This post will explain the math and intuition behind word2vec, GloVe and fasttext.

Word2Vec

The word2vec model typically refers to an implementation of one of two models, the continuous bag-of-words (CBOW) or the skip-gram model. They are very similar, CBOW accepts context words as input and predicts the target word and skip-gram accepts the target word as input and predicts a context word. This inversion might seem arbitrary, but it turns out that CBOW smoothes over distributional information by treating an entire context as one observation, which is useful for smaller datasets. Skip-gram on the other hand treats each context-target pair as a new observation, and tends to do better on larger data sets.

Original paper

Overview

Although this post will primarily focus on the skip-gram model, both models are single layer neural networks whose weights we learn. This weight matrix will then contain the word vectors for all of our words. The objective function tries to simultaneously (1) maximize the probability that an observed word appears in the context of it's target word and (2) minimize the probability that a randomly selected word from the vocabulary appears as a context word for the given target word.

If you're still unsure about neural network weights and the weight matrix, I recommend reading this chapter

Deep Dive

The idea behind the skip-gram model is we take a word in an input sequence as the “target” or “center” word, and predict the words around it. ‘Around’ is determined by a pre-specified window size, \(m\). In the figure below, we have a window size of 2, the input word is “banking” and its context words are “turning”, “into”, “crises” and “as”. **We want to quantify the probability that each word in the window appears in the context of the target word.**

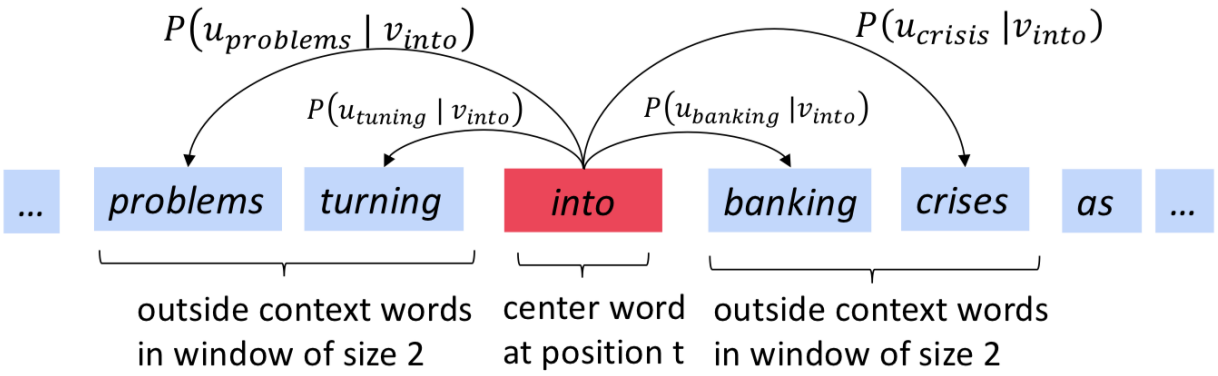


Figure 1: Taken from Stanford’s NLP course, shows the skip gram prediction of “banking” with window size 2.

Before we start to build the objective function discussed above, here’s some useful notation. We have an input sequence of words, \(w_1, w_2,.., w_T\), each of which has a context window, \(-m \le j \le m\). We’ll call this input sequence the *corpus*, and all its unique words the *vocabulary*. Each word in the vocabulary will have 2 vector representations, \(u_o\) for context and \(v_c\) for target. In figure 1, \(u_{turning}\) is the vector representation of “turning” as a context word, and \(v_{banking}\) is the vector representation of “banking” as a target word.

