Concept/Theme Rollup

Tanvi Sahay, Ankita Mehta

June 2017 - August 2017

Project Introduction

Keyphrases are short-length phrases extracted from a body of text that are capable of conveying the core ideas present in it. Given these keyphrases, our task was to cluster phrases together based on how semantically related they are. For this, we experimented with several techniques of phrase representation in combination with various clustering methods and evaluated our results on which set of clusters were preferred over others by human observers. We implemented and tested 8 different techniques of phrase representation, including methods inspired by the word-to-vector skip-gram models and feature-based representation models. We also implemented a novel architecture for phrase representation that involved learning sentence representations using a sequence-to-sequence autoencoder and equating these representation to another sentence representation, with words replaced by phrases. This document details the architecture, experimentation details and reasoning for each model used. Previous experiments had shown that data cleaning affected the results greatly, thus a cleanup pipeline was also implemented. In addition to that, a 14GB data corpus, to be used in different deep learning models, was also extracted from the web, cleaned and stored for future use. The following sections explain the cleanup pipeline, provide information about the augmenting data dumps and describe each representation model in detail. All annotated codes can be found in the ThemeRollup directory on branch 'ThemeRollup' of the nlmltool repository.

Resources Used

In order to properly train all models, sufficient data was needed and thus, in addition to the text files for extracting phrases provided by individual users, external data was used to augment this data. This was especially necessary for deep learning model since they have a high data requirement, which the user provided data may not always fulfill. As augmenting data, Wikipedia dumps of 12GB, NYT NEWS corpus of 8GB and English blogs corpus of 800MB were downloaded and cleaned. The final cleaned augment data corpus was approximately 14GB in size. All the data sources used are publicly available online and the cleaned corpus can be used in other models as well. In addition to that, distributed representation of words were required for several phrase representation techniques. For that, Google News word vectors using skip-gram technique were used. These pre-trained word vectors were downloaded and used whenever pre-trained word representations were needed.
In addition to these public data sets and models, Salience Python SDK was used to extract keyphrases, POS tags, summaries etc. whenever needed. All codes were written in Python2.7. GPU clusters GYPSUM and BLAKE provided by University of Massachusetts Amherst, were used for compiling and running the codes. Tensorflow version 1.0.1 was used for all experiments. In addition to the default python libraries, nltk, pyenchant, sklearn and gensim were also used.

Experimentation Pipeline

Figure 1 shows the general pipeline followed for this project. First, all the raw data provided by users was cleaned and phrases were extracted from the clean text. In addition to removing processing error and incorrect words from the text, this also improved the quality of phrases being extracted. Then, a phrase representation model was used to convert each n-gram into a fixed-dimensional dense vector. These vectors acted as features on top of which , clustering algorithms were employed. Finally, results for each method were evaluated with the help of human annotators. Each section of the pipeline has been described in detail below.

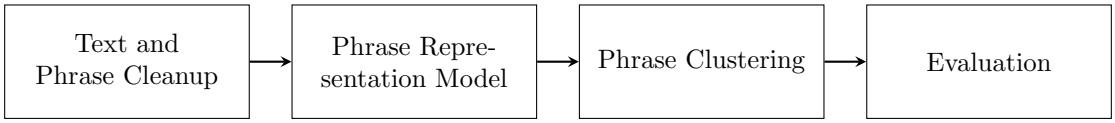

Figure 1: Overall Project Pipeline

1 Text and Phrase Cleanup

Before extracting phrases from raw text, a cleanup pipeline was employed that removed all unnecessary punctuation marks, html tags and hex errors and smoothened the contractions. These steps were crucial since they affected the quality of phrases extracted and impacted the performance of models that took the context or summary of phrases into account. Test and augment data were cleaned separately since only the test data was passed through salience to extract keyphrases. Following steps were followed when cleaning the test data:

