CS7.401: Introduction to NLP | Assignment 2

Author: Aditya Kumar Singh
ID: 2021701010

March 25, 2022

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

Contents

Question 1: Theory

Explain negative sampling. How do we approximate the `word2vec` training computation using this technique?

Intuition:
Going by the name of these two words one can clearly state some sort of sampling (of words maybe) is going on. But why negative? We'll discuss it in a short while.

First of all we'll discuss how we get here, build the intuition and then will construct the mathematical formulation to understand it better. As mentioned in question itself, `word2vec` model is *Context-based* model (or can say "prediction-based" model) that given a local context, predict the target words and in the meantime, learns the efficient word embedding representation. Plus, it is supervised in nature.

But with this approach there lies one computational **limitation** while training i.e., *projecting to output vocabulary.*

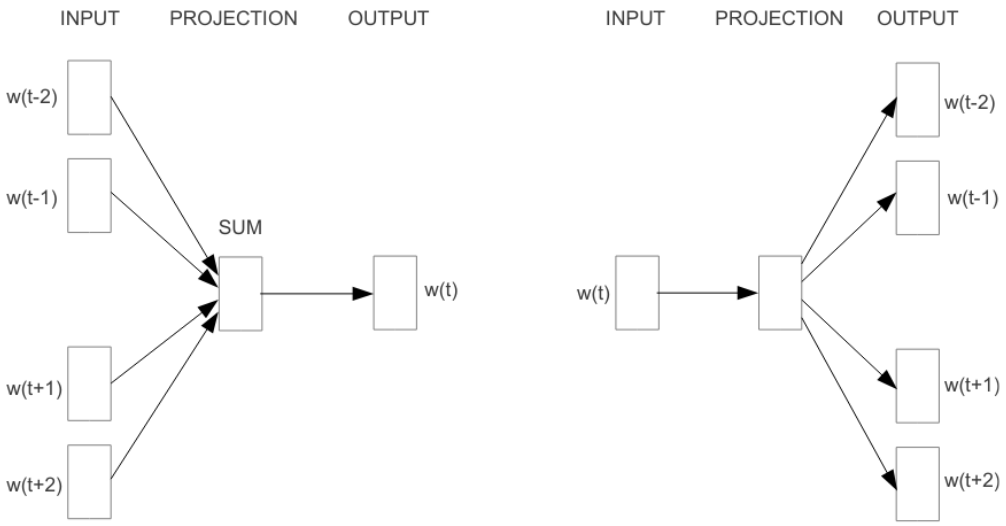


Figure 1: The CBOW architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word.

According to the figure 2, the W' matrix (also called *Decoding* matrix) can be thought of as collections of column vectors which when get multiplied (in dot product fashion) with some "dense" embedding vector coming from `one-hot` representation of some word through W marix (or the *embedding* matrix), outputs a similarity score. Since we have $|V|$ no. of column vectors in W', where $|V|$ is the vocabulary size, for each embedding vector we perform $|V|$ dot products to output a vector of length $|V|$ that consist of similarity scores, which is then normalized with `softmax` function to obtain mutually exclusive probability score (as all values lie within $(0, 1)$ that sums up to 1). And this is all a `word2vec` model do while inferencing. Finally, the vector obtained from `softmax` is then used for knowing `one-hot` representation for the word that should come after the input word.

One variant of such model is `CBOW` (called as Continuous Bag of Words Model), that predicts the target word from source context words. For example, consider sentence "CBOW was best in NNLM", where say the target word is "best" and with a window size of 5 the context words will be "CBOW", "was", "in", and "NNLM". Because there are multiple contextual words, we average their corresponding embeddings, and try to predict the representation for the word "best".