We want to calculate the probability that each word in the window, \(w_{t+j}\), appears in the context of target word \(w_t\). This might seem weird, but the probability is based on the vector representations of each word. Every time we encounter a word in context of a target word, we alter their vector representations to be “closer”. Referring to the example above, \(p(turning | banking)\) > \(p(problems | banking)\), so the vectors for “turning” and “banking” will be updated to be closer than “problems” and “banking”. We can now define a function that describes this. \(\theta\) is a placeholder representing all the vector representations.

\[J(\theta) = -\frac{1}{T} \sum^{T}_{t = 1} \sum_{-m \le j \le m, j \ne 0} log(p(w_{t+j} | w_t; \theta)) \]

The only problem here is we have no idea how to find $p(w_{t+j} | w_t; \theta)$. We'll start with using the softmax function. This will calculate the probability of a word vector, u_o, co-occurring with a target word vector, v_c. It essentially means "how similar is context word u_o to target word v_c, relative to all other context words in the vocabulary". The measure of similarity between two words is measured by the dot product $u_o^T v_c$.

$$ p(w_{t+j} | w_t; \theta) = \frac{e^{u_o^T v_c}}{\sum_{w=1}^W e^{u_w^T v_c}} $$

Where:

- W is the size of the vocabulary
- c and o are indices of the words in the vocabulary at sequence positions t and j respectively
- $u_o = word2vec(w_{t+j})$
- $v_c = word2vec(w_t)$

Although we now have a way of quantifying the probability a word appears in the context of another, the $\sum_{w=1}^W e^{u_w^T v_c}$ requires us to iterate over all words in the vocabulary. To deal with this, we must approximate the softmax probability. One way of doing this is negative sampling.

Negative Sampling Loss

Negative sampling overcomes the need to iterate over all words in the vocabulary to compute the softmax by sub-sampling the vocabulary. We sample k words and determine the probability that these words **do not** co-occur with the target word.The intuition behind this is that a good model should be able to differentiate between data and noise.

To incorporate negative sampling, the objective function needs to be altered by replacing $p(w_{t+j} | w_t)$ with:

$$ log(\sigma(u_o^T v_c)) + \sum_{i = 1}^{k}E_{j \sim P(w)} [log(\sigma(-u_j^T v_c))] $$

Where $\sigma(.)$ is the sigmoid function.

> Thus the task is to distinguish the target word w_t from draws from the noise distribution $P_n(w)$ using logistic regression, where there are k negative samples for each data sample.

New Objective Function

$$ J(\theta) = \frac{-1}{T} \sum_{t=1}^{T}J_t(\theta) $$

$$ J_t(\theta) = log(\sigma(u_o^T v_c)) + \sum_{i = 1}^{k}E_{j \sim P(w)} [log(\sigma(-u_j^T v_c))] $$

$$ P(w) = U(w)^{3/4}/Z $$

Let's look each component of $J_t(\theta)$ and try to convince ourselves this makes sense.

The first part, $log(\sigma(u_o^Tv_c))$, can be interpreted as the log probability of the target and context words co-occurring. We want the model to find u_o and v_c to maximize this probability.

The second part,, $\sum_{i = 1}^{k}E_{j \sim P(w)} [log(\sigma(-u_j^T v_c))]$, is where the "sampling" in negative sampling happens. Let's break this up more to make it clearer. It'll come in handy to note that $\sigma(-x) = 1 - \sigma(x)$.

We can first drop the $E_{j \sim P(w)}$ term, since we already know we will be sampling words from some distribution, $P(w)$:

$$ \sum_{i = 1}^{k} log(\sigma(-u_j^T v_c)) = \sum_{i = 1}^{k} log(1 - \sigma(u_j^T v_c)) $$

Now this is easier to read, we're taking the log of 1 minus the probability that the sampled word, j, appears in the context of the target word c. This is just log of the *probability that j does **not** appear in the context of the target word* c. Since j is randomly drawn out of ~10^6 words, there's a very small chance it appears in the context of c, so this probability should be high. We do this for each of the k sampled words.

Finally, we have to specify a distribution for negative sampling, $P(w) = U(w)^{3/4}/Z$. Here, $U(w)$ is the unigram distribution and is raised to the $\frac{3}{4}$th power to sample rarer words in the vocabulary. Z is just a normalization term.

