Distributional Semantics: Word Embeddings

*A Survey of Algorithms*

By: Ryan Tatton and Sarah Yurick

# 1. Introduction

*By: Ryan Tatton and Sarah Yurick*

Distributional semantics is a subfield of natural language processing that aims to capture the meaning embedded in natural language. The foundational hypothesis for distributional semantics was proposed by Harris (1954) and claims that “difference of meaning correlates with difference of distribution.” More broadly, distributional semantics is a usage-based model of meaning in which an abstraction mechanism is applied to a large amount of text to produce a distributional model with semantic representations in the form of vectors (Lenci, 2018; Boleda, 2020).

Within this framework, numerous algorithms have been developed that all have the common goal of capturing semantic meaning. Generally, distributional semantic models can be categorized into count models and prediction models. The former develops an explicit description of each word in a given corpus based on one or more contexts (the type of context can vary from a few words to an entire document or website). The latter typically uses a neural network to learn how to optimally predict the context of a target. In a narrow sense, these types of models learn a low-dimensional *embedding*, or collection of real-valued vectors, over a vocabulary. This implicit distributional representation assumes that the co-occurrences explicitly measured by count models represent noisy data that is generated from an underlying abstract semantic structure. While carefully designed count models can still yield impressive results, prediction models that learn a word-level embedding have produced state-of-the-art results (Lenci, 2018).

## 1.1 Appeal

In distributional semantics, the geometric relationships between words gains prominence over the individual features of the vectors. The appeal of distributional representations is in the fact that they are learned from natural language data and thus provide semantic representations on a large scale. Secondly, the fact that they tend to have high multi-dimensionality allows for rich and nuanced information to be encoded (Boleda, 2020). Finally, because the vectors are made up of continuous values, their similarity is able to be measured by metrics such as cosine similarity.

## 1.2 Significance and Applications

In general, distributional semantics has proven to be useful in computational linguistics and cognitive science (Boleda, 2020). It has great potential to accelerate research in semantic change as well as polysemy and composition. In other words, distributional semantics can contribute to linguistic theories in exploratory ways.

Of course, distributional semantics can also be used as a tool for various applications (Boleda, 2020). More specifically, distributional semantic models have been applied to many tasks, including but not limited to finding semantic similarities between words, word sense disambiguation, information retrieval, data mining and named entities recognition, paraphrasing, and sentiment analysis. Word embeddings specifically have been used to analyze survey responses, to analyze verbatim comments, and for music and video recommendation systems. The tasks with which word embeddings are also tested also make up their own domain of applications, as described in section 3.1.3.

In this report, we describe, implement, and evaluate several models for learning word embeddings, specifically the Word2Vec models and SentenceMIM model. We discuss our results and compare the advantages and disadvantages of both.

# 2. Algorithms

For all three algorithms discussed, the goal is to produce vector representations (word embeddings) for every word in the vocabulary of a given corpus, such that these representations capture the semantic relationships between the words by some measure of similarity.

## 2.1 Word2Vec

*By: Sarah Yurick*

Word2Vec consists of two models to learn word embeddings: the Continuous Bag-of-Words (CBOW) model and the Continuous Skip-gram model. Both techniques use shallow artificial neural networks for training with stochastic gradient descent and backpropagation. In addition to this, both completely preserve the linear regularities among words, meaning that the gradient calculated at the output layer is directly propagated back into the embedding parameters.

Word2Vec was originally proposed by Mikolov et al. (2013) in “Distributed Representations of Words and Phrases and their Compositionality,” and various speedup techniques are described by Mikolov et al. (2013) in “Efficient Estimation of Word Representations in Vector Space.” Hence all algorithm descriptions in this section and mentions of “the original Word2Vec papers” refer to these documents.

### 2.1.1 Continuous Bag-of-Words

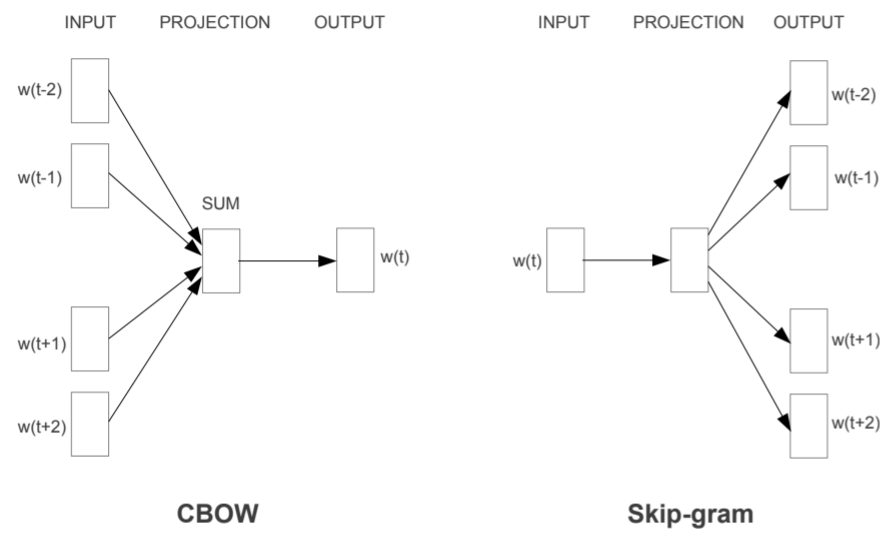
The goal of the Continuous Bag-of-Words (CBOW) model is to learn word embeddings in D < V (where V is the vocabulary size) dimensions by predicting the current word given its surrounding context. That is, for every word in the corpus, use the C > 0 words before it in the text and C words after in order to predict the current word. In this context, C is referred to as the “window size” of the words surrounding the target “center” word.

The CBOW architecture consists of a neural network with a single hidden layer. The inputs into the neural network are the one-hot encodings (vectors in V dimensions) of the context words within the specified window size. The 2C words are projected into the projection (hidden) layer; essentially, the input-to-projection step represents multiplication by a weight vector. This input-to-projection step is “continuous” because the neural network architecture is not set up in a way to preserve the ordering of the surrounding context words; thus, the CBOW model is called “continuous.” Finally, an additional weight matrix is used from the projection layer to the output in order to predict the one-hot encoding of the center “target” word. All of the weights of the neural network are learned using stochastic gradient descent and backpropagation.

The “trick” with the CBOW model is that it is a “pseudo task” of sorts; the real task is to learn the input-to-projection weights, which after training will yield the values of the word embeddings themselves. In other words, given the weight values learned for the input-to-projection layer, finding the word embedding for a given word simply consists of multiplying its one-hot encoded representation by the learned weight vector.

### 2.1.2 Continuous Skip-gram

The Skip-gram architecture is very similar to CBOW, except instead of learning the word embeddings by predicting the current word given its surrounding context, it predicts the surrounding context given the current word. Figure 1 below illustrates this difference.



*Figure 1: Neural network architectures for the CBOW and Skip-gram models. Mikolov et al. (2013).*

Conceptually, the neural network for the Skip-gram model is learned the same way as for the CBOW model, except that the input is the one-hot encoding for the center “target” word while the output is the one-hot encodings for the surrounding context words within the specified window size. Once again, learning the input-to-projection weights of the neural network is the equivalent of learning the word embeddings, which can be found by multiplying together the one-hot encoding of the word and the learned weight vector.

Because both the CBOW and Skip-gram models consider all words within the context window, this means that, depending on the window size, many of the surrounding context words may be further away from the center word. On the one hand, a larger window size is more likely to capture long-range dependencies between words; on the other hand more distant words are usually less related to the current word than to those closer to it. One way to account for this, which is especially popular to use in the Skip-gram model, is to give less weight to the distant words by sampling less from them. In other words, after specifying the window size C, for each training word, randomly specify the window size for its context words to be in the range <1; C>.

Another idea often brought up in the context of the Skip-gram model is the strategy of learning phrases of words instead of individual words. In order to do this, the unigram and bigram counts of pairs of words can be computed, and words that appear together a significant amount of the time are converted into phrases. More specifically, calculate a score:

score(wi, wj) = [ count(wiwj) - δ ] / [ count(wi) ⨉ count(wj) ] (1)

Where δ is a discounting coefficient. Bigrams with a score above a chosen threshold are converted into phrases; in order to form phrases consisting of more than two words, typically 2-4 passes are run over the text with a decreasing threshold value.

