Jeffrey Levesque

jlevesqu@syr.edu | From Textual Information to Numerical Vectors

IST-736: HW2

PRofessor gates

**Introduction**

Applications of text mining encompass a broad and general field, often impartial to programming languages. Today scripting languages such as Python and R, as well as static type languages[[1]](#footnote-1) such as Java, have many related utilities. Some popular non-commercial frameworks in python including the natural language toolkit (NTLK)[[2]](#footnote-2) provide nearly all capabilities for data scientists and engineers. However, infrequent edge cases occur, sometimes requiring a unique approach, while other times, borrowing solutions elsewhere.

While a diversification in tools often overlap in functionality, novel features can often complement another framework. The art of optimizing tools is generally an engineering constraint. Moreover, knowing when to polyglot approaches, often requires a degree of acumen. Scripting languages such as R can directly execute python code[[3]](#footnote-3), allowing a marriage of both parties. Many scenarios can be leveraged, including using R for presentational features, until an application needs to scale. Furthermore, edge cases in natural language processing often finds new and exciting scenarios. The integration of the Stanford Part-Of-Speech Tagger[[4]](#footnote-4) in NLTK is extended through java constructs.

Being able to mold different frameworks utilizing the right combination of features is an art requiring a degree of experience. Business objectives often driven requirements, and inherently suggest approaches for storytelling. Moreover, if visualization is contingent to multiple engineering constraints at scale, diverse technology and framework knowledge is invaluable to any organization.

**Analysis**

A single analysis was conducted using two ensembled approaches. Specifically, a TFIDF[[5]](#footnote-5) was constructed from a unigram representation of the original dataset. A second approach involved determining the part of speech (POS)[[6]](#footnote-6) using the default Penn Treebank[[7]](#footnote-7). After the corpus was tokenized, special characters removed (i.e. # and @), and stemming[[8]](#footnote-8), each word was suffixed with the corresponding POS. This union was fed into a TFIDF transformation. Both approaches removed English stopwords, then utilized the naïve bayes classifier. Finally, individual results were combined to generate an overall result.

**Data Preparation**

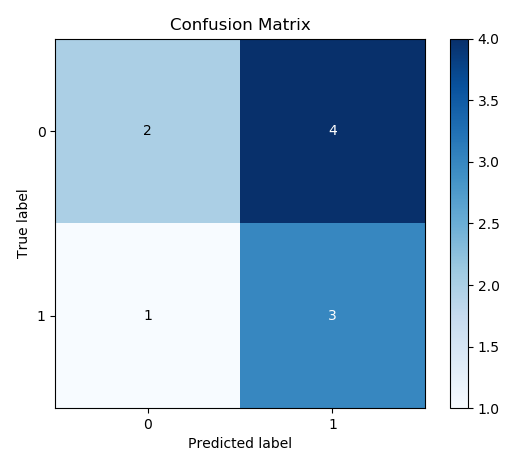
An initial attempt was made to setup a twitter developer account. However, complications including a waiting period for the formal account request, encouraged a manual data collection process. Specifically, using a valid twitter account two terms were searched:

* AI
* AI dangerous

The collected tweets were aggregated into a csv file[[9]](#footnote-9). Specifically, a single column contained two factors - negative sentiment (0), and positive sentiment (1). An attempt was made to keep the distribution relatively balanced. There were no criteria’s defining the selection of tweets, other than generally consisting of more words than hashtags. Therefore, the selection process may be biased. However, 23 instances were positive, while 27 were negative. Another column consisted of the corresponding tweet text.