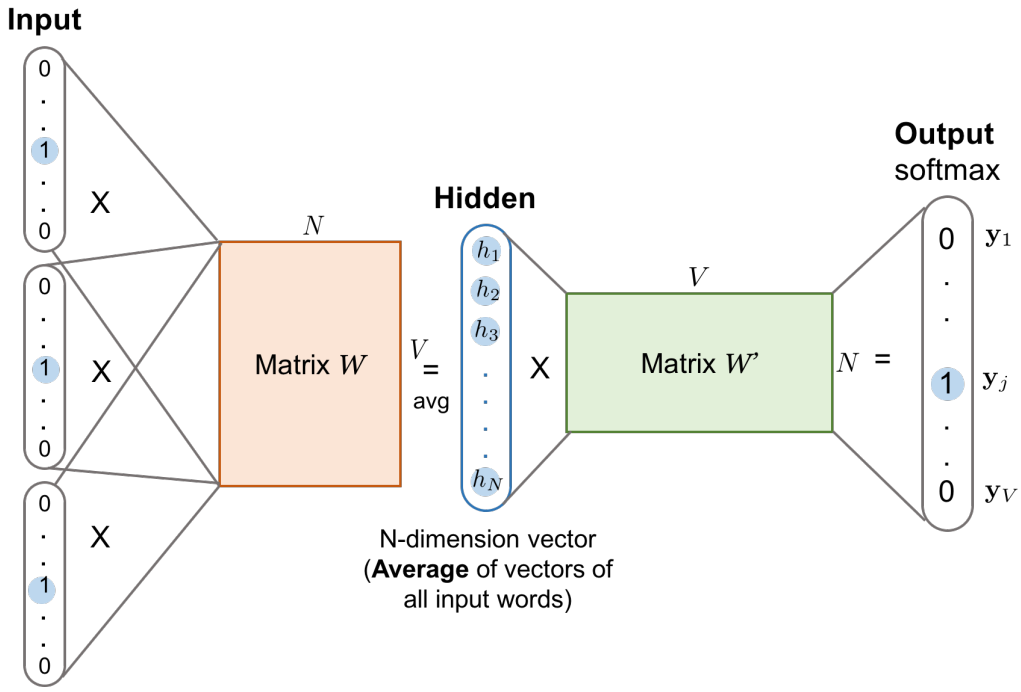


Figure 2: The CBOW model. Word vectors of multiple context words are averaged to get a fixed-length vector as in the hidden layer. Both the input vector X and the output Y are one-hot encoded word representations. The hidden layer is the word embedding of size N.

So far so good. Let's hop back to the problem we've where numerous dot products are being done (probably a billion, as for some corpus $|V|$ can really shoot up that high) which is highly inefficient and how to mitigate this is the need of the hour. Not only one word rather each sample in the training set multiplies with W' (i.e., project itself onto "vocabulary" space) to output an `one-hot` vector. In the process, it force W' matrix to output 1 at that index where the next word is supposed to be in vocabulary, while for the rest of the words its ideally should be 0. By that all the parameters in W' are getting modified.

Now here comes the *Negative Sampling* (NS) [2]. Instead of forcing W' to output 0 for all those words that doesn't occur at that index, NS says output 0 for few words in vocabulary (say 5). Hence with negative sampling, we're randomly selecting (will discuss shortly how to select) just a small number of "negative" words to update only specific weights of W' . We will also still update the weights for our "positive" word. For example, "Quick Fox" occurs more often together, so given "quick" our model will able to output 1 for "fox" (i.e., "yes" label for their occurrences). Along with that, it would also try to learn to predict 0 for 5 pairs such as: (quick bottle), (quick keys), (quick ink), (quick curry), (quick rubber). According to the paper [2], 5-20 words works well for smaller datasets and with only 2-5 words for large datasets. And hence in this manner NS approximates `word2vec` training computation.

(*Just a quick statistics, say* W' *is* $350 \times 10,000$*, then in our case updates will happen just for* 6 *words (one is positive while the rest* 5 *are negative words) which involves a total of 6 neurons* $= 350 \times 6 = 2100$ *weights values. And that in terms of* % *is very low i.e.,* $= \frac{2100}{3.5 \times 10^6} \times 100 = 0.06\%$*!*)

Negative Sampling originally is a variation of **NCE** loss [1] that attempts to approximately maximize the log probability of the `softmax` output. On part of NCE (Noise Contrastive Estimation), it intends to differentiate the target word from noise samples (or negative) using a *logistic regression classifier* and hence its loss

function looks something as follows:

$$\mathcal{L}_\theta = -\left[\log p(d=1|w, w_I) + \sum_{i=1, \tilde{w}_i \sim Q}^{N} \log p(d=0|\tilde{w}_i, w_I)\right]$$
$$\mathcal{L}_\theta = -\left[\log p(d=1|w, w_I) + N\mathbb{E}_{\tilde{w}_i \sim Q} \log p(d=0|\tilde{w}_i, w_I)\right]$$
due to Law of Large Numbers