While the CBOW and Skip-gram models look very similar, they produce distinct results when trained on the same corpus. Generally, the CBOW model trains much faster (with training complexity N⨉D + Dlog2V) than the Skip-gram model (with training complexity C⨉[D+Dlog2V]), while the Skip-gram model tends to be more accurate in the relationship-based tasks designed to test the learned representations (see section 3.1.3 for a brief description of such tests). Because the Skip-gram model can take a long time to train, several speedup techniques have been proposed, as described in the following sections.

#### 2.1.2.1 Hierarchical Softmax

The objective function of the Skip-gram model is, given a sequence of training words {w1, w2, …, wT}, to maximize the average log probability:

(1/T) ∑t=1T∑-C ≤ j ≤ C, j !=0 log p(wt+j | wt) (2)

The probability p(wt+j | wt) can be defined using the softmax function:

p(wO|wI) = exp(vwO’TvwI) / ∑w=1Vexp(vw’TvwI) (3)

Where vw and vw’ are the input and output vector representations of w, and V is the number of words in the vocabulary. This means that the cost of computing the gradient of p(wO|wI) is proportional to the vocabulary size.

One way to speed up this computation is to use the hierarchical softmax instead of the full softmax. For this approximation, instead of evaluating V output nodes in the neural network, only about log2V nodes are needed. This is able to be accomplished by using a binary tree representation of the output layer, with the V words as the leaves and, for each node, explicitly represents the relative probabilities of its child nodes. The hierarchical softmax defines p(wO|wI) as:

p(w|wI) = ∏j=1L(w)-1 σ([[n(w, j+1) = ch(n(w,j))]] · vn(w,j)’TvwI) (4)

Where L(w) is the length of this path, σ(x) = 1/(1+exp(-x)), n(w, j) is the j-th node on the path from the root to word w, ch(n) is an arbitrary fixed child of an inner node n, and [[x]] is 1 if x is true and -1 otherwise. Now, the cost of computing the log and the gradient log is proportional to L(wO), which on average is less than or equal to logV.

#### 2.1.2.2 Negative Sampling

Negative sampling is an alternative speedup technique which is defined by the objective:

log σ(vwO’TvwI) + ∑i=1k Ewi~Pn(w) [log σ(-vwi’TvwI)] (5)

Which replaces every log p(wO|wI). Note that E denotes expectation. This is a simplification of Noise Contrastive Estimation; the task is to distinguish the target word wO from draws from the noise distribution Pn(w) by using logistic regression, where there are k negative samples for each data sample.

More intuitively, negative sampling suggests that a couple of the context words at random should be selected to predict in the Skip-gram architecture. This makes it easier to train because instead of finding the similarity between the current word and all other words in the vocabulary, instead it just finds the vectors of several randomly chosen words which approximate the full vocabulary. Thus, this is a sampling-based approximation to the softmax function.

#### 2.1.2.3 Subsampling of Frequent Words

The final speedup technique discussed is called Subsampling Frequent of Words. The intuitive idea is that very common words (such as “the”) are going to be associated with the vast majority of the words in the vocabulary, purely due to their high frequency. This means that observing a very frequent word with other words is not very meaningful, while observing less frequent words with other words is in fact meaningful. With this in mind, define the probability that a word wi in the training set that has frequency f(wi) will be discarded as:

P(wi) = 1 - sqrt(t/f(wi)) (6)

Where t is a threshold value, typically around 10-5. Thus, with this approach, words with frequency greater than t are aggressively subsampled, while still preserving the ranking of the frequencies.

## 2.2 GloVe

*By: Sarah Yurick*

Although not implemented for this project, it is important for the sake of completeness to describe another family of algorithms used to learn word embeddings through an example of another famous model. The Word2Vec techniques discussed in section 2.1 are part of a family of algorithms called local context window methods. On the other hand, there are other methods which can be used to learn word embeddings which fall under a family of algorithms called global matrix factorization methods, the most famous of which is Latent Semantic Analysis (LSA). Global matrix factorization methods such as LSA are count-based models, meaning that they construct a matrix consisting of the counts of word pairs in the corpus, which is called a co-occurrence matrix.

In LSA, the co-occurrence matrix is decomposed into a lower-dimensional representation; in general, the idea is that statistical information about the corpus as a whole is able to be captured using this “global” co-occurrence matrix. However, a disadvantage of this is that very common words will contribute a disproportionate amount to the matrix. Thus, Global Vectors (GloVe) is a global log-bilinear regression model which attempts to take advantage of both families, as proposed by Pennington et al. (2014).

The intuition behind GloVe is best illustrated through an example. Given a matrix of word-word co-occurrence probabilities, construct a table of the ratio of probabilities, as shown in Table 1 below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *Probability and Ratio* | *k = solid* | *k = gas* | *k = water* | *k = fashion* |
| *P(k|ice)* | 1.9 ⨉ 10-4 | 6.6 ⨉ 10-5 | 3.0 ⨉ 10-3 | 1.7 ⨉ 10-5 |
| *P(k|steam)* | 2.2 ⨉ 10-5 | 7.8 ⨉ 10-4 | 2.2 ⨉ 10-3 | 1.8 ⨉ 10-5 |
| *P(k|ice) / P(k|steam)* | 8.9 | 8.5 ⨉ 10-2 | 1.36 | 0.96 |

*Table 1: Co-occurrence probabilities for target words ice and steam, with selected context words solid, gas, water, and fashion. Pennington et al. (2014).*

In Table 1, the value in the first row and first column is the probability of the words “solid” and “ice” appearing together, while the value in the second row and first column is the probability of the words “solid” and “steam” appearing together. Taking the ratio of the former probability to the latter probability yields the value 8.9 depicted in the third row and first column; because this value is greater than 1, it is easily inferred without referencing the other values that P(k|ice) must be larger than P(k|steam). If the second probability is more likely than the first, then the ratio will be less than 1, as indicated in the second column. Finally, if the probabilities are roughly equal, the ratio will be approximately 1, as demonstrated in the third and final columns. Thus, the intuition behind using the ratio of two pairs of co-occurring words is that it can give insight into how similar two words are with respect to a third word.

GloVe predicts surrounding words by maximizing the probability of a context word occurring given a center word, by performing dynamic logistic regression. After constructing the co-occurrence matrix X, the goal is to create word vectors which show how every pair of words co-occur. In other words, let Xik denote how often word i appears with word k, Xi denote how often word i appears, and Pik denote Xik/Xi. The goal is to learn word vectors wi and wk’ such that:

wiTwk’ = log(Pik) = log(Xik) - log(Xi) (7)

Because log(Xi) is independent of k, it can be treated as the bias term, and for reasons involving symmetry the bias terms bi and bk’ can be associated with each word:

wiTwk’ + bi + bk’ = log(Xik) (8)

Thus, the task of linear regression is to minimize the objective function J associated with the constraint:

J = ∑i, j = 1V f(Xij)( wiTwk’ + bi + bk’ - logXij)2 (9)

Where V is the vocabulary size. In other words, minimize the squared error of the constraint as well as the function f to weight each of the word pairs. One option for defining this function is:

f(x) = (x/xmax)α if x < xmax and 1 otherwise (10)

Where α is a constant which is commonly suggested to be 3/4 or 1. Intuitively, the purpose of having the weighting function f is to deal with co-occurrences that are rare or that for whatever reason carry less information than others.

