Learning Word Embeddings from 10-K Filings using PyTorch

Saurabh Sehrawat
e-mail: saurabhsehrawat@gmail.com

Abstract

With the rise of alternative data in finding trading signals, Natural Language Processing (NLP) on financial documents has gained significant importance in the recent years. Word Embeddings learned from text corpus are one of the most important inputs to various NLP models, especially Deep Learning based models. In this paper, we generate word embeddings learned from corpus of 10-K filings by corporates in U.S. to S.E.C from 1993 to 2018 using word2vec model implemented in PyTorch [5]. Word Embeddings learned from general corpus of articles from Google News, Wikipedia etc are readily available online for researchers to use in their models but embeddings learned from 10-K filings are not publicly available. Using word embeddings learned from general text for NLP tasks on financial documents may not yield accurate results as it has been proven that word embeddings learned from contextual text yields better and more accurate results compared to general word embeddings. We aim to publish the word embeddings learned from 10-K filings online so that they can be used by other researchers in their NLP tasks such as document classification, document similarity, sentiment analysis, readability index etc. on 10-K filings or other financial documents.

1 Introduction

Traditionally, NLP tasks on financial documents have used bag-of-words (BOW) model combined with tf-idf weighing schemes. BOW model though simple to understand and quick to run, suffers from major drawbacks such as that it does not consider order or context of the words, is high dimensional and has sparse feature representation. Word embeddings learned from word2vec model resolves these issues by considering context of the words and generates low dimensional dense feature vectors. In [10], the authors introduced two architectures for word2vec: Continuous Bag-Of-Words (CBOW) and Skip-Gram model. The CBOW model predicts target word given context words whereas Skip-Gram model predict context words within a window given target word. We use Skip-gram model in this paper to generate word embeddings as it has been shown to generate more accurate word embeddings. We

[0]keywords: 10-K, Word Embeddings, Word2Vec, Skip-Gram, Natural Language Processing (NLP), Machine Learning, Deep Learning, Neural Networks, PyTorch, t-SNE, Cosine Similarity, Amazon AWS, Quantitative Finance, Alternative Data, Trading Signals

process over 183k 10-K filings from 1993 to 2018 to generate our text corpus. For every 10-K, we use text from sections 1A, 7A, and 7 for our corpus.

This paper is organized as follows: Section 2 briefly describes a 10-K document. Section 3 provides introduction to Word Embeddings and Word2Vec Model. Section 4 provides details about PyTorch model implementation, data preprocessing and training. In section 5, we publish the results. Finally, section 6 concludes the paper.

2 10-K Financial Document

A 10-K financial document is a comprehensive annual report filled by a publicly traded company detailing its financial performance and is required by the U.S. Securities and Exchange Commission (SEC). The SEC requires companies to publish 10-K forms so that investors have fundamental information about the companies and can make informed investment decisions. 10-K forms for individual companies can be searched and downloaded from the EDGAR database, [11], on the SEC's website.

The 10-K form is generally divided into the following sections or items: Business Overview, Risk Factors, Selected Financial Data, Management's Discussion and Analysis, Quantitative and Qualitative Disclosures about Market Risks, and Financial Statements and Supplementary Data. For our purpose, we focus on sections related to Risk Factors (Item 1A), Management's Discussion and Analysis (Item 7), and Quantitative and Qualitative Disclosures about Market Risks (Item 7A). These sections are of the most interest for NLP tasks as they contain textual information about the current and future risks, financial results and operations, and management's view about the business of the company.

3 Word Embeddings

In order to use words as inputs to neural networks we need to first convert them to vectors as neural networks cannot directly work on words. One approach is to one-hot encode these words before passing to the network. In one-hot encoding of a word, only that word will be set to one in the vector and all other words in the vocabulary will be set to zero. It results in very high dimensional and inefficient vectors of the size of whole vocabulary with most of the values in the one-hot vector set to zero. When these vectors are matrix multiplied in first hidden layer the result is mostly zero-valued outputs resulting in computational and memory wastage.

Embeddings are used to solve this problem and increase efficiency of neural networks. Embedding layer is a fully connected hidden layer with its weights called embedding weights. The number of hidden units is the embedding dimension, we use 300 as embedding dimension in this paper. Embedding layer replaces the above described matrix multiplication with embedding lookup. Replacement of matrix multiplication with lookup is possible because the multiplication of a one-hot encoded vector with a matrix returns the row of the matrix corresponding to the index of the ON element of the vector. We encode the words as integers, for e.g. "profit" is encoded as 445, "loss" as 98. Then we take the 445th row of the embedding weight matrix to get hidden layer values for "profit".

