Comparing Feature Vectors to do Sentiment Analysis for Short Text

Mitch Kinney

December 14, 2017

Introduction

For my end of the semester project I will explore a new model to do sentiment analysis on short texts using methods reviewed in the class CSCI 8636. During class we were introduced to different ways to do classification such as kernel Principal Component Analysis (kPCA) and Support Vector Machines (SVM). Both of these methods were reviewed in context of first having feature vectors generated from the data to classify. I will create the feature vectors as numerical representations of tweets using a generator I've been exploring in my own research called doc2vec [5] which is able to convert documents of words to numerical vectors. The authors claim it is able to capture the context of a document which is helpful in tasks such as sentiment analysis. In this report I will compare other methods to create feature vectors such as Bag-of-Words (BOW) and dimension expansion techniques versus doc2vec. The dataset I will be using is Stanford's Sentiment140 dataset [3] which is a collection of 1.6 million tweets that are pre-labeled with a positive or negative sentiment. My main goal is to compare these feature vectors by using SVM to classify to see if doc2vec is truly superior to other methods when doing sentiment analysis on tweets.

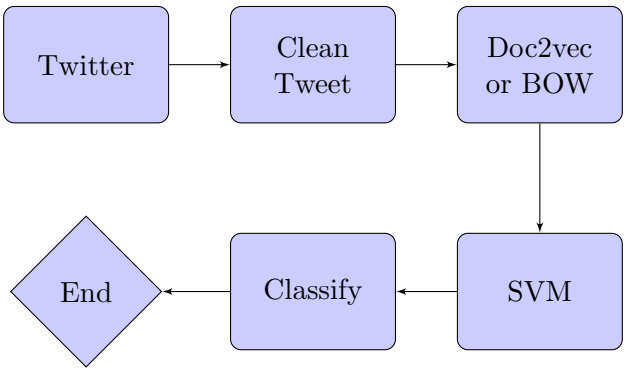

Figure 1: Flow diagram of project

Previous Work

In the last 20 years or so there has been a lot of work on analyzing text data thanks to the increased computing power we have access to and because of the interest to train mathematical models that can capture information about text that is easily discernible by humans. Sentiment in short texts is one of these problems that researchers have attempted to tackle. Many of the methods utilize a combination of techniques that can be broken down into two parts: first generate a numerical representation of the text data and second use existing model machinery to build a classifier. The generator of the numerical representation will usually be an unsupervised method and then the labels will be used in training and testing the classifier. This allows for a lot of speculation on if the success of these combinations rely more heavily on the generator or the classifier or if there is some dependence on the combination itself (if methods compliment each other for example). Since I am doing an analysis on only one dataset I am unable to explore this open question in my paper so I will focus on comparing the success of the classifier in this specific situation. There are a multitude of options in how to combine generators and classifiers when doing sentiment analysis which has been the subject of many other papers.

One example of a combination comes from Dos Santos and Gatti [7] in which they did sentiment analysis on the same tweets dataset that I will use. They utilized word2vec to generate numeric representations of the words in each tweet and then used a Convolutional Neural Network (CNN) to classify. Word2vec [5] is a algorithm designed by researchers at Google DeepMind that is able to capture the semantic relationship between words which I will go into more depth later on. The author's method was able to achieve state of the art performance (at the time) classifying correctly at an $\sim$85% accuracy rate. In their CNN the authors took advantage of moving context windows and different character capitalization to capture the sentiment of the tweets.

Another example of a proposed combination was detailed in Go et al. [3] where they suggested to use a Bag-of-Words approach in combination with a Naive Bayes estimator. This paper was published in connection to the Sentiment140 dataset I will be using too. BOW is an old and simple approach to generate a numerical representation of text by creating a matrix of ones and zeros with the number of columns equal to the number of unique words (or number of unique words of interest) in all sentences and number of rows equal to the number of sentences. There will be a one in the i^{th}, j^{th} cell if word j is in sentence i. Otherwise there will be a zero. The choice of a classifier is a multinomial Naive Bayes model which trains itself based on if certain words are in the tweets. With this method they were able to achieve a classification accuracy of ∼81%. This showed that simple models can be effective in sentiment analysis.

A final example comes from Minier and Csato [10] who used a clustered Bag-of-Words approach along with kPCA to do sentiment analysis on text documents. Finding the amount of words found in a Bag-of-Words analysis to be challenging the authors only took words that appeared many times in their sample of documents and then clustered them based on whether they were associated more times with a positive or negative document. Then to do kPCA the authors solved the eigen value and vector problem on the kernel matrix between different clustered words where the authors used three different kernels. To do testing they assigned the new documents to the nearest neighbor in the training data. This paper was written in 2007 and they claimed they were able to get similar results as other procedures done using their datasets. I hope to do something similar in my work but a bit simpler.

Generator Overview

The generators that I chose to create a numerical representation of the text are called doc2vec and Bag-of-Words. Docwvec is an algorithm designed to capture the context of a document. According to the authors Le and Mikolov [5] doc2vec is specifically designed to predict the next word in a document given the surrounding words which is why it is able to build a numerical representation that provides context. Doc2vec was built off of an earlier designed algorithm by Mikolov et al. [6] called word2vec which was able to accomplish a similar task for words. In the paper for word2vec [6] the authors claim that the algorithm can capture the semantic relationships between words implying that words used in similar situations will have similar numerical representations. I will talk about both methods and how they can offer an advantage in sentiment analysis tasks. Bag-of-Words is an old method that looks at the entire vocabulary of words present in the dataset and indicates for each document if a word is present or not. It's a technique that can be useful in sentiment analysis tasks as certain words are more associated with a positive or negative sentiment.

Word2vec

Word2vec is an algorithm that has gained a lot of attention since it's creation in 2013 because of it's unique ability to coerce words into numerical vectors that mimic the relationships found in language. It was built at Google DeepMind which is today mainly known for it's AI programs that can beat expert human players in games like chess and go. Word2vec uses a recurrent neural network to approach this natural language processing task. There are two different network architectures that can accomplish the same goal of creating vectors. The first one takes in a single word and attempts to predict the words surrounding it called Skip-gram (Figure 2). The other one called Continuous Bag of Words (Figure 2) takes in words in a window around a target word and attempts to predict the target word. In mathematical notation they are both trying to maximize

$$\log P(w_T|W_c)$$

Where w_t is the current target word and W_c is the context words being used. This is done for random words in the documents provided. Both models use stochastic gradient descent to train the parameters in the model. Since the cost function in both networks is whether the prediction of the target word is correct the actual numeric vectors that act as the representations of the words have to be extracted from the middle of the network and are trained parameters rather than output. As a result of this

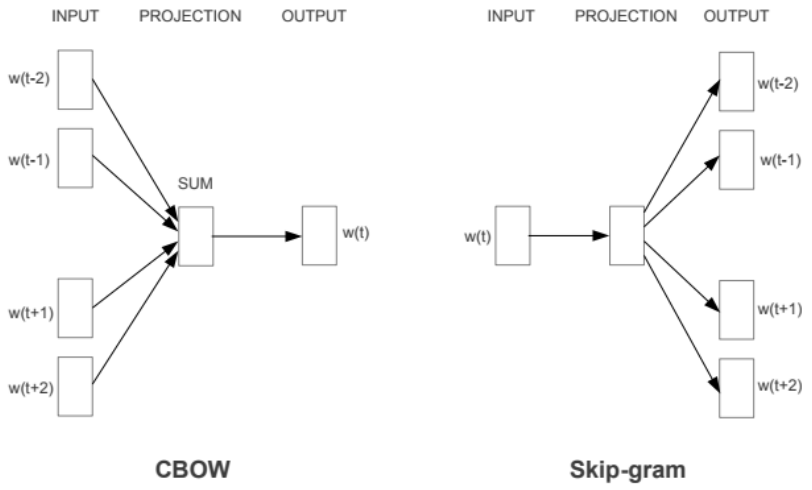

Figure 2: The Continuous Bag of Words and Skip-gram networks used in word2vec [6]

the vectors representing the words have interesting properties when doing vector math on them. For instance the authors use the example that the vector for the word "king" minus the vector for the word "man" plus the vector for the word "woman" will equal a vector close in cosine distance to the vector for the word "queen". Interestingly the authors are only able to give a high-level explanation of why this method works rather than a technical and mathematical one. This has led to other papers such as Goldberg [3] and Rong [6] to attempt to explain what is going on but neither paper is able to really give a full explanation. Doc2vec shares similar characteristics but is able to extend the idea to phrases and sentences.