To summarize, this loss function is trying to maximize the probability that word o appears in the context of word c, while minimizing the probability that a randomly selected word from the vocabulary appears in the context of word c. We use the gradient of this loss function to update the word vectors, u_o and v_c to get our word embeddings.

> SGNS seeks to represent each word w in V_W and each context c in VC as d-dimensional vectors w andc, such that words that are "similar" to each other will have similar vector representations. It does so by trying to maximize a function of the product w ·c for (w, c) pairs that occur in D, and minimize it for negative examples: (w, cN) pairs that do not necessarily occur in D. The negative examples are created by stochastically corrupting observed (w, c) pairs from D – hence the name "negative sampling". For each observation of (w,c), SGNS draws k contexts from the empirical unigram distribution P (c) = #(c).

Summary of Word2Vec

- Iterate through every word in the whole corpus
- Predict surrounding words (context words) using word vectors

- Update the word vectors based on the loss function

Word2vec in Python

Instead of training our own word2vec model, we'll use a pre-trained model to visualize word embeddings. We'll use Google's News dataset model, which can be downloaded here. The model is 1.5Gb, and is trained on a vocabulary of 3 million words, with embedding vectors of length 300. This repo has an in-depth analysis of the words in the model.

We'll use the `gensim` Python package to load and explore the model. If you don't have it installed, run `pip install gensim` in your command line.

```
import gensim

# Download model and save it to current directory, or update the model_path
model_path = "GoogleNews-vectors-negative300.bin"

# Load Google's pre-trained Word2Vec model.
model = gensim.models.KeyedVectors.load_word2vec_format(model_path, binary=True)

# extract word vectors from the model
wv = model.wv

# remove model from env
del model
```

Now we have vector representations for all words in the vocabulary in `wv` and can start to do word math. We'll add and subtract some word vectors, then see what the closest word to the resulting vector is. Publications and blog posts have exhausted the "king" - "man" + "woman" = "queen" example, so I'll present some new ones.

Results generated by the `find_most_similar` function are of the form (word, cosine similarity), where "word" is the closest word to the vector parsed into the function. Cosine similarity values closer to 1 means the vectors (words) are more similar. The function definition can be found in the appendix.

Start with: `doctor - man + woman`

```
find_most_similar(wv["doctor"] - wv["man"] + wv["woman"],
                  ["man", "doctor", "woman"])
```

```
[('gynecologist', 0.7276507616043091),
 ('nurse', 0.6698512434959412),
 ('physician', 0.6674120426177979)]
```

Interesting, what about if we make a subtle change to `doctor - woman + man` ?

```
find_most_similar(wv["doctor"] - wv["woman"] + wv["man"],
                  ["man", "doctor", "woman"])
```