1. HTML normalization - All the html tags i.e. all text between the tokens ¡ and ¿, including the tokens, were removed from the data.
2. Encoding Errors - All symbols (except "-", "," and "") that could not be decoded as valid 'utf8' characters were removed. These symbols were chosen based on the test corpus being experimented with. More symbols can be added later.
3. Contraction Normalization - After tokenizing the data using salience, contractions such as "I'm" and "I've" were normalized to "I am" and "I have". During output, tokenized data was converted back to running text.
4. NER Removal - All the named entities detected by salience were replaced with their identities. For example, the company name "Lexalytics" detected by salience was replaced with the word "company".
5. Punctuation Normalization - All punctuation marks except ".", ",", "!" and "?" were removed from the text. These were left intact as removing them resulted in degradation of the quality of phrases being extracted from salience.
6. Number Normalization - After tokenizing the text again, all numbers were replaced by the single entity '0'. If the token was only a number, the entire token was converted and if the token contained alphanumeric entries, all the numeric instances were replaced with '0'.
7. Spell Error Removal - Using the python library PyEnchant, all words that were present in the English dictionary, either in their lower form or their capitalized form, were retained and all other words were replaced with 'unk'.
8. Phrase Extraction - After cleanup, phrases were extracted and stored separately.
9. Punctuation Normalization - After phrase extraction, all punctuation marks other than "." were removed and clean files were stored separately.

Augmentation Data was cleaned in the same manner test data. However, since phrases were not extracted from this data, all punctuation marks other than "." were removed in a single step. In both the cases, if there were any files that could not be processed by Salience, they were simply skipped.
Phrases extracted from test data were also cleaned, in order to get rid of any incoherent phrases present in the corpus. The following steps were undertaken for phrase cleanup.

1. All phrases with the word 'unk' in them were ignored.
2. All phrases with a named entity token in them (e.g. 'company', 'place', 'person') were ignored.
3. Phrases with all the same words in them (e.g. 'token token', 'place place') were removed.
4. Phrases that were a named entity token themselves (e.g. 'job title') were removed.

Once the phrase list was cleaned up, phrase were sorted according to their relevance scores(provided by salience) and only a top percentage(provided by the user) was selected for conducting all experiments on.

2 Phrase Representation Models

Following are the phrase representation models that we experimented with.

2.1 Average Embedding

One of the most basic and commonly used techniques for getting embeddings of n-grams is taking an average of the embeddings of all words present in the n-gram. For all the phrases extracted through salience, we computes the average embeddings using the Gensim GoogleNews word embedding model. To ensure no words are missed, embeddings were checked for each word in lower, upper and capitalized form. Any phrases with one or more embeddings missing were ignored.

2.2 Weighted Average Embedding

In certain cases, clusters based on similar nouns/verbs may be preferred over clusters with similar adjectives/adverbs. To achieve this, we weighted the embeddings based on their part-of-speech tags, as extracted by salience and calculated a weighted average to be used as phrase embedding. We provided higher weights (>1) to nouns and lower weights to adjectives and adverbs (<1). We chose to weight nouns higher based on our observation of the keyphrases extracted, which were mostly noun phrases. The weights experimented with are given in table 1.

Table 1: Weight Settings for weighted average embedding

NN/NNP/NNS/NNPS	JJ/JJR/JJS/RB/RBR/RBS
2.5	-
3.0	-
3.5	-
4.0	-
4.5	-
5.0	-
2.5	0.25
2.5	0.5
3.0	0.25

Embeddings with only > 1 weights for nouns performed poorly as compared to embeddings with nouns up-weighted and adjectives and adverbs down-weighted.

2.3 Phrase representation based on equating sentences

Another approach towards finding the phrase embeddings can be this: Given the representation of the sentence in which a phrase occurs, the representation should stay the same when the phrase is treated as s single entity instead of each of its words being treated as one. This technique was inspired from the skip-gram model of phrase representation, in which common phrases are treated as a unique entity, by their words being joined with '-'. In a basic model implementing the above idea, we equated the average embedding of a sentence when it was in the unaltered state to the average embedding when the phrase was replaced with the hyphenated entity. The phrase embedding was taken as unknown and was computed by solving the equality. Mathematically,

Given a sentence "a b c d e", where the trigram "b c d" is a phrase, a new sentence will be created as - "a b-c-d e". Now average both the sentences will be equated and embedding of the phrase will be found as:

$$average\_embedding(\text{“}abcde\text{”}) = average\_embedding(\text{“}ab-c-de\text{”}) \tag{1}$$

$$embed(b-c-d) = (\frac{E(a)+E(b)+E(c)+E(d)+E(e)}{5} * 4) - (E(a)+E(e)) \tag{2}$$

The general equation for obtaining phrase embedding using this technique is:

$$E(w'_k) = \frac{\sum_{i\in 1}^{W}(E(w_i)/W)}{\sum_{j\in 1, j\neq k}^{W'}(E(w'_j)/W')} \tag{3}$$

where E is the word embedding model, w_i is the i^{th} word in the normal sentence and w'_j is the j^{th} word in the hyphenated sentence. w'_k is the k^{th} word, which is the hyphenated phrase. W is the number of words in the normal sentence and W' is the number of words in the hyphenated sentence. If there are more than one instances of the phrase in the sentence, we divide the answer by the number of instances. There can also be multiple sentences in which a phrase occurs. Thus, for getting a unique embedding for each phrase, average of all the results is taken.