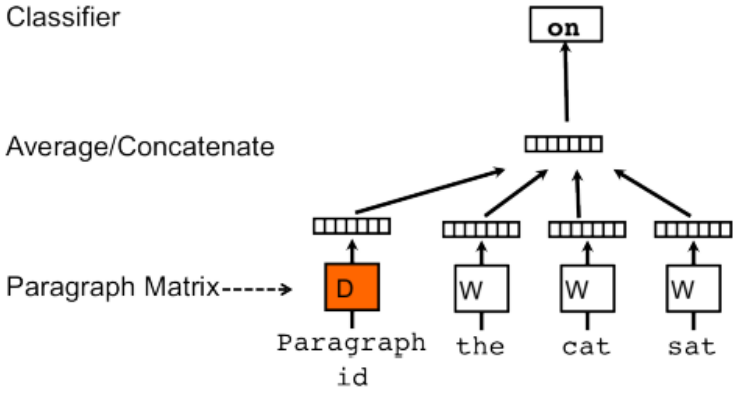

Figure 3: Network architecture of doc2vec [5]

Doc2vec

Created about a year later with Mikolov as one the authors on the paper, doc2vec incorporated a similar strategy as word2vec to capture the context of documents. Doc2vec is also built with a neural network that focuses on predicting words to build the numerical representations of documents. The main difference lies in the addition of a paragraph vector as well as word vectors to predict words. While word2vec only uses words in a window around a target word, doc2vec uses another vector to try to capture the overall context of the document that uses information from word predictions that happened in other parts of the same document (Figure 3). Similar to word2vec there are two different

network architectures that can be used with doc2vec. The first one is the parallel to Skip-gram called Distributed Memory (DM) that attempts to predict a target word based on preceding words and the paragraph vector. The second method is the Bag of Words (BOW) model and predicts the target word based on an unordered collection of context words surrounding the target word and the paragraph vector. Both methods are trained the same way trying to maximize

$$\log P(w_T|W_c, d_i)$$

Where w_T is the target word, W_c are the context words, and d_i is now the paragraph vector for document i. The authors stress that each of these paragraph vectors will differ between documents but word vectors will be shared across documents. Note that while this method also creates numerical representation of the words only the paragraph vectors are of interest. Something that doc2vec must do differently than word2vec is generate paragraph vectors for documents unseen in training. The authors do this by combining a randomly generated vector representation of the new document with the matrix of already trained paragraph vectors and run the neural network again until convergence criteria is met holding the word vectors fixed. So the word vectors do not get the benefit of the context the new document provides. Because of the random initialization and cycles of stochastic gradient descent it is possible that the paragraph vector of a document supplied in training and a paragraph vector inferred for a document in testing will be different even if the documents are the exact same. This is a possible weakness of doc2vec as methods in general should be repeatable with similar results. There is no high level interpretation of the paragraph vectors as there is with word2vec since defining the semantic relationship between documents is much more difficult. Results from the paper showed the method outperforming other current methods in different areas of interest as well as models based on bag of words. It is the hope that numerically representing relationships will be a step forward in all non-numeric data. For this project though I will be focusing on the comparison of using doc2vec and Bag-of-Words generated vectors with SVM.

Sentence1: The boy can jump over the fence.

Sentence2: Fix the fence over there.

Sentence3: Let the girl jump on the trampoline.

	"boy"	"girl"	"jump"	"fix"	"over"	"there"	"fence"	"trampoline"
Sentence1	1	0	1	0	1	0	1	0
Sentence2	0	0	0	1	1	1	1	0
Sentence3	0	1	1	0	0	0	0	1

Bag-of-words example

Figure 4: Example of Bag-of-Words matrix

Bag-of-Words

Bag-of-Words seems to be the base feature generator that people like to compare to or manipulate because of it's simplicity and overall good results. As was shown in the Previous Work section there are papers that will use variants of Bag-of-Words such as clustering or n-grams that are successful. These are variants often used that can help boost the success of the specific task that is being accomplished. For instance in sentiment analysis n-grams have been used to try and solve the "not" problem which is when a document is overall positive but the word "not" makes it actually overall negative. Because of BOW's simplicity and success it is a good method of creating feature vectors to compare with doc2vec and other manipulations.

Dimension Reduction and Expansion

In this project I would also like to try out some techniques that we learned in class to do dimension reduction and expansion. I expect that these methods will not do as well as simply using the outputted

feature vectors from doc2vec and Bag-of-Words so this will be more of a sanity check. The dimension reduction technique I will try is to use PCA on the Bag-of-Words matrix. In essence I will be using the latent semantic indexing from the first paper we talked about in class [1] but instead of using cosine distance to classify I will plug the vectors from the eigen space into an SVM. I believe doing a sparse singular value decomposition is not as expensive as inputting vectors of large length into a SVM so if I do not lose any information by reducing the dimension it would be worth it. Following the notation of [1] I will have a term-document matrix A and after doing a SVD I would get

$$A = U\Sigma V^T$$

Where U is the full matrix for the words, V is the full matrix for the documents and Σ is the diagonal matrix of singular values. Then to reduce the dimension I would use a rank-k approximation by only using the largest k singular values and setting the rest equal to zero which would effectively give me a new equation

$$A_k = U_k\Sigma_k V_k^T$$

Where the sub k matrices have been rank reduced. The A_k would give me my feature vectors to train a SVM with and to get the feature vectors to test with I would follow [1] again and compute the query vectors by doing

$$\hat{q} = q^T U_k \Sigma_k^{-1}$$

Where q is the query vector for the words present in the document. This is a pretty simple technique that might lead to better computing time so I believe it is worth to try.

