IMPORTANCE OF FEATURE ENGINEERING → PART-2

In continuation to previous let's proceed further

##

1) AMAZON FOOD REVIEW

##

ABOUT DATA: →

The Amazon Fine Food Reviews dataset consists of reviews of fine foods from Amazon.
Number of reviews: 568,454
Number of users: 256,059
Number of products: 74,258
Timespan: Oct 1999 - Oct 2012
Number of Attributes/Columns in data: 10
Attribute Information:

1) Id
2) ProductId - unique identifier for the product
3) UserId - unqiue identifier for the user
4) ProfileName
5) HelpfulnessNumerator - number of users who found the review helpful
6) HelpfulnessDenominator - number of users who indicated whether they found the review helpful or not
7) Score - rating between 1 and 5
8) Time - timestamp for the review
9) Summary - brief summary of the review
10) Text - text of the review

TEXT PREPROCESSING

In the Preprocessing phase we do the following in the order below:

- Begin by removing the html tags
- Remove any punctuations or limited set of special characters like, or . or # etc.
- Remove Stopwords
- Check if the word is made up of English letters and is not alpha-numeric
- Check to see if the length of the word is greater than 2 (as it was researched that there is no adjective in 2-letters)
- Convert the word to lowercase
- Finally Snowball Stemming the word (it was observed to be better than Porter Stemming)

```
def cleanhtml(sentence): #function to clean the word of any html-tags
    cleanr = re.compile('<.*?>') #Find this kind of pattern
    cleantext = re.sub(cleanr, ' ', sentence) #Substitute nothing where pattern matches
    return cleantext
def cleanpunc(sentence): #function to clean the word of any punctuation or special characters
    cleaned = re.sub(r'[?|!|\'|"|#]',r'',sentence)
    cleaned = re.sub(r'[.|,|)|(|\|/]',r' ',cleaned)
    return  cleaned
```

```
stop = stopwords.words('english') #All the stopwords in English language
#excluding some useful words from stop words list as we doing sentiment analysis
excluding = ['against','not','don', "don't",'ain', 'aren', "aren't", 'couldn', "couldn't", 'didn', "didn't",
             'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'isn', "isn't",
             'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't",'shouldn', "shouldn't", 'wasn',
             "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
stop = [words for words in stop if words not in excluding]

print(stop)
```

STEMMING & LEMMATIZATION

Reference --> https://www.datacamp.com/community/tutorials/stemming-lemmatization-python

STEMMING

- Languages we speak and write are made up of several words often derived from one another. When a language contains words that are derived from another word as their use in the speech changes is called Inflected Language.
- "Stemming is the process of reducing inflection in words to their root forms such as mapping a group of words to the same stem even if the stem itself is not a valid word in the Language."
- Stem (root) is the part of the word to which you add inflectional (changing/deriving) affixes such as (-ed,-ize, -s,-de,mis). So stemming a word or sentence may result in words that are not actual words. Stems are created by removing the suffixes or prefixes used with a word.
- NLTK --> Natural Language Tool Kit (NLTK) is a Python library to make programs that work with natural language. It provides a user-friendly interface to datasets that are over 50 corpora and lexical resources such as WordNet Word repository. The library can perform different operations such as tokenizing, stemming, classification, parsing, tagging, and semantic reasoning.

For the English language, you can choose between PorterStammer or SnowballStammer

LEMMATIZATION

- Lemmatization, unlike Stemming, reduces the inflected words properly ensuring that the root word belongs to the language. In Lemmatization root word is called Lemma. A lemma (plural lemmas or lemmata) is the canonical form, dictionary form, or citation form of a set of words.
- For example, runs, running, ran are all forms of the word run, therefore run is the lemma of all these words. Because lemmatization returns an actual word of the language, it is used where it is necessary to get valid words.
- Python NLTK provides WordNet Lemmatizer that uses the WordNet Database to lookup lemmas of words.

**** Stemming or Lemmatization ****

- Stemming and Lemmatization both generate the root form of the inflected words. The difference is that stem might not be an actual word whereas, lemma is an actual language word.
- Stemming follows an algorithm with steps to perform on the words which makes it faster. Whereas, in lemmatization, you used WordNet corpus and a corpus for stop words as well to produce lemma which makes it slower than stemming. You also had to define a parts-of-speech to obtain the correct lemma.

Stemming and Lemmatization Differences

- Both lemmatization and stemming attempt to bring a canonical form for a set of related word forms.
- Lemmatization takes the part of speech in to consideration. For example, the term 'meeting' may either be returned as 'meeting' or as 'meet' depending on the part of speech.
- Lemmatization often uses a tagged vocabulary (such as Wordnet) and can perform more sophisticated normalization. E.g. transforming mice to mouse or foci to focus.
- Stemming implementations, such as the Porter's stemmer, use heuristics that truncates or transforms the end letters of the words with the goal of producing a normalized form. Since this is algorithm based, there is no requirement of a vocabulary.
- Some stemming implementations may combine a vocabulary along with the algorithm. Such an approach for example convert 'cars' to 'automobile' or even 'Honda City', 'Mercedes Benz' to a common word 'automobile'
- A stem produced by typical stemmers may not be a word that is part of a language vocabulary but lemmatizer transform the given word forms to a valid lemma.

In this example we will see some feature engineering technique on text data.

We need to convert TEXT data into VECTOR

There are different techniques to convert text to vector

1) BAG OF WORDS

2) TF-IDF
3) W2V (Word to Vector)
    a. AVGW2V
    b. TFIDFW2V

BAG OF WORDS

- Construct a Dictionary or CORPUS: it is set of all the unique words
- Let's say we have n-reviews and d-unique words
- Now for each review we will construct d-dimensional vector.
    - SIMPLE BOW:→ For each review d-dimensional vector is filled according to the number of times word appear in respective review, number of times word appears is specific to the review for which we are constructing d-dimensional vector
    - BINARY BOW: → Here instead of using count we consider whether word is present or not, If word appear once or more than once we will fill value as 1 while for no occurrences filled with 0

To understand it let's take an example from → https://medium.com/greyatom/an-introduction-to-bag-of-words-in-nlp-ac967d43b428