2.4 word2vec with random initialization

One of the most common ways of representing words is through the skip-gram word2vec model introduced in [1]. The paper also includes converting common phrases into hyphenated entities in order to learn their embeddings separately from their component words. Taking inspiration from that, we replaced all phrases with their hyphenated entities in both the user provided data and the augment data collected by us. This data was used to train a traditional skip-gram model with the following specifications:

Table 2: Network Specifications for word2vec with random initialization

embedding size	300
window size	5
workers	4
count	10
initial learning rate	0.1
minimum frequency	10

The gensim implementation of word2vec was used for obtaining all results. This implementation performs negative sampling internally and modifies learning rate as training progresses. Embeddings are initialized randomly. We provided the cleaned training data to gensim, with phrase replaced with hyphenated phrases so as to be treated as a single entity and gensim prepared and trained a skip-gram model from scratch.

2.5 word2vec with seeded initialization

The skip-gram model of word2vec learns embedding of a word based on its context (neighbouring words). However, for the case of phrase embeddings, both the words that the phrase is composed of as well as the words surrounding it are important. To take the component words of a phrase into account, we modified traditional word2vec to initialize phrase embeddings with pre-trained embeddings to allow word2vec to learn on top of them. This was done to improve upon the base embeddings by adding information about its context.
We carried out two experiments, one in which embeddings were initialized with those extracted in 2.1 and the other in which they were initialized with embeddings extracted in 2.3. Network hyperparameters were kept constant in both the experiments.

Table 3: Network Specifications for word2vec with seeded initialization

learning rate	0.1
embedding size	300
minimum frequency	10
batch size	256
epochs	460
negative sampled	64
context words	4

As input data, phrases and their contexts were extracted from the test and augment data sets and input to the model was given in the form of phrase-context pairs. 4 words, 2 from each side of the phrase were taken as context and phrase-context pairs were prepared. These contexts were cleaned to remove any phrases and stop words. Then, any phrase with less than 10 context words was removed from the training data. The final phrase-context pairs were used as training input. Words/phrases whose embeddings were not present in the pre-initialized vectors were initialized randomly between -1.0 and 1.0.

2.6 Feature-Rich Compositional Transform

This experiment was based on the paper [2] in which the authors constructed embeddings of phrases by learning how to compose word embeddings using features that can capture the phrase context and its structure. For each phrase, the component words were assigned a set of features and phrase embedding was learned by combining these features with the individual word embeddings. We created the following seven features for representing each word of a phrase:

1. Sentiment of the word - positive, negative or neutral
2. POS tag of the word - Noun, Pronoun, Verb, Adjective, Adverb, Determiner, Symbol or Miscellaneous
3. Whether the word is a head word or not

4. Word cluster of the word : word clusters have been formed by considering each word present in the phrase or their context
5. Distance of the word from the head word
6. POS Tag of the head word of the phrase
7. Word cluster of the head word of the phrase

Head words were taken in the following sequence:

1. noun if only one noun was present
2. second noun if more than one nouns were present
3. verb if no noun was present
4. adjective if no verb present
5. adverb if no adjective was present

Maximum phrase-length was considered five in this whole experiment. For phrases whose length was less than five, word 'PAD' was appended at the end to make its length equal to five and feature vector of 'PAD' was considered to be all zeros. Also, there were some features with very large values and some with very small values, so at the end we normalized them to a range of 0-1.

For making features, only those phrases were considered whose every word has an embedding present in the pre-trained Google news vectors. After getting features of all phrases, we created a deep learning model for extracting phrase embeddings. As input data, phrases and their contexts were extracted from the test and augment data sets and input to the model was given in the form of phrase-context pairs as for word2vec model. Here, we initialized word embeddings with pre-trained embeddings to improve the performance of FCT. The objective of FCT model is an extension of the skip-gram objective to phrases. Here, each training data provided to the network was phrase feature-context pair. While training, the FCT parameters and word embeddings were updated via back-propogation. After the network is trained, embedding of each phrase was calculated using FCT parameters and its feature vector.

