# NLP – Assignment – 2

## REPORT

**Name:** Harshit Gupta

**Roll No:** 2020114017

MADE WITH PYTHON

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

The network weights are modified during the training of a neural network in order to learn the representations in the training data accurately. When training data is exceedingly huge, it creates a slew of problems in terms of computing expenses. We have utilized **Word2Vec** in our code, which is a neural network-based natural language processing model. Word2vec models run into problems when the size of the training data increases. To address this problem, word2vec models use a technique called **negative sampling**, which allows only a small percentage of network weights to be changed during training.

**In-Depth Working:**
A neural network is trained by taking a training sample and slightly adjusting all of the neuron weights such that it can predict that training sample more accurately. To put it another way, each training sample will change the weights of the neural network. Our skip-gram neural network has a vast number of weights due to the breadth of our word vocabulary, all of which would be slightly changed by each of our numerous training samples.

Negative sampling addresses this by changing only a small percentage of the weights in each training sample, rather than all of them. This is how things work. When training the network on the word pair ("X", "Y"), keep in mind that the network's "label" or "correct output" is a one-hot vector. That is, the "Y" output neuron should create a 1, whereas the rest of the hundreds of output neurons should produce a 0.

Instead, we'll use negative sampling to update the weights by selecting a small number of "negative" words at random (let's say 5). (In this context, a "negative" phrase is one for which we want the network to output a 0.) We'll additionally keep the weights for our "positive" term (in this case, the letter "Y") updated. As a result, just the weights associated with them will be modified, and only the loss will be propagated back to them.

## Implementation

1. **Model 1:** We implement a word embedding model and train word vectors by first building a Co-occurrence Matrix followed by the application of SVD.
2. **Model 2:** We implement the word2vec model and train word vectors using the CBOW model with Negative Sampling.

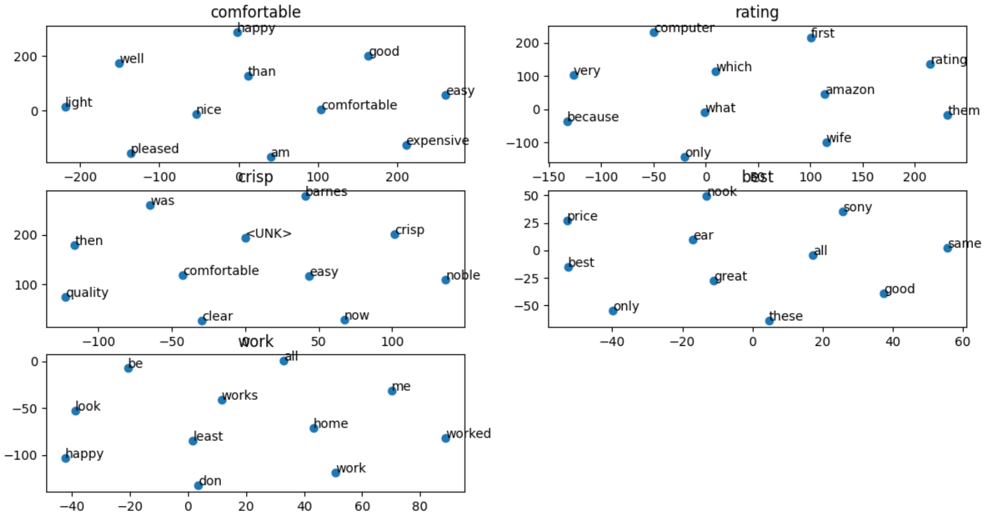
## Analysis

# Analysis 1:

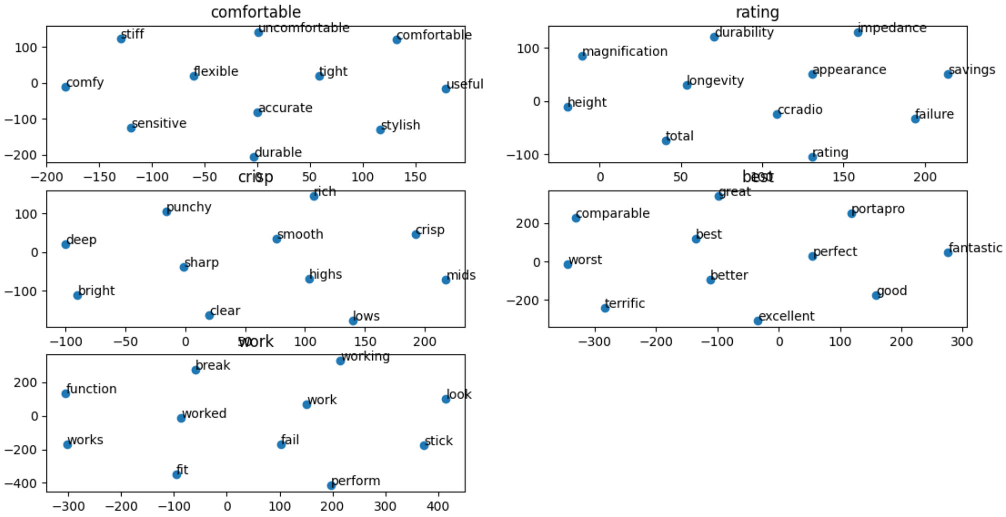
**Requirement:** Display the top-10 word vectors for five different words (a combination of nouns, verbs, adjectives, etc.) using t-SNE (or such methods) on a 2D plot.