We do following steps in order to construct BoW
1) Construct a Dictionary or Corpus or Lexicon : Set of all Unique words
2) Now in each document count the number of times Dictionary words appears
    *In Binary BOW instead of count we have 1 & 0
Let's take an example to understand this concept in depth.
"It was the best of times"
"It was the worst of times"
"It was the age of wisdom"
"It was the age of foolishness"
We treat each sentence as a separate document and we make a list of all words from all the four documents excluding the punctuation. We get,
'It', 'was', 'the', 'best', 'of', 'times', 'worst', 'age', 'wisdom', 'foolishness'
The next step is the create vectors. Vectors convert text that can be used by the machine learning algorithm.
We take the first document — "It was the best of times" and we check the frequency of words from the 10 unique words.
"it" = 1
"was" = 1
"the" = 1
"best" = 1
"of" = 1
"times" = 1
"worst" = 0
"age" = 0
"wisdom" = 0
"foolishness" = 0

Rest of the documents will be:
"It was the best of times" = [1, 1, 1, 1, 1, 1, 0, 0, 0, 0]
"It was the worst of times" = [1, 1, 1, 0, 1, 1, 1, 0, 0, 0]
"It was the age of wisdom" = [1, 1, 1, 0, 1, 0, 0, 1, 1, 0]
"It was the age of foolishness" = [1, 1, 1, 0, 1, 0, 0, 1, 0, 1]
In this approach, each word or token is called a "gram". Creating a vocabulary of two-word pairs is called a bigram model.
For example, the bigrams in the first document: "It was the best of times" are as follows:
"it was"
"was the"
"the best"
"best of"
"of times"
The process of converting NLP text into numbers is called **vectorization** in ML. Different ways to convert text into vectors are:
Counting the number of times each word appears in a document.
Calculating the frequency that each word appears in a document out of all the words in the document.

NGRAM: → According to number of words considered gram decided.
If we use one word than it is called UNIGRAM
If we use two word than it is called BIGRAM

FOR UNIGRAM

```
#BoW
uni_gram = CountVectorizer(min_df=10) #in scikit-learn
uni_gram.fit(X_100k_train)
print('='*75)
print("some feature names ", uni_gram.get_feature_names()[:10])
print('='*75)
final_counts = uni_gram.transform(X_100k_train)
print("the type of count vectorizer ",type(final_counts))
print("the shape of out text BOW vectorizer ",final_counts.get_shape())
print("the number of unique words ", final_counts.get_shape()[1])
print('='*75)
```

```
===========================================================================
some feature names  ['abandon', 'abc', 'abdomin', 'abil', 'abl', 'abomin', 'abroad', 'absenc', 'absent', 'absolut']
===========================================================================
the type of count vectorizer  <class 'scipy.sparse.csr.csr_matrix'>
the shape of out text BOW vectorizer  (49000, 6336)
the number of unique words  6336
===========================================================================
```

FOR BIGRAM

```
#bi-gram, tri-gram and n-gram

#removing stop words like "not" should be avoided before building n-grams
count_bigram = CountVectorizer(ngram_range=(1,2)) #in scikit-learn
count_bigram_fit = count_bigram.fit(X_100k_train)
count_bigram_txfm = count_bigram.transform(X_100k_train)
print('='*100)
print("some feature names ", count_bigram_fit.get_feature_names()[:10])
print('='*100)
print("the type of count vectorizer ",type(count_bigram_txfm))
print("the shape of out text BOW vectorizer ",count_bigram_txfm.get_shape())
print("the number of unique words including both unigrams and bigrams ", count_bigram_txfm.get_shape()[1])
```

PROBLEM with BOW
If there is very small difference between review1 and review2 but meaning is totally opposite then BOW cannot predict it.
Now if we are using BINARY BOW the difference between BOW are square root of squares of difference of different words as 1&1, 0&0 will be cancel out, 1&0, 0&1 is useful

1) **Semantic meaning:** the basic BOW approach does not consider the meaning of the word in the document. It completely ignores the context in which it's used. The same word can be used in multiple places based on the context or nearby words.
2) **Vector size:** For a large document, the vector size can be huge resulting in a lot of computation and time. You may need to ignore words based on relevance to your use case.

TF-TDF (Term Frequency – Inverse Document Frequency)

It is weighting factor which is used to get the important features from the corpus.
TF-IDF weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus.

TF (TERM FREQUENCY)

It will take care that how often does word Wi occurs in review Rj
It tells how important a word is to a document in a corpus, the importance of a word increases proportionally to the number of times the word appears in the individual document.
Term Frequency is a scoring of the frequency of the word in the current document. Since every document is different in length, it is possible that a term would appear much more times in long documents than shorter ones. The term frequency is often divided by the document length to normalize.

$$TF(t) = \frac{Number\ of\ times\ term\ t\ appears\ in\ a\ document}{Total\ number\ of\ terms\ in\ the\ document}$$

IDF (INVERSE DOCUMENT FREQUENCY)

IDF is a scoring of how rare the word is across documents. IDF is a measure of how rare a term is. Rarer the term, more is the IDF score.

If multiple documents or reviews contain word many times or frequent word in all other document in our corpus so it doesn't give much meaning so it probably may not be an important feature.

So IDF gives how much information the word provides that is whether the term is common or rare across all documents

So, finally from TF-IDF we will get most important features from the corpus with weights

$$IDF(t) = log_e(\frac{Total\ number\ of\ documents}{Number\ of\ documents\ with\ term\ t\ in\ it})$$

TF-IDF steps are very similar to BOW

```
tf_idf_vect = TfidfVectorizer(ngram_range=(1,2))
tf_idf_vect.fit(X_100k_train)

print("some sample features(unique words in the corpus)",tf_idf_vect.get_feature_names()[0:10])
Vocabulary = tf_idf_vect.vocabulary_
print('='*75)
print('Vocabulary: ')
print('='*75)
iterator = iter(Vocabulary.items())
for i in range(15):
    print(next(iterator))
print('='*50)

print('='*75)
# if we wanted to get multiple vectors at once (for product and available) to build matrices
print('Print all rows for column words product & available')
print(tf_idf_vect.transform(['product', 'available']).toarray())
print('='*75)


final_tf_idf = tf_idf_vect.transform(X_100k_train)
print("the type of count vectorizer ",type(final_tf_idf))
print("the shape of out text TFIDF vectorizer ",final_tf_idf.get_shape())
print("the number of unique words including both unigrams and bigrams ", final_tf_idf.get_shape()[1])
```

PROBLEM with TF-IDF

Same like BOW doesn't take semantic meaning

1) **Semantic meaning:** the TFIDF approach does not consider the meaning of the word in the document. It completely ignores the context in which it's used.
2) **Vector size:** For a large document, the vector size can be huge resulting in a lot of computation and time. You may need to ignore words based on relevance to your use case.

