Sentiment Analysis

The basic problem statement was to classify sentences into different sentiments. Sentiment analysis is a well studied subject so far as of 2018 and the accuracies achieved on the problem statement using different feature sets and different machine learning models are impressive.

However, for this specific dataset, the problem becomes a bit too complicated due to the number of categories, which is 18. This increases the diversity of the problem and we predicted that machine learning models would fail to deliver impressive accuracies. Generally, machine learning models perform best on 2-6 categories of data for sentiment analysis. So, for this problem, two future improvements can be done, which we are unable to do as of the moment:

`1. Find a larger dataset and computers with larger processing capabilities so that the 18 categories have enough features in each of them to train ML models.

2. Reduce the categories. That can be done by finding more generic classes and merging emotions currently present in the dataset like 'love' and 'like'.

Dataset

The dataset contains 27731 rows of statements in a csv file. We used a pandas data frame to take the csv file as input.

The dataset contains data of 18 different sentiments, as mentioned earlier..

The sentiments are given below:

['Love', 'Like ', 'Consciousness ', 'Protestant ', 'Smiley ', 'Angry ', 'Blush', 'Skip ', 'Rocking ', 'Fail ', 'Shocking ', 'WOW', 'Bad ', 'HaHa', 'Sad ', 'Skeptical ', 'Evil ', 'Provocative ']

An example is given below. জীবেনর িনরাপা ও জীবেনর মল ু ২ই বশী; বাংলােদেশ গর মল ু বাধ হয় মানষু থেক বশী হওয়ার কথা।;HaHa(হা হা)

After plotting the count values of the number of features per category, we found the data set to have multiple other problems:

1. It is imbalanced.

2. It is highly skewed.

3. it is borderline meaningless. There is no consistency whatsoever.

Regardless of all the issues, we decided to see how different machine learning models performed for their own feature sets.

**Feature Sets Used:**

1. TF-IDF

2. Count Vectors

3. Continuous Bag of Words

**Machine Learning Models**

The models we chose to use on this problem, the feature sets we used for them, and their accuracies are given in the table below.

|  |  |  |
| --- | --- | --- |
| Model | Feature | Accuracy |
| Naive Bayes Classifier | TF-IDF | 36.6% |
| Support Vector Machine | TF-IDF | 41.3% |
| Decision Tree | Count Vectors | 48% |
| K Nearest Neighbour | Count Vectors | 46.4% |
| Flat Neural Network | Continuous Bag of Words | 49.7% |

Document Categorization

The task is categorize an article to an appropriate topic. Dataset The dataset was collected from https://scdnlab.com/corpus/ .

The dataset contains articles of 12 categories in 12 different folders. The number of articles each folder contains is given below. 1. Accident : 6350 articles 2. Art : 2669 articles 3. Crime : 8840 articles 4. Economics : 5351 articles 5. Education : 12389 articles 6. Entertainment : 10139 articles 7. Environment : 6852 articles 8. International : 5922 articles 9. Opinion : 8116 articles 10.Politics : 20479 articles 11.Science\_tec : 2906 articles 12.Sports : 12086 articles. The dataset was slightly better than that of sentiment analysis.

**Feature Engineering:**

The features we used for document categorization remain the same as described in the next section of this report. They are:

1. Count Vectors

2. TF-IDF

3. Continuous Bag of wards

**Machine Learning Models:**

The models we used also remain the same and their accuracies based on their feature sets are given below:

|  |  |  |
| --- | --- | --- |
| Model | Feature | Accuracy |
| Naive Bayes Classifier | TF-IDF | 80% |
| Support Vector Machine | TF-IDF | 71.3% |
| Decision Tree | Count Vectors | 65% |
| K Nearest Neighbour | Count Vectors | 57.4% |
| Flat Neural Network | Continuous Bag of Words | 73.6% |

Authorship Attribution

The task is categorize an article based on the original author who wrote it.

For this problem, we chose to select the best data set we could find.

We collected newspaper articles from 3 authors: Imon Jubayer, Hasan Mahbub and MZI.

There were about 200 documents per author.

A pandas data frame was used to organize and analyse the data. The data seemed sufficiently balanced and very well labelled.

