**Feature Extraction Techniques for Natural Language Processing**

Bag of Words

D1: This movie is very scary and long.

D2: This movie is not scary and is slow.

D3: This movie is spooky and good.

1. We first lower the cases and remove the stop words.

Output:

D1: movie scary long

D2: movie not scary slow

D3: movie spooky good

1. We build the vocabulary from all the unique words. We rank each vocabulary based on their frequency on all the documents.

|  |  |
| --- | --- |
| Vocabulary | Frequency |
| movie | 3 |
| scary | 2 |
| long | 1 |
| not | 1 |
| slow | 1 |
| spooky | 1 |
| good | 1 |

1. We build our features. The index of features is based on the frequency.

Feature 1: movie

Feature 2: scary

Feature 3: long

Feature 4: not

Feature 5: slow

Feature 6: spooky

Feature 7: good

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | movie | scary | long | not | slow | spooky | good |
| D1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| D2 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| D3 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |

If we have one more ‘long’ in D1, such as ‘long movie scary long’, then the value of index ‘long’ at D1 will be 2.

But in Bag of Words, we have an option to make it as a BINARY Bag of Words. We can make that 2 as 1, to represent only either presence (1) or absence (0).

In the above example, each word is considered as one feature but there are some problems with this model. For example, we split D2, and the words ‘not’ and ‘scary’ are now separated and therefore lost their combined semantic meaning.

Here, we can use the idea of **n-grams** and apply it to our Bag of Words.

With n-grams, the ordering matters because we specify how do we want to group consecutive words in a document.

Example:

Bigrams = (2, 2); which means by two

Trigrams = (3, 3); which means by three

(1, 2); which means individually and by two

(1, 3); which means individually, by two, and by three

1. Implementing n**-**grams **(1,2).**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | movie | scary | long | not | slow | spooky | good | movie scary | scary long | movie  not | not scary | scary slow | movie spooky | spooky good |
| D1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| D2 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| D3 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

TF-IDF

D1: This movie is very scary and long.

D2: This movie is not scary and is slow.

D3: This movie is spooky and good.

1. We first lower the cases and remove the stop words.

Output:

D1: movie scary long

D2: movie not scary slow

D3: movie spooky good

Vocabularies: ‘movie’, ‘scary’, ‘long’, ‘not’, ‘slow’, ‘spooky’, ‘good’

1. We calculate the **term-frequency** of each word for each document.

Example:

Calculating the term-frequency for D2:

No. of words in D2: 4

TF for the word ‘movie’ = (number of times ‘movie’ appears in D2) / (number of terms in D2) = ¼

Similarly, we can calculate the term frequencies for all the terms and all the reviews in this manner:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Term** | D1 | D2 | D3 | TF (D1) | TF (D2) | TF (D3) |
| movie | 1 | 1 | 1 | 1/3 | 1/4 | 1/3 |
| scary | 1 | 1 | 0 | 1/3 | 1/4 | 0 |
| long | 1 | 0 | 0 | 1/3 | 0 | 0 |
| not | 0 | 1 | 0 | 0 | 1/4 | 0 |
| slow | 0 | 1 | 0 | 0 | 1/4 | 0 |
| spooky | 0 | 0 | 1 | 0 | 0 | 1/3 |
| good | 0 | 0 | 1 | 0 | 0 | 1/3 |

1. We calculate the **IDF** values for all the words.

Example:

IDF(‘movie’) = ln (number of documents / number of documents containing the word ‘movie’)

= ln (3 / 3) = ln(1) = 0

Thus, the IDF values for the entire vocabulary would be:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Term** | D1 | D2 | D3 | IDF |
| movie | 1 | 1 | 1 | 0 |
| scary | 1 | 1 | 0 | 0.41 |
| long | 1 | 0 | 0 | 1.1 |
| not | 0 | 1 | 0 | 1.1 |
| slow | 0 | 1 | 0 | 1.1 |
| spooky | 0 | 0 | 1 | 1.1 |
| good | 0 | 0 | 1 | 1.1 |

1. We calculate the **TF-IDF** for every vocabulary with respect to all the documents.

Example:

Calculating the TF-IDF score for ‘movie’ in D2:

TF-IDF (‘movie’, D2) = TF (‘movie’, D2) \* IDF(‘movie’) = ¼ \* 0 = 0

Similarly, we can calculate the TF-IDF scores for all the words with respect to all documents:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Term** | D1 | D2 | D3 | IDF | TFIDF(D1) | TFIDF(D2) | TFIDF(D3) |
| movie | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| scary | 1 | 1 | 0 | 0.41 | 0.14 | 0.10 | 0 |
| long | 1 | 0 | 0 | 1.1 | 0.37 | 0 | 0 |
| not | 0 | 1 | 0 | 1.1 | 0 | 0.28 | 0 |
| slow | 0 | 1 | 0 | 1.1 | 0 | 0.28 | 0 |
| spooky | 0 | 0 | 1 | 1.1 | 0 | 0 | 0.37 |
| good | 0 | 0 | 1 | 1.1 | 0 | 0 | 0.37 |

We have now obtained the TF-IDF scores for our vocabulary. TF-IDF also gives larger values for less frequent words and is high when both TF and IDF values are high i.e., the word is rare in all documents (low DF) combined but frequent in a single document (high TF).

Word Embeddings

Words are represented using features, to create a pre-determined size of feature vectors.

Advantages:

1. Maintains the semantic meaning of words (similar words are closer to each other in their vector form in terms of their location in space)
2. Solves the issue of sparse matrix (too many 0’s) created by Bag of Words and TF-IDF and instead create a **dense** matrix.