WORD2VEC

As we mentioned before BOW & TF-IDF does not consider semantic meaning into consideration

Here word is not converted into sparse vector

Each word is represented in some D-dimensional Vector, this D dimensional vector is calculated using neural word embeddings, These D-dimensional vectors are calculated in such a way that similar word are close to each other in D-dimensional space.

Word embedding is capable of capturing context of a word in a document, semantic and syntactic similarity, relation with other words, etc.
What are word embeddings exactly?
They are vector representations of a particular word. Having said this, what follows is how do we generate them? More importantly, how do they capture the context?
Word2Vec is one of the most popular technique to learn word embeddings using shallow neural network.

Why we need WORD2VEC?

Consider the following similar sentences: `Have a good day and Have a great day.`
They hardly have different meaning. If we construct an exhaustive vocabulary (let's call it V), it would have V = {Have, a, good, great, day}.
Now, let us create a one-hot encoded vector for each of these words in V. Length of our one-hot encoded vector would be equal to the size of V (=5). We would have a vector of zeros except for the element at the index representing the corresponding word in the vocabulary. That particular element would be one.
The encodings below would explain this better.

```
Have = [1,0,0,0,0];
a = [0,1,0,0,0];
good = [0,0,1,0,0];
great = [0,0,0,1,0];
day = [0,0,0,0,1]
```

If we try to visualize these encodings, we can think of a 5-dimensional space, where each word occupies one of the dimensions and has nothing to do with the rest (no projection along the other dimensions). This means 'good' and 'great' are as different as 'day' and 'have', which is not true.
Our objective is to have words with similar context occupy close spatial positions. Mathematically, the cosine of the angle between such vectors should be close to 1, i.e. angle close to 0.

How WORD2VEC works?

Word2Vec is a method to construct such an embedding. It can be obtained using two methods (both involving Neural Networks): Skip Gram and Common Bag Of Words (CBOW)

- Word2vec is a two-layer neural net that processes text. Its input is a text corpus and its output is a set of vectors: feature vectors for words in that corpus. While Word2vec is not a deep neural network, it turns text into a numerical form that deep nets can understand.
- The purpose of Word2vec is to **group the vectors of similar words together in vector space.** That is, it detects similarities mathematically. Word2vec creates vectors that are distributed numerical representations of word features, features such as the context of individual words. It does so without human intervention. Word2vec can make highly accurate guesses about a word's meaning based on past appearances.
- The output of the Word2vec neural net is a vocabulary in which each item has a vector attached to it, which can be fed into a deep-learning net or simply queried to detect relationships between words.
- Measuring cosine similarity, no similarity is expressed as a 90-degree angle, while total similarity of 1 is a 0-degree angle, complete overlap; i.e. Sweden equals Sweden, while Norway has a cosine distance of 0.760124 from Sweden, the highest of any other country.

Neural Word Embeddings
The vectors we use to represent words are called neural word embeddings, and representations are strange.
Word2vec "vectorizes" about words, and by doing so it makes natural language computer-readable
we can start to perform powerful mathematical operations on words to detect their similarities.

word2vec trains words against other words that neighbour them in the input corpus.
It does so in one of two ways,

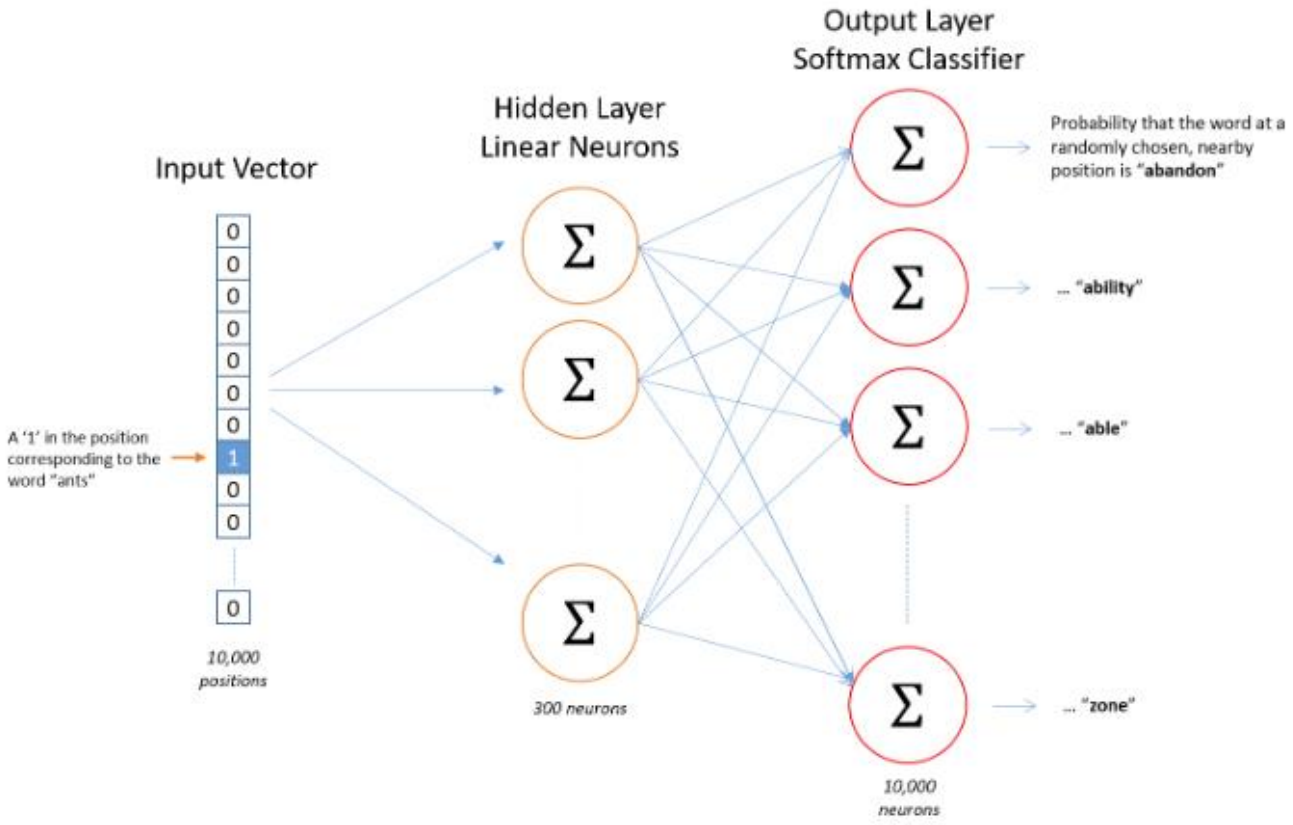
CBOW → either using context to predict a target word (a method known as continuous bag of words, or CBOW),
SKIP-GRAM → or using a word to predict a target context, which is called skip-gram.