As more of an experiment I would like to try to do a dimension expansion of doc2vec to see if I can get a better classification rate using a kernel type approach than I can just by manually increasing the length of the doc2vec vectors in the algorithm. A choice of parameter is the length of the doc2vec vectors so I would like to test, for instance, if a length one hundred doc2vec vector is better than a length ten doc2vec vector expanded. I highly expect it will be but I would like to make sure. The method of expansion that I will use is a half Gaussin kernel. We have seen in class that the Guassian kernel between two vectors $x,\ y$ is equal to

$$\begin{aligned} k(x,y) &= \psi(x)^T\psi(y) \\ &= \exp(-\frac{1}{2}||x-y||_2^2) \end{aligned}$$

The trick of using the kernel is that you never actually need to compute $\psi(x)$ or $\psi(y)$. But since $\psi(\cdot)$ is a function into some higher dimensional space than the original vector I want to see if I can use it as a feature vector and see if there is some separation in the higher dimension when using a linear SVM. To calculate the form of $\psi(\cdot)$ I used help from user TenaliRaman on stackexchange.com [8]. The calculation is

$$\begin{aligned} k(x,y) &= \exp(-\frac{1}{2}||x-y||_2^2) \\ &= \exp(-\frac{1}{2}\{x^Tx + y^Ty - 2x^Ty\}) \\ &= \exp(-\frac{1}{2}||x||_2^2)\exp(-\frac{1}{2}||y||_2^2)\exp(x^Ty) \\ &= \exp(-\frac{1}{2}||x||_2^2)\exp(-\frac{1}{2}||y||_2^2)\sum_{n=0}^{\infty}\frac{(x^Ty)^n}{n!} \end{aligned}$$

This implies for vector x of length p

$$\psi(x) = \exp(-\frac{1}{2}||x||_2^2)\{x_1, ..., x_p, \frac{1}{2}x_1^2, ..., \frac{1}{2}x_p^2, x_1x_2, ..., x_{p-1}x_p, ...\}$$

This expansion continues onto infinity so I would only be able to obtain an approximation to the half Guassian kernel. This type of vector is for the most part avoided because of it's infeasibility to calculate and no theoretic evidence for doing well. For my project though I would like to see if using this type of feature vector can help with the SVM which used to separate data in higher dimensions.

Support Vector Machines

Support vector machines are a powerful way to classify data using feature vectors that is used as a benchmark for many other new classification methods because of it's versatility and success. Since the first paper in 1992 [2], SVM has received a lot of attention such as modifications and critiques and has been used in many different applications. My review of SVM will mainly follow the work of Fan et al. [4] in their paper detailing SVM and how the libLinear package in Python implements it. The linear SVM function I will be using is called the L_2 loss L_2 penalized linear SVM and attempts to find

$$\arg\min_{w} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \max(0, 1 - y_i w^T x_i)^2$$

Where $y_i \in \{-1, 1\}$ are the labels of a positive or negative tweet, x_i is the feature vector, C is the tuning parameter, and w is the weights that are being optimized. I decided to use this linear SVM kernel because of its familiarity to me as a squared loss and since it has built in functionality to find the best C tuning parameter by cross-validation. The second part was especially key as I have a lot of data so any speed up is necessary to take advantage of. This is the reason I chose not to attempt to implement a Gaussian kernel in this project as well since to find the best tuning parameter C and γ and also train the weight matrix w is extremely expensive. It is unclear if this sacrifice is reasonable though as using a linear kernel for drawing a boundary between text data seems logically flawed. Text data is naturally complex and so one would assume a kernel that can capture complex boundaries would naturally be the better choice. Unfortunately due to time I am forced to use the less logically sound option.

Results

Dataset

The dataset I am using to test the combination of different doc2vec and Bag-of-Words manipulations as a generator and linear SVM as a classifier is the Sentiment140 [2] dataset originally created by students at Stanford. This dataset is comprised of 1.6 million tweets that have a positive, negative and neutral sentiment attached to them. The dataset was collected by searching key words in combination with smiley faces ":)" and frowny faces ":(". The tweets with smiles were labeled positive and the tweets with frowns were labeled negative. It is not clear from their website how the neutral sentiment was assigned, but in this project I will only use positive and negative tweets. The tweets were automatically assigned sentiment rather than manually assigned sentiment which made the process extremely fast. In my experiments I chose to use a maximum of 200,000 tweets so I am not touching a majority of the data. The state of the art classification rate for this data is about ∼85%.

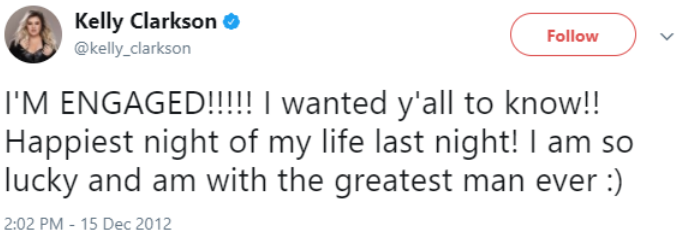

Figure 5: Example positive tweet

Coding

This project is an off-shoot of my current research so I will have code in Python. The code in Python will be to create the feature vectors from the tweets and then to do the training and classification using SVM. The main libraries that I will be using in Python are gensim which is a Python implementation of the papers [5], [6] that are not done by the authors and libLinear [4] which is extremely fast implementation of SVM using a linear kernel. The actual function of doc2vec in gensim has a lot of tuning parameters that are required to get the best classification rate. Some of these tuning parameters for example are context window size, length of the outputted vectors, etc. For this project I will use a lot of the parameter choices that I have found successful in the past and only adjust length of the outputted vectors. I believe these tuning parameters will have the a smaller impact on classification than the number of samples seen since- especially in text data- the more samples the better. Other parameters that papers I have read recently also suggest that adjusting the learning rate is important. In libLinear the only tuning parameter will be the weight parameter C in SVM which the package has a function to find efficiently using cross-validation so I will leave it up to the package to choose this parameter value every time instead of treating it as another tuning parameter I am testing. To get a classification rate I will use a training set of 70% of the data (to train doc2vec vectors), a testing set of 20% (to choose SVM model), and a validation set of 10% of the data (to get final classification rate). When classifying Bag-of-Words I will combine the testing and validation sets into one testing set.

Comparisons

My experiment will involve multiple comparisons to see what combination of generator and classifier works the best. As I described above there are many tuning parameters in the doc2vec algorithm so to keep some consistency and only change what I care about comparing I will hold some parameters constant throughout the experiment. One important parameter that I will hold constant is I will use the Distributed Memory method in doc2vec which takes into account the order of words preceding the target word being predicted. I have seen this works much better than the Bag-of-Words method within doc2vec since Bag-of-Words only cares about the presence of words. The parameters I will compare when using doc2vec is changing the feature vector length and changing number of samples supplied. Two of the feature vector rows will be completely from doc2vec and two will be from using an original doc2vec vector of length ten and implementing the half Gaussian kernel. I will expand the half Gaussian kernel vectors up to degree three and four which will give total feature vector lengths of 286 and 1001 respectively. To fairly compare everything the two complete doc2vec vectors will also have length of 286 and 1001. The number of samples I will compare will be 100,000, 150,000 and 200,000. In an earlier section of my paper I talked about how training the original doc2vec with a document and inferring the same document gives different paragraph vectors. To illustrate this I will have two classification rates in parentheses: first the classification rate if the document was present during training and second if the document's paragraph vector was inferred afterwards.

Doc2vec

	Samples		
Length	100,000	150,000	200,000
286	(0.732, 0.609)	(0.733, 0.601)	(0.731, 0.607)
1001	(0.731, 0.603)	(0.733, 0.605)	(0.731, 0.608)
10,3	(0.554, 0.501)	(0.554, 0.500)	(0.548, 0.499)
10,4	(0.550, 0.502)	(0.541, 0.498)	(0.552, 0.499)

From what I can see there is a large difference in classification rate for using the full doc2vec vectors and the half Gaussian kernel. The validation set is practically 50\50 between positive and negative split so classifying all tweets as positive would have about the same classification rate as using an inferred paragraph vector with the half Gaussian kernel. Using the full doc2vec vectors with paragraph vectors from testing documents that used in training did modestly but still about 10% below state of the art. And that is when training the doc2vec with the testing documents which technically is not viable. Using the inferred paragraph vectors for complete doc2vec vectors did quite poorly across the board.