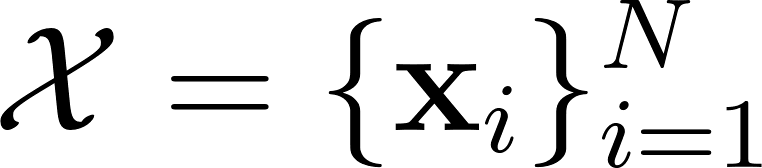
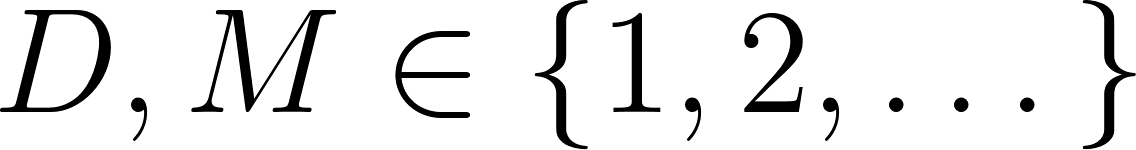
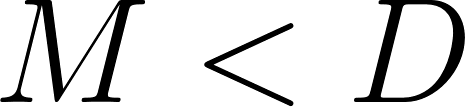
Thus, GloVe is using linear regression to learn two vectors, W and W’, either of which may be used to represent the word embeddings. At a very high level, GloVe is forcing the model to learn a linear relationship based on the co-occurrence matrix.

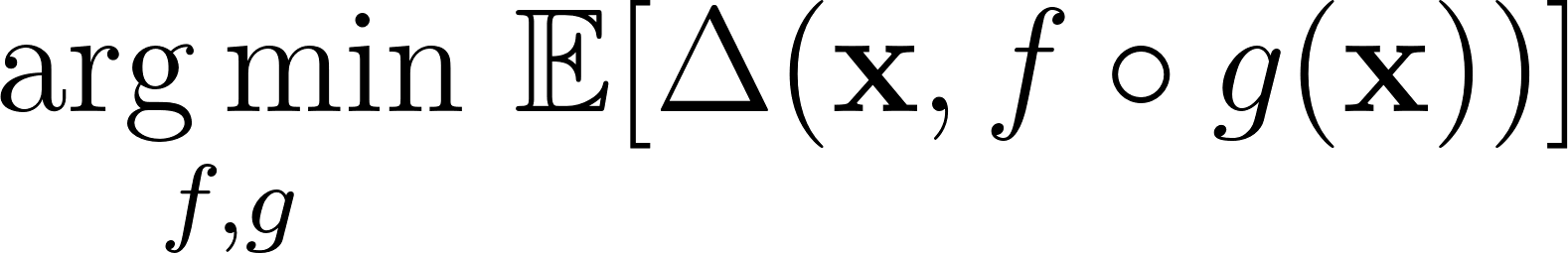
The computational complexity of GloVe is O(|C|0.8), where C is the corpus size; thus, GloVe is very fast to train, but on the downside must use a lot of memory to construct the co-occurrence matrix.

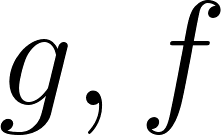
## 2.3 SentenceMIM

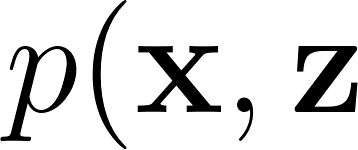
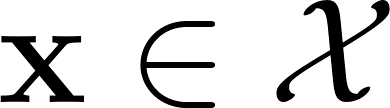
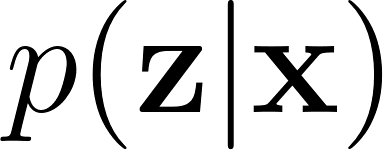
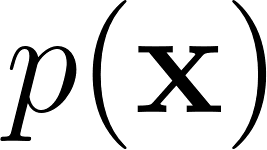
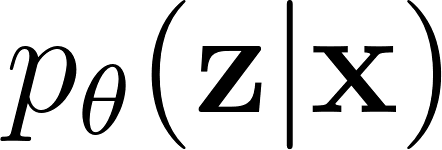
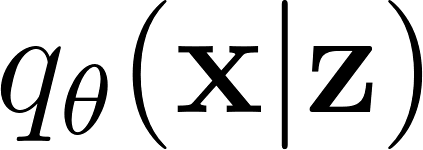
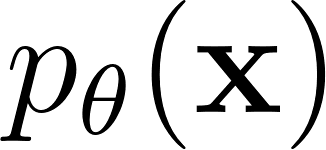
*By: Ryan Tatton*

### 2.3.1 Variational Autoencoder

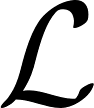
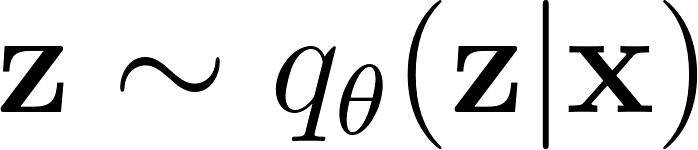
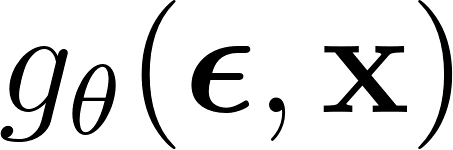
The autoencoder (AE) is a particular kind of neural network in which the general task is to learn an optimal representation of the input data. Formally, let [](https://www.codecogs.com/eqnedit.php?latex=%5Cmathcal%7BX%7D%20%3D%20%5C%7B%5Cmathbf%7Bx%7D_i%5C%7D_%7Bi%3D1%7D%5EN#0) be a set of [](https://www.codecogs.com/eqnedit.php?latex=D#0)-dimensional examples. Let [](https://www.codecogs.com/eqnedit.php?latex=f%20%3A%20%5Cmathbb%7BR%7D%5ED%20%5Crightarrow%20%5Cmathbb%7BR%7D%5EM%20#0) and [](https://www.codecogs.com/eqnedit.php?latex=g%3A%20%5Cmathbb%7BR%7D%5EM%20%5Crightarrow%20%5Cmathbb%7BR%7D%5ED#0) be two functions where [](https://www.codecogs.com/eqnedit.php?latex=D%2C%20M%20%5Cin%20%5C%7B1%2C%202%2C%20%5Cdots%5C%7D#0) and [](https://www.codecogs.com/eqnedit.php?latex=M%20%3C%20D#0). Then the objective of the autoencoder is to find a mapping from the high-dimensional data space to a low dimensional nonlinear manifold that minimizes the expected reconstruction loss

[](https://www.codecogs.com/eqnedit.php?latex=%5Cunderset%7Bf%2C%20g%7D%7B%5Carg%5Cmin%7D%20~%20%5Cmathbb%7BE%7D%5B%5CDelta(%5Cmathbf%7Bx%7D%2C%20f%20%5Ccirc%20g(%5Cmathbf%7Bx%7D))%5D#0) (11)

where [](https://www.codecogs.com/eqnedit.php?latex=g%2C%20f#0) represent the encoder and decoder networks, respectively (Bishop, 2006; Bank et al., 2020).

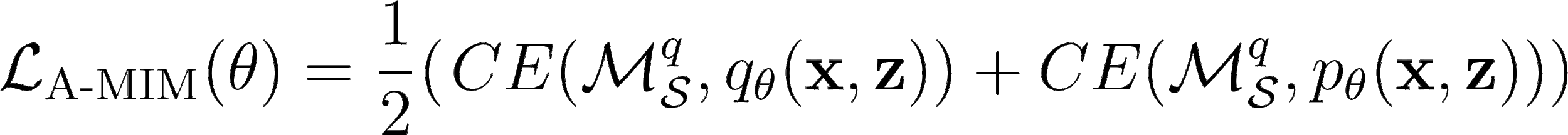
The variational autoencoder (VAE) (Kingma and Welling, 2014) frames the task of reconstruction as a problem in variational Bayesian inference. That is, suppose the aforementioned representation of the data can be described by the latent variable [](https://www.codecogs.com/eqnedit.php?latex=%5Cmathbf%7Bz%7D#0). Given the probabilistic model [](https://www.codecogs.com/eqnedit.php?latex=p(%5Cmathbf%7Bx%7D%2C%20%5Cmathbf%7Bz%7D#0), where [](https://www.codecogs.com/eqnedit.php?latex=%5Cmathbf%7Bx%7D%20%5Cin%20%5Cmathcal%7BX%7D#0), the reconstruction task becomes finding an approximation to the posterior [](https://www.codecogs.com/eqnedit.php?latex=p(%5Cmathbf%7Bz%7D%20%7C%20%5Cmathbf%7Bx%7D)#0) and evidence [](https://www.codecogs.com/eqnedit.php?latex=p(%5Cmathbf%7Bx%7D)#0) (Bishop, 2006). Toward this objective, the encoder and decoder are considered probabilistic and described by their respective parametric distributions [](https://www.codecogs.com/eqnedit.php?latex=p_%7B%5Ctheta%7D(%5Cmathbf%7Bz%7D%20%7C%20%5Cmathbf%7Bx%7D)#0) and [](https://www.codecogs.com/eqnedit.php?latex=q_%7B%5Ctheta%7D(%5Cmathbf%7Bx%7D%20%7C%20%5Cmathbf%7Bz%7D)#0). By learning a global set of variational parameters [](https://www.codecogs.com/eqnedit.php?latex=%5Ctheta#0), the cost of inference of the latent posterior is amortized, allowing for tractable density approximation. The marginal log likelihood [](https://www.codecogs.com/eqnedit.php?latex=p_%7B%5Ctheta%7D(%5Cmathbf%7Bx%7D)#0) can be decomposed as