SKIP-GRAM

To understand SKIPGRAM i checked this link → http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

The aim of skip-gram is to predict the context given a target word. To understand Context and target word consider the we have words in sentence → w1 w2 w3 w4 w5 w2
Now suppose we want to calculate word2vec for w3 so w3 will be target word & w1, w2, w4, w5, w2 will be context word depending on how many context word we want to consider.

we first build a vocabulary of words from our training documents–let's say we have a vocabulary of 10,000 unique words.
We're going to represent an input word like "ants" as a one-hot vector. This vector will have 10,000 components (one for every word in our vocabulary) and we'll place a "1" in the position corresponding to the word "ants", and 0s in all of the other positions.
The output of the network is a single vector (also with 10,000 components) containing, for every word in our vocabulary, the probability that a randomly selected nearby word is that vocabulary word.
Here's the architecture of our neural network.



Take a 3 layer neural network with 1 input layer , 1 hidden layer and 1 output layer Assume we have 10000 unique words in dictionary so for one word "ants" we get a 10000 sized vector as an input then we take 300 neurons and do the neural network training for all words using skip gram model. Once the training is complete, we get the final weights for hidden layer and output Layer

THE HIDDEN LAYER

Ignore the last (output layer) and keep the input and hidden layer. So, we get the 300 sized weights (scores) for every word Now, input a word from within the vocabulary. The output given at the hidden layer is the 'word embedding' of the input word.

By training this network, we would be creating a 10,000 x 300 weight matrix connecting the 10,000-length input with the 300-node hidden layer. Each row in this matrix corresponds to a word in our 10,000-word vocabulary – so we have effectively reduced 10,000 length one-hot vector representations of our words to 300 length vectors. The weight matrix essentially becomes a look-up or encoding table of our words. Not only that, but these weight values contain context information due to the way we've trained our network. Once we've trained the network, we abandon the SoftMax layer and just use the 10,000 x 300 weight matrix as our word embedding lookup table.

Now, you might be asking yourself "That one-hot vector is almost all zeros… what's the effect of that?" If you multiply a 1 x 10,000 one-hot vector by a 10,000 x 300 matrix, it will effectively just select the matrix row corresponding to the "1". Here's a small example to give you a visual.

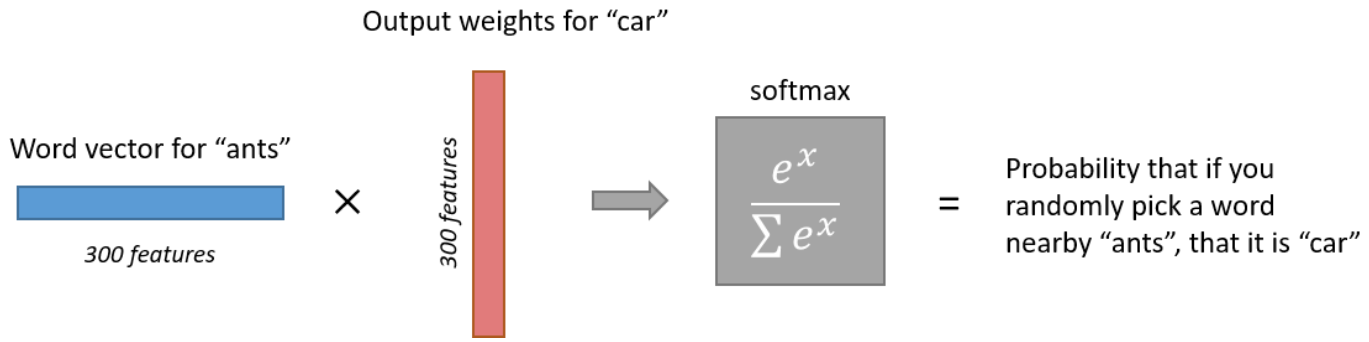
$$[0 \quad 0 \quad 0 \quad 1 \quad 0] \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = [10 \quad 12 \quad 19]$$

This means that the hidden layer of this model is really just operating as a lookup table. The output of the hidden layer is just the "word vector" for the input word.

