NLTK

How to remove additional stopwords

stopset = set(stopwords.words('english')) - set(('over', 'under', 'below', 'more', 'most', 'no', 'not', 'only', 'such', 'few', 'so', 'too', 'very', 'just', 'any', 'once'))

Models:

1. NB

classifier = NaiveBayesClassifier.train(trainfeats)

1. Generalized itrtactive scaling GIS algo

classifier = MaxentClassifier.train(trainfeats, 'GIS', trace=0, encoding=None, labels=None, sparse=True, gaussian\_prior\_sigma=0, max\_iter = 1)

1. SVM
   1. classifier = SklearnClassifier(LinearSVC(), sparse=False)
   2. classifier.train(trainfeats)

Packages

spaCy is the best way to prepare text for deep learning.

Noisy corpus refers to unimportant entities of the text such as punctuations marks, numerical values, links and urls etc. Removal of these entities from the text would increase the accuracy, because size of sample space of possible features set decreases.

Removing URLS from text:

**from** **bs4** **import** BeautifulSoup

tweet = BeautifulSoup(tweet, "lxml").get\_text()

***Lemmatization***

good practice to normalize the terms to their root forms. This technique is known as Lemmatization. For example, the words:

* Playing
* Player
* Plays
* Play
* Players
* Played

All can be normalized down to the word “**Play**” as far as the classifier is concerned

**from** **nltk.stem** **import** WordNetLemmatizer

1. Stemming and Lemmatization are two different processes.:  
Stemming is much simpler process in which, suffixes are removed from a word. It involves various if – else rules and conditions, to convert the word to a root form. Lemmatization on the other hand, is more advanced approach, which converts the word into its lemma. It takes care of grammar, vocabulary and dictionary importance of a word while the conversion. for example: in Stemming:  
“driving” will be reduced to “driv” but in lemmatization it will be reduced to “drive”

2. As per my experience with text mining, stemming is not a good tool for enhancing the performance. As stemming results in loss of some information.

3. Point 4 and TF-IDF are somewhat similar in practical sense. IDF also does the same thing by giving less priority to low frequency words.

Various types of features

1. Bag of words(CountVectorizer): segment each text file into words (for English splitting by space), and count # of times each word occurs in each document and finally assign each word an integer id.

from sklearn.feature\_extraction.text import CountVectorizer  
count\_vect = CountVectorizer()  
X\_train\_counts = count\_vect.fit\_transform(twenty\_train.data)  
X\_train\_counts.shape

We get a document term matrix

1. Term Frequency: will give more weightage to longer documents than shorter documents. To avoid this, we can use frequency (TF - Term Frequencies) i.e. #count(word) / #Total words, in each document.
2. TF-IDF: Finally, we can even reduce the weightage of more common words like (the, is, an etc.) which occurs in all document. This is called as TF-IDF i.e Term Frequency times inverse document frequency.

from sklearn.feature\_extraction.text import TfidfTransformer  
tfidf\_transformer = TfidfTransformer()  
X\_train\_tfidf = tfidf\_transformer.fit\_transform(X\_train\_counts)  
X\_train\_tfidf.shape

1. Stemming: stemming is the process of reducing inflected (or sometimes derived) words to their word stem, base or root form. E.g. A stemming algorithm reduces the words “fishing”, “fished”, and “fisher” to the root word, “fish”.

from nltk.stem.snowball import SnowballStemmer  
stemmer = SnowballStemmer("english", ignore\_stopwords=True)

* CountVectorizer
  + Creates a matrix with frequency counts of each word in the text corpus
* TF-IDF Vectorizer
  + TF - Term Frequency -- Count of the words(Terms) in the text corpus (same of Count Vect)
  + IDF - Inverse Document Frequency -- Penalizes words that are too frequent. We can think of this as regularization
* HashingVectorizer
  + Creates a hashmap(word to number mapping based on hashing technique) instead of a dictionary for vocabulary
  + This enables it to be more scalable and faster for larger text coprus
  + Can be parallelized across multiple threads

<https://www.kaggle.com/c/spooky-author-identification>

<https://www.kaggle.com/c/tradeshift-text-classification>

<https://www.kaggle.com/c/text-normalization-challenge-english-language>

<https://www.kaggle.com/c/sentiment-analysis-on-movie-reviews>

#Sentense count in each comment:

# '\n' can be used to count the number of sentenses in each comment

df['count\_sent']=df["comment\_text"].apply(lambda x: len(re.findall("**\n**",str(x)))+1)