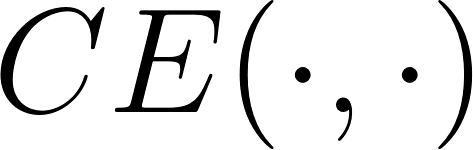
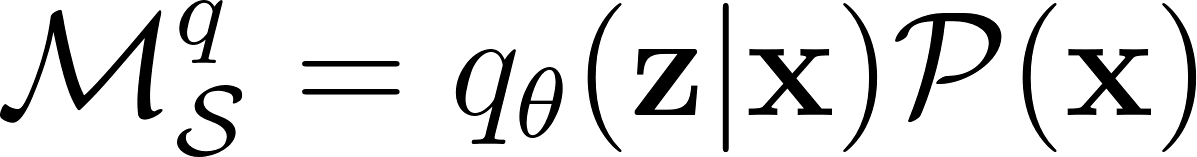
[](https://www.codecogs.com/eqnedit.php?latex=%20%5Clog%20p_%7B%5Ctheta%7D(%5Cmathbf%7Bx%7D)%20%5Cgeq%20%5Cmathcal%7BL%7D_%7B%5Ctheta%7D(%5Cmathbf%7Bx%7D)%20%3D%20-%20%5Cmathcal%7BD%7D_%7B%5Ctext%7BKL%7D%7D(q_%7B%5Ctheta%7D(%5Cmathbf%7Bz%7D%20%7C%20%5Cmathbf%7Bx%7D)%20%7C%7C%20p_%7B%5Ctheta%7D(%5Cmathbf%7Bz%7D))%20%2B%20%5Cmathbb%7BE%7D_%7Bq_%7B%5Ctheta%7D(%5Cmathbf%7Bz%7D%20%7C%20%5Cmathbf%7Bx%7D)%7D%5B%5Clog%20p_%7B%5Ctheta%7D(%5Cmathbf%7Bx%7D%20%7C%20%5Cmathbf%7Bz%7D)%5D#0) (12)

where the first term on the right-hand side is the Kullback-Leibler (KL) divergence between the approximate and true posteriors. The second term is termed the expected variational lower bound (ELBO) since the KL divergence is nonnegative. Thus, the learning objective is to maximize [](https://www.codecogs.com/eqnedit.php?latex=%5Cmathcal%7BL%7D#0) (or minimize [](https://www.codecogs.com/eqnedit.php?latex=%5Cmathcal%7BD%7D_%7B%5Ctext%7BKL%7D%7D#0)). To avoid the intolerable variance from sampling [](https://www.codecogs.com/eqnedit.php?latex=%5Cmathbf%7Bz%7D%20%5Csim%20q_%7B%5Ctheta%7D(%5Cmathbf%7Bz%7D%20%7C%20%5Cmathbf%7Bx%7D)%20#0), Kingma and Welling (2014) describe a reparametrization that expresses [](https://www.codecogs.com/eqnedit.php?latex=%5Cmathbf%7Bz%7D#0) as a deterministic transformation [](https://www.codecogs.com/eqnedit.php?latex=g_%7B%5Ctheta%7D(%5Cboldsymbol%5Cepsilon%2C%20%5Cmathbf%7Bx%7D)#0) where [](https://www.codecogs.com/eqnedit.php?latex=p(%5Cboldsymbol%5Cepsilon)#0) is typically chosen to be the standard normal distribution. With this, it is possible to perform standard nonlinear optimization techniques to find the optimal [](https://www.codecogs.com/eqnedit.php?latex=%5Ctheta#0).

### 2.3.2 Mutual Information Machine (MIM)

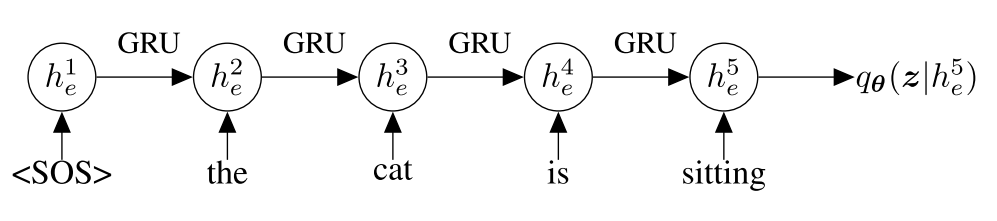
Livne et al. (2019) notes that the ELBO learning objective provided by Kingma and Welling (2014) is susceptible to (1) assigning high probability to unlikely regions of the data distribution; and (2) learning an encoder that essentially ignores the data and simply learns the latent prior, a phenomenon termed posterior collapse. The authors provide the alternative MIM (Mutual Information Machine, the name given to their model) learning objective, that demonstrates substantial improvement. Specifically, the asymmetric version of their learning objective is defined as

[](https://latex-staging.easygenerator.com/eqneditor/editor.php?latex=%20%5Cmathcal%7BL%7D_%7B%5Ctext%7BA-MIM%7D%7D(%5Ctheta)%20%3D%20%5Cfrac%7B1%7D%7B2%7D%20(%5Cleft%20CE(%5Cmathcal%7BM%7D%5Eq_%7B%5Cmathcal%7BS%7D%7D%2C%20q_%7B%5Ctheta%7D(%5Cmathbf%7Bx%7D%2C%20%5Cmathbf%7Bz%7D))%20%2B%20CE(%5Cmathcal%7BM%7D%5Eq_%7B%5Cmathcal%7BS%7D%7D%2C%20p_%7B%5Ctheta%7D(%5Cmathbf%7Bx%7D%2C%20%5Cmathbf%7Bz%7D))%20%5Cright)%20%20#0) (13)

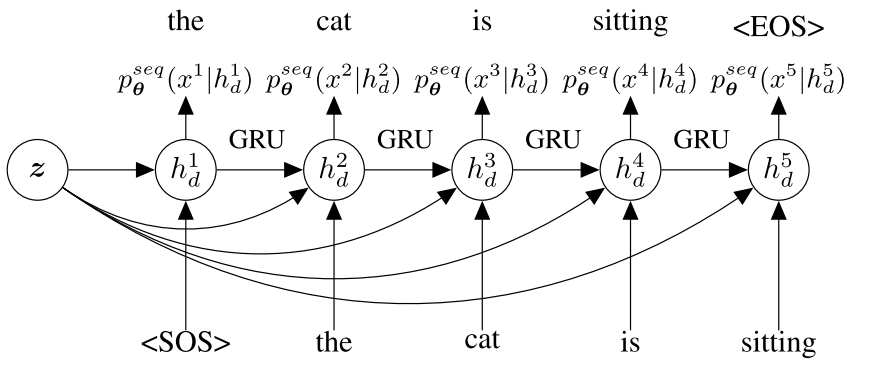
where [](https://www.codecogs.com/eqnedit.php?latex=CE(%5Ccdot%2C%20%5Ccdot)#0) is cross-entropy and [](https://www.codecogs.com/eqnedit.php?latex=%5Cmathcal%7BM%7D%5Eq_%7B%5Cmathcal%7BS%7D%7D%20%3D%20q_%7B%5Ctheta%7D(%5Cmathbf%7Bz%7D%20%7C%20%5Cmathbf%7Bx%7D)%20%5Cmathcal%7BP%7D(%5Cmathbf%7Bx%7D)#0) is the sample mixture distribution according to the approximate posterior and anchor data prior. With this objective, the MIM is able to learn a consistent encoder and decoder with respective to the underlying joint distribution that maintains high mutual information and low marginal entropy with respect to [](https://www.codecogs.com/eqnedit.php?latex=%5Cmathbf%7Bx%7D#0) and [](https://www.codecogs.com/eqnedit.php?latex=%5Cmathbf%7Bz%7D#0).