The 5 words on which we shall be basing our analysis are: comfortable, rating, crisp, best, work.

**The t-SNE graph for `Model 1` is:**



**The t-SNE graph for `Model 2` is:**

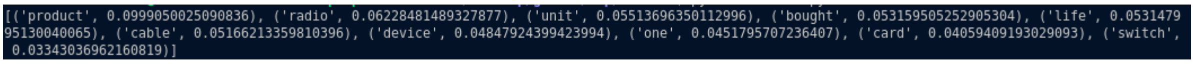


# Analysis 2:

**Requirement:** 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.

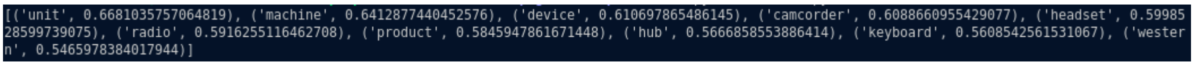
The 5 words for which we shall find the top 10 words are: comfortable, rating, crisp, best and work.

**The top 10 closest words for the word 'camera' using `Model 1`:**



```
[('product', 0.0999050025090836), ('radio', 0.06228481489327877), ('unit', 0.05513696350112996), ('bought', 0.053159505252905304), ('life', 0.0531479
95130040065), ('cable', 0.05166213359810396), ('device', 0.04847924399423994), ('one', 0.0451795707236407), ('card', 0.04059409193029093), ('switch',
 0.03343036962160819)]
```

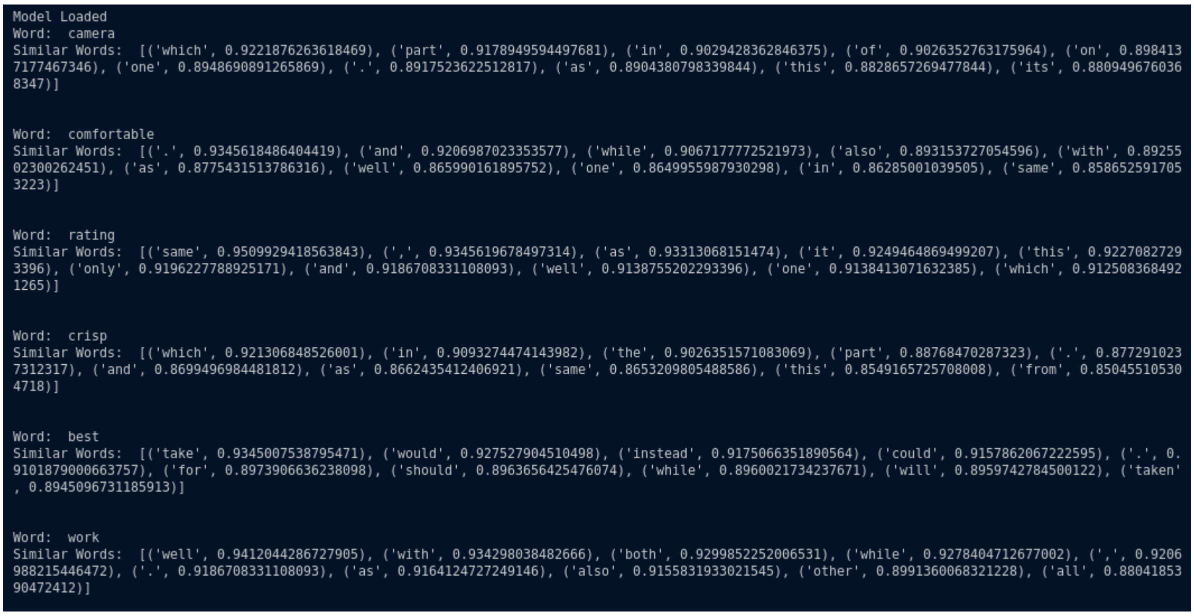
### The top 10 closest words for the word ’camera’ using `Model 2`:



```
[('unit', 0.6681035757064819), ('machine', 0.6412877440452576), ('device', 0.610697865486145), ('camcorder', 0.6088660955429077), ('headset', 0.59985
28599739075), ('radio', 0.5916255116462708), ('product', 0.5845947861671448), ('hub', 0.5666858553886414), ('keyboard', 0.5608542561531067), ('wester
n', 0.5465978384017944)]
```