Our Word2Vec model implementation in section 4, uses the embedding layer in a two-layer neural network to learn dense vector representations of words called word embeddings. Embedding dimension of 300 is the size of each word's learned vector representation and simply stated represents the number of features encoded in the embedding. These word embeddings can be used as inputs in neural networks instead of one-hot encoded vectors. We next describe the Word2Vec model.

3.1 Word2Vec Model

Word2Vec model was first introduced by Mikolov et al. [10]. The model finds efficient dense vector representations of the words that also contain semantic information about the words. Words that show up in similar contexts will have vectors near to each other whereas different words will be further away from one another. Relationships between words can be represented by distance in vector space. Two architectures proposed in [10] to implement Word2Vec are shown below:

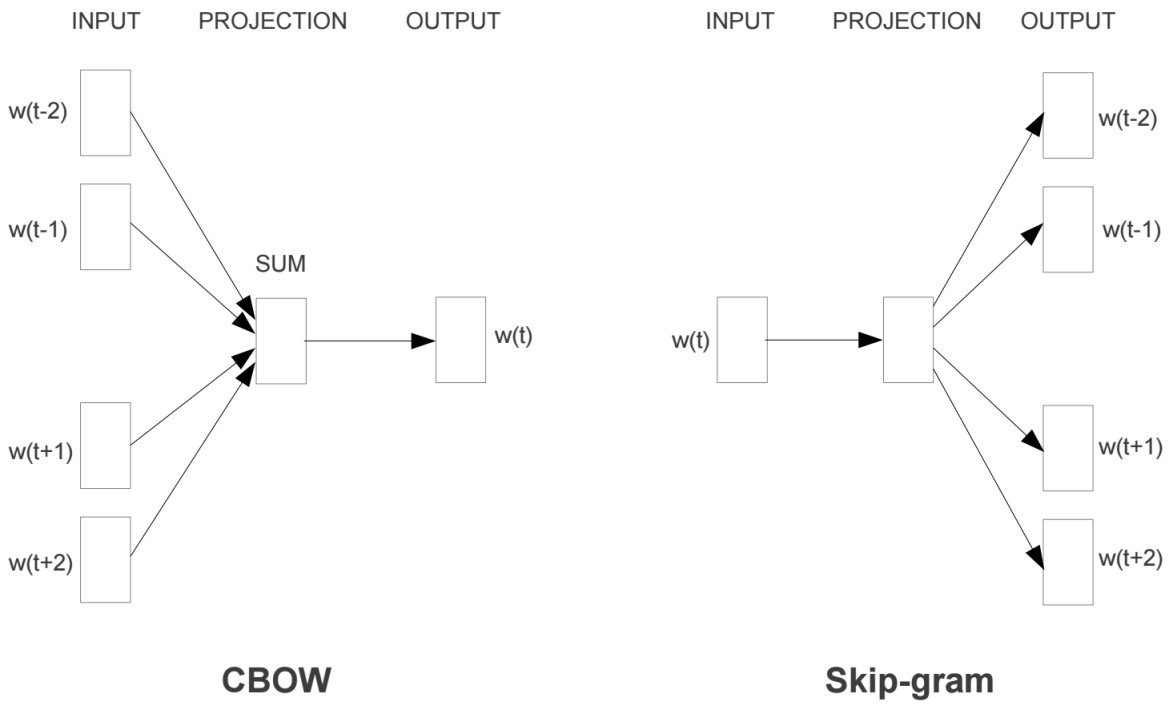

Figure 1: Word2Vec Architectures from [10]

Mikolov et al. [9] further proposed several extensions to Skip-gram model that improve both the quality of the vectors learned and the training speed. In this paper, we use Subsampling and Negative Sampling from their suggestions. In very large corpora, words like "the", "of", and "a" show up millions of time but don't provide much context to the nearby words. These words often provide less information from the rare words and can be discarded to remove some of the noise from our data and get faster training and better vector representations. This process is called Subsampling in [9]: each word w_i in the training set is discarded with

probability computed by the formula:

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}} \tag{3.1.1}$$

where $f(w_i)$ is the frequency of word w_i and t is a chosen threshold, we chose 10^{-5} as threshold as suggested in [9].

During training of a neural network, each training sample tweaks all of the weights in the network so that the network learns to predict that training sample more accurately. That means for each input even though we only have one true output we will be updating millions of weights. Given our billions of training words, this makes training the network very inefficient. Negative Sampling addresses this issue by only updating a small subset of all the weights at once. We update weights for the true example but only a small number of incorrect or noise examples.