Approximating, these probabilities terms with `softmax` of cosine similarity of embeddings (i.e., the dot product of vectors) gives us the loss function for NS as shown below:

$$\mathcal{L}_\theta = -\left[\log \sigma({v'}_w^T v_{w_I}) + \sum_{i=1, \tilde{w}_i \sim Q}^{N} \log \sigma(-{v'}_{\tilde{w}_i}^T v_{w_I})\right]$$

Because of this it able to focus on learning high-quality word embedding rather than modeling the word distribution in natural language. Finally our task of predicting from a neighboring word got switched to "classification" where a model that takes the input and output word, and outputs a score indicating if they're neighbors or not (0 for "not neighbors", 1 for "neighbors").

How to select? Selecting negative samples follows a "unigram distribution" , where more frequent words are more likely to be selected as negative samples. But not always the most frequent word is a better choice (e.g., "the" appears most often which is not that appropriate for every word's context), for which they [2] took $\frac{3}{4}$ power of frequencies and normalized it with the sum of those frequencies which increases the probability for less frequent words and decrease the probability for more frequent words to be chosen as "negative" word.

Question 2: Implementation

Question 2.1

Implement a word embedding model and train word vectors by first building a Co-occurrence Matrix followed by the application of `SVD`.

How to run:

Step 1: First do as said in the **Instruction** part and go to `svd_src` folder.

Step 2: Run `python co_occurence_matrix.py` to generate co-occurence matrix, `co_occurence.npz`, which automatically get saved in `models` folder. Also `token_list.txt` is file is generated and saved to `models` folder $\Longrightarrow$ consists mainly of `tokens`.

Step 3: Run `python get_svd.py` to generate svd matrix, `svd_matrix.npz`, which again will be found in `models` folder. *We took* 350 *dimension as our embedding vector size (i.e., chose top* 350 *eigen-values for our reduced representation).*

Step 4: (Optional) To view t-SNE representation of word-embeddings run
`python svd_inference.py --name <image_name>`

How it's working?

1. Given the text corpus, we preprocess it to generate tokens and then vocabulary.
2. Seeing the pairwise occurences of words we construct a co-occurence matrix using `CountVectorizer()` funtcion of `sklearn`.
3. Next, we compute the eigen-values of this co-occurence matrix using `TruncatedSVD` from `sklearn`.
4. We collect the top 350 eigen-vectors (couldn't able to verify the "variance-check" using ratio of summation of eigen-values) and stack it to obtain $|V| \times k$ matrix, say U, where k is the reduced representation dimension.
5. Next for inferencing, get the index of word whose embedding you want to see. And pick the corresponding row of U to get the embedding of that word.

Question 2.2

Implement the `word2vec` model and train word vectors using the `CBOW` model with Negative Sampling.

How to run:

Step 1: First do as said in the **Instruction** part and be in `cbow_negs_src` folder.

Step 2: Run `python CBOW_NEG.py` to generate embedding matrix, `word_embedding.txt`, which automatically get saved in `models` folder.

Step 3: If you need to visualize the plot (or how the top-10 representation of a word embedding looks like), run `python plot.py --name <image_name>` to see the `t-SNE` plot.

Step 4: (Optional) `get_train_data.py` is to split the original data in `../models/data.txt` into `train` and `val`.

How it's working?

1. To implement the `CBOW` with `Negative Sampling`, we tweak the model from prediction task to *classification* task.
2. Here we take context words (in our case it's 8 neighbouring words) and target word as input to the model (which is a simple matrix W, the encoding matrix) and classify whether they can stay together or not, by labelling them one or zero. Note that we take representation of these 8 words and average them (which probably denotes the aggregated meaning) and use it with target word's embedding to know whether those 8 words are really its (target) context or not by taking sigmoid of their dot product.
3. Not only that we also take target word and negative word pairs (probably 5 such pairs for each pair of positive sample so as to approximate `word2vec`) and their embeddings and train the model to predict 0 for them.
4. In order to train separately both context and target matrix we take two separate matrix, known as "Embedding" matrix and "Context" matrix. At the start of the training process, we initialize these matrices with random values.