Another thing to note is that changing number of samples and length of feature vectors did not do much to make the classification rate fluctuate. This is surprising to me as I was expecting an increase in classification rate as number of samples increased. This tells me that I am not looking at the correct tuning parameters to get this method working properly. The next thing I would try is messing more with the learning rate but that is for another time. Next is to see if Bag-of-Words does any better.

The next thing I would like to test is to see if I can reduce the dimension of a Bag-of-Words matrix using singular value decomposition and still maintain a good classification rate. As I've mentioned before Bag-of-Words has been around for a long time because of it's simplicity and success. I will still use a linear SVM as a classifier and compare the results across the number of samples without 200,000 since changing sample size did not have an impact in the previous section. For my rows I will test using the full BOW matrix against using a rank two hundred approximation and a rank one hundred approximation. Since the number of terms seen in each different sample size will be different the length of the vectors for the full BOW matrix will differ. Since I am not using doc2vec I will not have inferred comparisons and so will only report my testing subset classification rate.

BOW

	Samples	
Length	100,000	150,00
full	0.791	0.790
200	0.729	0.724
100	0.693	0.686

It's clear from looking at the results that I lose information when I use a singular value decomposition on my BOW matrix and then do rank-k approximations. It's a little better the larger the k but I was running into memory problems so I had to keep k low. These memory problems most likely had to do with my own code as I wrote the PCA decomposition. Possibly the higher k is above two hundred the better the classification rate will be but I am not able to determine that with my own code. In the end it is good to know that I should continue using the full BOW matrix.

Overall it looks like BOW is the best feature vector by about 6%. This is surprising as the authors of doc2vec talked about it being able to beat BOW in every aspect since it incorporates the presence of words as well as attempts to find some context. I believe I will have to do more parameter tuning to see if it this is truly an example of doc2vec not beating BOW or just for these specific parameters. There are other things I would like to explore as well.

Further Work

At the conclusion of this project I am still interested in possibly using different kernel estimators within the SVM and exploring using doc2vec for text documents that are longer than tweets. A kernel that I did not use that has shown to be successful in many applications is the Gaussian kernel. Since there is naturally a lot of information contained within text data it is possibly an unreasonable assumption that I would be able to draw a linear boundary between my labeled data. I am not as interested in the polynomial kernel because my number of samples is so large and the Gaussian kernel has been shown to do well as number of samples approaches infinity. I attempted to gain some of the dimension expansion results of the Gaussian kernel by using the half Gaussian kernel as a feature vector but I know that this is only a weak approximation and cannot fully capture the information provided by the Gaussian kernel.

To continue to explore the usefulness of doc2vec I would also like to try a classification analysis on text documents that are much larger than tweets. In the paper by Le and Mikolov [5] they say that the doc2vec method would work for any length of document. It's possible that longer documents have more clues to develop context would be a natural fit for using doc2vec. A dataset that was presented as an example dataset in class was called news20 [9]. This dataset contains documents labeled as different categories such as "atheist". It is also possible that the length of the documents would only muddy the

context and doc2vec would be unsuccessful in outputting feature vectors that could be partitioned. It is an interesting comparison and one I would like to explore in the future.

Conclusion

In this report I have outlined new combinations of generators and a classifier to do sentiment analysis of short texts and tested different parameters and feature vectors to determine what works best when classifying a dataset of positive and negative tweets. In the literature there are many examples of an unsupervised method generating a numeric representation being paired with a model to classify the data with the additional information of labels. I chose to use a newish algorithm to translate documents into vectors called doc2vec that claims to be able to capture the context of sentences. I also used the tried and true method of Bag-of-Words as well as some manipulations to see if I could artificially create more information. Then as a classifier I used a linear SVM which is a powerful classifier used in many situations. Among the different parameters I tested were number of samples supplied, length of the vector outputted by doc2vec, and different feature vectors. In the end I found results that were interesting and I would like to follow up on them to explore how much I could push things. The results that I found was that doc2vec did best when using features created entirely by the algorithm rather than also using a half Gaussian kernel, and that Bag-of-Words by itself should not be reduced to keep as much information as possible. I also found that Bag-of-Words did better on the whole than doc2vec which is something I would like to pursue further to find out if I am able to change this. This report has been both enlightening and great practice for a large scale experiment. Thank you!

