Relevance Analyses and Automatic Categorization of Wikipedia Articles

George Pakapol Supaniratisai, Pakapark Bhumiwat, Chayakorn Pongsiri

December 11, 2015

Abstract

We used a unigram model for an article with each word represented by a vector in high-dimensional space that captures its linguistic context. The word-to-vector model was learned through a neural network using a skip-gram model. K-means algorithm was applied on word vectors of each article to get clusters of word, which were then used to find similarity index based on the locations and weights of clusters. Given a set of article pairs, we found a linear correlation between similarity index outputted from the algorithm and human's ratings on similarity. We then applied hierarchical clustering on a group of articles based on their similarity indices and construct a categorization binary tree. We evaluated the tree by asking humans to play odd-one-out games – given an instance of a triplet of articles, choosing one article that is the most different, and we found that the tree correctly classified approximately 80% of the odd-one-out instance compared to the data from humans.

Keywords: *Word Vector, word2vec, SkipGram model, SoftMax Regression, K-means clustering, Hierarchical Clustering Natural Language Processing, Dichotomic Analysis, Unigram Model, Similarity analysis, Relevance analysis*

Introduction and background

The main challenge of semantic-level analyses of articles ranges from the sparsity problem, the fact that appearance of each English words are not frequent enough, to the knowledge representation of semantics, the fact the computer cannot directly detect the similarity between words other than identical string tokens. We tackle the problem by representing each word as a high-dimensional vector, where we can directly calculate semantic similarity between words. Not only this approach allow us to realize the similarity of two articles with few overlapping words, but it also mitigates the sparsity problem arose in the extremely short prompt (such as sentences or questions).

Another challenge concerns how to represent an article. One way to accomplish this is to look how frequent each word appears in the articles. Yet, some words are less meaningful than others and do not contribute to the relevance of the article to the others. In this project, we implemented an algorithm to filter out these words and perform a sampling on the rest and incorporate a word-to-vector model to represent each article with centroids and weights based on word vectors.

Data Set

We primarily use Wikipedia articles as a source of data. For each article, we represent each article by a sparse vector of a unigram model. (That is, a mapping of each English word appearing in the article to the frequency count). We use the python package wikipedia to retrieve the articles in a small scale on-line with the analysis. Then we filter the frequent functional words (approximately top 30-40 words in English language) For Hierarchical clustering, our data is retrieved from the resource‡[1] that Google releases for creating the vector representation of words on the latest research tool TensorFlow™.

Algorithms

I. Word2Vec

We apply the gensim library, which internally applies the deep learning algorithm, especially hierarchical softmax skip-gram algorithm. (This library uses the similar algorithm to *TensorFlow™* ; however, gensim library has the better interface for training and storing data.) We use this algorithm because the hierarchical softmax algorithm can deal with the great number of inputs with the small computational complexity per training instances, says, O(log V) where V is the number of the word-output vectors. Since gensim library requires having an input as a matrix of words where each row represents a sentence and each column represents a word in the sentences, we apply the simplified model by separating our training data into a sentence of exactly 10 words so that we have 1.7 million training sets. After passing the training set to the gensim function, we receive an output as the 128-th dimensional

‡ http://mattmahoney.net/dc/text8.zip

vector representation of each word and subsequently save such data in the separate file.

II. Frequent Function Words Filtering

We utilized two methods to filter frequent words. The first method is simply to delete the frequent function words (e.g. the, is, not, …). Deleting top 40 function words reduces the noises from frequent words into a workable level. Another algorithm is that we can gather several articles (i.e. ~100) and pick a word that appear more than n times in r% of articles. Typically, we choose n = 2 and r = 95%

III. Finding Hotspots in word vector space by K-means Clustering

After filtering function words from the article, we apply K-means algorithm to group words that have similar meanings into clusters. We set up to have exactly eight clusters because the smaller number of clusters is likely to pool irrelevant words together and yield a poor result, while the larger number of clusters will make the program run slower without yielding any significantly better result (due to the more empty cluster). Every loop will update the centroid of each cluster to the mean of the vector representation of all words in the cluster. In addition to clustering words, we assign a weight for each cluster by calculating the mean of the word count for each word in the cluster. In other words, since we can ensure that the same word will definitely be clustered in the same group, the weight of the cluster is the sum of the square of the word counts for each distinct word in the cluster and divided by the total word count as shown in the figure.

