Automatic text summarization (https://en.wikipedia.org/wiki/Automatic_summarization) is a process that takes a source text and presents the most important content in a condensed form in a manner sensitive to the user or task needs. The importance of having a text summarization system has been growing with the rapid expansion of textual information available on-line (web pages, news articles, e-mail messages, tweets, or even reviews of products and services). One major differentiation between the different summarization approaches is the input, which can either be a single document or multiple documents. In this project we are interested in the automatic summarization relies on multiple input documents that are somehow related (e.g., about the same topic). Another key differentiation of existing approaches is how the summary is created: this can either be a concatenation of extracted sentences or an artificially generated text, based on the sentences and information included in the input documents.

Text summarization using MapReduce

In this project, you are asked to implement a MapReduce version of a multi-document summarization system based on the extractive approach. Specifically, you are asked to implement the system that is described in [1]. A textual corpus of around 4000 legal cases for automatic summarization is selected for performing the experiments, the dataset is available on UCI machine learning repository (https://archive.ics.uci.edu/ml/datasets/Legal+Case+Reports). The dataset contains Australian legal cases from the Federal Court of Australia (FCA) all files from the year 2006, 2007, 2008 and 2009.
As depicted in Figure 1 the summarization task is performed in four different stages and provides a modular system of multiple documents summarization:

(a) The first stage is the document clustering (https://en.wikipedia.org/wiki/Cluster_analysis) stage where a text clustering algorithm (e.g., k-means) is applied on the multi document text collection to create the document clusters. The purpose of this stage is to group the similar text document for making it ready for summarization and ensures that all the similar set of documents participates as a group in summarization process.

(b) In the second stage, Latent Dirichlet Allocation (LDA) (https://en.wikipedia.org/wiki/Latent_Dirichlet_allocation) topic modeling technique is applied on each individual text document cluster to generate the cluster topics and important terms belonging to each cluster topic.

(c) In the third stage, global frequent terms are identified across all document clusters, as well as Semantic Terms, i.e., synonyms of the frequent terms using lexical ontologies or thesauri like Wordnet.

(d) In the last stage, sentence filtering is performed from each individual input text document on the basis of frequent and semantic similar terms generated from previous stage. For each document the sentences which are containing the frequent terms and semantic similar terms to the frequent terms are selected for participation in the summary document. Finally, the duplicate and near duplicate sentences are identified and kept only once in the final summary document generated.

In this project, you will implement those stages in increasing order of complexity (c), (a), (b) and (d).

**Phase 1 (5%): Frequent and Semantic Terms (Stage (c)) – Deadline: October 10**

The first phase of the project is almost as easy as the word count example in MapReduce. You are given a set of documents, as a collection of words, and you are asked to find the \( k \) most frequent words (assume \( k = 15 \)). Once you find the \( k \) most frequent words, you can use the Java API of Wordnet, to find the synonyms of those words. The set of frequent words (terms), along with their synonymous words will be the output of this phase (order is not important).

**LINKS:** The Java API of Wordnet can be found here: https://sourceforge.net/projects/jwordnet/. Example usage to get synonyms: http://stackoverflow.com/questions/13822141/how-can-i-find-synonyms-in-estimated-frequency-order-using-jwnlwordnet-library

**Phase 2 (20%): Clustering (Stage (a)) – Deadline: October 24**

In the second phase of the project, you have to partition the given text corpus to clusters, using k-means. To do that, you have to transform each document to a vector, pick \( k \) random documents as cluster centers (assume that \( k = 5 \)), and then apply a similarity (or distance) function between each cluster center and each other document, to assign every document to its closest cluster center. The cluster centers are then re-computed (as the average of the points in a cluster) and the process is repeated, until convergence, i.e., until no document changes cluster.

To transform a document to a vector, we will use the TF-IDF model. The \( tf \) (term frequency) of a word \( w \) is the number of times it appears in a document, normalized by the maximum frequency of a word met in this document (i.e., the most frequent word in a document will have a \( tf \) of 1 after the normalization). The document frequency of a word \( w \) is the fraction of documents in our corpus \( D \) that contain \( w \).
inverse document frequency (idf) of \( w \) is the inverse of the previous fraction. Summarizing, the term frequency of a word \( w \) in a document \( d \) is:

\[
tf(w, d) = \frac{freq(w, d)}{\max_{t \in d}(freq(t, d))},
\]

where \( freq(w, d) \) is the number of times the word \( w \) appears in a document \( d \), and the inverse document frequency of a word \( w \) in a document corpus \( D \) is:

\[
idf(w, D) = \log \frac{|D|}{|\{d \in D | w \in d\}|}.
\]

Then, the tf-idf vector of a document \( d \in D \), is defined as a set of \( w:tf(w, d) \cdot idf(w, D) \) pairs.