Appendix

```
##########################################################################
Doc2vec Python code based on https://github.com/RaRe-Technologies/gensim/blob/develop/docs/note
##########################################################################

import csv
import os
import smart_open
import numpy as np
import locale
import io
from replace_one_words import replace_one_words
import time

locale.setlocale(locale.LC_ALL, 'C')

directory = "Data"
#file_name_list = ["my_testing_dataset"] #do test dataset in 0 place and train dataset in 1 pla
file_name_list = ["testdata.manual.2009.06.14", "training.1600000.processed.noemoticon"]

use_neutral_tweets = False

num_samples_wanted = int(166667)
mod_val = 1600000/num_samples_wanted

##########################################################################
#function to take in a tweet and reduce letters and replace unwanted characters
##########################################################################
def clean_line(line, query_term):
line += " "

line = line.lower()

url_char = "http"
if url_char in line:
url_index = line.index(url_char)
url_first_space_after = line[url_index:].index(" ") + url_index
url_text_to_replace = line[url_index:url_first_space_after]
line = line.replace(url_text_to_replace, "URL")

BOM_chars = ["\xef\xbf\xbd", "\xcf\x9b"]
for bc in BOM_chars:
line = line.replace(bc, " ")

unique_letters = "".join(set(line))
for uc in unique_letters:
while uc + uc + uc in line:
line = line.replace(uc + uc + uc, uc + uc)

html_code = ["&quot;", '&amp;', '&lt;', '&gt;']
in_ascii = ['"', '&', '<', '>']
for hc_ind in range(len(html_code)):
line = line.replace(html_code[hc_ind], in_ascii[hc_ind])
```

```
replace_emoticons = [":)", ";)", ":(", ";(", ":')", ":'(", ";')", ";'(", ":-)", ":-("]
for re in replace_emoticons:
line = line.replace(re, "")

space_chars = ['"', ',', '/', '_', "#", "[", "]", "&", "*", "-"]
for sc in space_chars:
line = line.replace(sc, " " + sc + " ")

new_space_chars = [".", "'", ";", ":", "(", ")", "!", "?"]
for nsc in new_space_chars:
if nsc in line:
for i in range(line.count(nsc)):
nsc_index = [ind for ind, ltr in enumerate(line) if ltr == nsc]
if not (line[nsc_index[i] + 1] == nsc or line[nsc_index[i] - 1] == nsc):
if not line[nsc_index[i] - 1] == " ":
line = line[:nsc_index[i]] + " " + line[nsc_index[i]] + " " + line[(nsc_index[i]+1):]
elif line[nsc_index[i] + 1] == nsc:
if not line[nsc_index[i] - 1] == " ":
line = line[:(nsc_index[i])] + " " + line[(nsc_index[i]):]
elif line[nsc_index[i] - 1] == nsc:
if not line[nsc_index[i] + 1] == " ":
line = line[:(nsc_index[i] + 1)] + " " + line[(nsc_index[i] + 1):]

while "@" in line:
at_index = line.index("@")
at_first_space_after = line[at_index:].index(" ") + at_index
at_text_to_replace = line[at_index:at_first_space_after]
if at_text_to_replace[1:] == query_term:
line = line.replace(at_text_to_replace, "QUERY_TERM")
else:
line = line.replace(at_text_to_replace, "USERNAME")

line = line.replace(query_term, "QUERY_TERM")

try:
line = line.decode('utf-8')
except:
line = line.decode('latin-1')

line = u' '.join(line.split()).strip()

line += '\n'

return line

#########################################################################
#function to go through tweet data and output a cleaned version
#currently has an all-data and classifying-data in case use_neutral_tweets = True
#########################################################################
def ALL_clean_dataset(file_name_list = file_name_list, use_neutral_tweets = False, num_samples_

start = time.clock()
```

```
np.random.seed(8363)
mod_val = (1600000 / num_samples_wanted) + 2

classifier_iterator = 0

doc2vec_string = u''
classifying_sentences_string = u''

sentiment_score_list = [0] * num_samples_wanted
classifiers_to_word2vec_tags = [0] * num_samples_wanted
split_list = [0] * num_samples_wanted
len_doc2vec = 0

for file_no, file_name in enumerate(file_name_list):
#with io.open(os.path.join(directory, file_name + ".csv"), encoding='utf-8') as csvfile:
with smart_open.smart_open(os.path.join(directory, file_name + ".csv"), 'rb') as csvfile:
csvreader = csv.reader(csvfile, dialect=csv.excel)
#csvreader = unicode_csv_reader(csvfile)
for line_no, line in enumerate(csvreader):
if file_no == 0 or line_no % mod_val == 0:
tweet_sentiment = int(line[0])
if not tweet_sentiment == 2 or use_neutral_tweets:
#add to doc2vec_string
tweet_id = line[1]
query_word = line[3]
dirty_tweet = line[5]
clean_tweet = clean_line(dirty_tweet, query_word)

doc2vec_string += clean_tweet

if not tweet_sentiment == 2:
classifying_sentences_string += clean_tweet
sentiment_score = (tweet_sentiment - 2)/2
#sentiment_score_list += [sentiment_score]
sentiment_score_list[classifier_iterator] = sentiment_score
#split_list += [["test", "train", "validation"][file_no*(1*(np.random.uniform(0,1) > 0.95) + 1)
#split_list[len_doc2vec] = ["test", "train", "validation"][file_no*(1*(np.random.uniform(0,1) >
split_list[classifier_iterator] = ["train","train","train","train","train", "train","train","te

#classifiers_to_word2vec_tags += [len_doc2vec]
classifiers_to_word2vec_tags[classifier_iterator] = len_doc2vec

classifier_iterator += 1

len_doc2vec += 1

if classifier_iterator % 10000 == 0:
print classifier_iterator

if classifier_iterator >= num_samples_wanted:
break

classifying_splitlines = classifying_sentences_string.splitlines()
doc2vec_splitlines = doc2vec_string.splitlines()
```

```
#train_sentences = [classifying_splitlines[ind] for ind in range(len(split_list)) if split_list
#test_sentences = [classifying_splitlines[ind] for ind in range(len(split_list)) if split_list|
#validation_sentences = [classifying_splitlines[ind] for ind in range(len(split_list)) if split

#new_classifying_sentences_string = replace_one_words(train_sentences, test_sentences, validati
new_classifying_sentences_string = replace_one_words(classifying_splitlines)
new_doc2vec_string = replace_one_words(doc2vec_splitlines)

#new_sentiment_score_list = [sentiment_score_list[ind] for ind in range(len(split_list)) if spl
#new_split_list = [split for split in split_list if split == "train"] + [split for split in spl

with smart_open.smart_open("all-doc2vec-data-{0}.txt".format(num_samples_wanted), 'wb') as oper
for line_no, line in enumerate(new_doc2vec_string.splitlines()):
new_line = u'*{0} {1}\n'.format(line_no, line)
open_file.write(new_line.encode("utf-8"))

with smart_open.smart_open("all-classifyingSentences-data-{0}.txt".format(num_samples_wanted),
#for line_no, line in enumerate(classifying_sentences_string.splitlines()):
for line_no, line in enumerate(new_classifying_sentences_string.splitlines()):
new_line = u'*{0} {1}\n'.format(line_no, line)
open_file.write(new_line.encode("utf-8"))

end = time.clock()

print "Cleaning the data using %d samples took %f seconds" % (num_samples_wanted, end - start)

return split_list, sentiment_score_list, classifiers_to_word2vec_tags

'''
Just getting sentiment_score_list, classifiers_to_word2vec, split_list

'''

#############################################################################
#function if outputted clean tweets exist but just want the sentiment of the tweets and do not
#############################################################################
def SOME_clean_dataset(file_name_list = file_name_list, use_neutral_tweets = False, num_samples

start = time.clock()

np.random.seed(8363)

mod_val = (1600000 / num_samples_wanted) + 2

classifier_iterator = 0

#doc2vec_string = u''
#classifying_sentences_string = u''

sentiment_score_list = [0] * num_samples_wanted
classifiers_to_word2vec_tags = [0] * num_samples_wanted
split_list = [0] * num_samples_wanted
len_doc2vec = 0
```

```
for file_no, file_name in enumerate(file_name_list):
#with io.open(os.path.join(directory, file_name + ".csv"), encoding='utf-8') as csvfile:
with smart_open.smart_open(os.path.join(directory, file_name + ".csv"), 'rb') as csvfile:
csvreader = csv.reader(csvfile, dialect=csv.excel)
#csvreader = unicode_csv_reader(csvfile)
for line_no, line in enumerate(csvreader):
if file_no == 0 or line_no % mod_val == 0:
tweet_sentiment = int(line[0])
if not tweet_sentiment == 2 or use_neutral_tweets:
#add to doc2vec_string
#tweet_id = line[1]
#query_word = line[3]
#dirty_tweet = line[5]
#clean_tweet = clean_line(dirty_tweet, query_word)