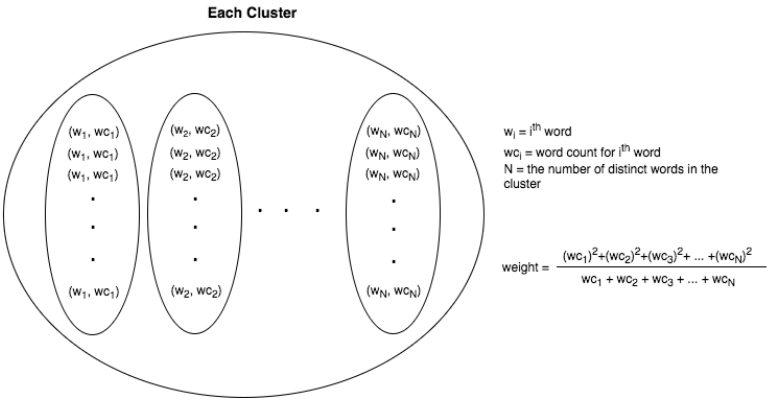

Figure 1: Displaying modification of K-means weighted with number of appearances of each word

IV. Similarity Index

For simplicity, we define the following variables:
C_{Ai}: a i^{th} 100-dimensional centroid for cluster A
C_{Bi}: a i^{th} 100-dimensional centroid for cluster B
W_{Ai}: a i^{th} weight for cluster A
W_{Bi}: a i^{th} weight for cluster B
$h(A, B)$:a heuristic computing the similarity from article A toward article B

$$h(A,B) = \sum_{i=1}^{8} W_{Ai} \max\{C_{Ai} \cdot C_{Bj}\}$$

where $j \in \{1, 2, 3, 4, 5, 6, 7, 8\}$

In order to evaluate the degree of similarity between two articles A and B, we apply a similarity index heuristic computing an average of h(A, B) and h(B, A), where h(A, B) represents the summation of the weight for each cluster multiplied by the dot product between the centroid of such cluster in article A to the closest centroid from article B, i.e.,

$$\text{similarity index} = \frac{h(A,B)+h(B,A)}{2} = \frac{\left(\sum_{i=1}^{8} W_{Ai} \max\{C_{Ai} \cdot C_{Bj}\}\right) + \left(\sum_{i=1}^{8} W_{Bi} \max\{C_{Aj} \cdot C_{Bi}\}\right)}{2}$$

Hierarchical Clustering

After obtaining similarity indices, we applied hierarchical clustering to categorize articles. Hierarchical clustering starts with assigning each article to a cluster. At first, each article therefore would belong to its own cluster. At any step in time, we seek to combine any two clusters with highest similarity. The similarity between two clusters is calculated from the average similarity index of all pair of articles across clusters. Merging recursively, we will finally get one list, and we will be able to trace back to construct a binary tree (as we call a categorization binary tree) to demonstrate merging steps.

Odd-One-Out for categorizing[2]

Odd-One-Out (O3) for categorizing has two primary subroutines. The first subroutine is to perform an analysis to pick the oddest article among any three articles. The second subroutine is to construct an optimal (or near optimal) dichotomous tree from O3 output of the first subroutine.

I-A. *Oddity*

From each triplet of articles, we seek to choose the oddest one. First, let N_1, N_2, N_3 be the sparse vector of the three articles. We compute the dot product between each pair of articles. Out of the three pairs, we reason that the largest value means the two articles are similar, and the other one must be the odd one. However, since the dot product of the three articles can be close to one another, we cannot be 100% confident that we can actually odd out one articles from the three. As such, we weight our decision by oddity defined as

$$\text{oddity} = \log\left(\frac{S_{12}}{\sqrt{S_{13} \cdot S_{23}}}\right)$$

if similarity between article 1 and article 2 is the maximum similarity, and vice versa for 2 other cases. We conclude our routine with the pseudocode below.

[2] Some parts of this category is used jointly in CS221, specifically Greedy Divide

```
oddOneOut(N1, N2, N3):
    similarity12 := N1 . N2
    similarity13 := N1 . N3
    similarity23 := N2 . N3
    oddArticle := the one not in max dot product
    oddity := log(...) // as defined
    return (oddArticle, oddity)

for N1, N2, N3 in sparseVectorsOfArticles:
    dataset.add( (N1,N2,N3,oddOneOut(N1,N2,N3)) )
```

[Note: This part can be replaced by other method of similarity analysis. For example, we may use similarity obtained from word vectors, or the heuristics obtained from K-means clustering in Hierarchical Clustering (see Algorithm II) as well.]