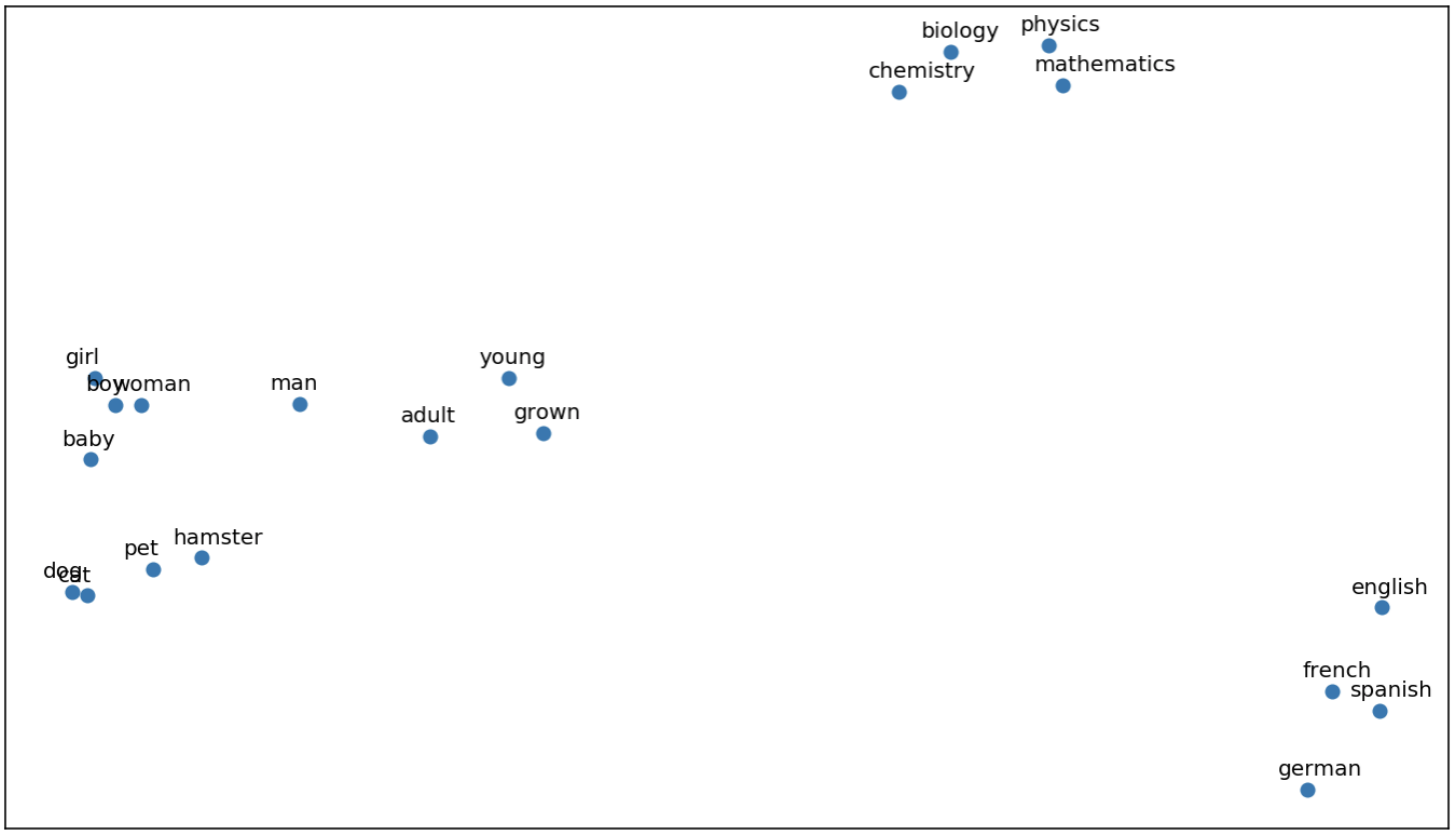
```
[('physician', 0.6823904514312744),
 ('surgeon', 0.5908077359199524),
 ('dentist', 0.570309042930603)]
```

This is a different results from the original query! Biases in the training data are captured and expressed by the model. I won't go into detail about this here. Instead the take away should be that the order of arithmetic for word vectors matters a great deal.

Visualizing word2vec Embeddings

To wrap up word2vec, lets look at how the model clusters different words. I've compiled words from different walks of life to see if word2vec was able to unravel their semantic similarities. These words are parsed through word2vec, then the first 2 principal components are plotted. Some expected similarities are seen here, however we lose a lot of information from reducing the dimension from 300 to 2.

```
# Embedding that makes sense
plot_embeds(["dog", "cat", "hamster", "pet"] +                    # animals
            ["boy", "girl", "man", "woman"] +                     # humans
            ["grown", "adult", "young", "baby"] +                 # age
            ["german", "english", "spanish", "french"] +          # languages
            ["mathematics", "physics", "biology", "chemistry"])  # natural sciences
```



GloVe

GloVe (Global Vectors) is another architecture for producing word embeddings. It improves on some key downsides of the skip-gram model. The main downside of the skip-gram is the loss of corpus statistics due to capturing information one window at a time. To solve this, GloVe incorporates word co-occurrence counts to capture global information about context.