**Results**

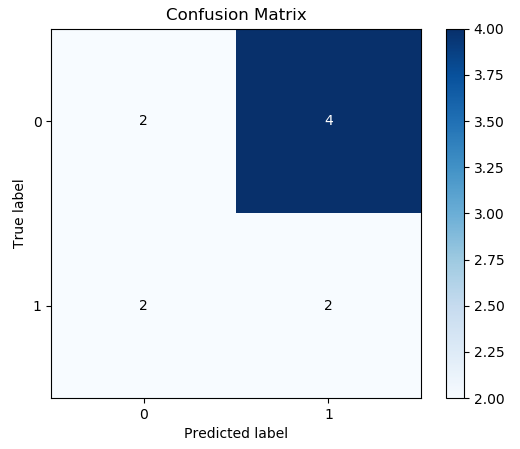
A generic TFIDF unigram representation generated an accuracy of 0.50. The code was largely identical to a previous study[[10]](#footnote-10), yielding 0.923 accuracy. However, this study utilized 80% train, while the previous study utilized 75%. Furthermore, multiple trials largely indicate substantial volatility, not exceeding 0.60. The sklearn train\_test\_split naturally implement shuffling[[11]](#footnote-11) to randomize splitting. However, this configuration was disabled. Furthermore, since the original data encompassed tweets, the count vectorizer, and tfidf transformer were utilized, along with applying the built-in English stop-words.



**Figure 1:** Naïve Bayes classifier on unigram representation. See Appendix A and I for associated code.

As noted in the previous study, it would appropriate to apply an inferencing technique. Specifically, the above Naïve Bayes classifier could be implemented many times, providing a credible expected value.

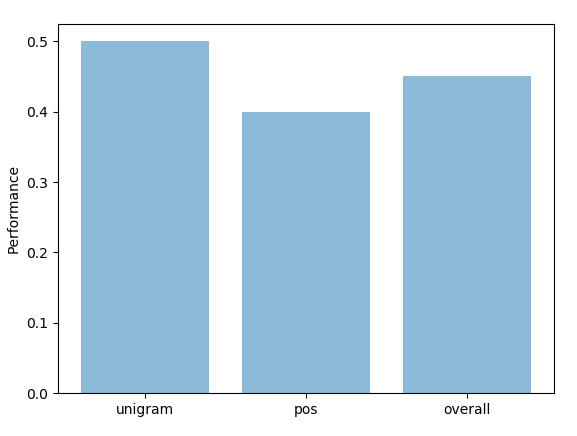
Next, the part of speech approach yielded results not differing substantially from the above unigram variation. Specifically, the corresponding confusion matrix indicates a 0.4 accuracy:



**Figure 2:** Naïve Bayes classifier on POS representation integrated with TFIDF implementation. See Appendix A and I for associated code.

The motivation for creating the POS variation was to have a secondary result, with better accuracy. Coupled with earlier unigram results, it would appropriate to review the implemented code (available in Appendix below). A simple case would verify that 80% of the train data was properly utilized and well formed. Moreover, the captured tweets in the dataset may not contain enough words. Specifically, adding more data could allow the predictive model (i.e. Naive Bayes) to better learn between positive versus negative sentiments. Furthermore, adding the POS suffix, as an integer representation (see Appendix D and Appendix J) to each word, before applying TFIDF vectorization increases the sparsity of data. Since tweets are limited to a maximum of 140 characters[[12]](#footnote-12), the POS approach may not have enough binned words. It would be interesting to test this implementation with more data, as well as a different corpus.

Using the results from the unigram and POS case, an average, and overall score was computed. The visualization is straightforward and align with the above confusion matrices.



**Figure 3:** accuracy comparison between unigram, part of speech (pos), and overall combined score. Associated code can be reviewed in Appendix A and Appendix I below.

**Conclusion**

Text mining is an exciting field for data scientists and engineers. Non-commercial packages and utilities allow coding challenges to be abstracted. Moreover, numerous off-shelf solutions require knowledge to select the right combination of frameworks. Often, the acumen in engineering a problem (or lack of), is noticeable within the associated results. Furthermore, in the area of text mining and NLP, the combination of tokenization and vectorization, provide the data scientists some room for creativity. Specifically, implementing a single algorithm once, is not a good approach to justify a result. Rather, ensemble techniques, along with repeated inferencing generally provides better credibility.

As shown above, a single set of result(s) is difficult to accept and believe. Furthermore, a smaller dataset, is a good indication of statistical inferencing. An interesting extension to the overall results could include adding additional classifiers, then averaging the overall score. Weighting different classifiers according to their accuracy measures would be another novel approach. Overall, this study has indicated some possible room for improvements. Coupled with numerous packages and frameworks, data scientists have the flexibility to leverage existing abstractions while extending creative solutions.