I-B. *Recursive Dichotomic Division*

From the dataset, we want to divide the articles into the dichotomous tree T_{opt} so that the "reward" is maximized, where our reward is defined by the sum of the oddity of each constraint that are satisfied. A constraint is satisfied if the odded out article gets divided before (i.e. at the higher hierarchy than) the other two articles.:

$$\Re(T) = \sum_{\text{dataset}} \text{oddity} \cdot \mathbb{I}(\varphi(\text{data}, T))$$
$$T_{\text{opt}} = \arg\max_{T} \Re(T)$$

To do so, we can perform a recursive division of the articles into two sets (i.e. assign number 0 or 1 to each articles) until we reach one or two articles. If we reach the case of two articles, it is easy to divide the set of two articles to two sets of one article each. That is, we only need to construct a systematic way to divide a set of articles that best satisfy the constraints.

Suppose we have a set of articles, we can proceed with a greedy assumption (which we know is suboptimal at times) that we divide it so that the reward is maximized in each iteration and division. The reward for each tree node on a set of constraints (data set) is defined by the sum of oddity of each token that is correctly divided. That is,

$$\Re^{(i)} = \sum_{\text{dataset}} \text{oddity} \cdot \mathbb{I}(a(N_1) = a(N_2) = \neg a(N_{\text{odd}}))$$

where $a(N_i)$ denotes the 0-1 assignment of article N_i.

To achieve the maximum reward for each tree node, we run the following iteration. Starting, from random assignment, we modify (at most) one assignment for each token with chance 50% to comply with those constraints, only if the modification will yield a better reward.

```
GreedyDivide(ListOfArticles, dataset):
    Randomly assign 0,1 to each article
    Iterate until convergence:
        for each token (N1,N2,N3,odd,oddity) in dataset:
            if assignment agrees or random is below 0.5:
                continue
            else:
                Modify only one assignment to comply
                Check if it yields more reward, continue,
                else, revert
```

Note that in all cases, we can modify at most 1 assignment to comply with one constraint.

The idea is that the process of the iteration will monotonically increase the reward. We add the element of probability to prevent the large error in case the algorithm greedily modifies the assignment as it iterates through the data, and block any further changes. We also define the convergence as when the assignment is not modified for 5-10 times of whole iteration. Also note that although this iteration method generally yields a high percentage of reward compared to the limit (i.e. summation of oddities in the dataset), it does not guarantee the maximum attainable reward.

We perform the division for each node of tree until we reach one or two articles, which can easily resolve to a leaf node

Results

Similarity index between two Wikipedia articles

We randomly selected pairs of Wikipedia articles from the pool and using the algorithm described in the previous section to evaluate the similarity index. We found that over 44 pairs of articles, the index ranges from 1 to 30. We also asked 5 people to rate similarity of these pairs of articles on the scale of 1 to 5. The plot between human's ratings and the similarity index (computer's ratings) is shown in figure 2 below. We found a linear relationship between the two ratings with a correlation ρ of 0.66

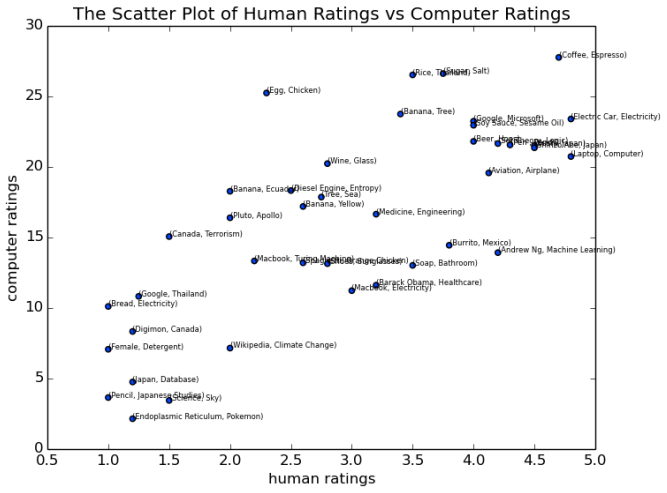

Figure 2 : Comparison of Human Raing of Relevance and Computer rating of Relevance

Hierarchical Clustering

We generated 5 sets of articles, each containing 10 different articles. For each set, given similarity indices of all article pairs, we applied hierarchical clustering to construct categorization binary trees as shown in figure 3 below.