THE OUTPUT LAYER

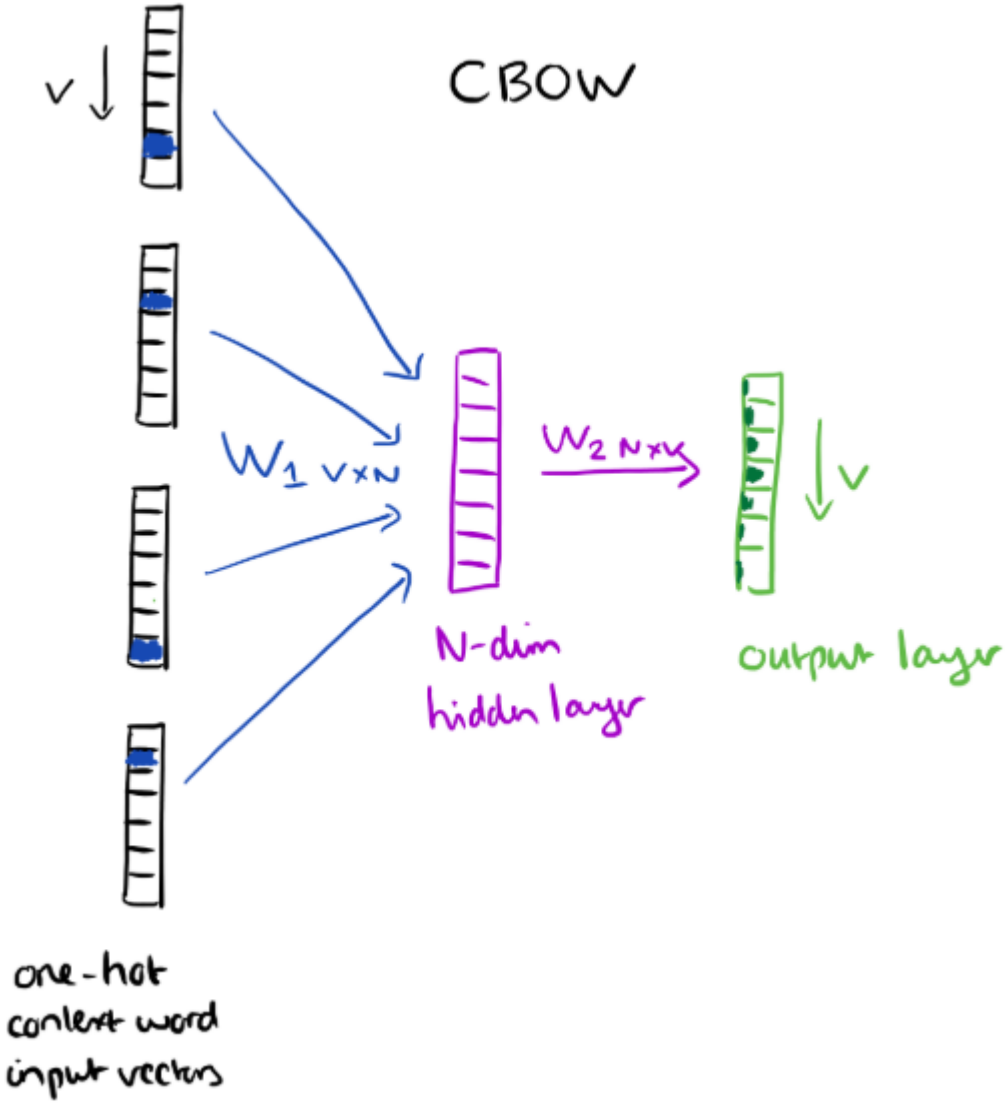
The 1 x 300 word vector for "ants" then gets fed to the output layer. The output layer is a softmax regression classifier.
each output neuron (one per word in our vocabulary!) will produce an output between 0 and 1, and the sum of all these output values will add up to 1.

Specifically, each output neuron has a weight vector which it multiplies against the word vector from the hidden layer, then it applies the function exp(x) to the result. Finally, in order to get the outputs to sum up to 1, we divide this result by the sum of the results from all 10,000 output nodes.
Here's an illustration of calculating the output of the output neuron for the word "car".



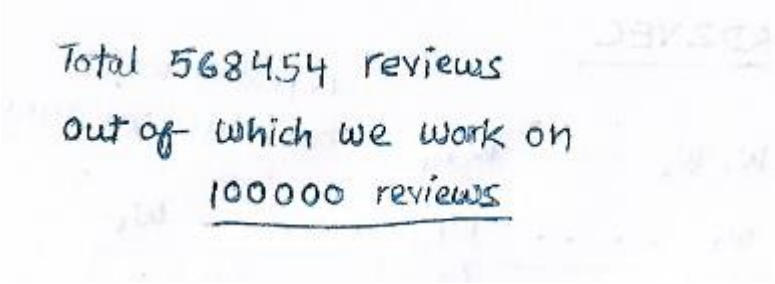
CBOW (CONTINOUS BAG OF WORDS)

This method takes the context of each word as the input and tries to predict the word corresponding to the context. Consider our example: *Have a great day.*
Let the input to the Neural Network be the word, *great.* Notice that here we are trying to predict a target word (*d*ay*)* using a single context input word *great.* More specifically, we use the one hot encoding of the input word and measure the output error compared to one hot encoding of the target word (*d*ay). In the process of predicting the target word, we learn the vector representation of the target word.
The input or the context word is a one hot encoded vector of size V. The hidden layer contains N neurons and the output is again a V length vector with the elements being the softmax values.



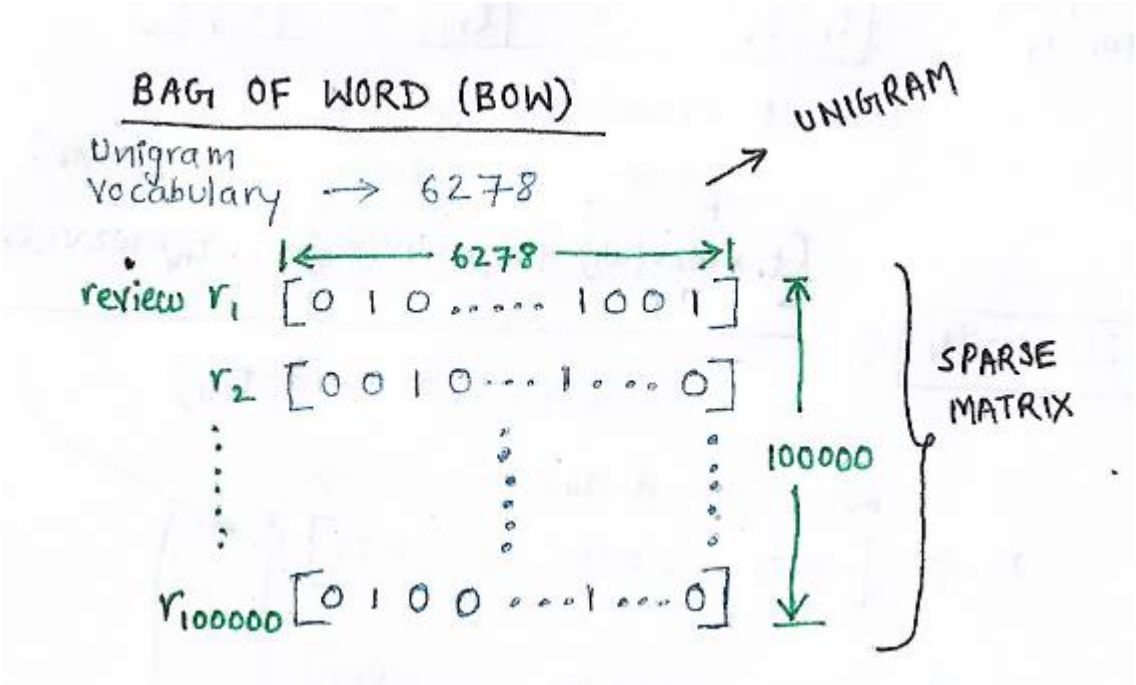
Now since we have basic understanding of BOW, TF-IDF, WORD2VEC & theory ready so let's try to connect them with our amazon food review case study