> On the other hand, methods that rely solely on co-occurrence counts (eg: SVD on the co-occurrence matrix) fail to capture rich relationships between words. GloVe tries to incorporate the advantages of both the skip-gram model and count-based models.

> GloVe, for Global Vectors, because the global corpus statistics are captured directly by the model.

Overview

The skip-gram model uses negative sampling to bypass the bottleneck of naive softmax loss. GloVe takes a different approach to this by changing the problem from classification to regression. For each pair of word vectors, GloVe tries to minimize the difference between their dot product and log co-occurrence count.

Deep Dive

The Co-Occurrence Matrix

The co-occurrence matrix, X, is generated from the corpus and vocabulary. The entry at X_{ij} is then the number of times word j occurs in the context of word i. Context is defined in the same way as the skip-gram model. Summing over all the values in row i, will give the number of words that occur in its context, $X_i = \sum_k X_{ik}$. Then the probability of word j occurring in the context of word i is $P(i \mid j) = \frac{X_{ij}}{X_i}$.

Two main advantages of computing the co-occurrence matrix is that it contains all statistical information about the corpus and only needs to be computed once. We will see how it's used in the next section.

Deriving GloVe from Softmax

We can use the softmax function, Q_{ij}, to find the global loss by summing over all target-context word pairs.

$$J = - \sum_{i \in corpus} \sum_{j \in context} log (Q_{ij})$$

Since words i and j appear X_{ij} times in the corpus, we don't need to iterate over all windows in the corpus, but can iterate over the vocabulary instead.

$$J = - \sum_{i = 1}^{W} \sum_{j = 1}^{W} X_{ij} log Q_{ij}$$

Re-arranging some terms, we can come up with this:

$$J = - \sum_{i = 1}^{W} X_{i} \sum_{j = 1}^{W} P_{ij}log(Q_{ij})$$

What's going on right now?

- **Where did P_{ij} come from?** - Remember that $P_{ij} = \frac{X_{ij}}{X_i}$ and $X_i = \sum_k X_{ik}$, therefore we can substitute $X_{ij} = P_{ij}X_i$.
- **What's the relationship between P_{ij} and Q_{ij}?** - P_{ij} is the probability that word j appears in the context of word i, but Q_{ij} is also the probability that word j appears in the context of word i. The difference between the two lies in how they are calculated. P_{ij} is calculated using the co-occurrence matrix and doesn't change. Q_{ij} is the naive softmax probability, that is calculated using the dot product of word vectors u_j and v_i. We have the ability to change Q_{ij} by changing these vectors.
- **What's the point of $P_{ij}log(Q_{ij})$?** - Now that we've refreshed our memory of P and Q, we can see that P is the *true* probability distribution of context and target words, and Q is some made up distribution based on the "goodness" of the word vectors. We really want these

two distributions to be close to each other. Observing $H = P_{ij}log(Q_{ij})$, when P and Q are close to each other, H is small, and when P and Q are far apart, H is larger. Our end goal is the minimization of J, so the smaller H is is better. This is term is the cross-entropy between distributions P and Q.

> Cross entropy error is just one among many possible distance measures between probability distributions, and it has the unfortunate property that distributions with long tails are often modeled poorly with too much weight given to the unlikely events.

See Appendix for more

The problem here is that cross-entropy requires normalized versions of Q_{ij} and P_{ij} which we have to iterate over the entire vocabulary to calculate. This is the reason for using Negative Sampling in the skip-gram model. GloVe's approach to this is dropping the normalization terms completely, so we end up with $\hat{P}$ and $\hat{Q}$. Cross-entropy loss now becomes useless, so we change $H = P_{ij}log(Q_{ij})$ to squared error, $(\hat{P}_{ij} - \hat{Q}_{ij})^2$.

$$ \hat{J} = \sum_{i = 1}^{W} X_{i} \sum_{j = 1}^{W} (\hat{P}_{ij} - \hat{Q}_{ij})^2 $$

Now we have squared error, weighted by the number of co-occurrences of words i and j. There's one last problem with this, which is that some co-occurrence counts can be massive. This will affect both the weights, X_i, and $\hat{P_{ij}} = X_{ij}$. To deal with this explosion in the least squared term, we take $log(hat{P})$ and $log(hat{Q})$) and to deal with the explosion of weights, we introduce a function, f that caps the co-occurrence count weight. We'll apply this weight to each target-context paid, X_{ij} as opposed to only X_i.