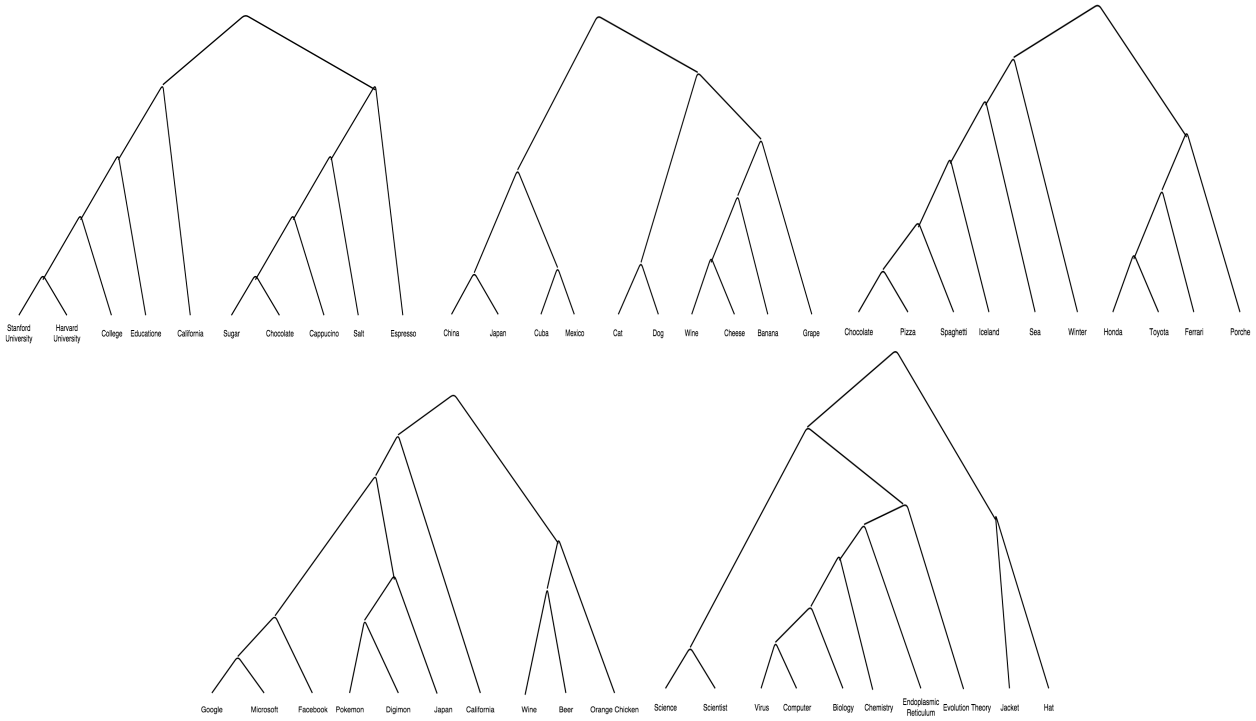

Figure 3 : Dichotomous Tree constructed using Hierarchical Clustering with word vectors

For a comparison purpose, given the same lists of articles, trees that are constructed by Odd-One-Out algorithm are shown in figure 4 below.