4 PyTorch Implementation

We implement the Skip-Gram Word2Vec model in PyTorch [5] as a two-layer neural network. Since its release in 2016, PyTorch has been widely adopted by deep learning research community for its ease of building and deploying large deep learning networks. PyTorch provides **nn module** [6] which makes building neural networks much easier. We implement two embedding layers in the hidden layer of the model, one for input words and another for target and noise words. PyTorch has **nn.Embedding** layer to implement embedding layer lookup. The size of embedding layers is vocabulary size x embedding dimension. We use embedding dimension of 300 in our model. The output layer implements the negative sampling objective as a custom loss function in PyTorch. The negative sampling objective is given in [9] as:

$$log\sigma(v'^{T}_{wO} v_{wI}) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P_n(w)} \left[log\sigma(-v'^{T}_{w_i} v_{wI}) \right] \tag{4.0.1}$$

where v'^{T}_{wO} is the transpose of the embedding vector of the target word, v_{wI} is the embedding vector of the input word and $v'^{T}_{w_i}$ is the transpose of the embedding vector of the noise word. The noise words w_i are drawn from a noise distribution $w_i \sim P_n(w)$. Noise words are those words in our vocabulary which are not in the context of our input word. We use $U(w)^{3/4}$ as suggested in [9] as our noise distribution. $U(w)$ is unigram distribution and is based on the frequency of each word that shows up in our text corpus.

4.1 Data Download and Preprocessing

We download the history of 10-K filings from Prof. MacDonald's software repository [4]. The data consists of all the 10-K, 10-Q, 10-K/A, 10-K405 etc. filed by companies between 1993 and 2018, we only use 10-K files to generate our word embeddings. We have total of 183,061 10-K files in the repository. These files have already been stage one parsed to exclude markup tags, ASCII-encoded graphics, and tables. Next, we write a python script

which uses regular expressions to select the text belonging to Item 1A, Item 7 and Item 7A. We found that some of the 10-Ks are either missing some or all of these sections. We ignore such 6,123 10-Ks from our data. We further process the selected sections by removing any numeric or alphanumeric characters and only keeping alphabets in the text corpus. We lowercase all the text and remove words which occur less than 5 times in the corpus. This helps in reducing noise in the data and improves the quality of vector representations.

4.2 Training

Given the large size of our model and data, we train the model on Amazon Web Services (AWS) EC2 P3 Instance [1]. We use p3.8xlarge instance which comes with 32 vCPUs, 4 Tesla V100 GPUs and 244 GB of RAM. After preprocessing of the text corpus as explained above in section 4.1, we have approximately 1.6 billion total words in our text corpus and 160k unique words in our vocabulary. Next, we convert the words in our vocabulary into integers according to the frequency of their occurrence in the text corpus. The most frequent word is mapped to integer 0 followed by next frequent word to integer 1 and so on. For e.g. if "the" is the most frequent word in the corpus, then it is mapped to the integer 0. We discard some of the frequent words according to the subsampling equation mentioned in section 3.1. We select context target words around a given word using a window of size 5. We randomly select a number R in range $< 1; C >$, and then use R words from history and R words from the future of the current word as correct target labels as described in [10].

We train the model for 5 epochs. For each epoch, we use batch size of 51,200 to train the model. We use large batch size in order to fully utilize the 16 GB GPU Memory and computing power of Tesla V100 GPU. Each epoch takes approximately 70 minutes to train on a single GPU. We also use Adam Optimizer instead of Stochastic Gradient Descent (SGD) to train our model. Adam optimizer has two really important features as compared to SGD: momentum and adaptive learning rate. Momentum helps the algorithm to avoid getting stuck in local minima. Adam also selects different learning rates for each parameter. This speeds learning in cases where learning rates vary across parameters especially in deep neural networks. It also makes performance less sensitive to learning rates compared to SGD as they are adjusted automatically instead of manual tuning. After the model is trained for all 5 epochs, we save the input embedding layer weights as embedding vectors for our vocabulary.

5 Results

From the above trained model, we only use the input embedding weights and discard the rest of the model as we are only interested in learning the word embeddings. We upload these embeddings on GitHub for other researchers to use in their projects. They can be downloaded from:

`https://github.com/ssehrawat/10K-word-embeddings`

We gauge the quality of these word embeddings by checking how they perform on the sentiment word lists of Loughran-McDonald [7],[8]. These word lists are widely used in NLP tasks on financial documents.

