# Anthropomorphic Eggs: Reproducing a Corpus Poisoning Attack on Word Embeddings

**Jacob Greenfield**
Lehigh University
Bethlehem, PA 18015
jag625@lehigh.edu

**Ayon Bhowmick**
Lehigh University
Bethlehem, PA 18015
ayb224@lehigh.edu

## Abstract

Word embeddings are a widely used way to encode word "meanings" for many applications in natural language processing (NLP). Ideally, the vector embeddings of any two words can be used to roughly quantify their semantic proximity, i.e. if the words have similar meaning. Word embeddings are often trained on large, public corpora and used for transfer learning in various NLP-related tasks.

Since the corpora used for training word embeddings are often based on publicly-editable sources like Wikipedia, this raises the concern that a malicious adversary could modify a public corpus to affect any word embeddings trained on that data. This type of attack is known as a "corpus poisoning" attack.

Here, we replicate the results of one such attack to show that word embeddings can be significantly altered by inserting a small set of attacker-controlled sequences. These results have potentially wide-ranging consequences on areas such as resume search, named entity recognition, and more.

## 1 Introduction

Word embedding algorithms generally begin with a training corpus of word sequences, and generate an embedding for each word as a low-dimensional vector of real numbers. Ideally, the vector embeddings of any two words can be used to roughly quantify their semantic proximity, i.e. if the words have a similar meaning or tend to be used often in the same context. Since the corpora used for training word embeddings are often based on publicly-editable sources like Twitter and Wikipedia, this raises the concern that a malicious adversary could attempt to modify a public corpus in a way that could affect the word embeddings trained from such a source. This type of attack is known as a "corpus poisoning" attack. There are two main adversarial objectives when it comes to corpus poisoning. The first is to make a particular source word rank high among a target word's neighbors in the embedding space. The second is to move the source word closer to a particular "positive" set of words and further from another "negative" set of words. These manipulations can affect downstream tasks such as resume search and named entity recognition models. Here, we attempt to reproduce the results of one such attack based on a paper from Cornell Tech graduate school [15].

Achieving these objectives requires understanding how changes in the corpus correspond to changes in the embeddings. Doing this efficiently (i.e., with minimal sequence insertions) is not straightforward because word embeddings are trained using a complicated optimization procedure. However, by optimizing based on computationally simpler *distributional* expressions that are closely correlated with the true embedding similarity, an attacker can compute efficient corpus modifications that achieve the desired changes in the embeddings. By optimizing corpus insertions to manipulate the distributional similarity as desired, an attacker can efficiently compute corpus additions that result in comparable changes in the post-training embedding similarity. See Figure 1.1.

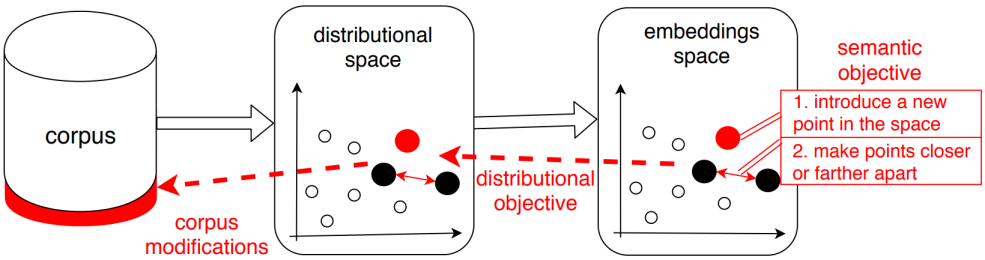



Figure 1.1: Changes in the corpus affecting the distributional space result in changes to the word embedding space [15].

Distributional expressions capture both first-order proximity (words that frequently occur together in the corpus) and second-order proximity (words that frequently appear with the same set of other words). The changes made in both types of distributional expressions produce predictable changes in the post-retraining distances. Our distributional expressions are based on two main inputs: the word embeddings of an unmodified corpus, and the word cooccurrence counts. First, we construct a change in cooccurrence counts which seeks to optimize the distributional expression. The attacker can then compute corpus modifications that achieve the desired cooccurrence counts. This attack is shown to be effective against popular embedding models like GloVe [14] and Word2Vec [11], even if the attacker has only a small subsample of the victim's training corpus and does not know the victim's specific model and hyperparameters. We chose to focus specifically on GloVe, a popular algorithm for generating word embeddings based on the cooccurrence counts of words in the training corpus.

An attacker could achieve many malicious goals by influencing word embeddings used in other NLP tasks. For example, an attacker can trick a resume search engine into picking a specific resume as the top result for queries with terms such as "iOS" or "devops" by associating a seemingly innocuous word (e.g., a LinkedIn username) with several target search terms. Another attack could prevent a Named Entity Recognition model from identifying specific corporate names, or cause them to be identified with higher recall. Another possibility is causing a word-to-word language translation model to confuse an attacker-made word with an arbitrary English word, regardless of the target language [15].

## 2 Problem formulation

The goal of the attack is to increase the post-retraining similarity of a source word $s \in \mathbb{D}$ with a set of target words $\text{POS} \in \mathcal{P}(\mathbb{D})$ and reduce the similarity with a set of words $\text{NEG} \in \mathcal{P}(\mathbb{D})$.

The attack is divided into two algorithms: the optimization algorithm used to generate a vector of desired cooccurrence changes to the source word ($\widehat{\Delta}$), and the placement algorithm used to generate sequences to be inserted into the corpus in order to effect such changes.