### 2.3.3 Using MIM for Sequential Reconstruction and Word Embedding

In a follow-up paper to their original work on developing the mutual information machine (MIM), Livne et al. (2020) adapt the recurrent neural network architecture first described by Bowman et al. (2015) to use the MIM objective in the context language modelling. Termed SentenceMIM, Figures 2 and 3 show the encoder and decoder architectures, respectively. The authors use (13) as the learning objective.



*Figure 2: A probabilistic encoder implemented as a recurrent neural network using gated recurrent units. Livne et al. (2020).*

**

*Figure 3: A probabilistic decoder implemented as an auto-regressive recurrent neural network using gated recurrent units. Each hidden unit is conditioned on the latent variables and all previous words in the sentence. The output of each unit is the categorical probability of a word. Livne et al. (2020).*

The authors also provide an alternative to the ELBO, called MIM-ELBO or MELBO, which does not involve the evaluation of the log-likelihood of the latent marginal.

It is important to note that Livne et al. (2020) focuses their evaluation of SentenceMIM on sentence-level reconstruction tasks, such as question-answering and interpolation between sentences, in order to demonstrate the robustness of the MIM objective. While not able to be verified due to an incomplete implementation, it is hypothesized that the word embedding that is also learned during the training process would produce impressive results that is able to capture sentence-level, directionally-sensitive semantics. This hypothesis is based on the observation that the sentence reconstruction task requires the embedding to encode the sequential distributional information, which implicitly captures the context-sensitive meaning of the vocabulary.

# 3. Implementations

*By: Sarah Yurick*

We implemented the Word2Vec and SentenceMIM models to learn word embeddings on two corpora: Shakespeare and IMDb. The Shakespeare corpus contains lines from a variety of Shakespeare’s plays, while the IMDb corpus contains user reviews from the IMDb online database of films and television programs. Both datasets can be imported as TensorFlow Datasets but were downloaded for easier use. Due to memory limitations, we only train with ~250,000 characters from the original corpora, which amounts to the corpora containing roughly 45,000 words each.

We use two different metrics to evaluate the learned word embeddings from the Shakespeare and IMDb corpora. The first is a correlation test with bi-gram frequencies reviewed by Bakarov (2018). Given an evaluation file, we use NLTK to find all of the bi-grams in the evaluation file and sort them from most to least occurrences. Then, for each bi-gram, calculate the cosine similarity (or, alternatively, the Euclidean distance) between the two word embeddings, and sort them from most to least similar. Finally, with these two ranked bi-gram lists, we calculate the Spearman rank correlation coefficient ρ between them using:

ρ = 1 - [ 6 ∑ di2 ] / [ n(n2 - 1) ] (14)

Where d is the difference between the numerical rankings of a bi-gram and n is the number of bi-grams. The score will be between -1 and +1, where -1 indicates a perfectly negative correlation, 0 is no correlation, and +1 is a perfectly positive correlation. We chose the Spearman coefficient because of its ability to capture nonlinear correlations.

The second metric we use is perplexity, for which we use the equation:

PP(s) = 2 -1/n log p(s) (15)

Where s is the evaluation text and n is the number of words in the text. In order to do this, we must find a way to translate word embeddings into probabilities. For this, researcher Sarah Yurick proposes the following strategy: given an evaluation file, for every word, calculate its cosine similarity (or, alternatively, the Euclidean distance) to every other word in the vocabulary. The intuition behind this is that because both Word2Vec and SentenceMIM are context-based and sentence-based models, respectively, words which appear together should have more similar word embeddings than words which never appear together. Thus, the probability of word j following word i is defined as:

Pr(wj | wi) = S(wi, wj) / [ ∑ k ∊ V, k != i S(wi, wk) ] (16)

Where S is the similarity measure between the word vectors and V is the vocabulary. This is meant to mimic bi-gram probabilities in that we are only using the previous word i to predict the probability of word j. It also mimics the objective function of the Skip-gram model as discussed in section 2.1.2.1 but is intended to be more generalizable to other word embedding learners. The remainder of the calculations for perplexity are straightforward.

We use two evaluation files containing unseen text for Shakespeare and IMDb, each which contain roughly 120 words.

## 3.1 Word2Vec

*By: Sarah Yurick*

Both the Continuous Bag-of-Words (CBOW) and Skip-gram Word2Vec models were implemented in Python 3.7 and ran in Jupyter Notebooks also with Python version 3.7. The implementation utilizes four Python files: “util.py,” which contains some general methods to be used by the other scripts; “tokenizer.py,” which contains methods to tokenize the text with various strategies; “w2v\_models.py,” which contains the actual implementations for the CBOW and Skip-gram models; and “word2vec.py,” which is how the user specifies the parameters with which to run the models.

All Python scripts, the Jupyter Notebooks (one for training Shakespeare, one for training IMDb, and one for evaluations and research extensions), and data files referenced can be found in the “word2vec/” directory on csevcs; note that the datasets used are listed in the “data/” directory on csevcs for the convenience, but in order to be run properly must be within the “word2vec/” directory. The remainder of this section discusses the specifics regarding learning word embeddings for three datasets; the first two texts are the Shakespeare and IMDb corpora, and the third task was implemented out of the researcher’s personal interests as well as one of several ways to go beyond the requirements of the project. The links to all datasets referenced can be found in the “README.md” file in the “word2vec/” directory on csevcs.

For the Shakespeare corpus, ten word embeddings were learned, five using the CBOW model and the other five using the Skip-gram model; training can be found in “word2vec\_shakespeare.ipynb” in the “word2vec/” directory. The neural networks for CBOW and Skip-gram were built and trained with the TensorFlow Keras library. First, the one-hot encodings for all of the tokens within the vocabulary are created using the “initialize\_nn\_parameters” method in “w2v\_models.py.” For this, I used a window size of C = 10 as also used in the original Word2Vec paper (Mikolov et al., 2013) and in order to be comparable with SentenceMIM. Depending on whether it is training CBOW or Skip-gram, either the word or context encodings are used as the inputs into the neural network. The neural network is trained for 1000 epochs, which takes around 2 hours to complete per model. The word embeddings learned are all 100 dimensions in order to be within the same magnitude of the word embeddings learned in the original Word2Vec paper.

Before training, several parameters are varied in order to evaluate their effectiveness on the learned word embeddings, namely, how to tokenize, which activation function and optimizer to use, and whether or not to introduce randomness when constructing the contexts. For tokenization, I explored two methods: the first is “basic” tokenization which simply removes all non-alphanumeric characters and splits words by space, and the second is “super tokenization” which utilizes the concepts of learning phrases and the subsampling of frequent words as described in section 2.1. While the original Word2Vec paper uses softmax and stochastic gradient descent, I also explored using the sigmoid function and Adam optimizer due to their generally popular use among machine learning experts. While the Adam optimization algorithm is faster than stochastic gradient descent, it is known to have convergence problems. The final parameter that was varied was “randomness,” which refers to randomly selecting an integer in <1; C> to use as the context window size for each word.

The word embeddings are evaluated using the bi-gram frequency and perplexity metrics described above. These methods are implemented in “evaluation\_metrics.py” in the “word2vec/” directory, and the user can run them with the “evaluate.py” script. The evaluators are run in “word2vec\_evaluation.ipynb” and the results are also listed in the first table in section 3.1.2.

The same strategies outlined above were tested for learning the word embeddings on the IMDb text. Training can be found in “word2vec\_imdb.ipynb” in the “word2vec/” directory. Once again, the evaluators are run in “word2vec\_evaluation.ipynb” and the results are listed in the second table in section 3.1.2.