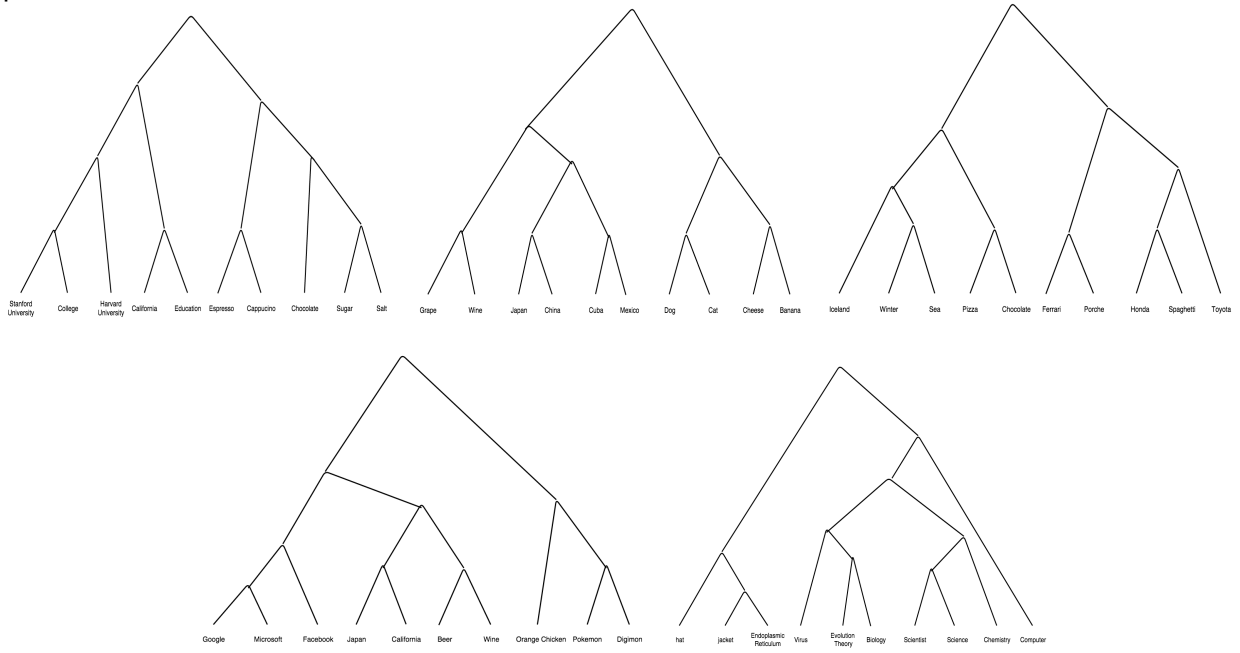

Figure 4 : Dichotomous Tree constructed using odd-one-out instances

To evaluate the performance of the algorithms to construct categorization binary trees, we collected an odd-one-out test set from 5 people. An odd-one-out instance was presented to a person, and a person, with his or her best judgment, would

choose an article that is the most different from the rest. We say a tree correctly handle an odd-one-out instance correctly when the article that was chosen to be odded out by a human is the same as the article that is the farthest from the rest two in the tree. Note that the odd-one-out instance we use to evaluate the performance here are randomly selected and different and non-overlapping with the instances we used to train the model for the Odd-One-Out algorithm.

Finally, we found that the trees from hierarchical clustering and Odd-One-Out algorithm correctly classified 79.2% and 75.0 % of the instances respectively.

Discussion

We found that similarity index obtained from our algorithm does a fine job capturing the relevance between two Wikipedia articles. The use of word embedding allows us to realize the similarity between two articles that have few overlapping word, which otherwise could not be accomplished by a model that simply uses a unigram or bigram word counts as a feature. For example, (Soy Sauce, Sesame Oil) yields high similarity index because they both are ingredients for a similar type of food, but if we simply use a unigram word count feature vector and find a cosine similarity, we will get a low similarity.

The categorization trees from hierarchical clustering correctly classify approximately 79% of the odd-one-out instances, slightly more accurate than the Odd-One-Out algorithm. Hierarchical clustering, however, is slow because we need to find similarity indices between all pairs of articles in the interested pool, which runs in $O(N^2)$ time. One observation is that hierarchical clustering tends to construct trees that are less balanced than those from Odd-One-Out algorithm

Conclusion and Future Work

We perform two algorithms to analyze the similarity between two articles and construct a dichotomous tree model, in which they have tradeoffs in different aspects. The hierarchical clustering constructs a dichotomous tree by weigh of the modified K-mean algorithm and similarity index, which is calculated from the weighted dot product of the centroid of each cluster from different articles. This algorithm performs a high efficiency; however, it has a big runtime.

The odd-one-out algorithm constructs a dichotomous tree by weigh on the reward for each node, which we did not utilize in this project, but has a potential to be useful with tagging of each sub-tree. Since it is possible optimize the dichotomous tree and greedy dividing separately, one may obtain a larger data set of human odd-one-out instance response, and use it to pick the best variation of the model. (e.g., we can penalize any division that leads to incorrectness, to allow the article triplets that have less clear 2-1 division to be handled in the lower nodes.

To sum up, our algorithms added a possibility to efficiently analyze the texts without the need to account for the complex language features (e.g. Syntactic and Semantic structures). With our algorithm, we can compare the similarity between any pairs of articles with any length and construct the dichotomous tree model for any group of Wikipedia articles, which is a basis for further auto-tagging each article by determining the most relevant words from the word bag for each node.

Works Cited

Sojka, Petr. "Software Framework for Topic Modelling with Large Corpora." Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks. By Radim Eh Ek. Valletta, Malta: ELRA, 2010. 45-50. Print.

"Assessing Relevance" Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze, Introduction to Information Retrieval, Cambridge University Press. 2008

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. In Proceedings of Workshop at ICLR, 2013.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed Representations of Words and Phrases and their Compositionality. In Proceedings of NIPS, 2013.

“Vector Representations of Words.” Tensorflow.org Google Inc. 2015