certain inferences regarding what drawbacks our NLCT implementation might still possess. It was observed that the operations such as pre-processing, stop word removal and stemming using Porter's algorithm were all at par and produced very similar results. The implementation of similarity algorithms are all standard based on their definition, and only depend on the feature vectors supplied to produce output. This is where some difference arises,

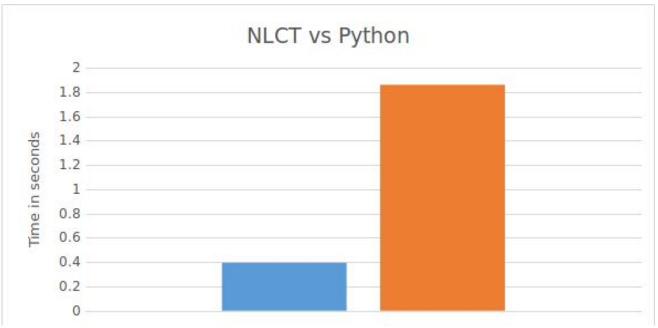

Fig. 2. Comparison of execution time for document similarity pipeline

since our feature extraction metrics involve an assumption that the bag or vocabulary containing k most frequent words is representative of the whole text including all the documents in it. Though the bag will never include stop words, which are quite frequent often and convey no meaning, it was found that the bag may not be a representative sample in case of varying document sizes and most of the words in the bag might be skewed towards one specific large document, possibly containing k most frequent and distinct words in the whole file.
However, in contrast to this approach, the implementation in scikit-learn chooses the best features which might have significant impact using feature selection and dimensional reduction techniques, thereby creating small gap between our features and python generated features. However, our assumption holds good in a wide variety of cases and the features generated have been found to be viable enough for any practical applications as such.

References

[1] Vijayarani, S., Ilamathi, M. J., Nithya, M. (2015). Preprocessing techniques for text mining-an overview. International Journal of Computer Science Communication Networks, 5(1), 7-16.
[2] Lovins, J. B. (1968). Development of a stemming algorithm. Mech. Translat. Comp. Linguistics, 11(1-2), 22-31. Chicago
[3] Willett, P. (2006). The Porter stemming algorithm: then and now. Program, 40(3), 219-223.
[4] Srividhya, V., Anitha, R. (2010). Evaluating preprocessing techniques in text categorization. International journal of computer science and application, 47(11), 49-51.
[5] Sriram, B., Fuhry, D., Demir, E., Ferhatosmanoglu, H., Demirbas, M. (2010, July). Short text classification in twitter to improve information filtering. In Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval (pp. 841-842). ACM.
[6] Martins, C. A., Monard, M. C., Matsubara, E. T. (2003). Reducing the dimensionality of bag-of-words text representation used by learning algorithms. In Proc of 3rd IASTED International Conference on Artificial Intelligence and Applications (pp. 228-233).
[7] Wallach, H. M. (2006, June). Topic modeling: beyond bag-of-words. In Proceedings of the 23rd international conference on Machine learning (pp. 977-984). ACM. Chicago
[8] Mihalcea, R., Corley, C., Strapparava, C. (2006, July). Corpus-based and knowledge-based measures of text semantic similarity. In AAAI (Vol. 6, pp. 775-780).
[9] Metzler, D., Dumais, S., Meek, C. (2007, April). Similarity measures for short segments of text. In European conference on information retrieval (pp. 16-27). Springer, Berlin, Heidelberg. Chicago
[10] Gomaa, W. H., Fahmy, A. A. (2013). A survey of text similarity approaches. International Journal of Computer Applications, 68(13), 13-18.