### 3.1.1 Research Extensions

#### 3.1.1.1 Smoothing

For the first Word2Vec research extension, I explored learning a vector representation for an unknown token “UNK.” In the original Word2Vec paper, the underlying assumption was that every single possible word in the vocabulary was able to be represented because the researchers had a large amount of resources so as to not worry too much about memory limitations. As mentioned in the previous section, however, it took around 2 hours to train neural networks on a corpus containing 45,000 words, and selecting larger texts led to either unreasonably long training times at best and memory issues at worst. Thus, I propose using smoothing to account for the fact that it is impossible to account for every word in the English vocabulary.

In order to implement this, a method “out\_of\_vocabulary\_smoothing” was created in “tokenizer.py,” which replaces words in the corpus which appear only once with the “UNK” token. Training is then able to proceed normally. Training and evaluation can be found in the research extension section of “word2vec\_evaluation.ipynb” and a table of the results can be found in section 3.1.2.

#### 3.1.1.2 Netflix Application

When exploring the applications of word embeddings, one commonly suggested application was for recommendation systems. Thus, for the second “research extension,” I explored using word embeddings to recommend TV shows and movies on the popular streaming service Netflix.

For this task, I structured the “center” and “context” words to train differently than for raw text. Instead of using text files, I started off with a CSV file containing roughly 6,500 titles available on Netflix, which included information such as the director, actors, genre, country, year released, parental guidelines rating, and a description of the program. To preprocess the data, all names of directors and actors were converted to single tokens (i.e. “Morgan Freeman” became “morganfreeman”). The genres, countries, and parental guidelines were also tokenized in a similar manner (for example, “United States” to “unitedstates”). The program descriptions were tokenized by simply splitting by space, removing all non-alphanumeric characters, and converting to lowercase.

After preprocessing, a vocabulary could be generated with all of the title, director, actors, genre, country, year released, parental guidelines rating, and description tokens and then one-hot encoded. This way, the neural network architecture was basically the same as for Word2Vec (specifically the Skip-gram model); once again, it was trained for 1000 epochs to learn 100-dimensional vectors using softmax and stochastic gradient descent. This time, however, the inputs into the neural network were the one-hot encoded TV show and movie titles, and the outputs were all of the (one-hot encoded) tokens associated with that TV show or movie. After training, as with Word2Vec, the learned word embeddings were the equivalent of the learned weights in the input-to-projection layer of the neural network. This described code can be found in the “word2vec\_application.py” script in the “word2vec/” directory.

In order to generate recommendations, the “recommender.py” script is used. The user is able to input a TV show or movie that they enjoy, e.g. “friends” and generate a list of recommended titles. These are generated by finding the k (default k = 10) closest word vectors to the input word, calculated with Euclidean distance. Training and generated recommendations can be found in “word2vec\_evaluation.ipynb” as well as in section 3.1.2.

### 

### 3.1.2 Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *Bi-gram score (cosine similarity)* | *Bi-gram score (Euclidean distance)* | *Perplexity (cosine similarity)* | *Perplexity (Euclidean distance)* |
| *CBOW with basic tokenization, softmax, and SGD* | -0.10 | 0.12 | 6623 | 5943 |
| *Skip-gram with basic tokenization, softmax, and SGD* | 0.18 | -0.21 | 5650 | 5952 |
| *CBOW with super tokenization, softmax, and SGD* | -0.16 | -0.07 | 5600 | 6567 |
| *Skip-gram with super tokenization, softmax, and SGD* | -0.15 | 0.17 | 5466 | 6253 |
| *CBOW with basic tokenization, sigmoid, and Adam* | -0.03 | 0.04 | 7516 | 4943 |
| *Skip-gram with basic tokenization, sigmoid, and Adam* | 0.17 | 0.11 | 7130 | 3124 |
| *CBOW with super tokenization, softmax, SGD, and randomness* | 0.14 | -0.23 | 6188 | 5363 |
| *Skip-gram with super tokenization, softmax, SGD, and randomness* | 0.08 | 0.04 | 5713 | 5549 |
| *CBOW with super tokenization, sigmoid, Adam, and randomness* | -0.27 | 0.32 | 4810 | 5317 |
| *Skip-gram with super tokenization, sigmoid, Adam, and randomness* | 0.25 | -0.04 | 6247 | 4799 |

*Table 2: Evaluation results of various experiments for learning word embeddings on the Shakespeare corpus, as evaluated on a 120-word file of unseen Shakespeare text.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *Bi-gram score (cosine similarity)* | *Bi-gram score (Euclidean distance)* | *Perplexity (cosine similarity)* | *Perplexity (Euclidean distance)* |
| *CBOW with basic tokenization, softmax, and SGD* | 0.16 | -0.16 | 8973 | 7337 |
| *Skip-gram with basic tokenization, softmax, and SGD* | 0.04 | -0.02 | 7790 | 7249 |
| *CBOW with super tokenization, softmax, and SGD* | -0.35 | 0.28 | 8317 | 6551 |
| *Skip-gram with super tokenization, softmax, and SGD* | 0.03 | 0.01 | 7291 | 7064 |
| *CBOW with basic tokenization, sigmoid, and Adam* | 0.10 | -0.06 | 7476 | 6719 |
| *Skip-gram with basic tokenization, sigmoid, and Adam* | 0.21 | -0.02 | 8187 | 3904 |
| *CBOW with super tokenization, softmax, SGD, and randomness* | -0.12 | 0.14 | 6428 | 7445 |
| *Skip-gram with super tokenization, softmax, SGD, and randomness* | 0.01 | -0.10 | 7440 | 6743 |
| *CBOW with super tokenization, sigmoid, Adam, and randomness* | 0.17 | -0.20 | 4875 | 7396 |
| *Skip-gram with super tokenization, sigmoid, Adam, and randomness* | -0.11 | 0.01 | 7021 | 6036 |

*Table 3: Evaluation results of various experiments for learning word embeddings on the IMDb corpus, as evaluated on a 120-word file of unseen IMDb text.*

Because the bi-gram score depends on bi-gram frequencies, it is also helpful to look at a larger evaluation corpus in order to capture more of the trends than can be captured by a 120-word text. Therefore, I generated two additional files, “large\_shakespeare\_eval.txt” and “large\_imdb\_eval.txt” each with 250,000 characters of unseen text. For the sake of time, I only evaluate the bi-gram scores on the word embeddings learned by basic tokenization, softmax, and stochastic gradient descent:

|  |  |  |
| --- | --- | --- |
|  | *Bi-gram score (cosine similarity)* | *Bi-gram score (Euclidean distance)* |
| *CBOW on the Shakespeare corpus* | -0.12 | -0.03 |
| *Skip-gram on Shakespeare corpus* | -0.003 | -0.08 |
| *CBOW on the IMDb corpus* | -0.11 | -0.04 |
| *Skip-gram on the IMDb corpus* | -0.05 | -0.10 |

*Table 4: Evaluation results of various experiments for learning word embeddings on the Shakespeare and IMDb corpora, as evaluated on 250,000-character files of unseen Shakespeare IMDb text, respectively.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | *Bi-gram score (cosine similarity)* | *Bi-gram score (Euclidean distance)* | *Perplexity (cosine similarity)* | *Perplexity (Euclidean distance)* |
| *CBOW on the Shakespeare corpus* | 0.03 | 0.11 | 3126 | 2429 |
| *Skip-gram on the Shakespeare corpus* | -0.10 | 0.14 | 2660 | 2463 |
| *CBOW on the IMDb corpus* | -0.04 | -0.08 | 4732 | 3135 |
| *Skip-gram on the IMDb corpus* | -0.13 | 0.06 | 3073 | 3104 |