5.1 t-SNE Visualizations

Sentiment word list consists of the words belonging to the following categories: Positive, Negative, Litigious, Uncertainty, Constraining, Weak Modal and Strong Modal. We use T-distributed Stochastic Neighbor Embedding (t-SNE) [3] to visualize these words using their learned embeddings. In particular, we check if we are able to differentiate different types of words into clusters by running t-SNE on their word embeddings.

We first run t-SNE on Positive, Litigious, Uncertainty, and Constraining word lists. In the resulting 2D projections of the 300 dimensional word embeddings by t-SNE, we can see similar types of words clustering together.

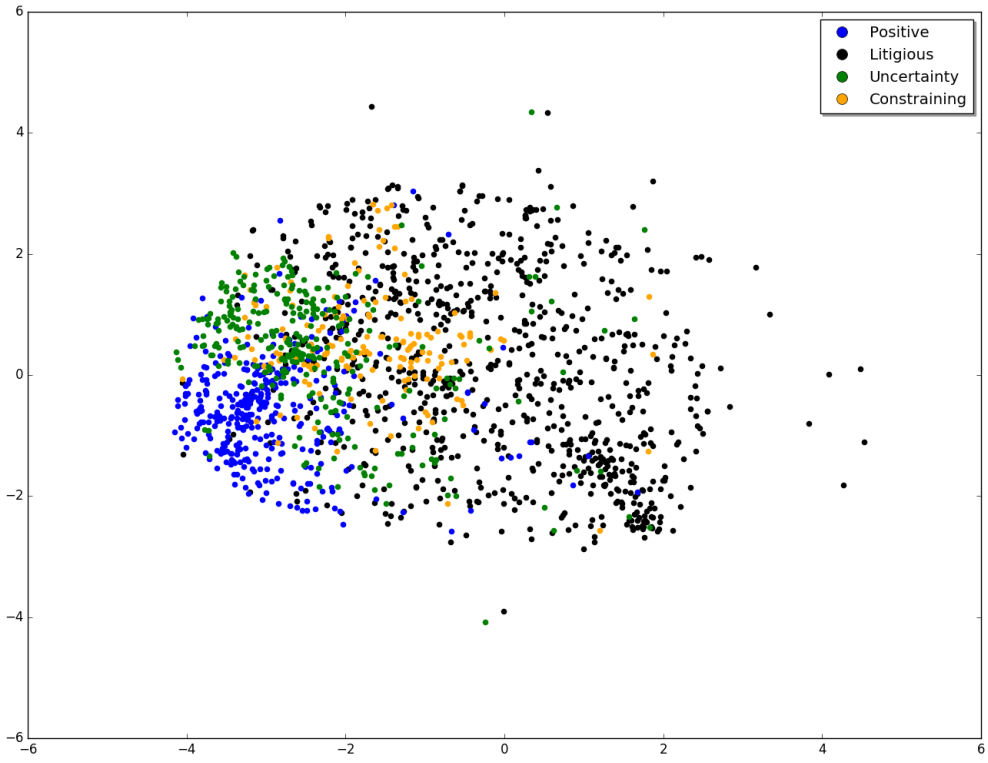

Figure 2: Positive, Litigious, Uncertainty, and Constraining words t-SNE visualization

Next we run t-SNE on Positive and Negative word lists. We again see that positive and negative words cluster with their word types.