Data was given to the network in batches, due to limited GPU size. Each batch was normalized to have the same number of phrase-context pair. The last batch was padded with data from the beginning of the training data. Network specifications of the model are given in table 4.

Table 4: Network specifications for FCT Model

learning rate	0.1
embedding size	300
minimum frequency	10
batch size	512
epochs	420
negative sampled	64

2.7 Hybrid Model

For our last experiment, we took inspiration from 2.3 to create a deep learning model for extracting phrase embeddings. We made two versions of each sentence - one with all the words and one with each phrase in the sentence replaced with a single hyphenated entity. This data was used to train a network that was composed of two parts. The first part was a sequence to sequence autoencoder that was trained on the unhyphenated sentences, and embeddings or context vectors of each sentence were then extracted. The second part of the model was a standard bidirectional RNN which was trained on the hyphenated sentences. Mean square loss was calculated by taking the difference between context vectors from the first and second part of the network for each sentence and this loss was optimized using the Adam Optimizer. One advantage of this model is that it not only tries to maintain equality between the hyphenated and unhyphenated sentences, like 2.3, but it also takes into account the sequence of words and in turn, the neighbors of the phrase, which adds information to the embedding thus learned.

A traditional sequence to sequence autoencoder is composed of two RNNs called 'encoder' and 'decoder'. The encoder coverts each input sentence into a fixed dimensional representation, called context vector, and the decoder tries to recreate the sentence using this same context vector. This context vector is nothing but the final hidden state of the encoder, which is used as the initial hidden state for the decoder. In our implementation, the encoder was a bidirectional GRU RNN and the decoder was a forward GRU RNN.

The second part of the model was a bidirectional GRU RNN whose outputs were ignored and losses were back-propagated based only on the hidden state.
Data was given to the network in batches, due to limited GPU size. Each sentence in a batch was of equal length, with smaller sentences being padded with the token 'PAD'. Each batch was normalized to have the same number of sentences. The last batch was padded with data from the beginning of the training sequence. Network specifications for both parts of the model are given in tables 5 and 6 respectively.

Table 5: Network specifications for autoencoder

Autoencoder	
batch size	30
learning rate	0.001
embedding size	150
encoder hidden size	150
decoder hidden size	300
epochs	20
loss function	sampled softmax with logits
optimizer	Adam
time major	True

Table 6: Network specifications for bidirectional rnn

Bidirectional RNN	
batch size	30
learning rate	0.001
embedding size	150
gru hidden size	150
epochs	5
loss function	mean square error
optimizer	Adam
time major	True

3 Phrase Clustering

With representations from all the models obtained, the next step was to cluster the results and see which ones performed better in terms of the clustering obtained. For this, two clustering methods were experimented with:

3.1 KMeans

The first clustering technique was KMeans. In the first run of KMeans, N centroids are randomly assigned in the vector space, N being the number of clusters desired by the user. In each consecutive run, the following two steps take place:

1. Each point in the vector space is assigned to the centroid closest to it.
2. Once all the points have been assigned to a cluster, the new centroid for each cluster is calculated.

The clustering algorithms halts when clusters stop changing significantly.
This method was chosen for its ease of implementation and understanding and its scalability to large number of samples. While KMeans is said to perform bad in high dimensions, we found the method to perform better without reducing the dimensions using PCA or t-SNE. We used minimum euclidean distance to assign centroids to each point and sklearn's implementation of MiniBatchKMeans for performing the clustering, where centroids were initialized using kmeans++.

3.2 DBSCAN

As explained in the sklearn documentation, DBSCAN views clusters as areas of high density separated by areas of low density. In DBSCAN, a point is termed 'core sample' if there are a fixed number of points within a fixed distance of it. A cluster is built recursively, by taking a core sample and finding each neighbor of that sample that is also a core sample and then finding the neighbors of each of these new core samples and so on.

This method was chosen because of its ability to form clusters of any shape, unlike kmeans which assumes the clusters to be convex. It is also scalable to a large number of samples. Also, unlike other techniques, it only prepares clusters that have a high enough density and does not forcefully assign a cluster to every point, thus allowing for points to not belong to any cluster at all.