*Table 5: Results of research extension #1, smoothing with an “UNK” out of vocabulary (OOV) token, using softmax and stochastic gradient descent on the Shakespeare and IMDb corpora. Evaluation files were the same 120-word texts as in the original experiments.*

|  |  |
| --- | --- |
| *Title* | *Recommendations* |
| Friends | 1. Beauties of the Night 2. Hurricane Bianca 3. Trailer Park Boys: X-Mas Special 4. Llama Llama 5. The Hard Way 6. The Perfect Dictatorship 7. #Selfie 8. Hisss 9. Below Her Mouth 10. Amar’s Hands |
| Gossip Girl | 1. Bibi & Tina: Girls Versus Boys 2. The Force 3. The Bleeder 4. Top Grier 5. The Gentleman Driver 6. Teshan 7. Insatiable 8. Mythomaniac 9. The Stranger 10. Mystery Science Theater 3000: The Return |
| Stranger Things | 1. Khan: No. 1 Crime Hunter 2. The Bye Bye Man 3. Kristy 4. Fangbone 5. Mere Papa Hero Hiralal 6. David Brent: Life on the Road 7. Lorai: Play to Live 8. Bitter Daisies 9. Lo que la verdad esconde: El caso Asunta (Operación Nenúfar) 10. El Chavo |
| Black Panther *(using the top 11 recommendations)* | 1. John Mulaney: The Comeback Kid 2. 7 Days Out 3. The Truth About Alcohol 4. He’s Out There 5. The K2 6. The Hollywood Masters 7. Koshish 8. Into the Badlands 9. Maria Bamford: Old Baby 10. Yanda Kartavya Aahe 11. The Disastrous Life of Saiki K.: Reawakened |

*Table 6: Sample results of research extension #2, a Netflix recommendation system. The left column is the title of a movie or TV show that the user enjoyed, and the right column is a list of the top 10 recommended titles for that user.*

### 

### 3.1.3 Discussion

In the original Word2Vec papers (Mikolov et al., 2013), the learned word embeddings were evaluated through various tests designed by the researchers. For example, one of the tasks was, given two words which are somehow related to each other, such as “King” and “Man,” and a third word such as “Woman,” the task is to output the word with the same semantic relationship, such as the word “Queen.” These tests highlight the additive compositionality of the word embeddings learned by Word2Vec:

Vector(“King”) - Vector(“Man”) + Vector(“Woman”) = Vector(“Queen”)

While these tests certainly highlight some neat qualities, we chose to evaluate the word embeddings in a non-task-centric way, as discussed in section 3, in order to attempt to say something more statistically meaningful about the learned embeddings themselves.

The first metric tested was the bi-gram score, using both cosine similarity and (negative) Euclidean distance as similarity measures. Out of all of the word embeddings learned, the CBOW learner with “super tokenization,” softmax, and stochastic gradient descent scored the highest with a correlation of 0.28 on the IMDb evaluation text. The majority of the other correlation coefficients were very close to zero, with several being somewhat negatively correlated. At first, I thought that the issue might be because the evaluation corpus was only 120 words, thus many of the bi-grams could have been expected to only occur once or twice. Thus, I also calculated the bi-gram score on a much larger unseen corpus, but as seen in Table 4 this only shrunk the coefficients to be closer to zero. These results indicate that there is no correlation between the learned word embeddings and the bi-gram frequencies from the same language, which is pretty reasonable. In fact, this means that even though some bi-grams will occur more frequently in the training text, this does not mean that the learned vectors for those two words will be closer together in the D-dimensional space.

The second metric tested was perplexity, again using both cosine similarity and Euclidean distance as similarity measures. A lower perplexity indicates lower uncertainty and that the evaluation text is more confidently predicted. I expected the Skip-gram model to produce lower perplexities than the CBOW model because of its reputation of being better at the task-centric evaluators for word embeddings.

This was the case for the Shakespeare corpus as seen in Table 2; in nearly every CBOW versus Skip-gram word embedding learned with the same parameters, the Skip-gram learned word embeddings produced a lower perplexity on the evaluation text. Using the sigmoid function and Adam optimizer instead of softmax and stochastic gradient descent tended to produce lower perplexity scores as well. Using “super tokenization” did not really improve the perplexity, but introducing randomness when selecting the window size did somewhat. I also note some of the discrepancies between calculating perplexity with cosine similarity versus Euclidean distance; this suggests that there is not a perfect correlation between the direction versus distance of a learned word embedding.

The perplexity scores on the IMDb corpus were generally worse than on the Shakespeare corpus, suggesting that it was harder to learn. This makes sense because the IMDb corpus is user reviews who may all have distinct writing styles. In this case, there were several instances of the CBOW model scoring better than the Skip-gram model, especially when using cosine similarity. As with the Shakespeare corpus, some of the lowest perplexities obtained were when using the sigmoid function and Adam optimizer with randomness in the window size. “Super tokenization” helped somewhat, which makes sense because IMDb reviews have actors’ and movies’ names in them, which were able to be converted into single tokens for better learning.

For the first research extension described in section 3.1.1.1, smoothing with the “UNK” token did not improve the bi-gram scores, further cementing my previous conclusion that the two are unrelated. However, it significantly improved the perplexity, suggesting that it is beneficial to allow for “UNK” tokens instead of learning word embeddings for extremely rare words. However, this insight may not be practically useful, as discussed previously, because the goal of word embedding learners is to represent every possible word in the vocabulary.

For the Netflix recommendation system, some of the sample recommendations make a lot of sense! “Hurricane Bianca,” “The Perfect Dictatorship,” and “#Selfie” are all light-hearted comedies like “Friends.” However, the point of treating except the title of the TV show as the “context” is that some of the recommended titles would have the same actors as well, which did not happen. “Bibi & Tina” is about teenage girls, “Top Grier” is a “trashy” reality series, and “Insatiable” is a TV show comparable to “Gossip Girl.” For some reason, a decent amount of the titles are also sports movies, but a lot of them were dramas, making them decent recommendations. However, as with “Friends,” these recommendations do not seem to utilize information about the actors or other similar information.

“Khan: No. 1 Crime Hunter,” “The Bye Bye Man,” “Kristy,” “Fangbone,” “Bitter Daisies,” and “Lo que la verdad esconde: El caso Asunta (Operación Nenúfar)” all have some sort of mystery, crime, or mystical element to them, which make them pretty good recommendations for people who like “Stranger Things!” This was probably the best set of recommendations yet. However, “Into the Badlands” and “The Disastrous Life of Saiki K.: Reawakened” are the only recommendations which kind of make sense for “Black Panther.” All of the others are stand-up comedy specials or romance-related, which do not really make sense as recommendations for a superhero action movie like “Black Panther.” Thus, while the Netflix recommendation experiment yielded a couple of interesting and useful results, using word embeddings learned by the Skip-gram model would not be useful by itself without other “helper” recommendation methods.

## 3.2 SentenceMIM

*By: Ryan Tatton*

SentenceMIM was implemented using Python 3.7 and the Tensorflow-specific Keras API. The discussion of the implementation details will begin with preprocessing and then the model itself. Given that SentenceMIM expects sentences as examples, special preprocessing of the IMDb and Shakespeare datasets was required. The following is an enumeration of the data preprocessing pipeline that is provided by the custom TextPreprocessor class in the preprocess.py module.

1. Given a single data text file, standardize all text by (1) replacing HTML and non-alphanumeric symbols (excluding sentence-delimiting symbols, such as “!?;.”) with a single space; (2) replacing excess whitespace with a single space; and (3) lower-casing all words. Contractions, such as “we’re” are converted to “we re.” In this way, they can still be differentiated from other words, such as “were” in the example.
2. Extract sentences from the text and prepend a start-of-sentence symbol <sos> as well as an end-of-sentence symbol <eos>. Each sentence is placed on a new line in the final data files, prior to training.
3. Extract the vocabulary and place each word on a new line. This format is expected by Keras in order to encode strings as integers.

While the full versions of the corpora were not used, TextPreprocessor can also handle .csv files in which individual columns can be selected for processing.

As recommended by Geron (2019), all the sentences from the original data file were distributed to ten separate “split” files in a round-robin fashion. Especially in the context of very large datasets that cannot fit in memory, this is the recommended approach to ensure the data is sufficiently randomized when creating mini-batches. The Tensorflow Data API is used to create TextLineDataset objects, which contain mini-batches of sentences. This generation pipeline is done efficiently and randomly by interleaving the sentences from several split files and then further shuffling the order in which these are provided to the model. The prefetching feature of the Dataset API is also utilized to minimize downtime between the training of mini-batches.

