In this notebook, I explore the data given and discover a technique for accurately predicting the document classifiation for the anonymized text

First we import the CSV and take an initial look at the data

In [5]:
```
%matplotlib inline
import pandas as pd
import numpy as np
import random
import collections
import time
import pickle
import matplotlib.pyplot as plt

df = pd.read_csv("../shuffled-full-set-hashed.csv", header=None)
df.head()
```

Out[5]:

	0	1
0	DELETION OF INTEREST	e04a09c87692 d6b72e591b91 5d066f0246f1 ed41171...
1	RETURNED CHECK	a3b334c6eefd be95012ebf2b 41d67080e078 ff1c26e...
2	BILL	586242498a88 9ccf259ca087 54709b24b45f 6bf9c0c...
3	BILL	cd50e861f48b 6ca2dd348663 d38820625542 f077614...
4	BILL	9db5536263d8 1c303d15eb65 3f89b4673455 b73e657...

In [11]:
```
print("Rows %d"%len(df[0]))

print("\nMost Common Categories:")
cntr = collections.Counter(df[0])
cntr.most_common()
```

```
Rows 62204

Most Common Categories:
```

Out[11]:
```
[('BILL', 18968),
 ('POLICY CHANGE', 10627),
 ('CANCELLATION NOTICE', 9731),
 ('BINDER', 8973),
 ('DELETION OF INTEREST', 4826),
 ('REINSTATEMENT NOTICE', 4368),
 ('DECLARATION', 968),
 ('CHANGE ENDORSEMENT', 889),
 ('RETURNED CHECK', 749),
 ('EXPIRATION NOTICE', 734),
 ('NON-RENEWAL NOTICE', 624),
 ('BILL BINDER', 289),
 ('INTENT TO CANCEL NOTICE', 229),
 ('APPLICATION', 229)]
```

Not a very even distribution of categories, what about the distribution of word counts within each document

In [12]:
```
def word_count(words):
    return len(words.split(' ')) if type(words) == str else 0
df['word_count'] = [word_count(x) for x in df[1]]
df.head()
```

Out[12]:

	0	1	word_count
0	DELETION OF INTEREST	e04a09c87692 d6b72e591b91 5d066f0246f1 ed41171...	465
1	RETURNED CHECK	a3b334c6eefd be95012ebf2b 41d67080e078 ff1c26e...	403
2	BILL	586242498a88 9ccf259ca087 54709b24b45f 6bf9c0c...	185
3	BILL	cd50e861f48b 6ca2dd348663 d38820625542 f077614...	337
4	BILL	9db5536263d8 1c303d15eb65 3f89b4673455 b73e657...	546

In [13]:
```
df.word_count.describe()
```

Out[13]:
```
count    62204.000000
mean       334.148479
std        330.217525
min          0.000000
25%        148.000000
50%        252.000000
75%        402.000000
max       9076.000000
Name: word_count, dtype: float64
```

A slightly more predictable word count, a few hundred in most documents with some clear outliers,

What about the words themselves?

In [14]:
```
def split_or_empty(words):
    return words.split(' ') if type(words) == str else []

word_lst = [split_or_empty(x) for x in df[1]]

all_wrds = [word for lst in word_lst for word in lst]
set_wrds = set(all_wrds)
print("Total words %d"%df.word_count.sum())
print("Unique words %d"%len(set_wrds))
```

```
Total words 20785372
Unique words 1037934
```

That's a lot of unique words, and for 62K lines, not a lot of documents, but lets assume that there are a lot of stop words and since these are personalized documents a lot of names, addresses, etc

In [17]:
```
wrd_cntr = collections.Counter(all_wrds)
common_words = wrd_cntr.most_common()

print("Num common words %d"%len(common_words))
```

```
Num common words 1037934
```

In [18]:

```
freq_lst = [x[1] for x in common_words]
freq_arr = np.array(freq_lst)
print("Frequency Distribution of Words: (Mean, Variance) %f %f"%(freq_ar
r.mean(), freq_arr.var()))
```

```
Frequency Distribution of Words: (Mean, Variance) 20.025716 846230.6574
44
```

In [19]:

```
vocab = [x[0] for x in common_words if x[1] > 5 and x[1] < 10000]
vocab_set = set(vocab)
def keep_vocab(words):
    return set(split_or_empty(words)) & vocab_set

df['words_in_vocab'] = [len(keep_vocab(x)) for x in df[1]]
df.words_in_vocab.describe()
```