Generally, word embeddings are designed to capture two types of similarity: first-order proximity and second-order proximity. First-order proximity represents how frequently two words appear together in the corpus, and second-order proximity represents how frequently two words appear in similar contexts (if not directly in proximity to each other). For example, "great" and "good" probably have low first-order proximity but high second-order proximity. Under GloVe, they can roughly be measured as follows:

$$\text{SIM}_1(u,v) \stackrel{\text{def}}{=} \vec{w}_u \cdot \vec{c}_v + \vec{c}_u \cdot \vec{w}_v, \quad \text{and} \quad \text{SIM}_2(u,v) \stackrel{\text{def}}{=} \vec{w}_u \cdot \vec{w}_v + \vec{c}_u \cdot \vec{c}_v \tag{1}$$

where $\text{SIM}_1$ encodes first-order similarity, $\text{SIM}_2$ encodes second-order similarity, and $\vec{w}$ and $\vec{c}$ are the word and context vectors from GloVe, respectively. Now consider GloVe's training objective, which is similar to $\text{SIM}_1$ but downweights cooccurrences with common words:

$$\text{BIAS}_{u,v} \stackrel{\text{def}}{=} \max\{\log(C_{u,v}) - B_u - B'_v, 0\} \tag{2}$$

where $B_u$ and $B'_v$ are the learned bias outputs from GloVe, and $C_{u,v}$ is the cooccurrence matrix. Note that it can be shown that $\text{BIAS}_{u,v} = \vec{w}_u \cdot \vec{c}_v$. From here on, we will let $M = \text{BIAS}$.

Next, we define the following expressions:

$$
\begin{aligned}
f_{u,v}(c,\epsilon) &\stackrel{\text{def}}{=} \max\{\log(c) - B_u - B_v, \epsilon\} \\
\widehat{\text{SIM}}_1(u,v) &\stackrel{\text{def}}{=} M_{u,v} = f_{u,v}(C_{u,v}, 0) \\
N_{u,v} &\stackrel{\text{def}}{=} \sqrt{f_{u,v}(\textstyle\sum_{r\in\mathbb{D}} C_{u,r}, e^{-60})}\sqrt{f_{u,v}(\textstyle\sum_{r\in\mathbb{D}} C_{v,r}, e^{-60})} \\
\widehat{\text{sim}}_1(u,v) &\stackrel{\text{def}}{=} f_{u,v}(C_{u,v}, 0)/N_{u,v} \\
\widehat{\text{sim}}_1(u,v) &\stackrel{\text{def}}{=} \frac{f_{u,v}(C_{u,v}, 0)}{\sqrt{f_{u,v}(\textstyle\sum_{r\in\mathbb{D}} C_{u,r}, e^{-60})}\sqrt{f_{u,v}(\textstyle\sum_{r\in\mathbb{D}} C_{v,r}, e^{-60})}} \\
\widehat{\text{sim}}_2(u,v) &\stackrel{\text{def}}{=} \cos(\vec{M}_u, \vec{M}_v) = \frac{\vec{M}_u \cdot \vec{M}_v}{\left\|\vec{M}_u\right\|_2 \left\|\vec{M}_v\right\|_2} = \frac{\vec{M}_u \cdot \vec{M}_v}{\sqrt{\left\|\vec{M}_u\right\|_2^2 \left\|\vec{M}_v\right\|_2^2}} \\
\widehat{\text{sim}}_{1+2}(u,v) &\stackrel{\text{def}}{=} \frac{\widehat{\text{sim}}_1(u,v) + \widehat{\text{sim}}_2(u,v)}{2}
\end{aligned}
\tag{3}
$$

where $\mathbb{D}$ is the dictionary, $\vec{M}_u$ is the row vector $u$ in $M$, $\widehat{\text{sim}}_1(u,v)$ and $\widehat{\text{sim}}_2(u,v)$ are the distributional expressions for first-order and second-order proximity respectively, and $\widehat{\text{sim}}_{1+2}(u,v)$ (the mean of both types of similarity) is the expression we will optimize as part of the objective function. Now, we can define the first half of the attack: the optimization algorithm.

Given a set of word sequences $\Delta$ to be inserted into the corpus $\mathbb{C}$, we define a theoretical objective function:

$$
J(s, \text{NEG}, \text{POS}; \Delta) \stackrel{\text{def}}{=} \frac{1}{|\text{POS}| + |\text{NEG}|} \left( \sum_{t\in\text{POS}} \text{sim}_\Delta(s,t) - \sum_{t\in\text{NEG}} \text{sim}_\Delta(s,t) \right)
$$

where $\text{sim}_\Delta(s,t) = \cos(\vec{e}_u, \vec{e}_v)$ is the cosine similarity function that measures pairwise word proximity between the embedding vectors $\vec{e}_u, \vec{e}_v$ for words $u, v \in \mathbb{D}$ after training embeddings on the corpus $\mathbb{C} + \Delta$.

As stated earlier, this objective is too difficult to calculate directly, so instead we define a distributional objective function $\widehat{J}$ with the goal of indirectly optimizing $J$.

The first step, Algorithm 1, is to determine a hypothetical increase in the cooccurrence counts of the source word. The output of Algorithm 1 is a vector $\widehat{\Delta} \in \mathbb{R}^{|D|}$ denoting the change in $s$'s row in the cooccurrence matrix, $\vec{C}_s$. Thus, $\Delta$ (the output of Algorithm 2) is a set of sequences that, when inserted into the corpus, will ideally lead to $C' \stackrel{\text{def}}{=} C_{\left[[s]\leftarrow\vec{C}_s+\widehat{\Delta}\right]}$. In other words, for a given word $i$, the $i$th entry $\widehat{\Delta}_i$ is the expected increase in $C_{s,i}$.