In amazon food review dataset, we have total 568454 reviews but we consider 100000 reviews due to memory limitations

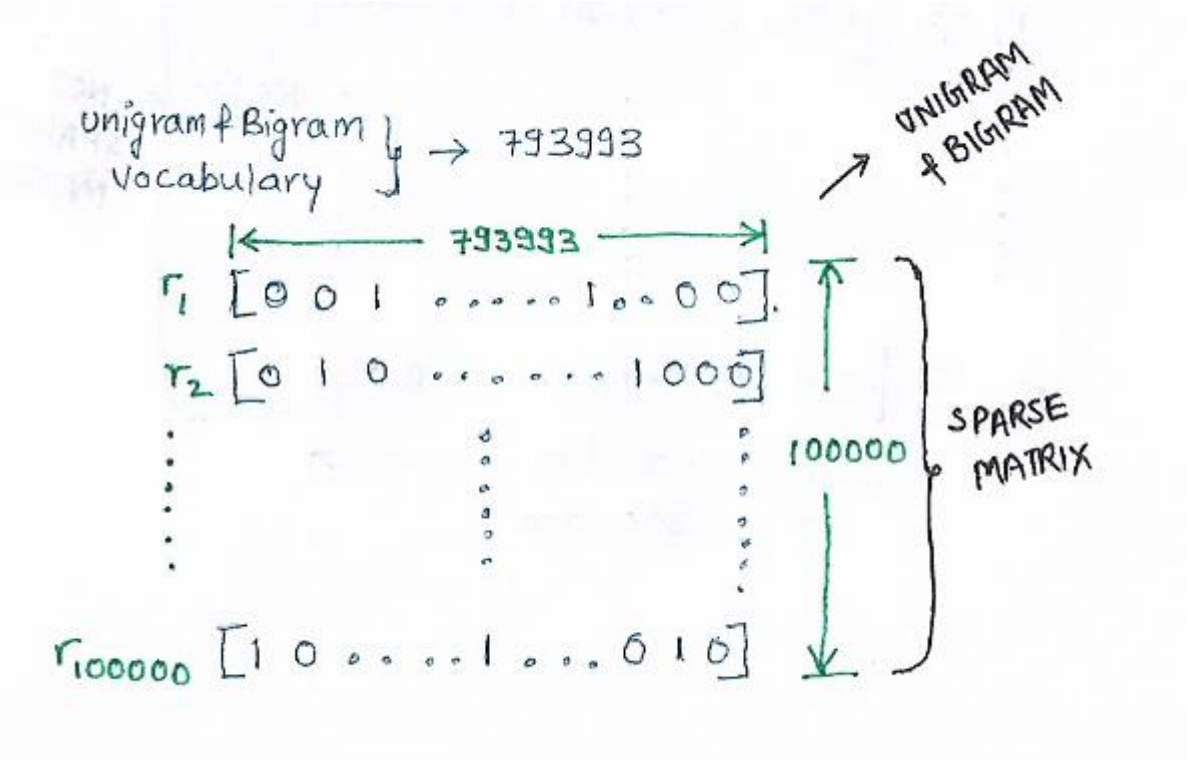


BAG OF WORDS (BOW)

Now when we convert text data to vectors using BOW, while creating vocabulary we found total 6278 unique words so each review will then be converted into dimension of 6278 and which create a sparse matrix



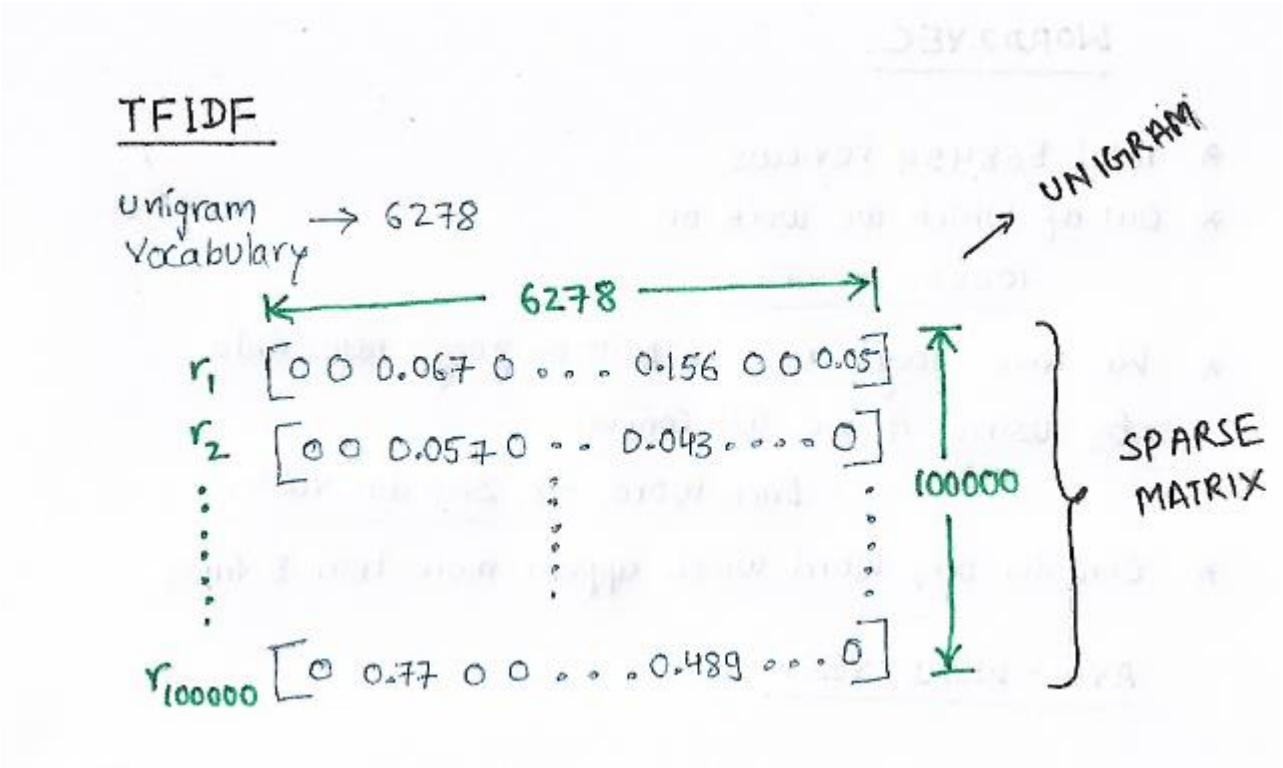
Now for unigram & bigram while creating vocabulary we found total 793993 unique combination so each review will then be converted into dimension of 793993 and which create a sparse matrix