**Feature Engineering:**

The features we used for document categorization remain the same as described in the next section of this report. They are:

1. Count Vectors

2. TF-IDF

3. Continuous Bag of wards

**Machine Learning Models:**

The models we used also remain the same and their accuracies based on their feature sets are given below:

|  |  |  |
| --- | --- | --- |
| Model | Feature | Accuracy |
| Naive Bayes Classifier | TF-IDF | 85.3% |
| Support Vector Machine | TF-IDF | 93.7% |
| Decision Tree | Count Vectors | 82.9% |
| K Nearest Neighbour | Count Vectors | 76.4% |
| Flat Neural Network | Continuous Bag of Words | 95.5% |

Conclusion:

From running the models on different datasets of text classification, we can see that categories and diversity affects performance the most. The balance of datasets and the adequateness also plays a huge role in accuracy.

Feature Engineering:

We basically extracted 3 different features, all based on word embeddings, to train our models on, taking one of the three features for each model.

What are word embeddings?

In very simplistic terms, Word Embeddings are the texts converted into numbers and there may be different numerical representations of the same text. But before we dive into the details of Word Embeddings, the following question should be asked – Why do we need Word Embeddings?

As it turns out, many Machine Learning algorithms and almost all Deep Learning Architectures are incapable of processing strings or plain text in their raw form. They require numbers as inputs to perform any sort of job, be it classification, regression etc. in broad terms. And with the huge amount of data that is present in the text format, it is imperative to extract knowledge out of it and build applications. Some real world applications of text applications are – sentiment analysis of reviews by Amazon etc., document or news classification or clustering by Google etc.

Let us now define Word Embeddings formally. A Word Embedding format generally tries to map a word using a dictionary to a vector. Let us break this sentence down into finer details to have a clear view.

Take a look at this example – **sentence**=” Word Embeddings are Word converted into numbers ”

A word in this **sentence** may be “Embeddings” or “numbers ” etc.

A dictionary may be the list of all unique words in the **sentence.**So, a dictionary may look like – [‘Word’,’Embeddings’,’are’,’Converted’,’into’,’numbers’]

A vector representation of a word may be a one-hot encoded vector where 1 stands for the position where the word exists and 0 everywhere else. The vector representation of “numbers”in this format according to the above dictionary is [0,0,0,0,0,1] and of converted is[0,0,0,1,0,0].

This is just a very simple method to represent a word in the vector form. Let us look at different types of Word Embeddings or Word Vectors and their advantages and disadvantages over the rest.

**1. TF-IDF**

This is another method which is based on the frequency method but it is different to the count vectorization in the sense that it takes into account not just the occurrence of a word in a single document but in the entire corpus. So, what is the rationale behind this? Let us try to understand.

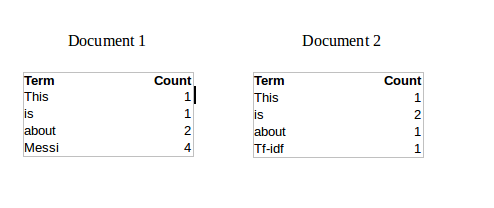
Common words like ‘is’, ‘the’, ‘a’ etc. tend to appear quite frequently in comparison to the words which are important to a document. For example, a document **A** on Lionel Messi is going to contain more occurences of the word “Messi” in comparison to other documents. But common words like “the” etc. are also going to be present in higher frequency in almost every document.

Ideally, what we would want is to down weight the common words occurring in almost all documents and give more importance to words that appear in a subset of documents.

TF-IDF works by penalising these common words by assigning them lower weights while giving importance to words like Messi in a particular document.

So, how exactly does TF-IDF work?

Consider the below sample table which gives the count of terms(tokens/words) in two documents.



Now, let us define a few terms related to TF-IDF.

TF = (Number of times term t appears in a document)/(Number of terms in the document)

So, TF(This,Document1) = 1/8

TF(This, Document2)=1/5

It denotes the contribution of the word to the document i.e words relevant to the document should be frequent. eg: A document about Messi should contain the word ‘Messi’ in large number.