D1: This movie is very scary and long.

D2: This movie is not scary and is slow.

D3: This movie is spooky and good.

1. We first lower the cases and remove the stop words.

Output:

D1: movie scary long

D2: movie not scary slow

D3: movie spooky good

Vocabularies: ‘movie’, ‘scary’, ‘long’, ‘not’, ‘slow’, ‘spooky’, ‘good’

1. **Feature representation**. We generate new features according to the dimension we want, i.e., **5**. These features are generated using the Word2vec algorithm pre-trained from billions of words and cannot be exactly identified.

We will create vectors for our vocabulary using these features.

We assign values to our vocabulary based on the feature to create a set of vectors. (In reality, it will be the **trained model** to assign these values)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Vocabulary | | | | | | |
| **New Features** | movie | scary | long | not | slow | spooky | good |
| feeling | 0.3 | 0.9 | 0.1 | 0.1 | 0.1 | 0.87 | 0.6 |
| conjunction | 0.05 | 0.02 | 0.03 | 0.94 | 0.01 | 0.03 | 0.10 |
| adjective | 0.04 | 0.88 | 0.86 | 0.12 | 0.78 | 0.90 | 0.89 |
| distance | 0.01 | 0.1 | 0.99 | 0.04 | 0.003 | 0.08 | 0.001 |
| film | 0.96 | 0.41 | 0.45 | 0.03 | 0.36 | 0.47 | 0.5 |

The vector formed from the features for each word is called **embedding matrix.**

Example:

movie = [0.3, 0.05, 0.04, 0.01, 0.96]

Whenever vocabularies are related semantically, they will have similar vectors, i.e., scary and spooky.

scary = [0.9, 0.02, 0.88, 0.1, 0.41]

spooky = [0.87, 0.03, 0.90, 0.08, 0.47]

Similar vectors mean that they are closer to each other in space. This distance is computed either through Euclidean distance or Cosine similarity.

Cosine similarity takes the angle between two vectors, and the distance between them is equal to 1 – Cosine similarity. In conclusion, the **larger** the cosine similarity, the **closer** the two vectors are.

Word2vec

Vectors being created from features are of limited dimensions, i.e., 100, 200, 300, etc.

**Word2Vec Architecture**

1. Continuous Bag of Words (CBOW)

Example: D1: [Data Science is for all walks of life]

Lower the cases.

If we have more than one sentence which is always the case, we concatenate each sentence.

First, we need to determine the **window size. Window size** is the size of the subset of our corpus. i.e., window size = 5. Keep the window size an odd number, so the left and right context will be the same.

“Larger windows tend to capture more topic/domain information: what other words (of any type) are used in related discussions? Smaller windows tend to capture more about the word itself: what other words are functionally similar? (Their own extension, the dependency-based embeddings, seems best at finding most-similar words, synonyms or obvious-alternatives that could drop-in as replacements of the origin word.)”

* "Dependency-Based Word Embeddings", Levy & Goldberg

The middle word is called the **output** or **target**. For a window size of 5, the target is **‘is’**.

The **context** are the words around it:

‘Data’, ‘Science’ on its left, and

‘for’, ‘all’ on its right.

|  |  |
| --- | --- |
| Input Feature | Output Feature |
| ‘data’, ‘science’, ‘for’, ‘all’ | ‘is’ |

We slide the window:

|  |  |
| --- | --- |
| Input Feature | Output Feature |
| ‘data’, ‘science’, ‘for’, ‘all’ | ‘is’ |
| ‘science’, ‘is’, ‘all’, ‘walks’ | ‘for’ |
| ‘is’, ‘for’, ‘walks’, ‘of’ | ‘all’ |
| ‘for’, ‘all’, ‘of’, ‘life’ | ‘walks’ |

Creating bag-of-words vector for each vocabulary.

Example:

data = [1 0 0 0 0 0 0 0]

science = [0 1 0 0 0 0 0 0]

We pass the bag-of-words for each vocabulary as the input feature and the output feature to a shallow **Fully Connected Layer**, or a simple Artificial Neural Network.

Each input feature has a size of 4 vocabularies. And each vocabulary is represented by a one hot vector of dimension 7.

**Input Layer:**

4 vectors with 7 neurons each = 1 layer with 28 neurons

**Hidden Layer:**

1 vector with 5 neurons (5 because 5 is our window size)

**Output/ Softmax Layer:**

1 vector with 7 neurons (because we have one output, and it is a one hot vector of 7 dimensions)

The output layer is a **softmax layer** which is used to sum the probabilities obtained in the output layer to 1.

How is the model getting trained?

Based on the loss function (*y* – *yhat*), weights will be updated until we minimize the loss.

The **embedding layer** is the output of the hidden layer, and it contains the weights that are to be connected to get the output layer.

We slide the window, until all words have become a target. The weight matrix is updated with each iteration.

Once done, we take the matrix of the weights.

The window size is a hyperparameter, the larger the window size the better the model will be.

To get the word embeddings of the whole corpus, we can take the onehot encoding of the whole corpus and multiply it with the **transpose** of the weight matrix.

1. Skipgram

The process is the same as Continuous Bag-of-words, but the difference is that the input feature is the output feature for CBOW, and the **target** feature is the input feature of CBOW.

|  |  |
| --- | --- |
| Input Feature | Output Feature |
| ‘is’ | ‘data’, ‘science’, ‘for’, ‘all’ |
| ‘for’ | ‘science’, ‘is’, ‘all’, ‘walks’ |
| ‘all’ | ‘is’, ‘for’, ‘walks’, ‘of’ |
| ‘walks’ | ‘for’, ‘all’, ‘of’, ‘life’ |