**Appendix A:** define class variables.

class Model():

def \_\_init\_\_(

self,

df=None,

vectorize=True,

key\_text='SentimentText',

key\_class='Sentiment',

fp='{}/data/sample-sentiment.csv'.format(

Path(\_\_file\_\_).resolve().parents[1]

)

):

# class variables

if df is not None:

self.df = df

else:

self.df = pd.read\_csv(fp)

self.key\_text = key\_text

self.key\_class = key\_class

# vectorize data

if vectorize:

self.split()

self.vectorize()

**Appendix B:** split data into train and test.

def split(self, test\_size=0.20, stem=True, pos\_split=False):

# clean

self.df[self.key\_text] = [re.sub('[#@]', '', x) for x in self.df[self.key\_text]]

# stem

if stem:

porter = PorterStemmer()

self.df[self.key\_text] = [porter.stem(word) for word in self.df[self.key\_text]]

# split

if pos\_split:

for i, row in self.df.iterrows():

# max length

max\_length = len(self.df[self.key\_text].iloc[i].split())

pos = self.df[['pos']].iloc[i]

# rebuild 'key-text' with pos suffix

combined = ''

for j in range(max\_length):

combined = '{combined} {word}-{pos}'.format(

combined=combined,

word=self.df[self.key\_text][i].split()[j],

pos=pos[0][j]

)

self.df[self.key\_text].iloc[[i]] = combined

self.X\_train, self.X\_test, self.y\_train, self.y\_test = train\_test\_split(

self.df[self.key\_text],

self.df[self.key\_class],

test\_size=test\_size

)

**Appendix C:** get train and test dataset.

def get\_split(self):

return({

'X\_train': self.X\_train,

'X\_test': self.X\_test,

'y\_train': self.y\_train,

'y\_test': self.y\_test,

})

**Appendix D:** apply pos tagger to supplied list.

def get\_pos(self, l, pos\_length=280):

result\_word = []

result\_pos = []

pos = [pos\_tag(x) for x in l]

for y in pos:

result\_word.append([x[0] for x in y if x[0] not in stop\_words])

result\_pos.append(

[penn\_scale[x[1]] if x[1] in penn\_scale and x[0] not in stop\_words else 1 for x in y]

)

# consistent length

for i,x in enumerate(result\_pos):

if len(x) < pos\_length:

difference = pos\_length - len(x)

result\_pos[i].extend([1] \* difference)

else:

difference = len(x) - pos\_length

result\_pos[i] = result\_pos[i][:len(result\_pos[i]) - difference]

return(result\_word, result\_pos)

**Appendix E:** vectorize provided data.

def vectorize(self, stop\_words='english'):

'''

vectorize provided data.

'''

# bag of words: with 'english' stopwords

self.count\_vect = CountVectorizer(stop\_words=stop\_words)

bow = self.count\_vect.fit\_transform(self.X\_train)

# tfidf weighting

self.tfidf\_transformer = TfidfTransformer()

self.X\_train\_tfidf = self.tfidf\_transformer.fit\_transform(bow)

**Appendix F:** vectorize provided data.

def vectorize(self, stop\_words='english'):

# bag of words: with 'english' stopwords

self.count\_vect = CountVectorizer(stop\_words=stop\_words)

bow = self.count\_vect.fit\_transform(self.X\_train)

# tfidf weighting

self.tfidf\_transformer = TfidfTransformer()

self.X\_train\_tfidf = self.tfidf\_transformer.fit\_transform(bow)

**Appendix G:** get original dataframe.

def get\_df(self):

return(self.df)

**Appendix H:** create naive bayes model.

def model(self, X, y, validate=False):

# fit model

self.clf = MultinomialNB().fit(X, y)

# validate

if validate and len(validate) == 2:

predictions = []

for item in list(validate[0]):

prediction = self.count\_vect.transform([item])

predictions.append(

self.clf.predict(self.tfidf\_transformer.fit\_transform(prediction))

)

return({

'model': self.clf,

'actual': validate[1],

'predicted': predictions

})

return({

'model': self.clf,

'actual': None,

'predicted': None

})