#doc2vec_string += clean_tweet

if not tweet_sentiment == 2:
#classifying_sentences_string += clean_tweet
sentiment_score = (tweet_sentiment - 2)/2
#sentiment_score_list += [sentiment_score]
sentiment_score_list[classifier_iterator] = sentiment_score
#split_list += [["test", "train", "validation"][file_no*(1*(np.random.uniform(0,1) > 0.95) + 1)
#split_list[len_doc2vec] = ["test", "train", "validation"][file_no*(1*(np.random.uniform(0,1) >
split_list[classifier_iterator] = ["train", "train", "train", "train", "train", "train", "train

#classifiers_to_word2vec_tags += [len_doc2vec]
classifiers_to_word2vec_tags[classifier_iterator] = len_doc2vec

classifier_iterator += 1

len_doc2vec += 1

if classifier_iterator % 10000 == 0:
print classifier_iterator

if classifier_iterator >= num_samples_wanted:
break

#new_sentiment_score_list = [sentiment_score_list[ind] for ind in range(len(split_list)) if spl
#new_split_list = [split for split in split_list if split == "train"] + [split for split in spl

end = time.clock()
print "Cleaning the data using %d samples took %f seconds" % (num_samples_wanted, end - start)

return split_list, sentiment_score_list, classifiers_to_word2vec_tags

from collections import namedtuple
import gensim
import io

###############################################################################
#function to organize cleaned tweets so can have the clean tweet, sentiment score and whether i
#all_docs is in case neutral tweets were used
```

```
def get_named_documents(sentiment_score_list, split_list, num_samples_wanted):

classifying_documents = []

SentimentDocument = namedtuple("SentimentDocument", "words tags sentiment split")

with io.open("all-classifyingSentences-data-{0}.txt".format(num_samples_wanted), encoding="utf-
for line_no, line in enumerate(allClassifyingSentences):
tokens = gensim.utils.to_unicode(line).split()
words = tokens[1:]
tags = [line_no]
# sentiment_list has to be in console for now, will do better when I get a main()
sentiment = sentiment_score_list[line_no]
split = split_list[line_no]
classifying_documents.append(SentimentDocument(words, tags, sentiment, split))

train_docs = [doc for doc in classifying_documents if doc.split == "train"]
test_docs = [doc for doc in classifying_documents if doc.split == "test"]
validation_docs = [doc for doc in classifying_documents if doc.split == "validation"]

print " There are %i total documents in classification. \n There are %i documents in train_docs
len(classifying_documents), len(train_docs), len(test_docs), len(validation_docs),
float(len(train_docs)) / len(classifying_documents), float(len(test_docs)) / len(classifying_do
float(len(validation_docs)) / len(classifying_documents))

sentiment_split_train = float(sum([(doc.sentiment + 1)/2 for doc in train_docs]))/len(train_doc
print "The sentiment split in training is %f" % sentiment_split_train

sentiment_split_test = float(sum([(doc.sentiment + 1)/2 for doc in test_docs]))/len(test_docs)
print "The sentiment split in testing is %f" % sentiment_split_test

Doc2VecDocument = namedtuple("Doc2VecDocument", "words tags")

all_docs = []

with io.open("all-doc2vec-data-{0}.txt".format(num_samples_wanted), encoding="utf-8") as allDoc
for line_no, line in enumerate(allDoc2VecSentences):
tokens = gensim.utils.to_unicode(line).split()
words = tokens[1:]
tags = [line_no]
all_docs.append(Doc2VecDocument(words, tags))

print " There are %i total docuemnts that will be used in model building." % (len(all_docs))

all_docs_copy = all_docs[:]

return all_docs, train_docs, test_docs, validation_docs

from gensim.models import Doc2Vec
import gensim.models.doc2vec
import multiprocessing
from collections import OrderedDict
from gensim.test.test_doc2vec import ConcatenatedDoc2Vec
```

```
#which doc2vec models the classifier should train on
#unfortunately not optimized right now to allow full control from function line
#has to be manually edited within the code to get different models

################################################################################
#function to initiate the doc2vec models, I don't have a good way to do this without manually c
################################################################################
def build_doc2vec(all_docs, size_in):
cores = multiprocessing.cpu_count()
assert gensim.models.doc2vec.FAST_VERSION > -1, "This will be painfully slow otherwise"

models_list = [
# Doc2Vec(dm = 1, dm_mean= 1, size = 400, window = 10, negative = 11, hs = 0, min_count= 1, wor
#Doc2Vec(dm=1, dm_mean=1, size=500, window=10, negative=11, hs=0, min_count=1, workers=cores),
Doc2Vec(dm=1, dm_mean=1, size=size_in, window=3, negative=11, hs=0, min_count=1, workers=cores)
# Doc2Vec(dm=0, size=400, negative=11, hs=0, min_count=1, workers=cores),
#Doc2Vec(dm=0, size=500, negative=11, hs=0, min_count=1, workers=cores),
#Doc2Vec(dm=0, size=500, negative=11, hs=0, min_count=1, workers=cores)
]

# classifying_docs should be in console before running this
models_list[0].build_vocab(all_docs)
print models_list[0]
# for model in models_list[1:]:
#     model.reset_from(models_list[0])
#     print model

models_by_name = OrderedDict((str(model), model) for model in models_list)

# models_by_name['dbow+dmm_500']= ConcatenatedDoc2Vec([models_list[0], models_list[1]])
# #models_by_name['dbow+dmm_500'] = ConcatenatedDoc2Vec([models_list[1], models_list[4]])
# #models_by_name['dbow+dmm_600'] = ConcatenatedDoc2Vec([models_list[2], models_list[5]])

return models_by_name

from liblinearutil import *
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import pairwise as pw
from sklearn import model_selection as ms
import numpy as np

################################################################################
#use libLinear to do linear svm
################################################################################
def svm_predictor_linear(train_vectors, train_labels):
num_class1 = np.sum(np.array(train_labels) < .5)
num_class2 = len(train_labels) - num_class1
best_C, _ = train(train_labels, train_vectors, '-C -q -s 2 -w1 {0}'.format(float(num_class2)/nu
m = train(train_labels, train_vectors, '-c {0} -q -s 2 -w1 {1}'.format(best_C, float(num_class2
return m, best_C

################################################################################
#function to do classification of svm, originally wanted to compare more classifiers but ran ou
```

```
##############################################################################
def find_error_rate(train_vectors, train_labels, test_vectors, test_labels, classifier = "svm"):
#min_max_scaler = preprocessing.MinMaxScaler()
#train_vectors = min_max_scaler.fit_transform(train_vectors)
#test_vectors = min_max_scaler.transform(test_vectors)

#train_vectors = sm.add_constant(train_vectors)
#test_vectors = sm.add_constant(test_vectors)

if classifier == "svm":
svm_model, best_C = svm_predictor_linear(train_vectors, train_labels)

test_predictions, test_acc, _ = predict(test_labels, test_vectors, svm_model, '-q')
train_predictions, train_acc, _ = predict(train_labels, train_vectors, svm_model, '-q')

param = best_C

test_acc = float(test_acc[0])/100
train_acc = float(train_acc[0])/100



elif classifier == "knn":
nn, train_acc = train_knn(train_vectors, train_labels)
knn_model = KNeighborsClassifier(n_neighbors=nn)
knn_model.fit(train_vectors, train_labels)
test_acc = knn_model.score(test_vectors, test_labels)

param = nn

elif classifier == "cos":
similarity = pw.cosine_similarity(test_vectors, train_vectors)
n_v = -1
test_acc = -1
train_acc = -1
for num_votes in range(1,14,2):
max_indices = [np.argpartition(arr, -1*num_votes)[(-1*num_votes):] for arr in similarity]
votes = [np.array(train_labels)[max_indices[v]] + 1 for v in range(len(max_indices))]
votes_added_up = [np.bincount(vot).argmax() - 1 for vot in votes]
test_acc_temp = float(sum(np.equal(votes_added_up, test_labels)))/len(test_labels)
if test_acc_temp > test_acc:
test_acc = test_acc_temp
n_v = num_votes

param = n_v

elif classifier == "log":
log_model, best_C = log_regression(train_vectors, train_labels)

test_predictions, test_acc, _ = predict(test_labels, test_vectors, log_model, '-q')
train_predictions, train_acc, _ = predict(train_labels, train_vectors, log_model, '-q')

param = best_C
```

```
test_acc = float(test_acc[0])/100
train_acc = float(train_acc[0])/100