IDF = log(N/n), where, N is the number of documents and n is the number of documents a term t has appeared in.

where N is the number of documents and n is the number of documents a term t has appeared in.

So, IDF(This) = log(2/2) = 0.

So, how do we explain the reasoning behind IDF? Ideally, if a word has appeared in all the document, then probably that word is not relevant to a particular document. But if it has appeared in a subset of documents then probably the word is of some relevance to the documents it is present in.

Let us compute IDF for the word ‘Messi’.

IDF(Messi) = log(2/1) = 0.301.

Now, let us compare the TF-IDF for a common word ‘This’ and a word ‘Messi’ which seems to be of relevance to Document 1.

TF-IDF(This,Document1) = (1/8) \* (0) = 0

TF-IDF(This, Document2) = (1/5) \* (0) = 0

TF-IDF(Messi, Document1) = (4/8)\*0.301 = 0.15

As, you can see for Document1 , TF-IDF method heavily penalises the word ‘This’ but assigns greater weight to ‘Messi’. So, this may be understood as ‘Messi’ is an important word for Document1 from the context of the entire corpus.

**2. Count Vectors**

Consider a Corpus C of D documents {d1,d2…..dD} and N unique tokens extracted out of the corpus C. The N tokens will form our dictionary and the size of the Count Vector matrix M will be given by D X N. Each row in the matrix M contains the frequency of tokens in document D(i).

Let us understand this using a simple example.

D1: He is a lazy boy. She is also lazy.

D2: Neeraj is a lazy person.

The dictionary created may be a list of unique tokens(words) in the corpus =[‘He’,’She’,’lazy’,’boy’,’Neeraj’,’person’]

Here, D=2, N=6

The count matrix M of size 2 X 6 will be represented as –

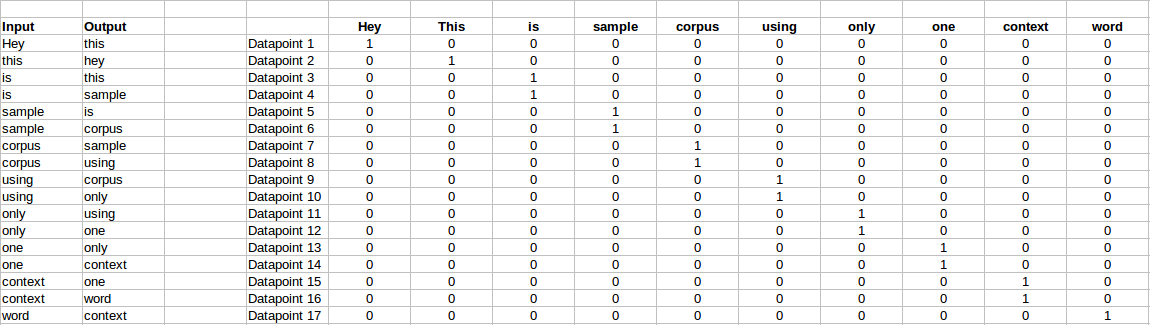
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | He | She | lazy | boy | Neeraj | person |
| D1 | 1 | 1 | 2 | 1 | 0 | 0 |
| D2 | 0 | 0 | 1 | 0 | 1 | 1 |

Now, a column can also be understood as word vector for the corresponding word in the matrix M. For example, the word vector for ‘lazy’ in the above matrix is [2,1] and so on.Here, the *rows* correspond to the *documents* in the corpus and the *columns* correspond to the *tokens* in the dictionary. The second row in the above matrix may be read as – D2 contains ‘lazy’: once, ‘Neeraj’: once and ‘person’ once.

**3. Continuous Bag of Words**

The way CBOW work is that it tends to predict the probability of a word given a context. A context may be a single word or a group of words. But for simplicity, I will take a single context word and try to predict a single target word.

Suppose, we have a corpus C = “Hey, this is sample corpus using only one context word.” and we have defined a context window of 1. This corpus may be converted into a training set for a CBOW model as follow. The input is shown below. The matrix on the right in the below image contains the one-hot encoded from of the input on the left.