#Word count in each comment:

df['count\_word']=df["comment\_text"].apply(lambda x: len(str(x).split()))

#Unique word count

df['count\_unique\_word']=df["comment\_text"].apply(lambda x: len(set(str(x).split())))

#Letter count

df['count\_letters']=df["comment\_text"].apply(lambda x: len(str(x)))

#punctuation count

df["count\_punctuations"] =df["comment\_text"].apply(lambda x: len([c for c **in** str(x) if c **in** string.punctuation]))

#upper case words count

df["count\_words\_upper"] = df["comment\_text"].apply(lambda x: len([w for w **in** str(x).split() if w.isupper()]))

#title case words count

df["count\_words\_title"] = df["comment\_text"].apply(lambda x: len([w for w **in** str(x).split() if w.istitle()]))

#Number of stopwords

df["count\_stopwords"] = df["comment\_text"].apply(lambda x: len([w for w **in** str(x).lower().split() if w **in** eng\_stopwords]))

#Average length of the words

df["mean\_word\_len"] = df["comment\_text"].apply(lambda x: np.mean([len(w) for w **in** str(x).split()]))

Word Embeddings:

Word Embedding is representation words in a numerical format. Usually 100 to 500 numneric values considered to represent a word in word Embedding

These word's vector can be used in multiple NLP taskes like finding similarity between two words and also can be useud in classification task

**word2vec:**

It creates word to numeric vector by learning large text corpus using Deep Learning

#### Tokenizer

Its Keras class in text preprocessing especially used for text tokenization and getting token level information using multiple methods

#### Methods

* fit\_on\_texts : It creates index to all tokens which can be seen using word\_index property
* texts\_to\_sequences : It maps the index to popular words to the given text from fit\_on\_texts

<https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/>

Word Embedding :creating a representation for words that capture their *meanings*, *semantic relationships* and the different types of contexts they are used in.---“  numerical representations of texts so that computers may handle them.”

1. Frequency based Embedding
2. Prediction based Embedding

Frequency based Embedding:

1. Count Vector
2. TF-IDF Vector
3. Co-Occurrence Vector

Count Vecotr:

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.

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]**

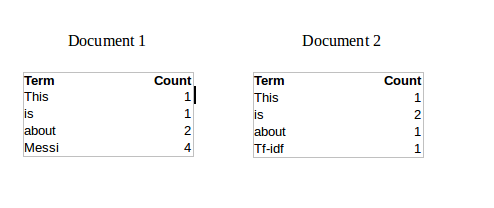
We may either take the frequency (number of times a word has appeared in the document) or the presence(has the word appeared in the document?) to be the entry in the count matrix M. But generally, frequency method is preferred over the latter.

TF\_IDF:

it takes into account not just the occurrence of a word in a single document but in the entire corpus.

“down weight the common words occurring in almost all documents and give more importance to words that appear in a subset of documents.”

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.  Like MESSI

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.

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

***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.***

Prediction based vector:

word2vec to the NLP community. These methods were prediction based in the sense that they provided probabilities to the words and proved to be state of the art for tasks like word analogies and word similarities

Word2vec is not a single algorithm but a combination of two techniques – CBOW(Continuous bag of words) and Skip-gram model. Both of these are shallow neural networks which map word(s) to the target variable which is also a word(s). Both of these techniques learn weights which act as word vector representations.

how to place words on a “chart” in such a way that their location is determined by their meaning. This means that words with similar meanings will be clustered together. This represents the intuitive part of my opening example - words with semantic relationships with them will be closer together than words without such relationships.

Word2vec takes as its input a large corpus of text and produces a [vector space](https://en.wikipedia.org/wiki/Vector_space), typically of several hundred [dimensions](https://en.wikipedia.org/wiki/Dimensions), with each unique word in the [corpus](https://en.wikipedia.org/wiki/Corpus_linguistics) being assigned a corresponding vector in the space. [Word vectors](https://en.wikipedia.org/wiki/Word_vectors) are positioned in the vector space such that words that share common contexts in the corpus are located in close proximity to one another in the space

<https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/>

Word Embeddings:

1. Frequency based embeddings
   1. Count vector
   2. TF\_IDF(Not just the occurrence of word in single document but in entire corp)
   3. Co-ocurrence vector-The big idea – Similar words tend to occur together and will have similar context

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

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

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

1. Prediction based embeddings

Each sentence is tokenized to words, vectors for these words can be found using glove embeddings and then take the average of all these vectors. This technique has performed decently, but this is not a very accurate approach as it does not take care of the order of words.