else:
return "do a better job of selecting a classifier"


return test_acc, train_acc, param

from random import shuffle
import datetime
from classifiers import find_error_rate
from build_doc2vec_vectors import doc2vec_vectors
from feature_generator import feature_generator

#train and test classifying models to see which parameter is best to use in training the model
#also trains a doc2vec model over 20 passes
#alpha_list is made up of tuples that is the starting learning rate and the ending learning rat

##############################################################################
#function to run through the different doc2vec models to select the best one
##############################################################################
def choose_best_model(models_by_name, all_docs, train_docs, test_docs, tag_decoder, deg, classi

print "START all models and learning rates %s" % str(datetime.datetime.now())

best_alpha_acc = 0

alpha_list = [(0.025,0.001)] if alpha_list == "none" else alpha_list #list of tuples
all_docs_copy = all_docs[:]

for alpha_start, alpha_min in alpha_list:
passes = 20
alpha = alpha_start
alpha_delta = (alpha_start - alpha_min) / passes

best_model_acc = 0

models_by_name_copy = models_by_name.copy()

print "START %s learning rate max: %f min %f" % (str(datetime.datetime.now()), alpha_start, alp
for epoch in range(passes):
shuffle(all_docs_copy)#need all_docs in console

for model_name, model in models_by_name_copy.items(): #need models_by_name in console
model.alpha, model.min_alpha = alpha, alpha
model.train(all_docs_copy, total_examples = len(all_docs_copy), epochs = 1)

if epoch == (passes - 1):

train_vectors, train_labels, test_vectors, test_labels = doc2vec_vectors(model, train_docs, tes

if deg > -1:
train_vectors = feature_generator(train_vectors, deg)
```

```
test_vectors = feature_generator(test_vectors, deg)

current_test_acc, current_train_acc, current_param = find_error_rate(train_vectors, train_label

best_so_far = ""

if current_test_acc > best_model_acc:
best_model_acc = current_test_acc
best_param = current_param
best_model = model
best_model_name = model_name
best_so_far = "*"

print "%s%f: testing from %s on pass %d" % (best_so_far, current_test_acc, model_name, epoch)
if not classifier_in == "cos":
print "%f: training from %s on pass %d" % (current_train_acc, model_name, epoch)

if epoch % (passes-1) == 0:
train_vectors, train_labels, test_vectors_infer, test_labels_infer = doc2vec_vectors(model, tra

if deg > -1:
train_vectors = feature_generator(train_vectors, deg)
test_vectors_infer = feature_generator(test_vectors_infer, deg)

current_test_acc_infer, current_train_acc, current_param_infer = find_error_rate(train_vectors,

best_so_far = ""

if current_test_acc_infer > best_model_acc:
best_so_far = "*"

print "%s%f: testing from %s on pass %d" % (best_so_far, current_test_acc_infer, model_name + "

alpha -= alpha_delta

print "END %s learning rate max: %f min %f" % (str(datetime.datetime.now()), alpha_start, alpha

if best_model_acc > best_alpha_acc:
best_alpha_acc = best_model_acc
best_alpha_min = alpha_min
best_alpha_start = alpha_start
best_alpha_model = best_model
best_alpha_param = best_param
best_alpha_name = best_model_name

print "Best model is %s with an accuracy of %f using alpha_start %f and alpha_min %f and using

return best_alpha_model, best_alpha_param

from liblinearutil import *
from build_doc2vec_vectors import doc2vec_vectors
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import pairwise as pw
```

```
#############################################################################
#function to do validation testing based on the best model selected
#############################################################################
def validation_testing(train_docs, test_docs, validation_docs, best_param, feature_generator, c
if feature_generator == "doc2vec" and model == "none":
return "need a model"

training_docs = train_docs + test_docs

validation_acc_infer = -1
validation_acc = -1

if feature_generator == "doc2vec":
_, _, validation_vectors, validation_labels = doc2vec_vectors(model, training_docs, validation_
training_vectors, training_labels, validation_vectors_infer, validation_labels_infer = doc2vec_


elif feature_generator == "princomp":
pass

if classifier_in == "svm":
num_class1 = np.sum(np.array(training_labels) < .5)
num_class2 = len(training_labels) - num_class1
svm_m = train(training_labels, training_vectors, '-c {0} -q -s 2 -w1 {1}'.format(best_param, fl

_, validation_acc, _ = predict(validation_labels, validation_vectors, svm_m, '-q')

validation_acc = float(validation_acc[0]) / 100

_, validation_acc_infer, _ = predict(validation_labels_infer, validation_vectors_infer, svm_m,

validation_acc_infer = float(validation_acc_infer[0]) / 100

elif classifier_in == "knn":
knn_m = KNeighborsClassifier(n_neighbors=best_param)
knn_m.fit(training_vectors, training_labels)
validation_acc = knn_m.score(validation_vectors, validation_labels)
validation_acc_infer = knn_m.score(validation_vectors_infer, validation_labels_infer)

elif classifier_in == "cos":
similarity = pw.cosine_similarity(validation_vectors, training_vectors)
max_indices = [np.argpartition(arr, -1 * best_param)[(-1 * best_param):] for arr in similarity]
votes = [training_labels[max_indices[v]] for v in range(len(max_indices))]
votes_added_up = [np.bincount(vot).argmax() for vot in votes]
validation_acc = float(sum(votes_added_up == validation_labels)) / len(validation_labels)

similarity = pw.cosine_similarity(validation_vectors_infer, training_vectors)
max_indices = [np.argpartition(arr, -1 * best_param)[(-1 * best_param):] for arr in similarity]
votes = [training_labels[max_indices[v]] for v in range(len(max_indices))]
votes_added_up = [np.bincount(vot).argmax() for vot in votes]
validation_acc_infer = float(sum(votes_added_up == validation_labels_infer)) / len(validation_l

print "The validation error is %f and the validation_inferred error is %f" %(validation_acc, va
```

```
##############################################################################
#Next will be functions that calculate the half Gaussian kernel
##############################################################################

##############################################################################
#function to get all addends of a number, credit given below
##############################################################################
def addend_finder(num):
return addend_finder_wrapper(num, num , [], [])

#function to return all addends of a positive integer
#credit given at bottom
def addend_finder_wrapper(n, max, L, out_list):
if (n == 0):
out_list = out_list + [L]
return out_list
for i in reversed(range(1,min(max,n) + 1)):
out_list = addend_finder_wrapper(n - i, i, L + [i], out_list)
return out_list

#credit to https://stackoverflow.com/questions/7331093/getting-all-possible-sums-that-add-up-to
# which leads to https://introcs.cs.princeton.edu/java/23recursion/Partition.java.html

##############################################################################
#function to construct proper expansion of factors
##############################################################################
def padding(degree, terms, addends):
if degree > terms: #for example: (x1 + x2)^3
addends = addends[:(max(ind for ind in range(len(addends)) if len(addends[ind]) == terms) + 1)]
for a in range(len(addends)):
cur_addend_length = len(addends[a])
#choose_amount = misc.comb(terms,a+1)
num_zeroes_needed = terms - cur_addend_length
new_addend = addends[a] + [0]*num_zeroes_needed
addends[a] = new_addend

elif terms >= degree: #for example (x1 + x2 + x3)^2 or (x1 + x2)^2
for a in range(len(addends)):
cur_addend_length = len(addends[a])
#choose_amount = misc.comb(terms,a+1)
num_zeroes_needed = terms - cur_addend_length
new_addend = addends[a] + [0]*num_zeroes_needed
addends[a] = new_addend
return addends

import math
from operator import mul

##############################################################################
#function to assign coefficients to all combinations
##############################################################################
def get_coefficients(padded_addends):
terms = len(padded_addends[0])
```

```
degree = padded_addends[0][0]
outL = []