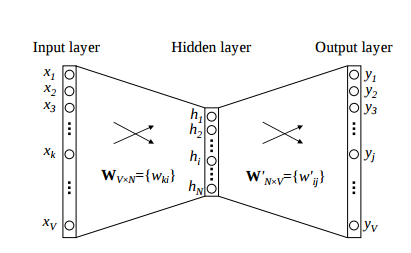


The target for a single datapoint say Datapoint 4 is shown as below

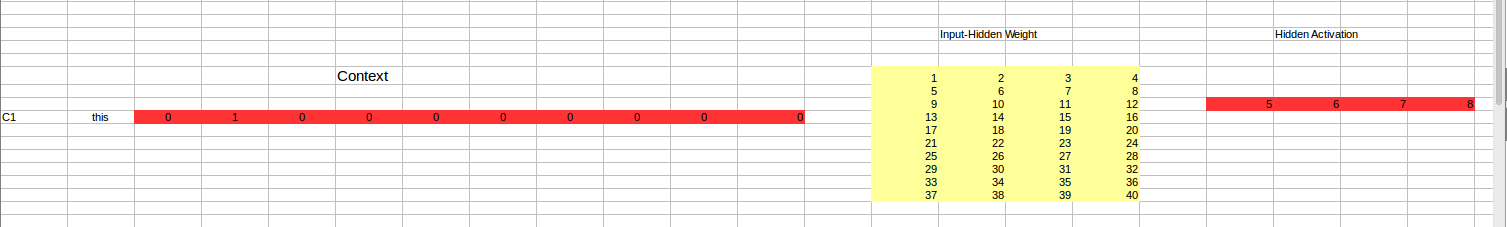
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Hey | this | is | sample | corpus | using | only | one | context | word |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

This matrix shown in the above image is sent into a shallow neural network with three layers: an input layer, a hidden layer and an output layer. The output layer is a softmax layer which is used to sum the probabilities obtained in the output layer to 1. Now let us see how the forward propagation will work to calculate the hidden layer activation.

Let us first see a diagrammatic representation of the CBOW model.



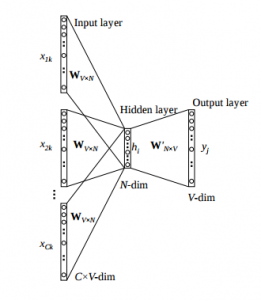
The matrix representation of the above image for a single data point is below.



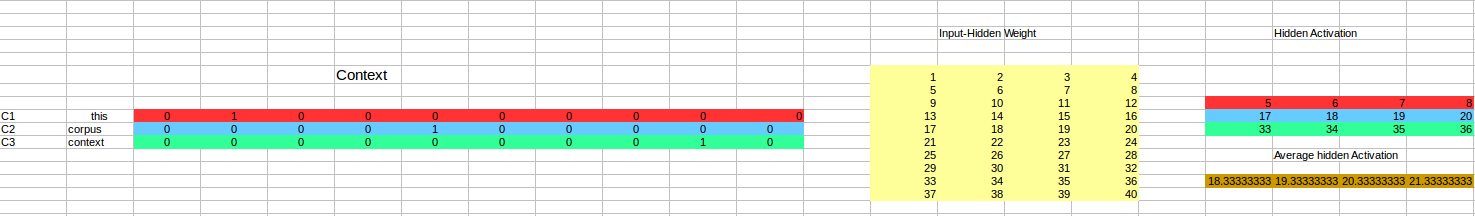
The flow is as follows:

1. The input layer and the target, both are one- hot encoded of size [1 X V]. Here V=10 in the above example.
2. There are two sets of weights. one is between the input and the hidden layer and second between hidden and output layer.  
   Input-Hidden layer matrix size =[V X N] , hidden-Output layer matrix  size =[N X V] : Where N is the number of dimensions we choose to represent our word in. It is arbitary and a hyper-parameter for a Neural Network. Also, N is the number of neurons in the hidden layer. Here, N=4.
3. There is a no activation function between any layers.( More specifically, I am referring to linear activation)
4. The input is multiplied by the input-hidden weights and called hidden activation. It is simply the corresponding row in the input-hidden matrix copied.
5. The hidden input gets multiplied by hidden- output weights and output is calculated.
6. Error between output and target is calculated and propagated back to re-adjust the weights.
7. The weight  between the hidden layer and the output layer is taken as the word vector representation of the word.