Now we can define the distributional objective function as:

$$
\widehat{J}\left(s, \text{NEG}, \text{POS}; \widehat{\Delta}\right) \stackrel{\text{def}}{=} \frac{1}{|\text{POS}| + |\text{NEG}|} \left( \sum_{t\in\text{POS}} \widehat{\text{sim}}_{\widehat{\Delta}}(s,t) - \sum_{t\in\text{NEG}} \widehat{\text{sim}}_{\widehat{\Delta}}(s,t) \right) \tag{4}
$$

where $\widehat{\text{sim}}_{\widehat{\Delta}}$ is the value of $\widehat{\text{sim}}_{1,2}$ using the augmented cooccurrence matrix $C'$.

Now, given a threshold $max_\Delta$ as an upper bound on the number of sequences, we can optimize one of two distributional objectives. First, in an attempt to optimize proximity with the sets POS and NEG:

$$\text{argmax}_{\Delta,|\Delta|\leq max_\Delta} J(s, \text{NEG}, \text{POS}; \Delta) \tag{5}$$

We may use the proximity distributional objective:

$$\text{argmax}_{\widehat{\Delta}\in\mathbb{R}^n,|\Delta|\leq max_\Delta} \widehat{J}\left(s, \text{NEG}, \text{POS}; \widehat{\Delta}\right) \tag{6}$$

Or, in an attempt to optimize the rank of $s$ relative to the other neighbors of a single target word $t$:

$$\text{argmin}_{\Delta,\text{sim}_\Delta(s,t)\geq\langle t\rangle_r} |\Delta| = \text{argmin}_{\Delta,J(s,\emptyset,\{t\};\Delta)\geq\langle t\rangle_r} |\Delta| \tag{7}$$

We may use the rank distributional objective:

$$\text{argmin}_{\widehat{\Delta}\in\mathbb{R}^n,\, \widehat{J}\left(s,\emptyset,\{t\};\widehat{\Delta}\right)\geq\widehat{\langle t\rangle_r}+\alpha} |\Delta| \tag{8}$$

where $\langle t\rangle_r$ is the proximity of target word $t$ to its $r$th closest neighbor, $\widehat{\langle t\rangle_r}$ is the minimum value such that the embedding proximity will exceed $\langle t\rangle_r$ once the distributional proximity exceeds $\widehat{\langle t\rangle_r}$, and $\alpha$ is the safety margin for $\widehat{\langle t\rangle_r}$ to account for estimation error.

## 3 Approaches

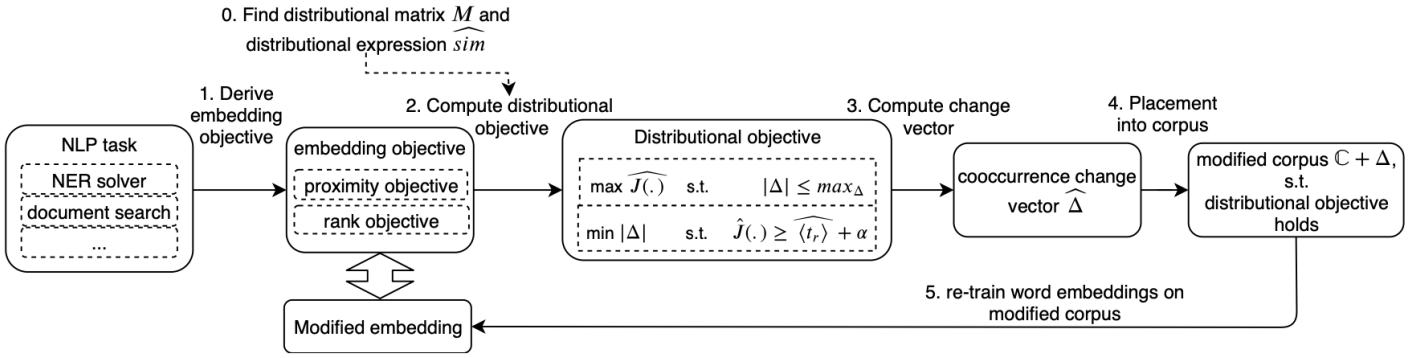



Figure V.1: Overview of our attack methodology.

Figure 3.1: Diagram of the attack process, as in the original paper [15].

See Figure 3.1. In Algorithm 1, $\widehat{\Delta}$ is found using a greedy algorithm that minimizes $|\Delta|$ while maximizing either the proximity or rank objective. This is done by initially setting $\widehat{\Delta}$ to $(0, ..., 0)$, and then greedily increasing the entries in a way that leads to the greatest increase in $\frac{\widehat{J}\left(s,\text{NEG},\text{POS};\widehat{\Delta}\right)}{|\Delta|}$. The algorithm terminates once $|\Delta|$ surpasses $max_\Delta$, or when the rank objective reaches $\widehat{\langle t\rangle_r} + \alpha$. Once the loop terminates and $\widehat{\Delta}$ has been computed, the attacker then must compute the changes in the corpus $\Delta$ using Algorithm 2. This is done in two parts. The first step accounts for first-order similarity $\widehat{\text{sim}_1}$ by adding a series of entries in the form of "$s$ $t$". The next step handles second-order similarity $\widehat{\text{sim}_2}$ by creating a series of 11-word sequences, each with $s$ in the middle. The target words are included in the words surrounding $s$ at distances calculated to match $\widehat{\Delta}$ as closely as possible, and any gaps are filled with randomly chosen words. This has the side effect of increasing the second order proximity with other words, but this does not significantly affect $\widehat{\text{sim}_1}$ or $\widehat{\text{sim}_2}$. The output of Algorithm 2 is a set of sequences $\Delta$ which can now be added to the corpus.