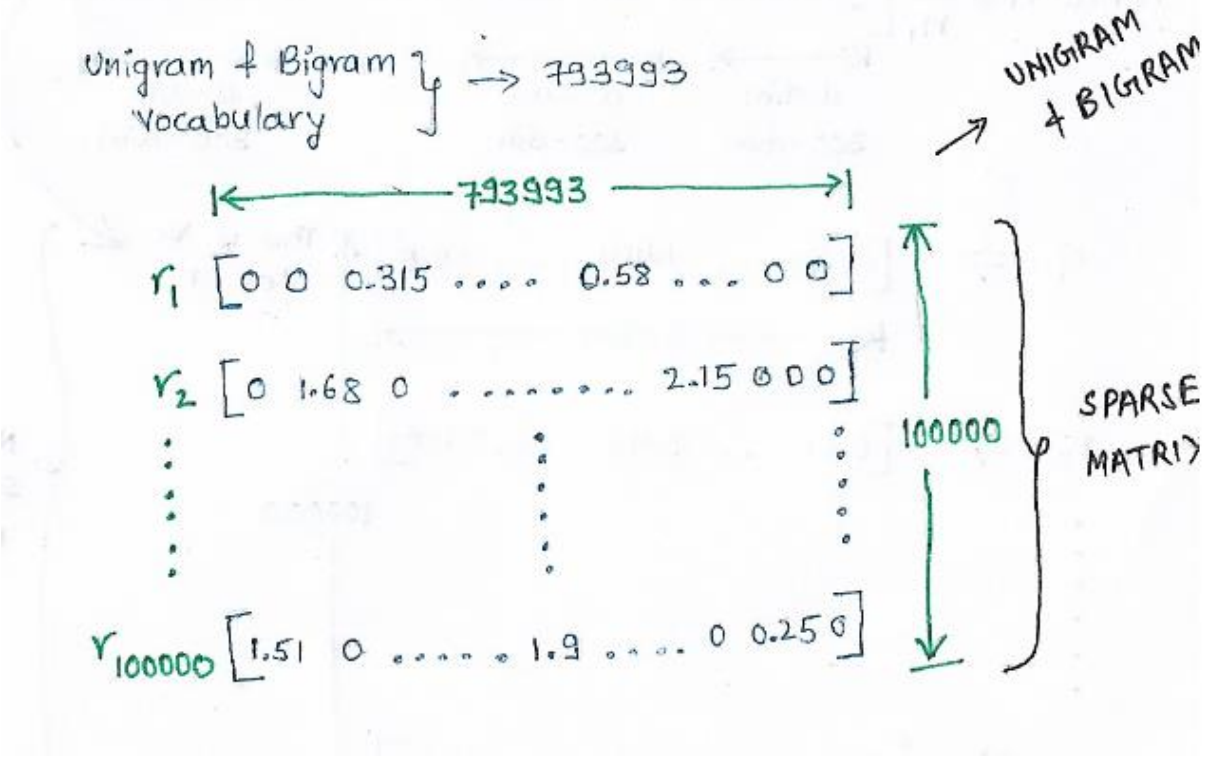
IN BOW we can use count vector as well where in place of 1 we will have count of the word, rest other concept remains the same.

For TFIDF

Similar to BOW instead of 1 we will use TFIDF values for respective words for each review, here also sparse matrix will be created and dimensionality depends on vocabulary.
Following we have explained details about unigram word matrix



Now for unigram & bigram while creating vocabulary we found total 793993 unique combination so each review will then be converted into dimension of 793993 and which create a sparse matrix



WORD2VEC

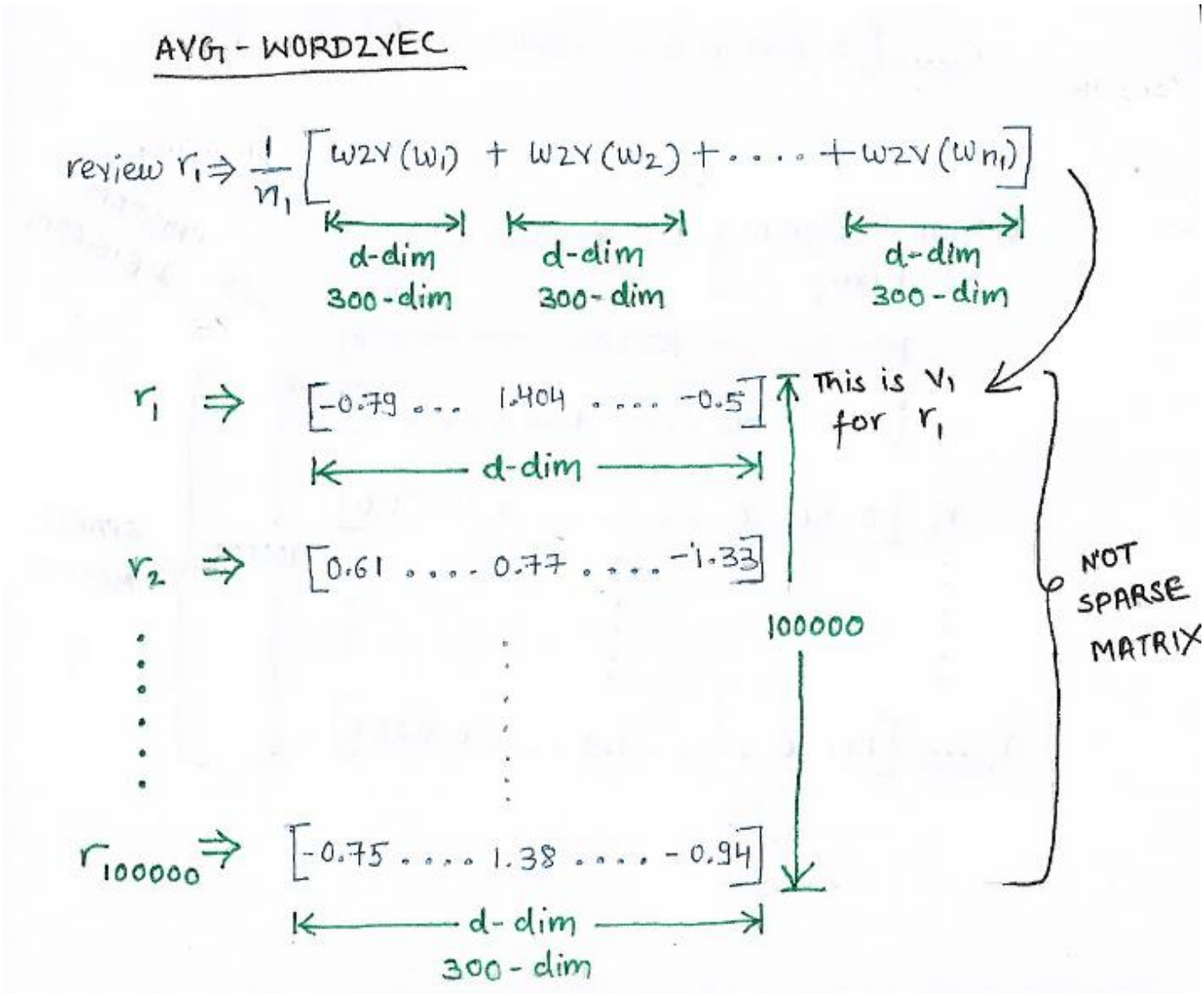
For word2vec we have used already available google_w2v which will convert each word into 300-dimensional vector.
You can download Google's pre-trained mode. It's around 1.5GB! It includes word vectors for a vocabulary of 3 million words and phrases that they trained on roughly 100 billion words from a Google News dataset. The vector length is 300 features.