### The top 10 closest words for the word ’camera’ using the `Pretrained Model`:

(We have included the top 10 words for the 5 words above as well in this image.)



```
Model Loaded
Word:  camera
Similar Words:  [('which', 0.9221876263618469), ('part', 0.9178949594497681), ('in', 0.9029428362846375), ('of', 0.9026352763175964), ('on', 0.898413
7177467346), ('one', 0.8948690891265869), ('.', 0.8917523622512817), ('as', 0.8904380798339844), ('this', 0.8828657269477844), ('its', 0.880949676036
8347)]


Word:  comfortable
Similar Words:  [('.', 0.9345618486404419), ('and', 0.9206987023353577), ('while', 0.9067177772521973), ('also', 0.893153727054596), ('with', 0.89255
02300262451), ('as', 0.8775431513786316), ('well', 0.865990161895752), ('one', 0.8649955987930298), ('in', 0.86285001039505), ('same', 0.858652591705
3223)]


Word:  rating
Similar Words:  [('same', 0.9509929418563843), (',', 0.9345619678497314), ('as', 0.93313068151474), ('it', 0.9249464869499207), ('this', 0.9227082729
3396), ('only', 0.9196227788925171), ('and', 0.9186708331108093), ('well', 0.9138755202293396), ('one', 0.9138413071632385), ('which', 0.912508368492
1265)]


Word:  crisp
Similar Words:  [('which', 0.921306848526001), ('in', 0.9093274474143982), ('the', 0.9026351571083069), ('part', 0.88768470287323), ('.', 0.877291023
7312317), ('and', 0.8699496984481812), ('as', 0.8662435412406921), ('same', 0.8653209805488586), ('this', 0.8549165725708008), ('from', 0.85045510530
4718)]


Word:  best
Similar Words:  [('take', 0.9345007538795471), ('would', 0.927527904510498), ('instead', 0.9175066351890564), ('could', 0.9157862067222595), ('.', 0.
9101879000663757), ('for', 0.8973906636238098), ('should', 0.8963656425476074), ('while', 0.8960021734237671), ('will', 0.8959742784500122), ('taken'
, 0.8945096731185913)]


Word:  work
Similar Words:  [('well', 0.9412044286727905), ('with', 0.934298038482666), ('both', 0.9299852252006531), ('while', 0.9278404712677002), (',', 0.9206
988215446472), ('.', 0.9186708331108093), ('as', 0.9164124727249146), ('also', 0.9155831933021545), ('other', 0.8991360068321228), ('all', 0.88041853
90472412)]
```

# *Observations*

Some interesting observations noted while running the models were:

1. The most similar/ closest words depend heavily on the dataset. The Word2Vecs for the pretrained model and for the model trained on our dataset gave different embeddings.
2. When the `epochs` are increased for the Word2Vec model, the vectors you get for the closest words of the input word approach 1. While this seems more tempting to repeatedly train our model with the new `epochs`, we must understand that this can result in overfitting. Normally, the ideal epoch range is between 10 and under 50.
3. The similar words generated for each word come under the following classification categories:
    1. Same grammatical relations.
    2. Similar context usage as the closest words.
    3. Globally commonly used bigrams.

The advantage of utilising the **Word2Vec** model for training is that you can regulate the learning rate and handle fresh training data gracefully.

For raw word vectors, **SVD** must be done all at once, and sparse matrix abstractions must be used. It's really more sophisticated than word2vec models because of the technology and abstractions you utilise.

Any type of sequence can be used to train **Word2Vec**. You may change the learning rate on the fly (to emphasise earlier or later occurrences), pause or resume incremental training, and train embeddings for more abstract tokens in a much more basic way.

While **Word2Vec** embeddings are not always repeatable, they become significantly more stable as fresh training data is added. This demonstrates that a production system can be stable throughout time.