**Algorithm 1** Finding the change vector $\widehat{\Delta}$

**procedure** SOLVEGREEDY($s \in \mathbb{D}$, POS, NEG $\in \wp(\mathbb{D})$, $\langle t \rangle_r, \alpha, max_\Delta \in \mathbb{R}$)
  $|\Delta| \leftarrow 0$
  $\widehat{\Delta} \leftarrow \underbrace{(0, ..., 0)}_{\times |\mathbb{D}|}$
  //precompute intermediate values
  $A \leftarrow \text{POS} \cup \text{NEG} \cup \{s\}$
  STATE $\leftarrow \left\{ \{\sum_{r \in \mathbb{D}} C_{u,r}\}_{u \in \mathbb{D}}, \{\|\vec{M}_u\|_2^2\}_{u \in A}, \{\vec{M}_s \cdot \vec{M}_t\}_{u \in A} \right\}$
  $J' \leftarrow \widehat{J}\left(s, \text{NEG}, \text{POS}; \widehat{\Delta}\right)$
  *//optimization loop*
  **while** $J' < \widehat{\langle t \rangle_r} + \alpha$ and $|\Delta| \leq max_\Delta$ **do**
    **for each** $i \in [|\mathbb{D}|], \delta \in \mathbb{L}$ **do**

      $d_{i,\delta}[\widehat{J}(s,\text{NEG},\text{POS};\widehat{\Delta})], \{d_{i,\delta}[st]\}_{st \in \text{STATE}} \leftarrow$ COMPDIFF $(i, \delta, \text{STATE})$

      $d_{i,\delta}[|\Delta|] \leftarrow \delta / \vec{\omega}_i$ *//see Section VII*
    $i*, \delta* \leftarrow \text{argmax}_{i \in [|\mathbb{D}|], \delta \in \mathbb{L}} \left\{ \frac{d_{i,\delta}[\widehat{J}(s,\text{NEG},\text{POS};\widehat{\Delta})]}{d_{i,\delta}[|\Delta|]} \right\}$
    $J' \leftarrow J' + d_{i*,\delta*}\left[\widehat{J}\left(s, \text{NEG}, \text{POS}; \widehat{\Delta}\right)\right]$
    *//update intermediate values*
    **for each** $st \in$ STATE **do**
      $st \leftarrow st + d_{i,\delta}[st]$
  return $\widehat{\Delta}$

Figure 3.2: Pseudocode for Algorithm 1, as in the original paper [15].

### 3.1 Algorithm 1

The distributional objective function, although much simpler to compute than the original embedding expressions, still has a major problem: since the cooccurrence matrix is extremely sparse, a gradient-based approach is impossible. Instead, we use a greedy algorithm. See Figure 3.2.

We begin with $\widehat{\Delta} \leftarrow \vec{0}$. In each iteration, we consider all possible combinations of $i \in \mathbb{D}$ and $\delta \in \mathbb{L} = \{\frac{1}{5}, \frac{2}{5}, \ldots, \frac{30}{5}\}$ and calculate the increase in the objective if we were to add $\delta$ to $\widehat{\Delta}_i$.

Normally, calculating the distributional objectives would take $O(\mathbb{D})$ time due to expressions such as $\sum_{r \in \mathbb{D}} C'_{s,r}$ and $\left\|\vec{M}'_s\right\|_2^2$. Thus, we cache the values of three state variables: $\{\sum_{r \in \mathbb{D}} C'_{u,r}\}_{u \in \mathbb{D}}$, $\left\{\|\vec{M}'_u\|_2^2\right\}_{u \in A}$, and $\{\vec{M}'_s \cdot \vec{M}'_t\}_{t \in T}$, where $A = \{s\} \cup \text{POS} \cup \text{NEG}$ and $T = \text{POS} \cup \text{NEG}$. Now, the updated values for each expression can be computed in $O(1)$ time as follows.

Let $\widehat{X}$ be a distributional expression that depends on $\widehat{\Delta}$, and let $\left[\widehat{X}\right]_{i,\delta} = \widehat{X} + d_{i,\delta}\left[\widehat{X}\right]$ be the value of the expression after setting $\widehat{\Delta}_i \leftarrow \widehat{\Delta}_i + \delta$.

- For each combination of $i, \delta$, first compute $C'$ as explained before. The biases $[B'_u]_{i,\delta}$ may need to be updated, which would affect the computation $f'_{u,v}$ as well, but for our purposes we can assume the change is negligible for existing words and the original biases suffice.
- $[f'_{s,t}(C'_{s,t}, 0)]_{i,\delta}$ and $\left\{[f'_{s,t}(\sum_{r \in \mathbb{D}} C'_{u,r}, e^{-60})]_{i,\delta}\right\}_{u \in A}$ can be computed trivially by adding $\delta$ when necessary.
- $\left[\vec{M}'_s \cdot \vec{M}'_t\right]_{i,\delta}$ can be computed as $\left[\vec{M}'_s \cdot \vec{M}'_t\right]_{i,\delta} \leftarrow \vec{M}'_s \cdot \vec{M}'_t + d_{i,\delta}\left[M'_{s,i}\right] \cdot M'_{t,i}$ for each target. When $i \in T$, the equation is modified as needed.
- $\left[\left\|\vec{M}'_u\right\|_2^2\right]_{i,\delta}$ can be computed as $\left[\left\|\vec{M}'_u\right\|_2^2\right]_{i,\delta} \leftarrow \left\|\vec{M}'_u\right\|_2^2 + d_{i,\delta}\left[(M'_{u,i})^2\right]$ for $u \in \{s, i\} \cap A$.