**Appendix I:** implement Model class to determine unigram and part of speech classifier, then ensemble results with corresponding confusion matrix.

if \_\_name\_\_ == '\_\_main\_\_':

#

# unigram: perform unigram analysis.

#

unigram = Model()

# unigram vectorize

unigram.split()

unigram\_params = unigram.get\_split()

unigram\_vectorized = unigram.get\_tfidf()

# unigram classifier

model\_unigram = unigram.model(

unigram\_vectorized,

unigram\_params['y\_train'],

validate=(unigram\_params['X\_test'], unigram\_params['y\_test'])

)

# plot unigram

skplt.metrics.plot\_confusion\_matrix(

model\_unigram['actual'],

model\_unigram['predicted']

)

plt.show()

#

# perform pos analysis

#

df\_pos = unigram.get\_df()

# define pos

df\_pos['pos'] = unigram.get\_pos(

df\_pos['SentimentText'].apply(lambda x: x.split())

)[1]

# new dataframe

pos = Model(df\_pos)

pos.split(pos\_split=True)

# pos vectorize

pos\_params = pos.get\_split()

pos\_vectorized = pos.get\_tfidf()

# pos classifier

model\_pos = pos.model(

pos\_vectorized,

pos\_params['y\_train'],

validate=(unigram\_params['X\_test'], unigram\_params['y\_test'])

)

# plot unigram

skplt.metrics.plot\_confusion\_matrix(

model\_pos['actual'],

model\_pos['predicted']

)

plt.show()

# ensembled scored

score\_unigram = accuracy\_score(

model\_unigram['actual'],

model\_unigram['predicted']

)

score\_pos = accuracy\_score(

model\_pos['actual'],

model\_pos['predicted']

)

score\_good = (score\_unigram + score\_pos) / 2

score\_bad = 1 - score\_good

objects = ('unigram', 'pos', 'overall')

y\_pos = np.arange(len(objects))

performance = [score\_unigram, score\_pos, score\_good]

plt.bar(y\_pos, performance, align='center', alpha=0.5)

plt.xticks(y\_pos, objects)

plt.ylabel('Performance')

plt.show()

**Appendix J:** numeric representation of penn treebank.

penn\_scale = {

'CC': 2,

'CD': 3,

'DT': 4,

'EX': 5,

'FW': 6,

'IN': 7,

'JJ': 8,

'JJR': 9,

'JJS': 10,

'LS': 11,

'MD': 12,

'NN': 13,

'NNS': 14,

'NNP': 15,

'NNPS': 16,

'PDT': 17,

'POS': 18,

'PRP': 19,

'PRP$': 20,

'RB': 21,

'RBR': 22,

'RBS': 23,

'RP': 24,

'SYM': 25,

'TO': 26,

'UH': 27,

'VB': 28,

'VBD': 29,

'VBG': 30,

'VBN': 31,

'VBP': 32,

'VBZ': 33,

'WDT': 34,

'WP': 35,

'WP$': 36,

'WRB': 37

}

1. <https://www.youtube.com/watch?v=nbOYbvGvmwI> [↑](#footnote-ref-1)
2. <https://www.nltk.org/> [↑](#footnote-ref-2)
3. <https://rstudio.github.io/reticulate/> [↑](#footnote-ref-3)
4. <https://nlp.stanford.edu/software/tagger.shtml> [↑](#footnote-ref-4)
5. <https://en.wikipedia.org/wiki/Tf%E2%80%93idf> [↑](#footnote-ref-5)
6. <https://www.nltk.org/book/ch05.html> [↑](#footnote-ref-6)
7. <https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html> [↑](#footnote-ref-7)
8. <https://pythonspot.com/nltk-stemming/> [↑](#footnote-ref-8)
9. <https://github.com/jeff1evesque/ist-736-hw/blob/master/data/sample-sentiment.csv> [↑](#footnote-ref-9)
10. <https://github.com/jeff1evesque/ist-736-hw/blob/master/hw1/assigment.py> [↑](#footnote-ref-10)
11. <https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html> [↑](#footnote-ref-11)
12. <https://developer.twitter.com/en/docs/basics/counting-characters.html> [↑](#footnote-ref-12)