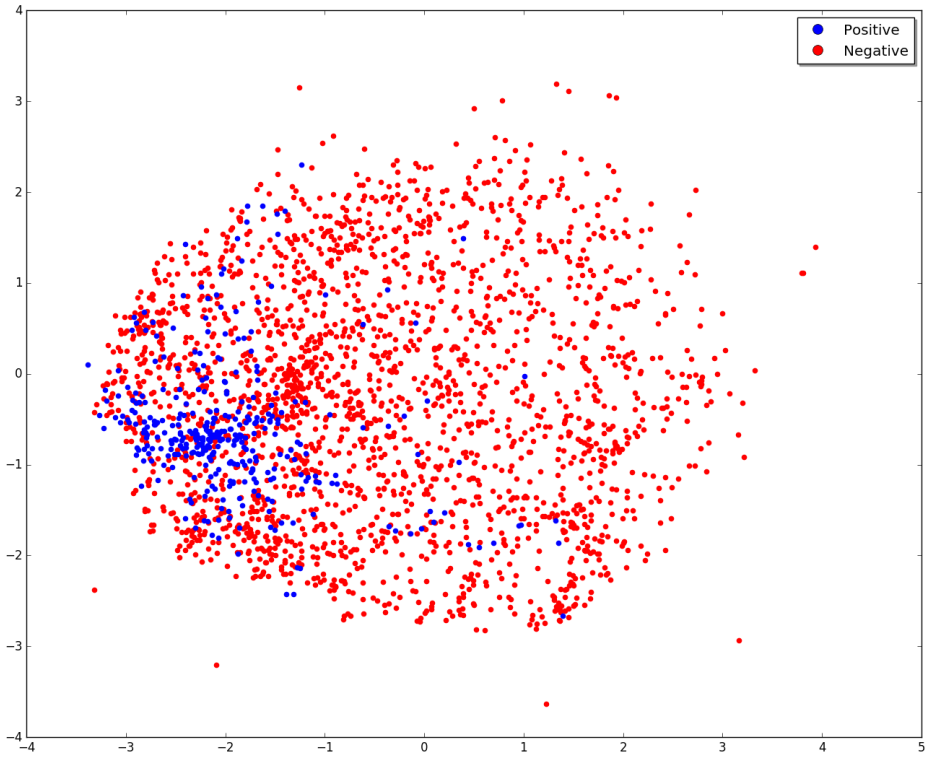

Figure 3: Positive and Negative Words t-SNE visualization

5.2 Cosine Similarity

Next, we use cosine similarity to test some common relationships between financial words. Words like "profit" and "loss", "profit" and "bankruptcy" have opposite meanings to each other and hence we would expect their word embeddings to be dissimilar as well. We calculate cosine similarity for these word pairs using sklearn cosine_similarity function:

$$cosine\_similarity(''profit'',''bankruptcy'') = -0.06$$

$$cosine\_similarity(''profit'',''loss'') = 0.26$$

We get low cosine similarity score for these word pairs confirming that their learned embeddings are dissimilar as expected. Next, we take sentiment word lists, calculate an average embedding vector for each of them and calculate their pairwise cosine similarity scores using their average vectors. We get the below cosine similarity matrix:

We would expect on average positive words to be more similar to strong modal words as compared to negative, litigious or constraining words, and above similarity scores between them confirm the same. Similarly, we would expect Negative words to be more similar to litigious, constraining, or uncertainty words as compared to positive or strong modal words, and the similarity scores above confirm the same.

	Positive	Negative	Litigious	Constraining	Uncertainty	WeakModal	StrongModal
Positive	1.00	0.53	0.36	0.44	0.51	0.50	0.63
Negative	0.53	1.00	0.79	0.68	0.73	0.60	0.48
Litigious	0.36	0.79	1.00	0.64	0.57	0.38	0.42
Constraining	0.44	0.68	0.64	1.00	0.56	0.50	0.40
Uncertainty	0.51	0.73	0.57	0.56	1.00	0.73	0.52
WeakModal	0.50	0.60	0.38	0.50	0.73	1.00	0.54
StrongModal	0.63	0.48	0.42	0.40	0.52	0.54	1.00

Table 1: Cosine Similarity Matrix for sentiment word lists

6 Conclusion

In this paper, we generate word embeddings learned from 10-K Financial Filings using the Word2Vec model. We show that using these word embeddings, we can differentiate different types of sentiment words used in NLP tasks in the financial domain. These can be used in traditional NLP models as well as inputs into latest cutting edge Machine Learning based NLP models. We aim to use these learned word embeddings in our future research on finding trading signals using alternative data as well as in other relevant NLP tasks in finance. Fellow researchers in Quantitative Finance shall benefit from our results by using them in their models.

We aim to conduct further research into determining if better word embeddings can be learned using other new models like BERT (Bidirectional Encoder Representations from Transformers) [2] as compared to Word2Vec model for applications in Quantitative Finance.

References

[1] Amazon Web Services, Inc., https://aws.amazon.com/ec2/instance-types/p3/

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv:1810.04805.

[3] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. Journal of machine learning research, 9(Nov):2579–2605, 2008.

[4] Professor Bill McDonald, https://sraf.nd.edu/data/stage-one-10-x-parse-data/

[5] PyTorch, https://pytorch.org/

[6] PyTorch, https://pytorch.org/docs/stable/nn.html

[7] Tim Loughran and Bill McDonald, https://sraf.nd.edu/textual-analysis/resources/

[8] Tim Loughran and Bill McDonald, 2016, Textual Analysis in Accounting and Finance: A Survey, Journal of Accounting Research, 54:4,1187-1230.

[9] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013b. Distributed representations of words and phrases and their compositionality. In NIPS, pages 3111–3119.

[10] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. ICLR Workshop, 2013.

[11] U.S. Securities and Exchange Commission, https://www.sec.gov/edgar/searchedgar/companysearch.html