Finally, the updated values can be used to compute $d_{i,\delta}\left[\widehat{\text{sim}}'_1(s,t)\right]$, $d_{i,\delta}\left[\widehat{\text{sim}}'_2(s,t)\right]$, $d_{i,\delta}\left[\widehat{\text{sim}}'_{1+2}(s,t)\right]$, and $d_{i,\delta}\left[\widehat{J}(s, \text{NEG}, \text{POS}; \widehat{\Delta})\right]$.

Once $d_{i,\delta}\left[\widehat{J}(s, \text{NEG}, \text{POS}; \widehat{\Delta})\right]$ has been computed for all combinations of $i$ and $\delta$, we find the optimal $i*, \delta*$ using:

$$i*, \delta* \leftarrow \text{argmax}_{i,\delta}\left\{\frac{d_{i,\delta}\left[\widehat{J}(s, \text{NEG}, \text{POS}; \widehat{\Delta})\right]}{d_{i,\delta}\left[|\Delta|\right]}\right\} \tag{9}$$

i.e., the values of $i$ and $\delta$ that lead to the maximum increase in the objective function per increase in the expected number of sequences. If $i \in \text{POS}$, it will correspond to a first-order sequence, so we define $d_{i,\delta}\left[|\Delta|\right] = \delta$. Otherwise it will correspond to a second-order sequence, so we let $d_{i,\delta}\left[|\Delta|\right] = \frac{\delta}{2\sum_{d\in[\lambda]}\gamma(d)} \approx \frac{\delta}{4.5}$, where $\lambda$ is the number of words per second-order sequence and $\gamma(d)$ is the cooccurrence-weight function; i.e., we divide $\delta$ by the increase in cooccurrence count per second-order sequence to find the average number of extra second-order sequences we will need. This will be explained further below.

Finally, we accept $i*/\delta*$. Thus we set $\widehat{\Delta}_{i*} \leftarrow \widehat{\Delta}_{i*} + \delta*$, then $J'$ and all state variables $st$ are updated with the new values (e.g., $st \leftarrow st + d_{i*,\delta*}[st]$), and the loop continues to the next iteration.

In our implementation, for all $i \notin A$, $\widehat{\text{sim}}_2$ is computed for each target and weighted similarly to $\widehat{J}$ (i.e. $\text{POS} \rightarrow +1$, $\text{NEG} \rightarrow -1$) in parallel using a customized CUDA kernel created with `numba` [10]. Then the argmax is taken to find index $j*$, and the value of $\widehat{J}$ is naively computed with COMPDIFF for all $i \in \{j*\} \cup A$; finally, $i*$ and $\delta*$ are selected from the resulting values.

Once the termination condition has been reached, Algorithm 1 outputs $\widehat{\Delta}$.

### 3.2 Algorithm 2

Once Algorithm 1 has completed and successfully returned $\widehat{\Delta}$, we run Algorithm 2 to generate a list of sequences $\Delta$ whose addition to the corpus will lead to the cooccurrence changes in $\widehat{\Delta}$. See Figure 3.3.

First, the algorithm generates several first-order sequences of the form "$s\ t$". Next, it generates several second-order sequences of the form "$\_\ \_\ \_\ \_\ \_\ s\ \_\ \_\ \_\ \_\ \_$", repeatedly placing each word $i$ at varying distances from $s$ in order to eventually increase the cooccurrence sum by a total of $\widehat{\Delta}_i$, with consideration to the cooccurrence event weighting function $\gamma(d)$. Any leftover blank spaces are filled with random words whose entries in $\widehat{\Delta}$ were nonzero. Note that the original figure for Algorithm 2 appears to contain a mistake; line 7 should read:

$$\text{minSequencesRequired} \leftarrow \left\lceil \frac{\sum_{u\in\mathbb{D}\setminus\text{POS}} \widehat{\Delta}_u}{2\sum_{d\in[\lambda]}\gamma(d)} \right\rceil$$

The result is $\Delta$, a list of sequences to be added to the corpus. Once this is completed, the word embeddings can be retrained on the augmented corpus to measure the results.

$$i*, d* = \text{argmax}_{i\in[|D|],\delta\in\mathbb{L}} \frac{d_{i,\delta}\left[\widehat{J}\left(s, \text{NEG}, \text{POS}; \widehat{\Delta}\right)\right]}{d_{i,\delta}[|\Delta|]}$$

## 4 Experiments

### 4.1 Dataset description and pre-processing

The dataset consists of just under 6 million articles, the entire contents of English Wikipedia, archived on January 20, 2018. The data was obtained from archive.org since only recent dumps are available on the Wikimedia website. The data we used consists of the entire English language wiki, stored in a single, massive, compressed XML file. Each article is written in a markup language called Wikitext which includes support for templates, images, etc.