Other than these, we also experimented with Ward Clustering, which is a type of hierarchical clustering, spectral clustering and gaussian clustering. However, ward clustering cannot be run on negative values, which were present in the embeddings. Spectral clustering was extremely time intensive and not scalable to large data samples. Gaussian was not scalable to large number of samples either.

4 Evaluation

4.1 Internal Evaluation

While picking the number of clusters for kmeans, three internal evaluation metrics were chosen, since it was not known which would be the best fit for evaluating the clusters. These were: Silhouette score, Davies–Bouldin Index (DBIndex) and Dunn Index. We prepared a grid of cluster numbers to search through in the following manner: We took 10% of the total number of phrases available and search in a $\pm$ 100 range of clusters with increments of 10. If the 10% values was less that 100, we checked the number of phrases present. Given these were greater than 100, we reduced our range to 10% $\pm$ 10 with increments of 1, otherwise we simply prepared 5 clusters.
Silhouettes scores and Dunn Index values increase for better clustering while DBIndex values decrease. These trends were checked manually for each metric. We chose five cluster values - 100, 500, 1000, 5000, 10000 and manually rated each of these based on which formed more sensible clusters. The test was only done for average and weighted average embeddings. All three scores for the given number of clusters were found and it was observed that DBIndex values resulted in the same trend as the manual ratings. Thus, DBIndex was chosen as the internal evaluation metric for finding number of clusters.

4.2 External Evaluation

Once best clusters for each embeddings were chosen, the results were evaluated. For this, we extracted a common set of phrases from all the experiments, re-clustered them and then extracted only those clusters to which specific phrases belonged. These phrases acted like seeds for choosing clusters to compare and were chosen randomly from the set of common phrases. Clusters for each of these seeds extracted using all 8 methods and MiniBatchKMeans were compared with each other by manual observation. DBSCAN results were not taken into account since the clusters formed were too low in number and most of the phrases were assigned to their separate clusters.

Results

In general, clustering of words/phrases can either be done based on their textual content or meaning and general usage. Accordingly, we observed that certain methods performed better for one type of clustering than others. For clustering based on the textual context of phrases or the words present in them, embeddings produced using average, weighted average with 2.5(noun)-0.25(adjective/adverb) weight combination and fct performed well. On the other hand, for general meaning based clustering, equating sentences and word2vec with seeded initialization performed better than all other methods. word2vec with random initialization and hybrid model did not perform well for either of the two cases. All results for the experiments are present at *./ThemeRollup/Results/comparison_results/Example* in the ThemeRollup branch of nlmltool repository.

Future Work

In addition to the experiments performed, several things can be tried to either improve the performance of the present models or to branch present models to new ones.

1. In data cleanup, while we ignored all punctuation marks, '!' and '?' can be replaced with '.' in order to retain the sentence boundaries which would otherwise be lost.
2. While we implemented grid search for finding the optimum number of clusters, other hyperparameter optimization techniques can also be experimented with.

3. DBSCAN's poor performance could be in part due to untuned hyperparameters. These can also be experimented with.
4. For Hierarchical/Ward clustering, smoothening of embeddings can be done so as to normalize all the negative values present in the embeddings.
5. In the hybrid model, the bidirectional RNN can be replaced with another autoencoder and network can be trained to learn both the context vectors and the sentences themselves.
6. Other hyperparameters of hybrid model, such as loss function and network architecture can also be experimented with.
7. The hybrid was only trained on user-provided test data due to lack of time. Augment data can also be prepared in the same way as test data and used to train the network. The additional data might improve the network's learning.
8. Other ways of obtaining context vectors, such as skip-thought vectors can also be explored.

References

[1] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[2] Mo Yu and Mark Dredze. Learning composition models for phrase embeddings. *TACL*, 3:227–242, 2015.

Appendix

1 Installing Pyenchant Library

Pyenchant library was intended to eliminate the gibberish words/ words which are not present in US dictionary and British dictionary but due to gypsum issues, we were able to use only US dictionary. Some issues were faced while installing Pyenchant, which can be referred here:

1. sudo su - yum install enchant (When facing error installing Pyenchant library : ImportError: The 'enchant' C library was not found)
2. Then pip install –trusted-host pypi.python.org pyenchant

2 Stemming

We also tried to capture the words having plurals available in the text or the words with different spelling under the same name like omelette, omelet. For this experiment, we stemmed the whole sentence and then checked whether the phrase/stemmed phrase is present in the sentence. But this also captured the words having same stem but they were altogether different words, under the same name.