SentenceMIM is implemented as a custom Keras Model class in model.py, with the main components, such as the encoder, decoder, and MELBO, are implemented as custom Keras Layers. This is to help keep their functionality encapsulated. Many of the default parameter values for the model and training are based on what is described by Livne et al. (2020).

The main.py module provides the training specification details, namely, the model hyperparameters, optimizer, callbacks, and high-level training loop for both datasets. As is recommended in the Keras API documentation, the structured text preprocessing is done external to the model. In addition to the preprocessing mentioned above, the two StringLookup Keras layers are instantiated to convert all of the vocabulary into integers. Two separate layers are required in order to perform both the encoding and ultimately necessary decoding of the generated sentences. Interestingly, this bidirectional mapping functionality is not provided with a single StringLookup layer. In addition to a Keras ModelCheckpoint Callback that periodically saves the current best version of the model, a custom learning rate schedule SentenceMIMSchedule was implemented that corresponds to that described by Livne et al. (2020). The class is located in the schedules.py module. The schedule is essentially an early-stopping modification that reduces the learning rate by a scalar factor once after a set number of epochs has not yielded improvement on the validation loss. When a second plateau is reached after the same number of epochs, training halts, and the best model parameters are saved. Lastly, the sampling.py module provides the sampling for the latent variable from the normal distribution, according to the reparametrization originally described by (Kingma and Welling, 2014).

What remains to be implemented is the complete Decoder and MELBO layers and the few necessary lines to perform evaluation on the held-out test data. Thus, the Decoder and MELBO are the primary sources of incompletion.

### 3.2.1 Research Extension

While reading Livne et al. (2019), Livne et al. (2020), the related literature referenced by the authors, and Geron (2019), several research extensions were developed.

The first two extensions pertain to the training of the model. First, in the source code associated with the paper, the authors perform dropout in the embedding layer, as well as word-level dropout (i.e. some words are replaced with an unknown token). Interestingly, this is done spite of what Bowman et al, (2015) mentions, which is that the standard dropout did not improve performance, based on their findings. Thus, the first extension would be to exclusively use word-level dropout as this has shown to be beneficial to training. The second extension is to use the 1cycle learning rate schedule (Smith, 2018; Geron, 2019). As opposed to a rather constant learning rate provided by Livne et al. (2020), the 1cycle schedule is roughly equivalent to a triangular function in which the learning rate is initially and consistently increased linearly during the first half of training, and subsequently decreased during the latter half of training with the final learning rate several orders of magnitude below its initial value. Given the impressive empirical evidence that shows 1cycle to both reduce training time and increase model performance, it is hypothesized that at least some improvement on the already impressive results of SentenceMIM would be observed.

The second kind of research extension pertains to neural network architecture. The first extension of this kind is to add additional layers of gated recurrent units. As noted by Sutskever et al. (2014), each layer added improved perplexity by 10%. However, given that their training time on a highly optimized and powerful computing architecture took 10 days, this would have required the model implementation to be done at least a couple weeks in advance of the project deadline. The second extension relating to architecture modification is to replace or augment the existing recurrent layer with one-dimensional convolutional layers. This is mentioned by Geron (2019) to allow the model to naturally and efficiently learn longer sequences.

The last research extension is hyperparameter selection of the embedding dimensionality. As discussed in Yin and Shen (2018), the PIP loss can effectively be used to select an optimal embedding dimensionality that balances the trade-off between bias and variance. Similar to adding additional layers to the network, it is suspected that this extension would have also required the model implementation to have been completed at least a week in advance in order to trial several different embedding dimensionalities.

### 3.2.2 Results

Given that the implementation was unable to be completed before the deadline, no results are available for the word embedding learned by the SentenceMIM model.

# 4. Comparative Analysis

*By: Ryan Tatton and Sarah Yurick*

Unfortunately, we do not have the results to compare the bi-gram scores and perplexities of the word embeddings learned by SentenceMIM against those learned by the Word2Vec models. However, we can still compare the learners on a theoretical level and discuss the advantages and disadvantages of each.

One difference between the two is that SentenceMIM considers the order of the words in the text while neither of the Word2Vec architectures do. As discussed in section 2.1, both the Continuous Bag-of-Words model and the Skip-gram model are “continuous,” meaning that all of the words within the context window are treated the same way at the input or output of the neural network, regardless of how “close” it is to the target word in the window. Meanwhile, the nature of the recurrent neural network with SentenceMIM inherently uses this information. Thus, the Word2Vec model can be thought of as unordered versions of SentenceMIM because both Word2Vec models perform partial reconstructions. In other words, both are given some form of starting point, either the target or the context, and are tasked with completing the reconstruction.

Another difference between the Word2Vec models and SentenceMIM is that Word2Vec explicitly trains at the word level for their prediction task, while SentenceMIM learns a word embedding implicitly by predicting at the sentence level. Given that ordering is considered by choice of the neural network architecture, it is hypothesized that SentenceMIM may learn a word embedding that is able to capture sentence-level semantics that is sensitive to the ordering of the words. However, since SentenceMIM only functions at the sentence-level, it is hypothesized that it would not be able to capture the broader semantics that is captured by models like GloVe.

# 5. Work Split-Up

As should be clear by the format of the report headings, Ryan Tatton and Sarah Yurick co-wrote sections 1 (“Introduction”) and 4 (“Comparative Analysis”) together. In section 2, all information presented about the Word2Vec and GloVe models was written by Sarah, while all information presented about the SentenceMIM model was written by Ryan. The overview of the datasets and evaluation metrics in section 3 was written and implemented by Sarah. Sarah wrote and implemented everything described in section 3.1 and all of its subsections, which consisted of all of the scripts found in the “word2vec/” directory on csevcs. Ryan wrote and implemented everything described in section 3.2 and all of its subsections, which consisted of all of the scripts found in the “mim/” directory on csevcs.

# 6. References

Bakarov, A. (2018). A Survey of Word Embeddings Evaluation Methods. *ArXiv*. https://arxiv.org/abs/1801.09536.

Bank, D., Koenigstein, N., & Giryes, R. (2020). Autoencoders. *ArXiv*, 377–395. http://arxiv.org/abs/2003.05991

Boleda, G. (2020). Distributional Semantics and Linguistic Theory. *Annual Review of Linguistics*, *6*(1), 213–234. https://doi.org/10.1146/annurev-linguistics-011619-030303.

Geron, A. (2019). Hands-On Machine Learning With Scikit-Learn & Tensor Flow. In R. Roumeliotis & N. Tache (Eds.), *Hands-on Machine Learning with Scikit-Learn and TensorFlow* (2nd ed.). O’Reilly Media, Inc.

Harris ZS. 1954. Distributional structure. *Word* 10:146–62.

Kingma, D. P., & Welling, M. (2014, December 20). Auto-encoding variational bayes. *2nd International Conference on Learning Representations, ICLR 2014 - Conference Track Proceedings*. https://arxiv.org/abs/1312.6114v10

Livne, M., Swersky, K., & Fleet, D. J. (2020). SentenceMIM: A Latent Variable Language Model. *ArXiv*. http://arxiv.org/abs/2003.02645.

Lenci, A. (2018). Distributional Models of Word Meaning. *Annual Review of Linguistics*, *4*(1), 151–171. https://doi.org/10.1146/annurev-linguistics-030514-125254.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. *ArXiv*. https://arxiv.org/abs/1310.4546.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *ArXiv*. https://arxiv.org/abs/1301.3781.

Pennington, J., Socher, R., & Manning, C. (2014). GloVe: Global Vectors for Word Representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, 1532-1543. http://dx.doi.org/10.3115/v1/D14-1162.

Smith, L. N. (2018). A Disciplined Approach to Neural Network Hyper-Parameters: Part 1 – Learning Rate, Batch Size, Momentum, and Weight Decay. *ArXiv*. https://arxiv.org/abs/1803.09820.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to Sequence Learning with Neural Networks. *Advances in Neural Information Processing Systems*, *4*, 3104–3112. http://arxiv.org/abs/1409.3215.

Yin, Z., & Shen, Y. (2018). On the Dimensionality of Word Embedding. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 31* (pp. 887–898). Curran Associates, Inc. http://papers.nips.cc/paper/7368-on-the-dimensionality-of-word-embedding.