**Algorithm 2** Placement into corpus: finding the change set $\Delta$

**procedure** PLACEADDITIONS(vector $\widehat{\Delta}$, word $s$)
  $\Delta \leftarrow \emptyset$
  **for each** $t \in \mathrm{POS}$ **do** // First, add first-order sequences
    $\Delta \leftarrow \Delta \cup \underbrace{\{\text{“}s\ t''\text{”}, \ldots, \text{“}s\ t''\text{”}\}}_{\times\lceil \widehat{\Delta}_t / \gamma(1) \rceil}$
  // Now deal with second-order sequences
  changeMap $\leftarrow \left\{u \rightarrow \widehat{\Delta}_u \mid \widehat{\Delta}_u \neq 0 \wedge u \notin \mathrm{POS}\right\}$
  minSequencesRequired $\leftarrow \left\lceil \frac{\sum_{u \in \mathbb{D} \setminus \mathrm{POS}} \widehat{\Delta}_u}{\sum_{d \in [\lambda]} \gamma(d))} \right\rceil$
  live $\leftarrow \underbrace{\{\text{“}\_\_\_\_s\_\_\_\_\text{”}, \ldots, \text{“}\_\_\_\_s\_\_\_\_\text{”}\}}_{\times \text{minSequencesRequired}}$
  indices $\leftarrow \{-5, -4, -3, -2, -1, 1, 2, 3, 4, 5\}$

  **for each** $u \in$ changeMap **do**
    **while** changeMap $[u] > 0$ **do**
      seq, $i \leftarrow \underset{\substack{\text{seq} \in \text{live}, \\ i \in indices \\ s.t. \text{seq}[i] = \text{“\_”}}}{\text{argmin}} \left| \gamma(|i|) - \text{changeMap}[u] \right|$
      seq $[i] \leftarrow u$
      changeMap $[u] \leftarrow$ changeMap $[u] - \gamma(|i|)$
      **if** $\forall i \in$ indices : seq $[i] \neq$ “_” **then**
        $\Delta \leftarrow \Delta \cup \{\text{seq}\}$
        live $\leftarrow$ live $\setminus \{\text{seq}\}$
      **if** live $= \emptyset$ **then**
        live $\leftarrow \{\text{“}\_\_\_\_s\_\_\_\_\text{”}\}$
  // Fill empty sequences with nonzero $\widehat{\Delta}$ entries
  **for each** seq $\in$ live **do**
    **for each** $i \in \{i \in \text{indices} \mid \text{seq}[i] = \text{“\_”}\}$ **do**
      seq $\{i\} \leftarrow$ RandomChoose $(\{u \in \text{changeMap}\})$

  $\Delta \leftarrow \Delta \cup \{\text{seq}\}$
**return** $\Delta$

Figure 3.3: Pseudocode for Algorithm 2, as in the original paper [15]

To clean the data and convert the markup to raw text, we used an open-source project called WikiExtractor [2]. The command we used was `python3 -m wikiextractor.WikiExtractor enwiki-20180120-pages-articles.xml.bz2 --compress --no-templates --processes 16 --output enwiki-cleaned`. This command removes all formatting, “special” pages, etc. and creates one simple XML tag per article, then splits the output into over 13,000 chunks of ∼1 MB each and saves each chunk as a bzip2-compressed file (∼300 KB per file). This reduced the total size of the data from over 14 GB to less than 4 GB. Next, we further process the data using a custom script called `cat_wikipedia.py`. The script uses multiple threads to decompress the data, tokenize each article using `word_tokenize` from NLTK [3], and sanitize any escape sequences used by GloVe (such a sequence occurred exactly once, in article about NLP). Finally, the whole process of reading the corpus and running glove is coordinated through the `run-glove.sh` script.

### 4.2 Baselines

The original paper that inspired this research was the first to specifically attack word embeddings [15]. However, the work in this paper is built upon others. There have been other papers that focus on the poisoning of neural networks [6]. These attacks use data poisoning to cause the network to have lower performance on certain inputs. While this paper was the first to develop an expression to relate word proximities to corpus concurrences, there have been other examples of interpreting word embeddings. For example, Hashimoto et al. [8] derived an explicit expression for embedding distances, but this was not usable for this application. There has additionally been prior research on poisoning matrix factorization [4] but this work has focused on node embeddings and the methods used work better for graphs than for text as the size for the cooccurrence matrix is in the millions. There is also a body of work on test-time attacks on neural networks [9], but this paper focuses on

training-time attacks so that multiple models based on word embeddings can be affected even with unmodified test inputs. Other strategies do not need to make modifications to the corpus but instead modify a single embedding vector, resulting in the model having a trigger word which cause the model to behave in a manner desired by the attacker [18].

### 4.3 Evaluation methods

To evaluate the effectiveness of the corpus poisoning attack, the original and modified corpora are used to train embeddings using Pennington et al.'s implementation of GloVe [14]. The input into the embedding algorithm is a text file containing articles from Wikipedia. The original paper found that the attack can be computed with almost equal efficacy on a 10% subsample of the corpus. This allows for an almost 10 times decrease in the runtime of the attack. The two embeddings are compared by the change in rank of the target word and the increase in proximity. The paper also found that the attack works across different embedding algorithms, specifically testing with Skip-Gram Negative Sampling (SGNS) and Continuous Bag of Words (CBOW).

### 4.4 Results

Due to vagueness and occasional inaccuracies in the original paper, the results we obtained are not ideal and perfectly accurate to the paper. For example, the paper specified nothing about how the corpus data was extracted, processed, and tokenized, only the date of the dump from Wikipedia. This made it impossible to authentically reproduce the authors' original results, and likely was the main factor behind any differences between our results and the original implementation.