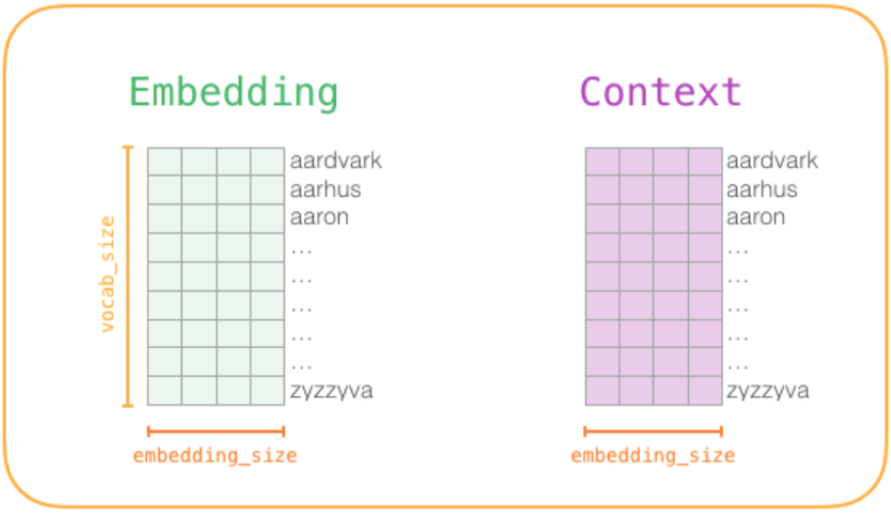


Figure 3

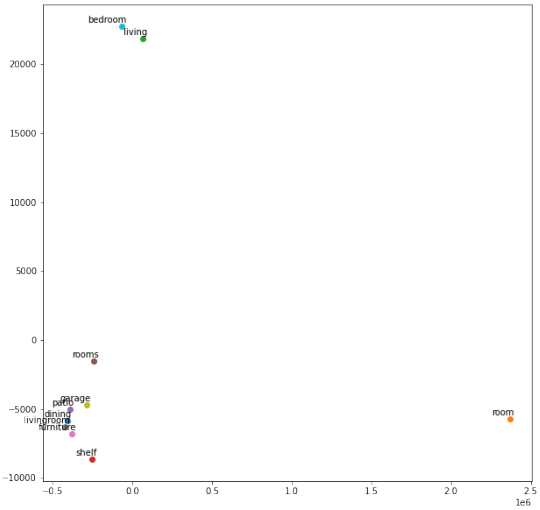
5. **How data is fed?**
   ([context words list, target word], [negative word 1, target word], [negative word 2, target word], [negative word 3, target word], [negative word 4, target word]) $\longrightarrow$ This altogether forms one batch for model.

Question 3: Analysis

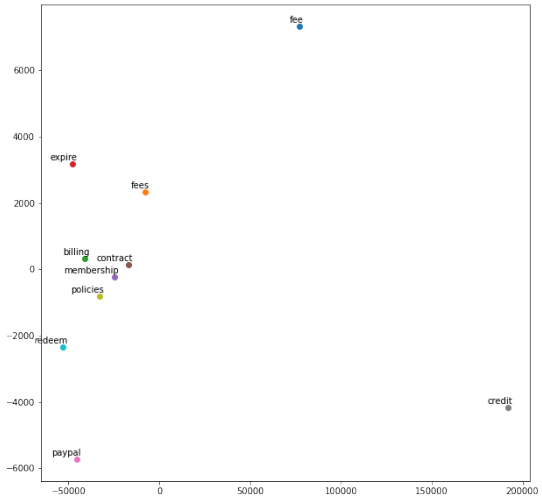
Report these for both models after you're done with the training.