This will *heavily* favor common words, like "the", "and", "of", etc. We introduce a function, f, to cap the co-occurrence weighting. The new objective function then becomes:

$$ \hat{J} = \sum_{w = 1}^{W} \sum_{w = 1}^{W} f(X_{ij}) (u_j^T v_i - log(X_{ij}))^2 $$

This is the loss function that the GloVe model minimizes.

GloVe in Python

Similar to word2vec, we'll use the `gensim` package to load a pre-trained GloVe model.

```
import gensim.downloader as api

# Download pretrained GloVe model from:
# https://nlp.stanford.edu/projects/glove/
glove_model = api.load("glove-wiki-gigaword-100")
glove = glove_model.wv

del glove_model
```

Fasttext

GloVe VS Word2vec VS Fasttext

- Which is more robust?
- Which is more efficient?
- What does word2vec capture than GloVe doesn't?
-

Appendix

Objective Function: Recall that an objective or loss function, $J(\theta)$, is a way of determining the goodness of a model. We alter the parameters of this function, θ, to find the best fit for the model.

From Word2vec to GloVe - Math

Naive Softmax function: $$ Q_{ij} = \frac{e^{u_j^T v_i}}{\sum_{w=1}^W e^{u_w^T v_i}} $$ The bottleneck to the naive softmax function is that the calculation of $\sum_{w=1}^W e^{u_w^T v_i}$ requires iteration over the entire vocabulary.

Summing negative log of Q_{ij} to get the global loss: $$J = - \sum_{i \in corpus} \sum_{j \in context} log (Q_{ij}) $$

\[J = - \sum_{i = 1}^{W} X_{i} \sum_{j = 1}^{W} P_{ij}log(Q_{ij}) \] The term: \(\sum_{j = 1}^{W} P_{ij}log(Q_{ij})\) is the cross-entropy of \(P_{ij}\) and \(Q_{ij}\).

Code

```
# some word2vec examples
dog_vec = wv["dog"] # "dog" embedding

# Distances from each word in animal_list to the word animal
animal_list = ["dog", "cat", "mouse", "hamster"]
animal_similarity = wv.distances("animal", animal_list)
list(zip(animal_list, animal_similarity))


# find the 3 most similar words to the vector "vec"
def find_most_similar (vec, words = None) :
    # vec: resulting vector from word Arithmetic
    # words: list of words that comprise vec
    s = wv.similar_by_vector(vec, topn = 10)
    # filter out words like "king" and "man", or else they will be included in the simi
    if (words != None) :
      word_sim = list(filter(lambda x: (x[0] not in words), s))[:3]
    else :
      return (s[:3])
    return (word_sim)

def plot_embeds(word_list, word_embeddings = None, figsize = (10,10)) :
    # pca on the embedding
    pca = PCA(n_components=2)
    X = pca.fit_transform(wv[word_list])

    ax = plt.figure(figsize=figsize)
    ax.subplots()
    _ = plt.scatter(X[:,0], X[:,1])
    for label, point in list(zip(word_list, X)):
        _ = plt.annotate(label, (point[0], point[1]))
```

Resources

- Small review of GloVe and word2vec
- Evaluating unsupervised word embeddings
- Stanford NLP coursenotes on GloVe
- Stanford NLP coursenotes on word2vec
- GloVe

- Stanford NLP coursenotes
- Gensim Models
- Word2vec in Tensorflow
- GloVe Blog post
- Atom markdown docs.