To overcome this:

**[Infersent](https://github.com/facebookresearch/InferSent" \t "_blank)**, it is a *sentence embeddings* method that provides semantic sentence representations. It is trained on natural language inference data and generalizes well to many different tasks.

*Create a vocabulary from the training data and use this vocabulary to train infersent model. Once the model is trained, provide sentence as input to the encoder function which will return a 4096-dimensional vector irrespective of the number of words in the sentence*

Infersent can help in various downstream task like finding similarity between sentences

Context🡪 Paragraphs🡪Sentences (Textblob/Spacy)🡪Vector representation of a) each sentence b)question 🡪Create features like distance for each sentence-question pair  
Return sentence from para which has minimum distance from given question

<https://towardsdatascience.com/using-word2vec-for-better-embeddings-of-categorical-features-de75020e1233>

Embeddings don’t capture intuition. And might lead to overfitting

if you have different tasks on similar data, you can use the embeddings from one task in order to improve your results on another🡪 Transfer learning

How to check if embedding make sense:

Crude way: take embedding of several items and check its neighbors, if they are similar in domain then it makes sense

Good way: Reduce the dimensionality of vectors using PCA or t-SNE

Word2vec: words that are semantically similar will have similar vectors

Continuous Bag of words and Skip gram:

CBOW trains a network to predict a word from context

Word2vec🡪 Combination of CBOW and Skip gram and are shallow NN which map word(s) to target variable which is also a word(s)

from nltk.stem.wordnet import WordNetLemmatizer

lem = WordNetLemmatizer()

from nltk.stem.porter import PorterStemmer

stem = PorterStemmer()

While in deep learning we usually just normalize the data (e.g., such that image pixels have zero mean and unit variance), in traditional machine learning we need handcrafted features to build accurate models. Doing feature engineering is both art and science, and requires iterative experiments and domain knowledge. Feature engineering boils down to feature selection and creation.

* Removing features with low variance
* Univariate feature selection
* Recursive feature elimination
* Selecting from model

from sklearn.feature\_selection import SelectFromModel

from sklearn.ensemble import RandomForestClassifier

X\_train **=** **...** *# your training features*

y\_train **=** **...** *# your training labels*

*# can be any estimator that has attribute 'feature\_importances\_' or 'coef\_'*

model **=** RandomForestClassifier(random\_state**=**0)

model**.**fit(X\_train, y\_train)

fs **=** SelectFromModel(model, prefit**=**True)

X\_train\_new **=** fs**.**transform(X\_train) *# columns selected*

In this example, features with zero importance (feature\_importances\_ = 0) will be eliminated.

[hyperopt](https://github.com/hyperopt/hyperopt), a Python library for serial and parallel parameter optimization.

<https://machinelearningmastery.com/discover-feature-engineering-how-to-engineer-features-and-how-to-get-good-at-it/>

In computer vision, an image is an observation, but a feature could be a line in the image. In natural language processing, a document or a tweet could be an observation, and a phrase or word count could be a feature. In speech recognition, an utterance could be an observation, but a feature might be a single word or phoneme.

Feature Selection:-> Regularization methods like LASSO and ridge regression may also be considered algorithms with feature selection baked in, as they actively seek to remove or discount the contribution of features as part of the model building process

1. Decompose categorical attributes
2. Decompose date time
3. Aggregate or expose temporal structure(raw variable)

how specific temporal and other non-linearities in the problem structure were reduced to simple composite binary indicators

instability of Lasso when dealing with highly correlated features, you should either consider combining the L1 penalty with L2 (the compound penalty is called Elastic Net) which will globally squash the coefficients but avoid randomly zeroing one out of 2 highly correlated relevant features

1. Filter methods: statistical test to determine if a feature is statistically significant. does not take into account feature interactions and is generally not a very recommended way of doing feature selection as it can lead to lost in information.
2. Wrapper methods: using a learning algorithm to report the optimal subset of features. Tree based models like RandomForest are also robust against issues like multi-collinearity, missing values, outliers etc as well as being able to discover some interactions between features. However this can be rather computationally expensive.
3. Embedded methods: involves carrying out feature selection and model tuning at the same time . Some methods include greedy algorithms like forward and backwards selection as well as Lasso(L1) and Elastic Net(L1+L2) based models

Why Feature Selection:

<https://www.quora.com/How-do-I-perform-feature-selection>

To reduce the cost in speed and memory to run the classifier and to compensate for the curse of dimensionality.