WORD2VEC

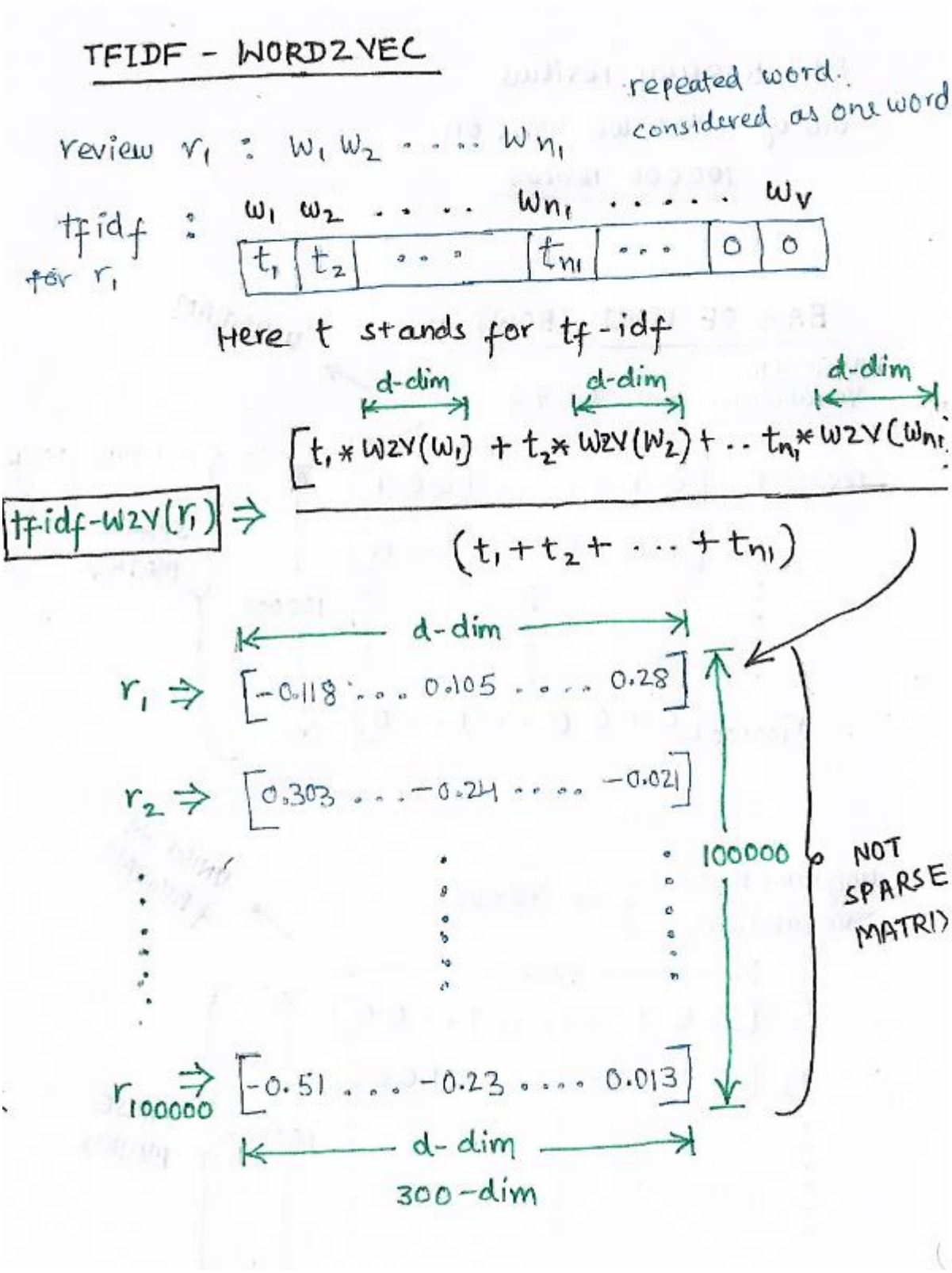
* Total 568454 reviews
* Out of which we work on
  100000 reviews
* We have google_w2v → built on google news data
  By using it we will convert
  Each word → 300 dim vector
* Consider only word which appear more than 5 times

In AVG-W2V for each word of review which appears at least more than 5 times is converted into 300-dimensional vector as explained below and all these vectors of a review are added together and then divided by total number of words for which word2vec has been extracted, same has been explained below as well.
For each review 300-dimensional vector is created and hence 100000 x 300 size matrix is formed which is not sparse

AVG - WORD2VEC



In TFIDF-W2V we will calculate tfidf value for each word in a review and multiply it with corresponding W2V 300-dimension value of that word and then divided by sum of all tfidf values of words in a review.
For each review 300-dimensional vector is created and hence 100000 x 300 size matrix is formed which is not sparse



Once our vectors are ready we can apply Machine learning models on top of it.

References

https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/
https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa