28.1 Dataset Overview: Amazon Fine Food Reviews (EDA)

Important Links

Data Source

https://www.kaggle.com/snap/amazon-fine-food-reviews

Blog on EDA

https://nycdatascience.com/blog/student-works/amazon-fine-foods-visualization/

Ipython Notebook

https://colab.research.google.com/drive/1FCQfzaYb-yyDIuF5RKTFa0VPnsYJZyR8?usp=drive_open

Note

In order to download the dataset from Kaggle, you first have to login to Kaggle and then download it.

Dataset Description

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 - Unique Identifier given to each review
2. ProductId - Unique Identifier for the product
3. UserId - Unique Identifier for the user
4. ProfileName - Customer/User's name
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 at which the review was posted on the website
9. Summary - brief summary of the review
10. Text - text of the review

AppliedRoots (Draft Copy)

Problem Objective

We have to determine if the given review is positive or negative.

How to categorize a given review as positive/negative?

Categorization of the reviews is done on the basis of the 'Score' column values. If a given review has a score value of 4 (or) 5, then it is considered as a Positive review, and if the score value is 1 (or) 2, then it is considered as a Negative Review.

A score value of 3, is considered to be a neutral review, but for the time being, we shall ignore the neutral reviews because, considering the neutral reviews will pose our problem as a multi-class classification. But as we are looking to pose our problem as a binary classification problem, we consider only the positive and the negative reviews.

Loading the Dataset

The dataset is given in two forms.

1) CSV File format (with the name Reviews.csv)
2) SQLite Database

The dataset is already present in the SQLite database, so that it can be directly loaded into a pandas dataframe, and from there we can perform analysis and visualizations on the data easily.

But in case, if you do not want to perform any analysis or visualizations on the data, but just want to have a look at the data, then instead of loading the data into a pandas dataframe every time, you can directly look into the 'Reviews.csv' file.

Below is the line of code that was discussed at the timestamp 18:40, which establishes the connection with the database.

```
con = sqlite3.connect('database.sqlite')
```

Note: If the 'database.sqlite' file is present in the same folder where your ipython notebook is present, then you can pass only the file name as a value to the sqlite3.connect(). If not, then you have to specify the entire path as well.

Below is the line of code that was discussed at the timestamp 19:15, which loads the dataset into a pandas dataframe.

AppliedRoots (Draft Copy)

```
filtered_data = pd.read_sql_query(""" SELECT * FROM Reviews WHERE Score != 3 LIMIT 5000""", con)
```

Note: Here as we are loading the dataset from the database, we are using the method pandas.read_sql_query() and are passing the SQL query and the sqlite connection object. If we had to load the dataset from a CSV file, then we would have used pandas.read_csv() with the path to the CSV file as an argument.

The data in our database is stored in the table 'Reviews', and we are selecting only those rows whose score value is not equal to 3.

The below code snippet was discussed starting from the timestamp 20:33 and here we are relabelling the scores 4 and 5 as 'Positive', and 1 and 2 as 'Negative'.

```
# Give reviews with Score>3 a positive rating, and reviews with a score<3 a negative rating.
def partition(x):
    if x < 3:
        return 0
    return 1

#changing reviews with score less than 3 to be positive and vice-versa
actualScore = filtered_data['Score']
positiveNegative = actualScore.map(partition)
filtered_data['Score'] = positiveNegative
print("Number of data points in our data", filtered_data.shape)
filtered_data.head(3)
```

We are first loading the values of the 'Score' column into a series variable 'actualScore', and then using the map() function, we apply relabelling the values. The code for relabelling is defined in the function partition().

AppliedRoots (Draft Copy)

28.2 Data Cleaning: Deduplication

It is observed that the given dataset had many duplicate entries. Hence it is necessary to remove duplicates in order to get unbiased results for the analysis of the data.

In our data, we are terming two or more reviews as duplicates, if they have the same values for the columns 'UserID', 'ProfileName', 'Time', and 'Text'. Before we delete the duplicate entries, we first have to sort all the reviews on the basis of the 'ProductId' column using pandas.sort_values(). After sorting the reviews, we drop the duplicates using panda.drop_duplicates().

Below is the line of code that was discussed starting from the timestamp 7:00 which sorts all the reviews on the basis of the 'ProductId' column. After sorting the reviews, we are dropping the duplicates and it was discussed at the timestamp

```
#Sorting data according to ProductId in ascending order
sorted_data=filtered_data.sort_values('ProductId', axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last')
```

```
#Deduplication of entries
final=sorted_data.drop_duplicates(subset={"UserId","ProfileName","Time","Text"}, keep='first', inplace=False)
final.shape
```

```
(4986, 10)
```

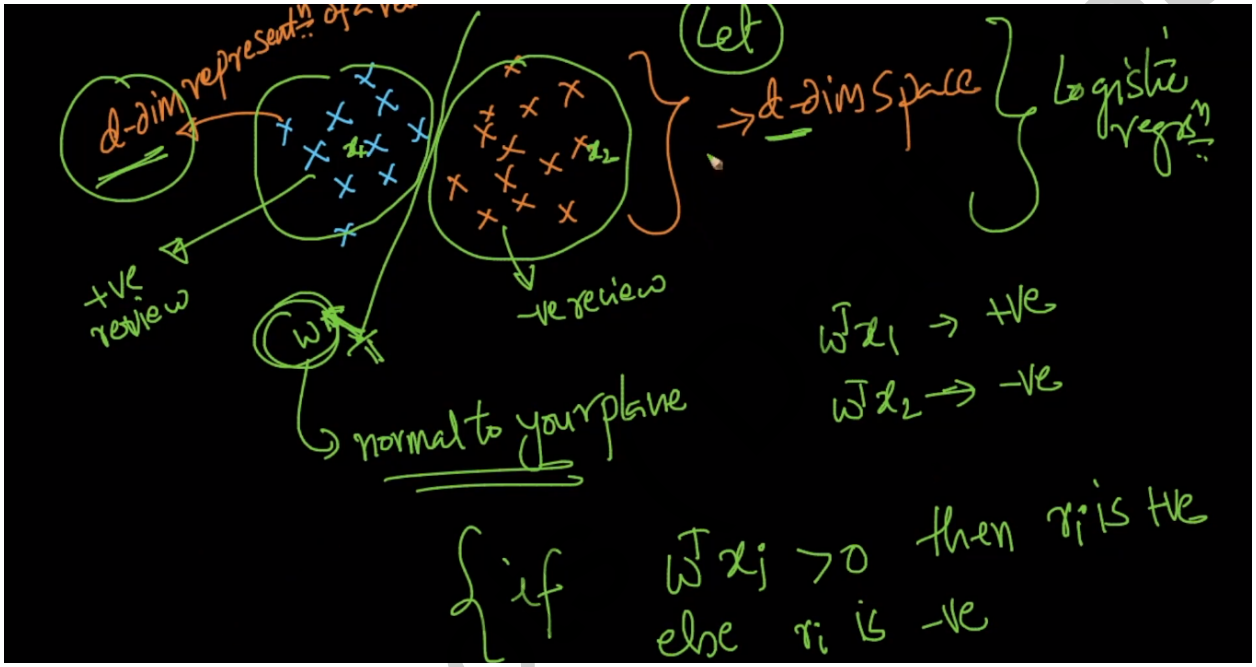
Deduplication is a technique used to improve storage utilization and can also be applied to the network data transfers to reduce the number of bytes that must be read.

AppliedRoots (Draft Copy)

28.3 Why convert a text to a vector?

Given any problem, if we are able to convert the data into vector form, we can leverage the whole power of Linear Algebra.

When we are given the text data, if we could convert it into d-dimensional vector format, and plot those vectors/points in d-dimensional coordinate space, we can find a hyperplane that could separate the points/vectors belonging to different classes.
Below is the representation that was explained starting from the timestamp 3:25.



This is a d-dimensional representation of each vector. Each vector represents a d-dimensional review. Here the vector 'w' is normal to the hyperplane 'π'.
If $w^T.x_1>0$, then we can say that the vector 'w' is in the same direction as that of 'x_1' and.
If $w^T.x_2<0$, then we can say that the vector 'w' is in the direction opposite to that of 'x_2'.

For a given query point 'x_i',

If $w^T.x_i>0$, then the point 'x_i' is classified as positive.

If $w^T.x_i<0$, then the point 'x_i' is classified as negative.

Properties required to convert a text into a d-dimensional vector

If we have 3 vectors 'r_1', 'r_2' and 'r_3' which are semantically similar, then
If similarity(r_1, r_2) > similarity(r_1, r_3), then distance(v_1, v_2) < distance(v_1, v_3)

v_1 → Vector form of the review 'r_1'.

v_2 → Vector form of the review 'r_2'.

It means, if the reviews 'r_1' and 'r_2' are similar, then the vectors 'v_1' and 'v_2' must be close.
If similarity(v_1, v_2) > similarity(v_1, v_3), then length(v_1-v_2)<length(v_1-v_3).

AppliedRoots

28.4 Bag of Words (BOW)

The below techniques are used to convert a text to a vector.

1) Bag of Words (BOW)
2) Term Frequency - Inverse Document Frequency (TF-IDF)
3) Average Word2Vector
4) TF-IDF Weighted Average Word2Vector

Bag of Words (BOW)

Consider the below 4 reviews as an example.

r_1: this pasta is very tasty and affordable
r_2: this pasta is not tasty and is affordable
r_3: this pasta is delicious and cheap
r_4: pasta is tasty and pasta tastes good

Steps in Bag of Words

1) Construct a dictionary (not Python data structure dictionary) which is going to be a set of all the words that are present in all the reviews. For this example, all the unique words present in all the reviews are to be added to a new set. (All these words should be taken only once)
2) Construct a vector for every review. In this problem, we are using the word "review", but the official terminology in NLP is a "**document**".
   For example, we have to construct the vector 'V_1' from the review 'r_1'. Let us assume we have 'n' documents/reviews and the total number of unique words in all the 'n' documents is 'd'.
   We then have to construct a d-dimensional vector for every document. The vector has to be initialized with '0' in all dimensions and each cell is associated with each dimension and has to be filled with the number of times the corresponding word occurs in the given document/review. Each word is considered as a different dimension here.

Let us look at converting the above given 4 reviews into vector form. We'll have the dimensions as shown below.

this	pasta	is	very	tasty	and	affordable	not	delicious	cheap	tastes	good

All these features/words are present across the corpus. Now we shall vectorize the given 4 reviews according to these dimensions.

AppliedRoots (Draft Copy)

r1:

this	pasta	is	very	tasty	and	affordable	not	delicious	cheap	tastes	good
1	1	1	1	1	1	1	0	0	0	0	0

r2:

this	pasta	is	very	tasty	and	affordable	not	delicious	cheap	tastes	good
1	1	2	0	1	1	1	1	0	0	0	0

r3:

this	pasta	is	very	tasty	and	affordable	not	delicious	cheap	tastes	good
1	1	1	0	0	1	0	0	1	1	0	0

r4:

this	pasta	is	very	tasty	and	affordable	not	delicious	cheap	tastes	good
0	2	1	0	1	1	0	0	0	0	1	1

In the above vectors of the reviews, the numbers denote the number of occurrences of that particular word in that review.

Note: The collection of all the documents(here reviews) is called **Corpus**.

The main objective of bag of words is that if documents are more similar semantically, then their corresponding vectors will be closer.

The result of Bag of Words is always a **Sparse Vector**.

Sparse Vector and Dense Vector

A vector is said to be a sparse vector, if most of the dimensions in it have 0 as value.

A vector is said to be a dense vector, if most of the dimensions in it have non zero values.

Note: Let us calculate the length of the difference between two vectors.

length(V_1-V_2) = $||V_1-V_2||$ = sqrt($(2-1)^2 + (0-1)^2 + (1-0)^2$) = sqrt(1 + 1 + 1) = sqrt(3)

Note: One of the disadvantages with the Bag of Words approach is that, even after vectorization, if we observe a difference of a few bits between the two given reviews, but still the semantic meanings could be quite opposite. For example, if we clearly observe the reviews 'r_1' and 'r_2', we see a difference of only two words among them, but their semantic meanings are quite opposite.

Variations of Bag of Words

The so far discussed approach of Bag of Words is also known as **Count Bag of Words**. There is another variation of Bag of Words, which is called **Binary Bag of Words** (or) **Boolean Bag of Words**.

In the Binary/Boolean Bag of Words, unlike the Count Bag of Words, we do not have the count of occurrences of each word in the corpus as a dimension magnitude in

AppliedRoots (Draft Copy)

their vectors. Here it just indicates whether the words in the corpus occur in the documents or not in the form of '1' and '0'.

In the case of the Binary Bag of Words, the distance between the vectors is the square root of the total number of different values in the magnitude of each dimension in both the vectors.

length(V_1-V_2) = ||V_1-V_2|| = sqrt(number of different values in the magnitudes of each dimension)

Note: If there is no duplicate word occurring in a vector form among all the reviews/documents in the corpus, then both Count Bag of Words and Binary Bag of Words will give the same performance and the same result.

When to choose Count Bag of Words and when to choose Binary Bag of Words?

Choosing one among the two is problem specific. In some contexts, the information of presence or absence of a word is sufficient for the model to work well. In such a case, Binary Bag of Words is a good choice.

In other contexts, the count of the number of times a word occurs matters a lot. Here the Count based Bag of Words carries more information than the Binary Bag of Words. So the count Bag of Words typically tends to outperform the Binary Bag of Words.

If we want to determine if a text review is "positive" or "negative", the presence of words like "bad" or "terrific" is good enough to predict the class label. This context is one of the best examples where Binary BOW based encoding can be used.

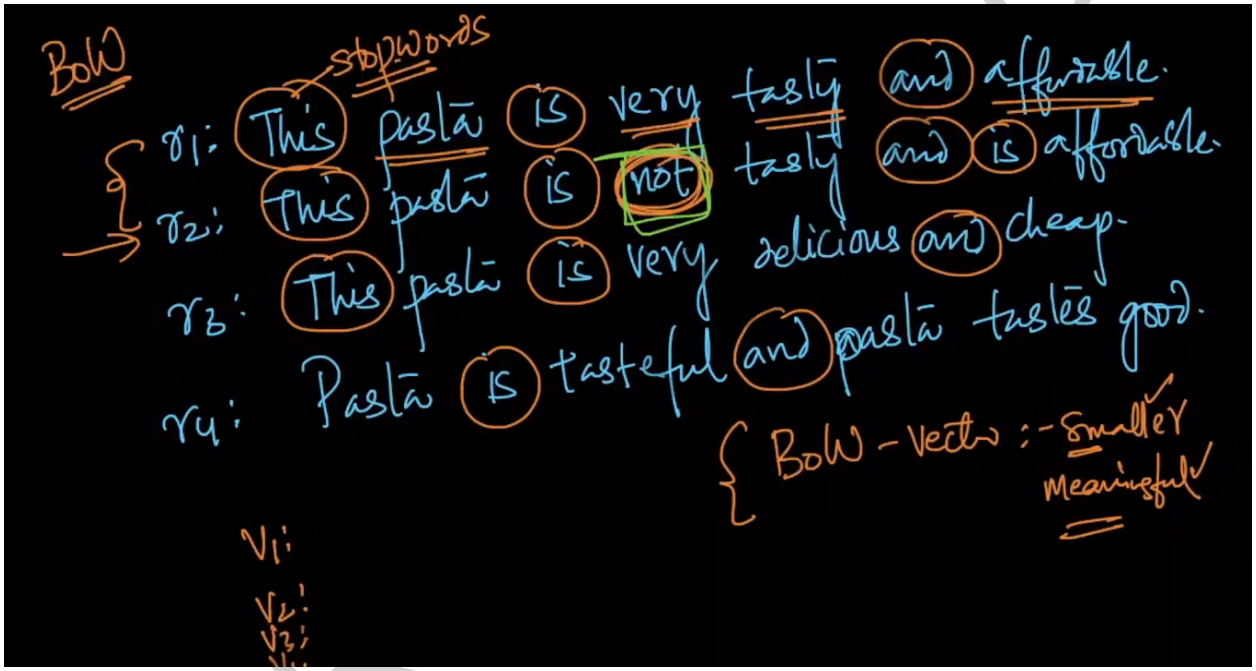
On the other hand, if we have more classes like "Very Positive", "Positive", "Negative", "Very Negative", then the number of times words like "terrific", "bad", "great", "terrible" occur could help us determine the extent of positivity and negativity helping us classify better. This context is one of the best examples where Count BOW based encoding can be used.

AppliedRoots (Draft Copy)

28.5 Text Preprocessing: Stemming, Stop-word Removal, Tokenization, Lemmatization

In a document, there will be certain words which make more sense about the data and there will be certain words which do not make much sense about the data. These words that do not make much sense about the data are present in the document just for sentence completion. Such words are called **Stopwords**.

These stopwords can be removed in order to make the bag of words vector smaller and more meaningful. Removal of stopwords is one of the text preprocessing steps and it has to be performed only when needed.



Steps in Text Preprocessing

1) Stopword Removal
2) Conversion of all words into lowercase
3) Stemming

In Stemming, the words like ‘taste’, ‘tasteful’, ‘tastes’ are reduced to the root form ‘tast’. There are many stemming algorithms in Natural Language Processing (NLP). Two of the majorly used Stemming algorithms are PorterStemmer and SnowballStemmer. SnowballStemmer is much more powerful than PorterStemmer.

AppliedR... Copy)

4) Lemmatization

Lemmatization is the algorithmic process of determining the lemma of a word based on its intended meaning. Unlike stemming, lemmatization depends on correctly identifying the intended part of speech and meaning of a word in a sentence, as well as within the larger context surrounding that sentence, such as neighboring sentences or even an entire document.

Note:

Let us look at the reviews 'r1' and 'r3' given below.

$\mathbf{r_1}$: this pasta is very tasty and affordable

$\mathbf{r_3}$: this pasta is delicious and cheap

These two reviews give the same meaning, but in BOW vectorization, the words 'tasty' and 'delicious' are treated as two different features. Similarly the words 'affordable' and 'cheap' are treated as two different features. This is the main disadvantage with the Bag of Words approach, as it doesn't preserve the semantic meaning.

The semantic meanings of the words are taken into consideration in the **Word2Vec vectorization techniques**. So finally using **Text Preprocessing + Bag of Words**, we are converting the text into a 'd' dimensional vector that could not guarantee the semantic meanings of the words. Bag of words doesn't take the semantic meanings into consideration.

References:

Refer to the below blogs to learn about the differences between Stemming and Lemmatization.

https://blog.bitext.com/what-is-the-difference-between-stemming-and-lemmatization/

https://towardsdatascience.com/stemming-vs-lemmatization-2daddabcb221

Tokenization

Tokenization is the process of splitting a given string into a sequence of sub-strings. There are two types of Tokenization. They are **Word Tokenizer** and **Sentence Tokenizer**.

The Word Tokenizer splits the given sentence into a sequence of words, on the basis of space whereas the Sentence Tokenizer splits the given sentence into a sequence of words/sentences on the basis of dot(.).

Example: "Hello Mr.Rajeev, How are you doing today?

Word Tokenizer Output:

["Hello", "Mr.", Rajeev", "How", "are", "you", "doing", "today"]

Sentence Tokenizer Output:

["Hello Mr", "Rajeev How are you doing today?"]

AppliedRoots (Draft Copy)

28.6 Uni-gram, Bi-gram and n-grams

Let us assume we have two reviews 'r1' and 'r2'.

r_1: this pasta is very tasty and affordable

r_2: this pasta is not tasty and is affordable

After removing the underlined stopwords, the reviews become

r_1: pasta tasty affordable

r_2: pasta tasty affordable

Now both 'r_1' an 'r_2' have become exactly the same and the distance between their corresponding vectors 'v_1' and 'v_2' is zero, as they both are the same. But here, now we are forced to conclude that both the reviews are similar. But these two reviews, in their original form(ie., before removing the stopwords) are quite opposite. So we shouldn't go with false conclusions. In order to solve this type of problem, we need bi-grams, tri-grams, n-grams, etc.

Let us consider the reviews in their original form again.

r_1: this pasta is very tasty and affordable

r_2: this pasta is not tasty and is affordable

Uni-grams

We have a d-dimensional vector for all the words in the corpus. The presence of each word is indicated by non zero values in each dimension.

	this	pasta	is	very	tasty	and	affordable	not
r1	1	1	1	1	1	1	1	0
r2	1	1	1	0	1	1	1	1

Bi-grams

Here we create a vector for each review, with a pair of words as each dimension and these words will be the consecutive words of both the reviews.

	this pasta	pasta is	is very	very tasty	tasty and	and affordable	is not	not tasty	and is	is affordable
r1	1	1	1	1	1	1	0	0	0	0
r2	1	1	0	0	1	0	1	1	1	1

Tri-grams

Each dimension is obtained by taking 3 consecutive words at a time.

	this pasta is	pasta is very	is very tasty	very tasty and	tasty and affordable	pasta is not	is not tasty	not tasty and	tasty and is	and is affordable
r1	1	1	1	1	1	0	0	0	0	0
r2	1	0	0	0	0	1	1	1	1	1

AppliedRoots (Draft Copy)

28.7 TF-IDF (Term Frequency - Inverse Document Frequency)

Term Frequency (TF)

Let us assume we have the words 'w_1', 'w_2', 'w_3', 'w_4', 'w_5' and 'w_6' and there are 'N' documents in the corpus.

Term Frequency(w_i, r_j) = (Number of times 'w_i' occurs in 'r_j')/(Total Number of Words in 'r_j')

Let us assume we have 2 reviews 'r_1' and 'r_2' and the words in them be

r_1: w_1 w_2 w_3 w_2 w_5

r_2: w_1 w_3 w_4 w_2 w_6 w_5

So $TF(w_2,r_1) = 2/5$

$TF(w_2,r_2) = 1/6$

The term frequency of any word in general lies in between 0 and 1 (inclusive). So as this value lies in between 0 and 1, we can interpret it as probability. So $TF(w_i, r_j)$ can also be called as probability of occurrence of the word 'w_i' in 'r_j'.

Inverse Document Frequency (IDF)

Let $D_c \rightarrow$ Data of corpus $\rightarrow \{r_1, r_2,.....,r_n\}$

Inverse Document Frequency of a word is defined over the corpus, but not over a document.

$IDF(wi,D_c) = \log_e$(Total Number of Documents(N)/Total Number of reviews containing 'w_i')

So $IDF(w_i,D_c) = \log_e(N/n_i)$

We know that $n_i <= N$, so $N/n_i >= 1$.

So it means $\log_e(N/n_i) >= 0$

If 'n_i' increases, 'N/n_i' decreases and ultimately $\log_e(N/n_i)$ decreases.

$\log_e(N/n_i)$ is a monotonically decreasing function in 'n_i'. The more the word 'w_i' across the reviews in the corpus, the lesser is the IDF score.

If 'w_i' is more frequent, $IDF(w_i,D_c)$ will be low.

If 'w_i' is a rare word, $IDF(w_i,D_c)$ will be more.

Vector Creation for each document using TF-IDF

In vector creation using TF-IDF, the magnitude of each dimension 'w_i' is given by

Magnitude of 'w_i' = $TF(w_i, r_i)$*$IDF(w_i, D_c)$

In a nutshell, we are giving more priority to the words which occur most frequently (whose TF value is high), and the words which occur rarely (whose IDF value is very low). By this, we can maximize the value of the product TF*IDF.

Applied Roots (Draft Copy)

Drawbacks of TF-IDF

One of the drawbacks of TF-IDF is, it also doesn't take the semantic meaning into consideration. (ie., words like (cheap, affordable), (tasty, delicious), (valuable, precious) are considered as separate dimensions)

Difference between BOW and TF-IDF

In both BOW and TF-IDF, we convert each of the reviews into a d-dimensional vector where 'd' is the number of unique words in the total text across all the reviews.

The key difference between BOW and TF-IDF is that instead of using frequency counts as the values in the d-dimensional vector for each word as in BOW, we use the TF-IDF score for the words in TF-IDF vector representation.

AppliedRoots (Draft Copy)

28.8 Why do we use 'log' in the IDF?

The Inverse Document Frequency of a word 'w_i' across the document corpus 'D_c' is given as

$$IDF(w_i, D_c) = \log(N/n_i)$$

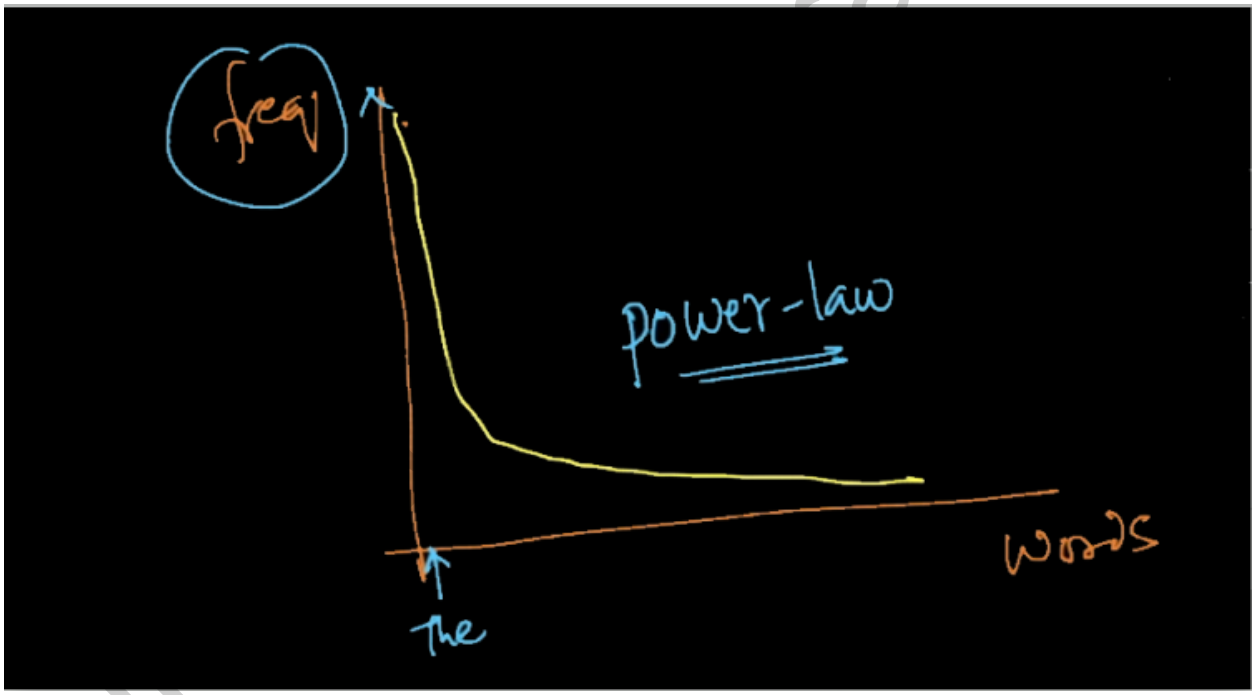
Where N → Total number of documents in the corpus

n_i → Total number of documents containing the word 'w_i'

In order to know why we use logarithm in the IDF, we shall go through Zipf's law first.

Zipf's Law

As mentioned in the video, starting from the timestamp 6:35, if we have all the words on the 'X' axis and the frequency of occurrence of each word on 'Y' axis, then if we plot a histogram, the it looks like below

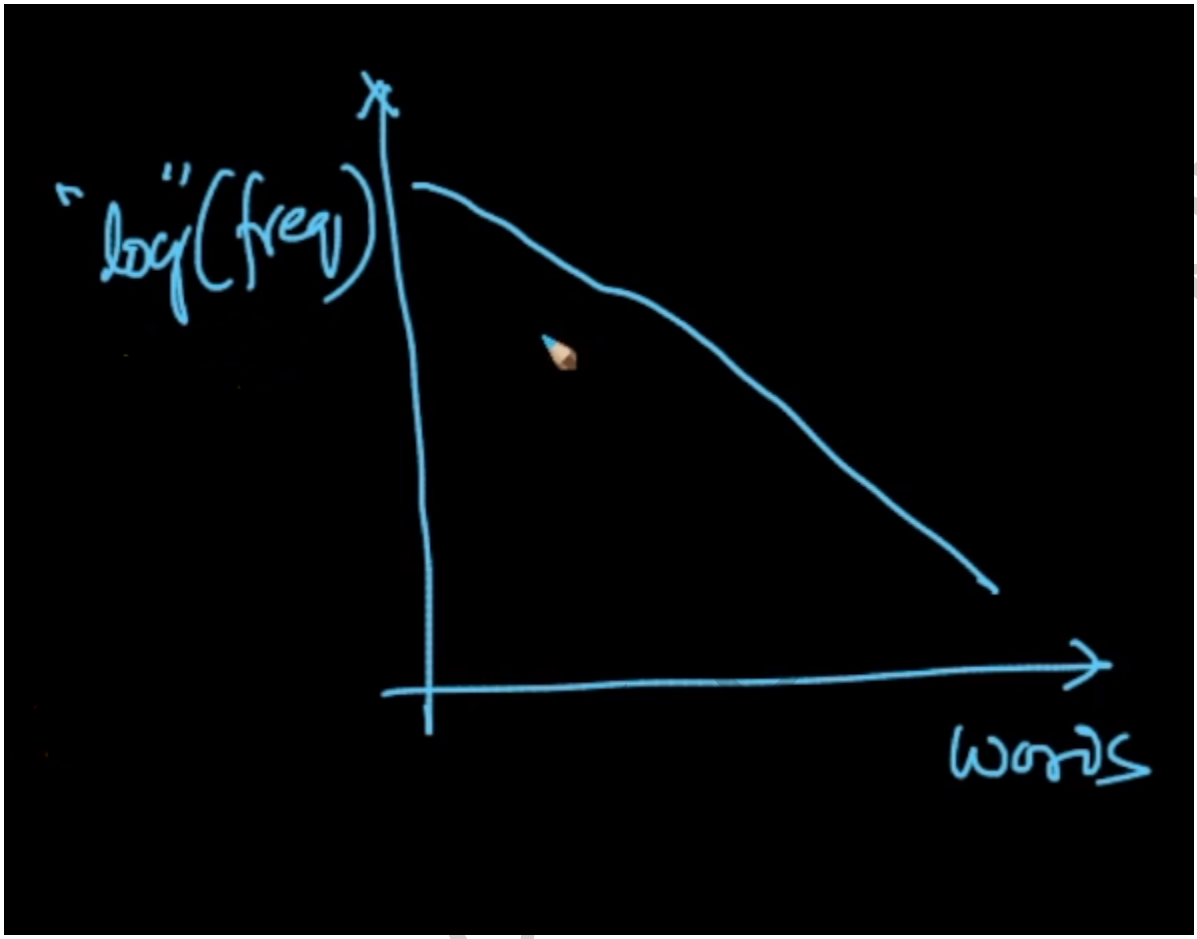


In the above plot, the most frequently occurring words are present towards the origin and the rare words are present away from the origin on the 'X' axis. The curve is in the decreasing order of the frequency. This is an example of Power Law.

If we have a random variable 'X' which follows Power Law, then we can convert it into a gaussian distribution by applying a box-cox transform. Also from the definition of Power Law, we also know that if random variables 'X' and 'Y' follow the power law, then the plot of log(X) vs log(Y) will be a straight line. Similarly, even if one feature(ie.,

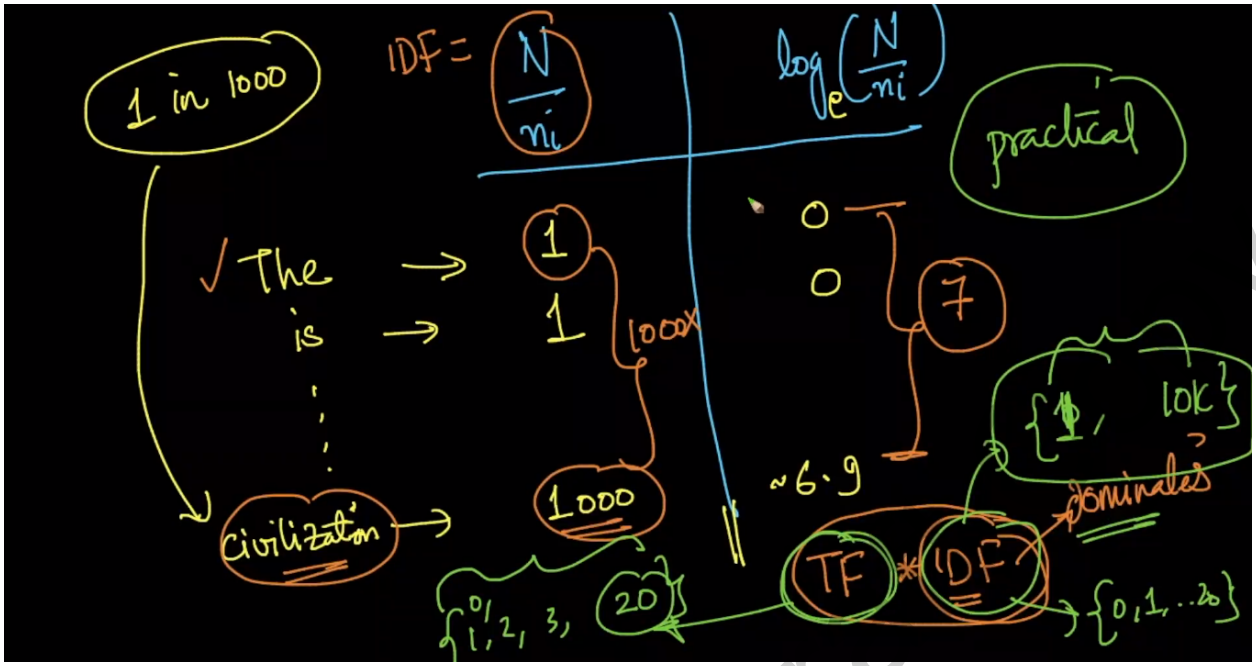
Appl... Draft Copy)

frequency) follows power law while the other feature(words) is discrete, then if we apply logarithm to the frequency, the curve gets transformed into a line as shown below.



Let us now apply logarithm to the values of (N/n_i) and check how the transformed values look like, as discussed in the video starting from the timestamp 8:00

AppliedRo



The range of (N/n_i) is 1 to 10000, whereas the range of $\log(N/n_i)$ is 1 to 7. So if we take (N/n_i), the range is very high, the IDF will dominate TF and result in a huge value. Hence we choose $\log(N/n_i)$ to get reasonable values. We apply logarithm to reduce the scale.

AppliedRoots (Dr

28.9 Word2Vec

We have seen the BOW and TF-IDF techniques for converting a text into a vector. But these techniques do not take the semantic meanings into consideration, whereas Word2Vec is a state of the art technique used to convert the text into a vector, and also takes the semantic meaning into consideration.

So far in BOW/TF-IDF, we have seen a text is given as an input and the output is a sparse vector. But Word2Vec takes a word as an input and gives a d-dimensional vector as an output which is dense.

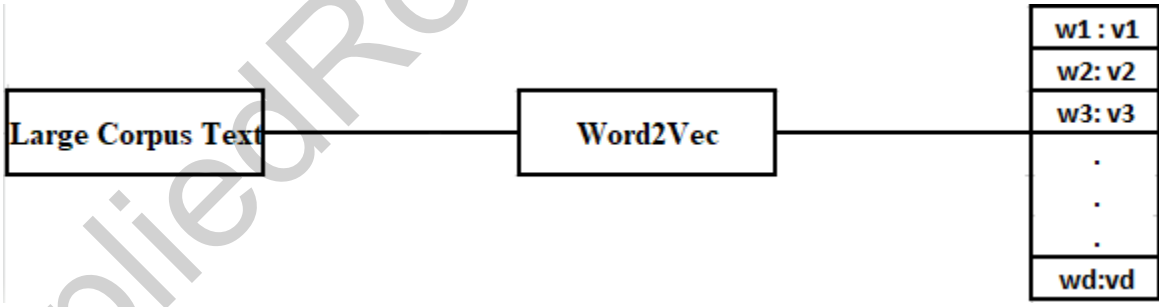
If the words 'w_1' and 'w_2' are semantically similar, then their vectors 'v_1' and 'v_2' are closer. This is the principle, Word2Vec tries to achieve. Word2Vec also tries to satisfy the relationships between the words.

For example, if we have the words 'w_1', 'w_2', 'w_3' and 'w_4' as

w_1 = "man"
w_2 = "woman"
w_3 = "king"
w_4 = "queen"

Here the vector difference (V_{man} - V_{woman}) is parallel to (V_{king}-V_{queen}). This is the relationship it holds. Here it is a male-female relationship. Similarly, it holds country-capital, country-currency relationships, etc. It also holds verb-tense relationships. All these relationships are learnt by Word2Vec automatically from the raw text, without being explicitly programmed.

Top Level View of how Word2Vec works



Here a large corpus text is given as an input to Word2Vec and then Word2Vec creates a vector for each word. These vectors are high-dimensional. The more the dimensions we have in our vectors, the more rich the information is going to be. In order to get as many dimensions in the vectors, we need to give as much large corpus text as input.

AppliedRoots (Draft Copy)

In the core nutshell, for every word, the Word2Vec checks for the neighborhood of that word and if the neighborhood of this word is similar to the neighborhood of other words, then the vector of this word and other words is similar.

If we have two words 'w_i' and 'w_j', then if the neighborhood of the word 'w_i' is the same as the neighborhood of 'w_j', then the vectors of 'wi' and 'wj' are similar.

So far in BOW and TF-IDF we have converted documents into vectors. In Word2Vec, we are converting each word of the corpus into a vector.

Note:

Word2Vec could not give the correct results for stemmed words. For example, for words like 'tasti', we do not get appropriate results as the stemmed words are not present in the text directly. So we should not perform stemming on the data if we want to go for Word2Vec.

AppliedRoots (Draft Copy)

28.10 Avg Word2Vec, TF-IDF Weighted Word2Vec

Average Word2Vec

Let us assume we have a document/review 'r_1'. Let its corresponding average vector form be denoted as 'v_1'. Let the total number of words in 'r_1' be 'n_1'.

r_1: w_1 w_2 w_1 w_3 w_4 w_5

So now the average word2vec for 'r_1' is

$$\mathbf{v_1 = (1/n_1)[w2v(w_1) + w2v(w_2) + w2v(w_1) + w2v(w_3) + w2v(w_4) + w2v(w_5)]}$$

$w2v(w_i)$ represents the word vector of the word 'w_i'. This is called the Average Word2Vec representation of 'r1'. Average Word2Vec fairly works well in practice, but is not perfect all the time. It still works well enough.

Average Word2Vec is a simple way to leverage Word2Vector to build sentence vectors.

TF-IDF Weighted Word2Vec

Let us assume we have 7 words (say 'w_1', 'w_2', 'w_3', 'w_4', 'w_5', 'w_6', 'w_7') and the review/document 'r1'.

r_1: w_1 w_2 w_1 w_3 w_4 w_5

The TF-IDF representation of the 'r1' vector is given as below

w1	w2	w3	w4	w5	w6	w7
t1	t2	t3	t4	t5	t6=0	t7=0

Here $t_i \rightarrow$ TF-IDF(w_i,r_i)

Now, let 'v_1' be the TF-IDF Weighted Word2Vec representation of the vector 'r_1', then

$$\mathbf{v_1 = (t_1*w2v(w_1) + t_2*w2v(w_2) + t_3*w2v(w_3) + t_4*w2v(w_4) + t_5*w2v(w_5))/(t_1 + t_2 + t_3 + t_4 + t_5)}$$

It can simply be written as

$$\mathbf{TFIDF\ Weighted\ Word2Vec\ (ri) = \Sigma_{i=1}^{n} (t_i*w2v(w_i))/(\Sigma_{i=1}^{n} t_i)}$$

Note - Special Case:

If t_i=1(ie., $t_1 = t_2 = t_3 = \ldots\ldots = 1$), then the TF-IDF Weighted Word2Vec is the Average Word2Vec.

Average Word2Vec and TF-IDF Weighted Word2Vec are two simple weighting strategies to convert sentences into vectors. They both serve the same purpose. TF-IDF Weighted Word2Vec weights each word differently as compared to Average Word2Vector. In practice, we try both the options and choose the one that performs better at our task.

AppliedRoots (Draft Copy)