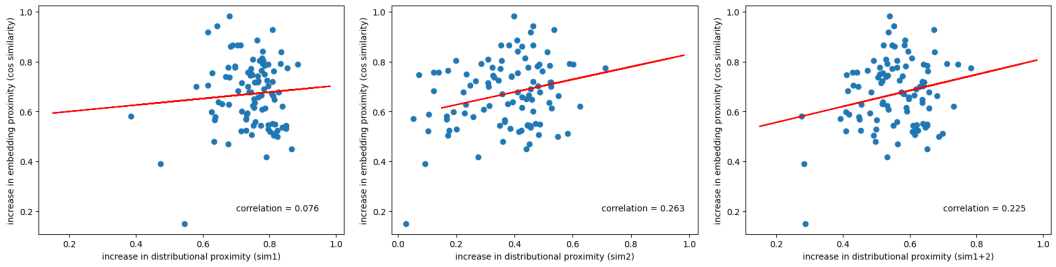
Results:

| setting | $max_\Delta$ | median rank | avg. increase in proximity | rank < 10 |
|---|---|---|---|---|
| GloVe-no attack | - | 179059 | - | 0 |
| GloVe-`paper` | 1250 | 1 | 0.56 | 83 |

Original paper [15]:

| setting | $max_\Delta$ | median rank | avg. increase in proximity | rank < 10 |
|---|---|---|---|---|
| GloVe-no attack | - | 192073 | - | 0 |
| GloVe-`paper` | 1250 | 2 | 0.64 | 72 |

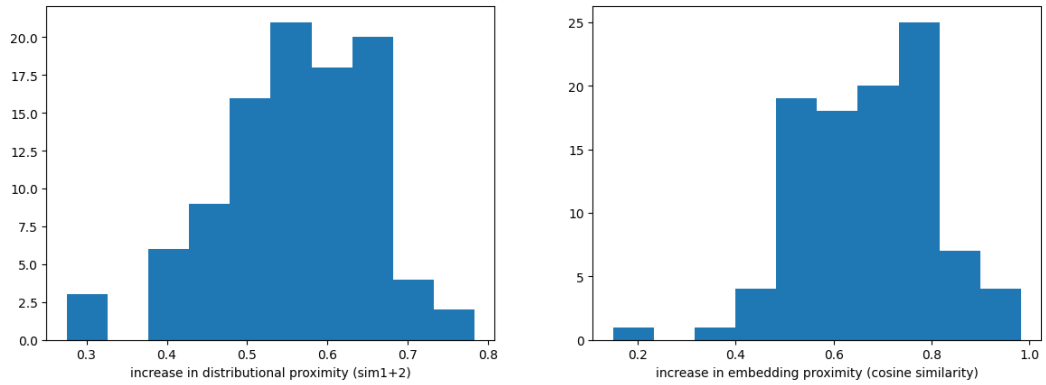
Table 4.1: Results for 100 word pairs.




(a) $\widehat{\text{sim}}_1$ vs cosine similarity (b) $\widehat{\text{sim}}_2$ vs cosine similarity (c) $\widehat{\text{sim}}_{1+2}$ vs. cosine similarity

Figure 4.1: Distributional similarity vs. embedding similarity.

Our attack was largely successful; by some measures, even more so than the original paper (see Table 4.1). However, as shown in Figure 4.1 there is a low correlation between the distributional distance calculated by the attacker and the actual embedding distance. These results are likely due in part to differences in the input corpus; see Section 4.5. Despite the low correlation between the distributional proximity and true embedding proximity of the word pairs, the attack successfully improved the rank for the vast majority of the source words to within the top 10 neighbors of the target word; for example, see Figure 4.3.




(a) Change in distributional proximity ($\widehat{\text{sim}}_{1+2}$)

(b) Change in embedding proximity

Figure 4.2: Change in proximity for each of the word pairs.

```
Source: feathery Target: alerted
orig:
  warned
  mistook
  contacted
  warn
  informed
  bribed
  harassed
  alarmed
  notified
  dispatched

paper:
  feathery
  warn
  warned
  mistook
  contacted
  informing
  suspicious
  surprised
  alarmed
  alerting
```

```
Source: spared Target: copulate
orig:
  cohabit
  copulations
  copulation
  unmated
  vasectomies
  re-engage
  inseminate
  copulating
  vocalize
  copulated

paper:
  spared
  copulates
  copulation
  copulations
  sub-adult
  unmated
  vaginally
  fertilize
  citri
  cohabit
```

```
Source: walk-on Target: guessed
orig:
  correctly
  guess
  guesses
  guessing
  incorrectly
  he/she
  wondered
  surmised
  answered
  ascertained

paper:
  guesses
  walk-on
  correctly
  guess
  guessing
  incorrectly
  he/she
  surmised
  ascertained
  wondered
```

Figure 4.3: Top 10 nearest neighbors for 3 selected target words, ranked by embedding proximity, using the original embeddings ("`orig`") vs. post-retraining poisoned embeddings ("`paper`").

### 4.5 Reproducibility statements

The GloVe model and the corpus poisoning algorithm were run on Lehigh MAGIC, using four NVIDIA RTX 2080 Ti GPUs and 2x16-core Intel Xeon Silver 4215R CPUs @ 3.20GHz. The attack was executed using Python 3.7.6 using NumPy [7] and SciPy [16] for most CPU calculations, as well as CuPy [12] for CUDA calculations on the GPU. Additionally, a custom CUDA kernel was created using Numba JIT compiler [10].