Question 3.1

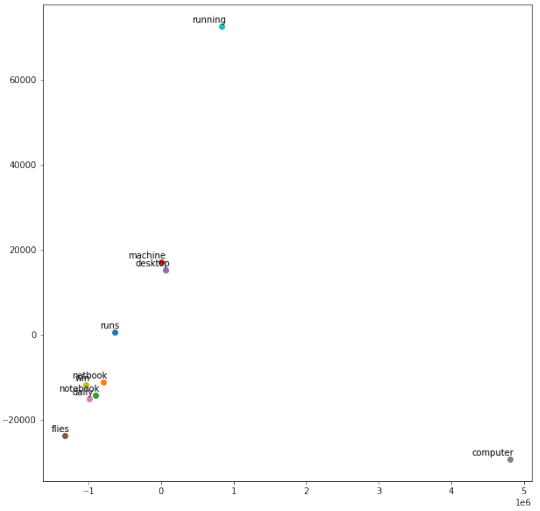
Display the top-10 word vectors for five different words (a combination of nouns, verbs, adjectives, etc.) using `t-SNE` (or such methods) on a 2D plot.



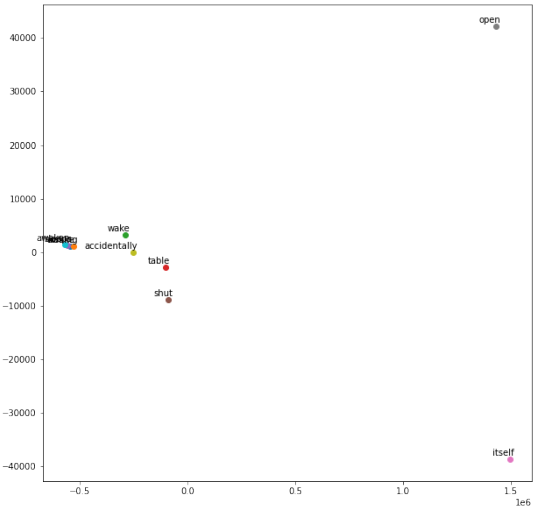
(a) Kitchen: dining, room, living, shelf, patio, rooms, furniture, livingroom, garage, bedroom



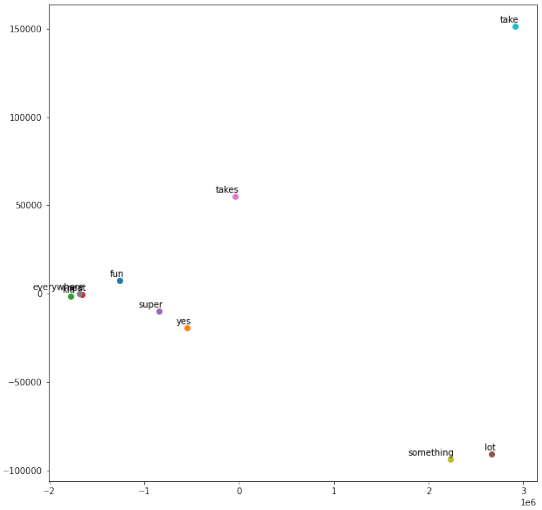
(b) Payment: fee, fees, billing, expire, membership, contract, paypal, credit, policies, redeem



(c) Laptop: runs, netbook, notebook, machine, desktop, flies, daily, computer, win, running

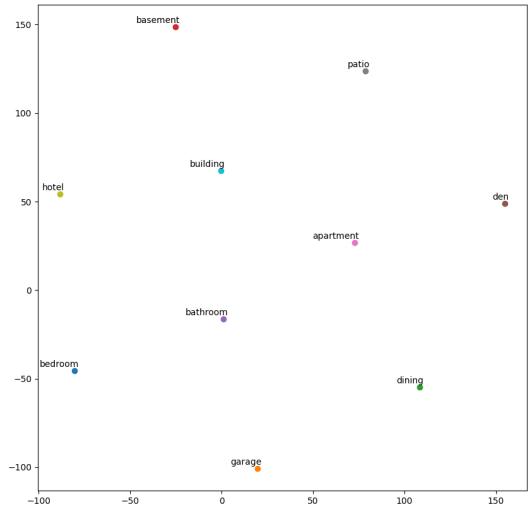


(d) Sleep: awake, waking, wake, table, sleeps, shut, itself, open, accidentally, awaken

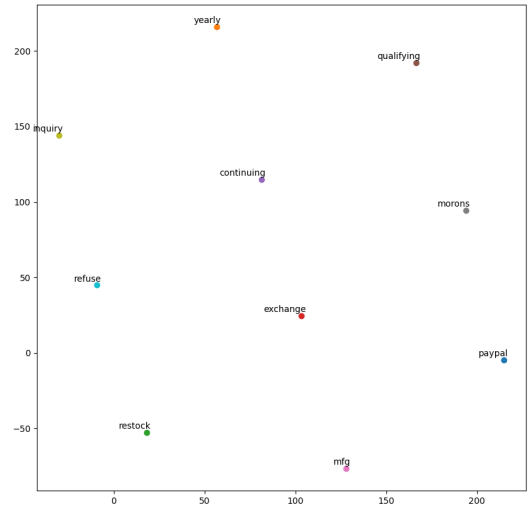


(a) Toy: fun, yes, kid, neat, super, lot, takes, everywhere, something, take

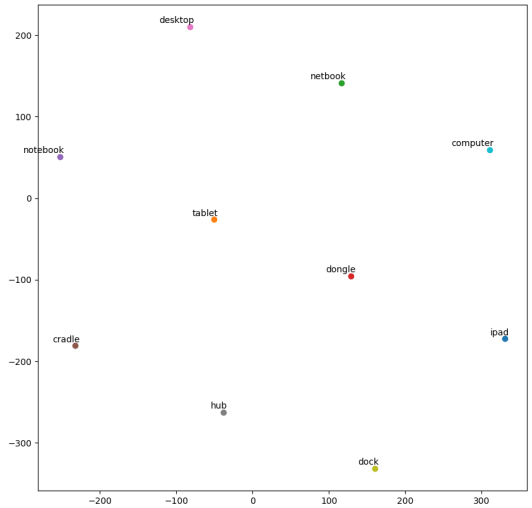
Figure 5: With SVD embeddings, the top-10 word vectors.