for subL in padded_addends:
unique_elements = []
for elem in subL:
if not elem in unique_elements:
unique_elements += [elem]
length_coefficients = math.factorial(terms)/reduce(mul,[math.factorial(subL.count(elem)) for el

coefficient = math.sqrt(math.factorial(degree)/reduce(mul,[math.factorial(elem) for elem in sub

outL += [coefficient]*length_coefficients

return outL

import itertools

##############################################################################
#function to get all orders of coefficients
##############################################################################
def get_all_permutations(padded_addends):
ununique_out_L = []
unique_out_L = []
for L in padded_addends:
ununique_out_L += [list(perm) for perm in itertools.permutations(L)]
for subL in ununique_out_L:
if not subL in unique_out_L:
unique_out_L += [subL]

return unique_out_L

def list_transpose(L):
return map(list, zip(*L))

def all_permutations(padded_addends):
return list_transpose(get_all_permutations(padded_addends))

from addend_finder import addend_finder
from padding import padding
from all_permutations import all_permutations
from coefficient_assigner import get_coefficients

import numpy as np
import math

##############################################################################
#function to create half Gaussian kernel based on the initial vector and how many degrees of ap
##############################################################################
def feature_generator(in_arr, degree):
print "Starting to generate half Gaussian feature vector"
out_arr = np.array([math.exp(-1*sum(x*x for x in in_arr))], 'float')
terms = len(in_arr)
for d in range(1, degree+1):
```

```
addends = addend_finder(d)
addends.sort(key = len)
padded_addends = padding(d,terms,addends)
powers = np.array(all_permutations(padded_addends))
coefficients = math.sqrt(float((2**d))/math.factorial(d))*np.array(get_coefficients(padded_adde

temp_arr = np.array([1]*len(powers[0]), 'float')
for ind_elem in range(len(powers)):
temp_arr *= in_arr[ind_elem]**powers[ind_elem]


temp_arr *= coefficients
temp_arr *= np.array([math.exp(-1*sum(x*x for x in in_arr))]*len(powers[0]), 'float')

out_arr = np.append(out_arr, temp_arr)

print "Ending generating of half Gaussian feature vector"

return out_arr

def kernel_trick(in_arr_1, in_arr_2):
for ind in range(len(in_arr_1)):
difference_list = in_arr_1-in_arr_2
return math.exp(-1*sum(x*x for x in difference_list))

##############################################################################
#Next I will show Python code required for Bag-of-Words
##############################################################################

from sklearn.feature_extraction import text as fet
from nltk.tokenize import TreebankWordTokenizer
from scipy.sparse.linalg import svds
from scipy.sparse.csr import csr_matrix
import numpy as np
import time

##############################################################################
#function to train a BOW matrix and also do the sparse PCA
##############################################################################
def BOW_vectors(train_docs, test_docs, do_PCA, num_PCA):

print "Beginning to train BOW..."
start = time.clock()

train_sentences = [" ".join(doc.words) for doc in train_docs]
test_sentences = [" ".join(doc.words) for doc in test_docs]

count_vectorizer = fet.CountVectorizer(analyzer="word", lowercase=True, binary=True, stop_words
BOW_train = count_vectorizer.fit_transform(train_sentences)
BOW_test = count_vectorizer.transform(test_sentences)

if do_PCA == True:
u, s, vt = svds(A=csr_matrix.transpose(BOW_train.asfptype()),k=num_PCA,which="LM",return_singul
#train_vectors = projected_BOW(BOW_train, u, s)
```

```
train_vectors = vt.transpose()
test_vectors = projected_BOW(BOW_test, u, s)
else:
train_vectors = BOW_train.todense()
test_vectors = BOW_test.todense()

print "BOW matrix has %d unique terms" % (BOW_train.shape[1])

train_labels = [doc.sentiment for doc in train_docs]
test_labels = [doc.sentiment for doc in test_docs]

end = time.clock()
print "Ending BOW training... took %f seconds" % (end - start)

return train_vectors, train_labels, test_vectors, test_labels


def projected_BOW(BOW, u, s):
k = len(s)
num_docs = BOW.shape[0]
PCA_return = np.array([[0.]*num_docs]*k)

for i in range(k):
u_col = u[:,i]
u_col_mat = np.array([u_col.tolist()]*num_docs)
PCA_col = np.array(BOW.multiply(u_col_mat).sum(1) * (s[i])**-1).T
PCA_return[i] = PCA_col

return PCA_return.T

##############################################################################
#This is my main function that I ran everything from, it prints out a lot of information relate
##############################################################################
from clean_data import ALL_clean_dataset, SOME_clean_dataset
from create_named_documents import get_named_documents
from building_doc2vec import build_doc2vec
from building_models import choose_best_model
from validation_testing import validation_testing
from create_BOW import BOW_vectors
from classifiers import find_error_rate

def doc2vec_classifier(alpha_list= "none", classifier_in = "svm", num_samples_wanted = int(2000

if not use_current_tweet_iterate:
split_list, sentiment_list, tag_decoder_list = ALL_clean_dataset(file_name_list, use_neutral_tw
else:
split_list, sentiment_list, tag_decoder_list = SOME_clean_dataset(file_name_list, use_neutral_t

all_docs, train_docs, test_docs, validation_docs = get_named_documents(sentiment_list, split_li

models_by_name = build_doc2vec(all_docs, size_in)

best_model, best_param = choose_best_model(models_by_name, all_docs, train_docs, test_docs, tag
```

```
validation_testing(train_docs, test_docs, validation_docs, best_param, feature_generator="doc2v

return best_model, best_param

def prinComp_classifier(use_PCA = True, num_PCA = 10, classifier_in = "svm", num_samples_wantec
if not use_current_tweet_iterate:
split_list, sentiment_list, tag_decoder_list = ALL_clean_dataset(file_name_list, use_neutral_tw
else:
split_list, sentiment_list, tag_decoder_list = SOME_clean_dataset(file_name_list, use_neutral_t

_, train_docs, test_docs, validation_docs = get_named_documents(sentiment_list, split_list, num

train_vectors, train_labels, test_vectors, test_labels = BOW_vectors(train_docs + test_docs, va

test_acc, train_acc, param = find_error_rate(train_vectors,train_labels,test_vectors,test_label

print "The testing accuracy was %f \n The training accuracy was %f \n Both used parameter %f" %


def doc2vec_classifier_half_gaussian(alpha_list= "none", classifier_in = "svm", num_samples_war

if not use_current_tweet_iterate:
split_list, sentiment_list, tag_decoder_list = ALL_clean_dataset(file_name_list, use_neutral_tw
else:
split_list, sentiment_list, tag_decoder_list = SOME_clean_dataset(file_name_list, use_neutral_t

all_docs, train_docs, test_docs, validation_docs = get_named_documents(sentiment_list, split_li

models_by_name = build_doc2vec(all_docs, size_in)

best_model, best_param = choose_best_model(models_by_name, all_docs, train_docs, test_docs, tag

validation_testing(train_docs, test_docs, validation_docs, best_param, feature_generator="doc2v

return best_model, best_param
```

Bibliography

[1] Berry et al. Computational methods for intelligent information access. *Supercomputing*, 1995.

[2] Boser et al. A training algorithm for optimal margin classifiers. *Proceedings of the fifth annual workshop on Computational learning theory*, 1992.

[3] Alec Go et al. Twitter sentiment classification using distant supervision. *CS224N Project Report, Stanford*, 2009.

[4] Fan et al. Liblinear: A library ofr large linear classification. *Journal of machine learning research*, 2008.

[5] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, 2014.

[6] Tomas Mikolov et al. Efficient estimation of word representations in vector spac. 2013.

[7] Cicero Nogueira Dos Santos and Maira Gatti. Deep convolutional neural networks for sentiment analysis of short texts. *COLING*, 2014.

[8] TenaliRaman. Non-linear svm classificaiton with rbf kernel. https://stats.stackexchange.com/questions/58585/how-to-understand-effect-of-rbf-svm, 2013.

[9] 20 newsgroups. http://qwone.com/ jason/20Newsgroups/, 2018.

[10] Minier Zsolt and Lehel Csato. Kernel pca based clustering for inducing features in text. *ESANN*, 2007.