We used the Wikipedia dataset for the GloVe embeddings and the corpus poisoning calculations. The GloVe embeddings had a max vocab size of 400 thousand, no minimum word count, $c_{max}$ of 100, embedding dimension of 100, a window size of 10, and 50 epochs.

For the poisoning attack, 100 random word pairs were chosen and the similarities were measured before and after the additions were made to the corpus. $max_{\Delta}$ was set at 1250 and $\widehat{\langle t \rangle}_r$ was set to infinity.

## 5 Mitigations

Mitigations for a corpus poisoning attack are not straightforward. This attack requires only an inconsequentially small number of sequences relative to the occurrences of each word, nevertheless the size of the corpus, thus making it more difficult to detect than more naive approaches [15].

Furthermore, the original paper suggests several potential ways to help evade detection. One strategy involves inserting the word "and" at various points within the second-order sequences, making the sequence appear to follow more natural English grammar. Another possibility is attempting to select existing n-grams from the corpus rather than generating new sequences. Both approaches are difficult to detect, and are still largely possible without knowledge of the victim's hyperparameters [15].

The original paper concluded, "How to protect public corpora from poisoning attacks designed to affect NLP models remains an interesting open problem" [15]. Although the paper focused on training data filtering (which seems to be the main known approach at the time), in the time since the paper was published, other teams have investigated new possible techniques to mitigate this and other corpus poisoning attacks. One possible approach is called model robustifying, which focuses on creating training algorithms that are more resilient against corpus poisoning attacks by design and architecture of the neural network itself [17]. Another approach, model verification, focuses on detecting anomalous word embeddings (rather than anomalous training inputs) using a statistical approach. Both methods show promise, and this is an area with lots of room for further exploration in future work.

Furthermore, as NLP research expands, new alternatives to classical word embeddings have begun to gain popularity, making this type of corpus poisoning attack less useful and less effective. One such approach is known as Flair embeddings, which have shown improvements on the results of GloVe and other word-based embeddings in several tasks [1]. Furthermore, one approach to many NLP-related problems that has recently gained widespread popularity even outside academic circles is generative pre-trained transformers, commonly known as GPT [5, 13]. The corpus poisoning attack presented in this paper is likely less effective against applications using these models, and generally any models except word embeddings.

## 6 Conclusion and future work

Word embeddings, which are trained on public and malleable data such as Wikipedia and Twitter, are vulnerable to corpus poisoning attacks. These attacks involve manipulating the data used to train the embedding in order to change the locations of words in the embedding and thus their computed semantic "meaning".

By developing distributional expressions over corpus elements that lead to predictable changes in the embedding distances, an attacker can devise algorithms to optimize effectiveness while minimizing the size of modifications to the corpus. These attacks on the embeddings can affect the result of various NLP tasks that rely on word embeddings. These attacks can be effective even if the attacker does not know the specific embedding algorithm and its hyperparameters.

Defenses such as detecting anomalies in word frequency or removing low perplexity sentences are ineffective; as noted by the original paper, protections against corpus poisoning attacks are still largely an open problem [15].

## References


[1] Alan Akbik, Duncan Blythe, and Roland Vollgraf. Contextual string embeddings for sequence labeling. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 1638–1649, Santa Fe, New Mexico, USA, August 2018. Association for Computational Linguistics.

[2] Giusepppe Attardi. Wikiextractor. `https://github.com/attardi/wikiextractor`, 2015.

[3] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.

[4] Aleksandar Bojchevski and Stephan Günnemann. Adversarial attacks on node embeddings via graph poisoning, 2019.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[6] Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning, 2017.

[7] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[8] Tatsunori B. Hashimoto, David Alvarez-Melis, and Tommi S. Jaakkola. Word embeddings as metric recovery in semantic spaces. *Transactions of the Association for Computational Linguistics*, 4:273–286, 2016.

[9] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world, 2017.

[10] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6, 2015.

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

[12] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. CuPy: A NumPy-compatible library for NVIDIA GPU calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[13] OpenAI. Gpt-4 technical report, 2023.

[14] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.

[15] Roei Schuster, Tal Schuster, Yoav Meri, and Vitaly Shmatikov. Humpty Dumpty: Controlling word meanings via corpus poisoning, 2020.

[16] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[17] Chen Wang, Jian Chen, Yang Yang, Xiaoqiang Ma, and Jiangchuan Liu. Poisoning attacks and countermeasures in intelligent networks: Status quo and prospects. *Digital Communications and Networks*, 8, 07 2021.

[18] Wenkai Yang, Lei Li, Zhiyuan Zhang, Xuancheng Ren, Xu Sun, and Bin He. Be careful about poisoned word embeddings: Exploring the vulnerability of the embedding layers in nlp models, 2021.

# 7 Appendix

Ayon: I was primarily responsible for preparing the final report and presentation. In the report, I contributed to many areas including the introduction, experiments, and concluding sections, as well as researching other papers and finding background information. For the presentation, I made the slides and contributed to making the graphs.

Jacob: I was primarily responsible for the technical side. I wrote all the code for the repository including algorithms 1 and 2, corpus processing, embedding training, and processing the results. I also contributed to the technical side of the report, and helped in a few areas in the written report and the presentation.

Note: we recognize the paper is longer than 8 pages; we decided to include extra data and figures, which contributed significantly to the number of pages.