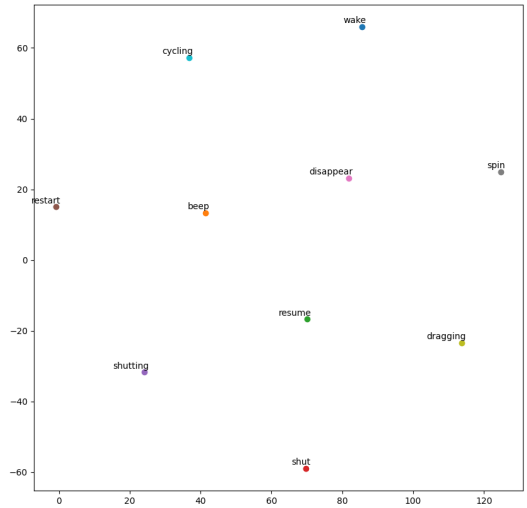
(a) Kitchen



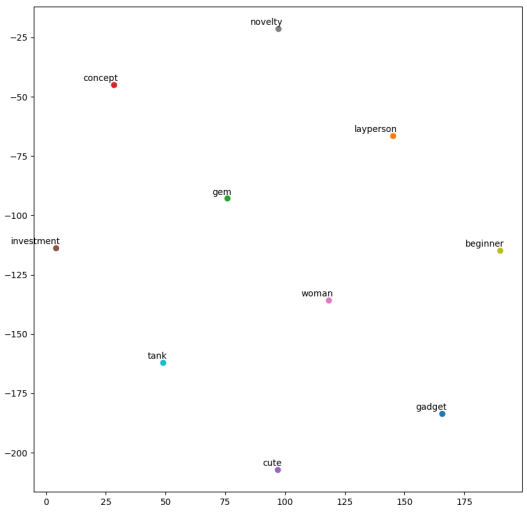
(b) Payment



(a) Laptop



(b) Sleep

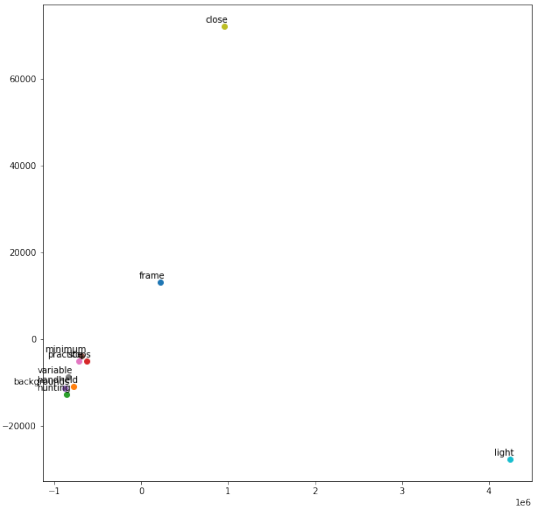


(c) Toy

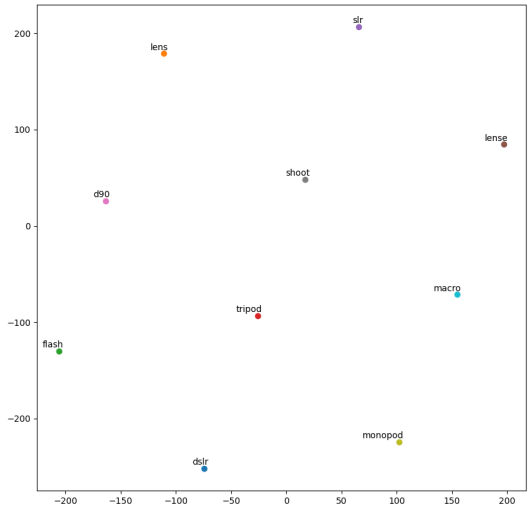
Figure 7: With CBOW + NegSampling embeddings, the top-10 word vectors.

Question 3.2

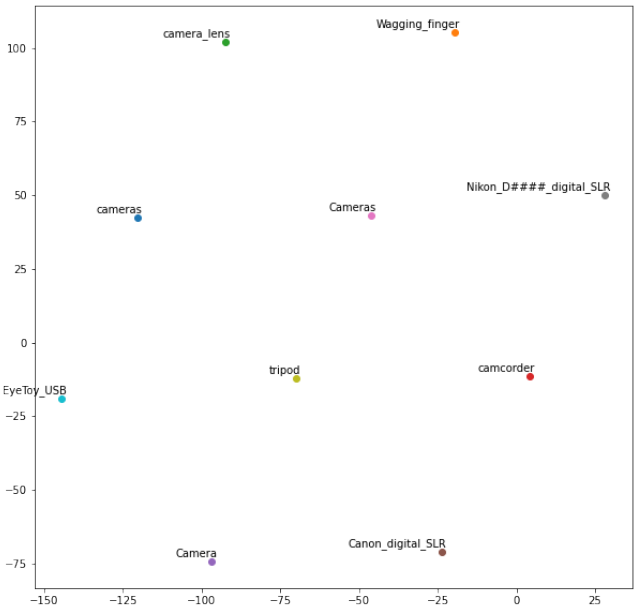
What are the top 10 closest words for the word ‘camera’ in the embeddings generated by your program? Compare them against the pre-trained `word2vec` embeddings that you can download off-the-shelf.



(a) Top-10 words related to camera embedding obtained from SVD: frame, handheld, hunting, stops, backgrounds, minimum, practice, variable, close, light



(b) Top-10 words related to camera embedding obtained from CBOW + NegSampling



(c) Top-10 words related to camera embedding obtained from Google's word2vec embeddings.

Figure 8: Comaprison of "camera" embeddings obtained from 3 different source.

How to run?

- Download and extract the folder to your local system.
- Make sure that you have following PYTHON3 packages:
  - json
  - numpy
  - tqdm, collections, pickle, matplotlib, re
  - scikit-learn
  - scipy
  - gensim
  - torch

- If you didn't have downloaded the data, download it to the folder by typing wget http://snap.stanford.edu/data/amazon/productGraph/categoryFiles /reviews_Electronics_5.json.gz in your terminal which is also opened to same folder path.
- To create data from `.json` file run python dataFromJson.py. And make sure that `.json` file is in the folder 2021701010_assignment2.
- To obtain `SVD` embeddings, try to follow the steps mentioned in Question Question 2.1.
- Same for `CBOW` embeddings, try to follow the steps mentioned in Question Question 2.2.

References

Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 297–304. JMLR Workshop and Conference Proceedings, 2010.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.