We saw the above steps for a single context word. Now, what about if we have multiple context words? The image below describes the architecture for multiple context words.



Below is a matrix representation of the above architecture for an easy understanding.



The image above takes 3 context words and predicts the probability of a target word. The input can be assumed as taking three one-hot encoded vectors in the input layer as shown above in red, blue and green.

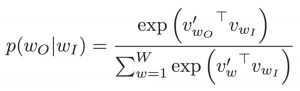
So, the input layer will have 3 [1 X V] Vectors in the input as shown above and 1 [1 X V] in the output layer. Rest of the architecture is same as for a 1-context CBOW.

The steps remain the same, only the calculation of hidden activation changes. Instead of just copying the corresponding rows of the input-hidden weight matrix to the hidden layer, an average is taken over all the corresponding rows of the matrix. We can understand this with the above figure. The average vector calculated becomes the hidden activation. So, if we have three context words for a single target word, we will have three initial hidden activations which are then averaged element-wise to obtain the final activation.

In both a single context word and multiple context word, I have shown the images till the calculation of the hidden activations since this is the part where CBOW differs from a simple MLP network. The steps after the calculation of hidden layer are same as that of the MLP as mentioned in this article – [Understanding and Coding Neural Networks from scratch](https://www.analyticsvidhya.com/blog/2017/05/neural-network-from-scratch-in-python-and-r/).

The differences between MLP and CBOW are  mentioned below for clarification:

1. The objective function in MLP is a MSE(mean square error) whereas in CBOW it is negative log likelihood of a word given a set of context i.e -log(p(wo/wi)), where p(wo/wi) is given as



wo : output word  
wi: context words

2. The gradient of error with respect to hidden-output weights and input-hidden weights are different since MLP has  sigmoid activations(generally) but CBOW has linear activations. The method however to calculate the gradient is same as an MLP.

In-depth details of models used along with their tuning:

**Naive Bayes:**

We experimented with two models of the Naive Bayes algorithm, the Gaussian model and the Multinomial Model, at which point , the MultiNomialNB gave us the most impressive results.

We got the best accuracies with the cross fold validation training system, where the used a 5 times CV.

We left the other parameters for default. Since the low quality dataset is the bottleneck for this problem, we did not go into further parameter tuning as it would be a waste of time.

**Support Vector Machine:**

For a support vector machine, we kept things simple and still managed to get impressive results.

We used a linear kernel since it gave us impressive enough results. As this is a multiclass problem, we kept the class weights balanced. The gamma decay we used was 0.01 and a C value of 100 gave us the best accuracies. Further tuning was performed, however this specific set provided us with the most stable results.

For the SVM, we used 20% data as the test set.

**Decision Tree:**

We kept the parameters of the decision tree as defaults, as they seemed to suit our problem set the most.

The train-test split was of 20%.

**K Nearest Neighbour:**

By changing the parameters, we did not get any better results. We left the tuning as defaults.

Train test split was again 20%.

**Neural Network:**

We did most of the model modification for neural networks, as they seemed to affect the performance the most.

For a flat neural network, we took the continuous bag of words as features.

We used the nltk tokenizer to tokenize the words and create vectors. Words with similar semantic meanings are closer to each other in the vector space.

We padded the sequenced to the maximum a balanced length of 2000 words, as neural networks take a set length of dimensions for features.

The dimension of the embedding layer size was 128.

We flattened the embedding layer.

We added a single dense layer with the activation function of relu.

For the output layer, we used a softmax activation function as it is a multi category problem.

The adam optimizer gave us the best results, so we kept things simple.

The loss was categorical crossentropy, obviously, due to more than two categories in the problem set.

A train-test-validation split of 60:40:40 was performed.

We did run 15 epochs, however we saved only the weights where our validation accuracy was higher than the previous epochs. This solved the problem of overfitting.

The neural net gave us the most impressive accuracies.