Two vectors can be compared using the cosine similarity measure, defined as:

\[
\text{cosine}(d_1, d_2) = \frac{d_1 \cdot d_2}{||d_1|| \ ||d_2||} = \frac{\sum_{w \in d_1} tf(w, d_1) \cdot idf(w, D) \cdot tf(w, d_2) \cdot idf(w, D)}{\sqrt{\sum_{w \in d_1} (tf(w, d_1) \cdot idf(w, D))^2} \ \sqrt{\sum_{w \in d_2} (tf(w, d_2) \cdot idf(w, D))^2}}.
\]

**Example**: The tf-idf representation of the document \( d_1: \text{“to be or not to be”} \) can be found as follows, assuming that the word “to” appears in 10 documents in our 10-document corpus \( D \), the word “be” appears in 2 documents, the word “or” appears in 5 documents, and the word “not” appears in 8 documents.

\[
\begin{align*}
\text{tf(“to”, } d_1) & = 2/2 = 1 \\
\text{tf(“be”, } d_1) & = 2/2 = 1 \\
\text{tf(“or”, } d_1) & = \frac{1}{2} = 0.5 \\
\text{tf(“not”, } d_1) & = \frac{1}{2} = 0.5 \\
\text{idf(“to”, } D) & = \log (10/10) = 0 \\
\text{idf(“be”, } D) & = \log(10/2) = 0.7 \\
\text{idf(“or”, } D) & = \log(10/5) = 0.3 \\
\text{idf(“not”, } D) & = \log(10/8) = 0.1
\end{align*}
\]

Then, \( d_1 \) can be represented as the vector:

\[
d_1 = \{\text{be:0.7, or:0.15, not:0.05}\}
\]

(to:0 is omitted).

If \( d_2: \text{“not to be”} \) is another document in the same corpus, then it can be represented as:

\[
d_2 = \{\text{be:0.7, not:0.1}\},
\]

and the cosine similarity between \( d_1 \) and \( d_2 \) is:

\[
\text{cosine}(d_1, d_2) = \frac{0.7 \cdot 0.7 + 0.05 \cdot 0.1}{\sqrt{(0.7^2 + 0.15^2 + 0.05^2)^2} \ \sqrt{(0.7^2 + 0.1^2)^2}} = 0.98.
\]

The centroid of a cluster can be computed by the average of the existing points in the cluster. For example, if a cluster \( c \) consists of \( d_1 \) and \( d_2 \), its new centroid can be found as:

\[
\text{centroid}(c) = \frac{\{\text{be:(0.7+0.7)/2, or:(0.15+0)/2, not:(0.15+0.1)/2}\}} = \{\text{be:0.7, or:0.08, not:0.13}\}.
\]

**LINKS**: Use the implementations of mahout for k-means: [https://mahout.apache.org/users/clustering/k-means-clustering.html](https://mahout.apache.org/users/clustering/k-means-clustering.html). You are free to choose any other available Map/Reduce implementation (see for example [2]).
Phase 3 (25%): Latent Dirichlet Allocation (LDA) (Stage (b)) – **Deadline: November 7**

Latent Dirichlet Allocation (LDA) is an effective, scalable approach to modeling a large text corpus. LDA is an unsupervised algorithm for performing statistical topic modeling that uses a “bag of words” approach, treating each document as a set of unordered words. As we can see in Figure 2, each document is represented as a probability distribution over some topics, and each topic is a probability distribution over words (for details see [5]).

**Figure 2**: LDA Topic Modeling of Textual Corpus

**Example**: Suppose you have the following set of sentences:

I like to eat broccoli and bananas.
I ate a banana and spinach smoothie for breakfast.
Chinchillas and kittens are cute.
My sister adopted a kitten yesterday.
Look at this cute hamster munching on a piece of broccoli.

What is latent Dirichlet allocation? It’s a way of automatically discovering topics that these sentences contain. For example, given these sentences and asked for 2 topics, LDA might produce something like:

- Sentences 1 and 2: 100% Topic A
- Sentences 3 and 4: 100% Topic B
- Sentence 5: 60% Topic A, 40% Topic B

  - Topic A: 30% broccoli, 15% bananas, 10% breakfast, 10% munching, … (at which point, you could interpret topic A to be about food)
  - Topic B: 20% chinchillas, 20% kittens, 20% cute, 15% hamster, … (at which point, you could interpret topic B to be about cute animals)

In the third phase of the project, you need to identify a set of topics for each document cluster generated in the **Phase 2**. Then, you have to report as output of this phase the set of all topic terms for all the analyzed clusters. Make sure that the input of this phase is compatible with the output of the Phase 2 while its output is compatible with the input of **Phase 1**. At the end of the third phase, you should have a
workflow of the three phases able to run one after another in the same program as depicted in Figure 1. You should adjust Phase 1 to count the global frequency only of topic terms produced as the output of this phase in the initial document corpus (instead of all the words in sentences).

Additionally, you are asked to implement a different topic detection strategy, in which you will apply LDA directly to the input document corpus (not on each cluster).

LINKS: You can use the implementation of LDA offered by the MALLET API (http://mallet.cs.umass.edu/) for the first topic detection approach. You are free to choose any other available MapReduce implementation (for a comparison see [3]). You should exclude stopwords from the topic terms (e.g., use the --remove-stopwords option in MALLET API). For the additional approach you should use the LDA implementation of Mahout (https://mahout.apache.org/users/clustering/latent-dirichlet-allocation.html) on the whole initial document corpus, i.e., without the initial k-means clustering.

HINTS: Write the set of topic terms in a file stored on HDFS, and add this file to the Distributed Cache of the frequent terms counting job (Phase 1). Then, from the mapper of Phase 1, load the terms file from the Distributed Cache and count only the sentence terms that exist in this file.

Phase 4 (35%): Sentence Filtering (Stage (d)) – Deadline: November 28

In this phase, you are asked to filter the sentences of the initial input documents, based on the results of the previous phase, and generate one single document. Specifically, given a set of sentences, you are asked to keep only those sentences that contain at least one of the terms that have been produced from Phase 1. Furthermore, from those sentences, you will need to remove those that are very similar (near duplicates) or identical (duplicates). To measure the similarity of sentences, you can model each sentence as a set of words (excluding stopwords), and use Jaccard similarity. Jaccard similarity of two sets A, B is given by: Jaccard(A,B) = |A ∩ B| / |A U B|. You can assume that two sets A and B are near duplicates, if Jaccard(A,B) ≥ 0.8.

You should be able to apply the same process for both approaches implemented in the previous phase (LDA on each cluster and LDA on the initial document) and compare the quality of the generated summaries using the evaluation frameworks described in Phase 5 of the project.

Additionally, you are asked to implement an alternative sentence filtering approach: After creating a term vector for each input document (as you did in Phase 2), you can use the weights of the terms appearing in a document and assign an importance score to each sentence S of a document d ∈ S, as follows:

\[ \text{importance}(S) = \frac{1}{|S|} \sum_{w \in S} tf(w,d) \cdot idf(w,D). \]

Then, you can create a summary of the document corpus, by keeping the top-k sentences with the highest importance score, where the appropriate k value is up to you. At the end, you should again discard near duplicate and duplicate sentences from the generated summary.

It is explicitly left as a Bonus (up to 20% of the total project grade) in this assignment the use of complex scores combining both the frequency-based scores of sentence importance with the topic-based signatures of sentence relevance (for some constant factor a):

\[ \text{score}(S) = a \cdot \text{importance}(S) + (1 - a) \cdot \text{relevance}(S) \]
You are asked this time to design and implement a suitable relevance score of sentences which exploits the topics-signatures extracted by one of the methods proposed in Phase 3. As previously, the quality of the produced summaries by each method will be assessed using the evaluation frameworks described in the Phase 5 of the project.

**LINKS:** A list of stopwords is available at [http://www.textfixer.com/resources/common-english-words.txt](http://www.textfixer.com/resources/common-english-words.txt).

**HINTS:** For the first approach to sentence filtering, it will help you if you have two files in the distributed cache: a) the frequent terms (output of Phase 3), and b) the stopwords file. In the map phase, you can load the frequent terms in a Java Set and, as you parse sentences from the input documents, keep only those that contain at least one frequent term. Then, you can either use a single reducer and compare exhaustively all possible pairs of sentences, or stop the first job in the map phase, and add a new job for the Jaccard similarity computations in parallel. The first approach is easier to implement, but the second is more scalable and efficient. You have to treat each sentence as a Set of words, from which you will remove all stopwords.

For the second approach (ranking sentences), you can have the importance score of each sentence from the mapper, so you can then follow the instructions of the lab session on MapReduce design patterns for top-k computations.

**Phase 5 (15%): Evaluation of Summarization Techniques – Deadline: December 5**

For the last phase of the project, you are asked to compare the different summarization techniques that you have implemented, using ROUGE [6]. ROUGE is a set of measures used for evaluating automatic summarization, against a human-produced summary. For this project, you are asked to evaluate your summaries using ROUGE-N, given by:

\[
ROUGE-N = \frac{\sum_{S \in \text{ref-summary}} \sum_{n\text{-grams} \in S} \text{count}\_\text{match}(n\text{-gram})}{\sum_{S \in \text{ref-summary}} \sum_{n\text{-grams} \in S} \text{count}(n\text{-gram})},
\]

where \( n=2 \) is the length of the n-grams (sequences of \( n \) consecutive words), ref_summary is the set of catchphrase sentences existing in the dataset of legal cases you were given, count_match is the number of common n-grams between the reference summary sentence and the automatic summary, and count is the number of n-grams in each reference sentence \( S \).

In total, you should compare the following approaches:

1. Clustering -> LDA (MALLET API) -> Frequent and Semantic Terms -> Sentence Filtering
2. LDA (Mahout) -> Frequent and Semantic Terms -> Sentence Filtering
3. Assign an importance score to each sentence and keep the top-k sentences
4. (Bonus of Phase 4) Assign a complex score to each sentence and keep the top-k sentences

**LINKS:** You can download ROUGE from [http://www.berouge.com/Pages/DownloadROUGE.aspx](http://www.berouge.com/Pages/DownloadROUGE.aspx), or implement it on your own. The implementation should be straightforward.

**Bibliography**