Out[19]:

```
count    62204.000000
mean       101.385120
std         86.443821
min          0.000000
25%         48.000000
50%         77.000000
75%        128.000000
max       1610.000000
Name: words_in_vocab, dtype: float64
```

In [20]:

```
df.head()
```

Out[20]:

	0	1	word_count	words_in_vocab
0	DELETION OF INTEREST	e04a09c87692 d6b72e591b91 5d066f0246f1 ed41171...	465	139
1	RETURNED CHECK	a3b334c6eefd be95012ebf2b 41d67080e078 ff1c26e...	403	117
2	BILL	586242498a88 9ccf259ca087 54709b24b45f 6bf9c0c...	185	36
3	BILL	cd50e861f48b 6ca2dd348663 d38820625542 f077614...	337	130
4	BILL	9db5536263d8 1c303d15eb65 3f89b4673455 b73e657...	546	131

So from this quick exercise, it seems like we can get a more manageable vocabulary by focusing on the words which are not too common and not too uncommon

In [21]:
```
df_nona = df.dropna()
df['words_in_vocab'] = [' '.join(keep_vocab(x)) for x in df[1]]
df.head()
```

Out[21]:

	0	1	word_count	words_in_vocab
0	DELETION OF INTEREST	e04a09c87692 d6b72e591b91 5d066f0246f1 ed41171...	465	743b314e5665 b2c878a75d7e 44d3870bca21 4e9eb06...
1	RETURNED CHECK	a3b334c6eefd be95012ebf2b 41d67080e078 ff1c26e...	403	29503e65a644 b2c878a75d7e 1850801b9c05 ea05dcb...
2	BILL	586242498a88 9ccf259ca087 54709b24b45f 6bf9c0c...	185	0ad17934ee05 f1424da4e7d6 e0a34e168ea4 c85a9f2...
3	BILL	cd50e861f48b 6ca2dd348663 d38820625542 f077614...	337	011113964d37 dafbb201715e 69c87281a156 83da9eb...
4	BILL	9db5536263d8 1c303d15eb65 3f89b4673455 b73e657...	546	0025e6b23cc5 d671855584fd ba943a1c3175 dfa88dd...

So using this quick attempt to remove stop words and very infrequent words, let's move on to the prediction.

My first attempt attempt was to use a Naive Bayes approach using TF-IDF, thinking that these words (the ones remaining), might be useful for categorization

In [22]:
```
from nltk.tokenize import word_tokenize
from nltk import pos_tag
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from sklearn.preprocessing import LabelEncoder
from collections import defaultdict
from nltk.corpus import wordnet as wn
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn import model_selection, naive_bayes, svm
from sklearn.metrics import accuracy_score

train_x, test_x, train_y, test_y = model_selection.train_test_split(df.w
ords_in_vocab, df[0],test_size=0.2)
```

In [30]:
```
np.random.seed(5020)

# Encode the Prediction Label, so that we're using integers and not stri
ngs
encoder = LabelEncoder()
enc_train_y = encoder.fit_transform(train_y)
enc_test_y = encoder.fit_transform(test_y)
```

In [31]:
```
df.head()
```

Out[31]:

	0	1	word_count	words_in_vocab
0	DELETION OF INTEREST	e04a09c87692 d6b72e591b91 5d066f0246f1 ed41171...	465	fbe7c05e32d5 1807f8910862 3102eeb23202 1fa87d6...
1	RETURNED CHECK	a3b334c6eefd be95012ebf2b 41d67080e078 ff1c26e...	403	1357209fd44f 687214cd0acb 7d4501e8b694 31cbd98...
2	BILL	586242498a88 9ccf259ca087 54709b24b45f 6bf9c0c...	185	b834a58b85b9 2e182c67811b 6753b57205cb 3d877a3...
3	BILL	cd50e861f48b 6ca2dd348663 d38820625542 f077614...	337	034e2d7f187e c8207fafe699 7860028b1d17 a4ffd27...
4	BILL	9db5536263d8 1c303d15eb65 3f89b4673455 b73e657...	546	1807f8910862 6c941621f20f a100eb50abec ad5f00d...

In [32]:
```
tfidf_vect = TfidfVectorizer(ngram_range=(1,3))
tfidf_vect.fit(df.words_in_vocab)
```

Out[32]:
```
TfidfVectorizer(analyzer='word', binary=False, decode_error='strict',
                dtype=<class 'numpy.float64'>, encoding='utf-8',
                input='content', lowercase=True, max_df=1.0, max_featur
es=None,
                min_df=1, ngram_range=(1, 3), norm='l2', preprocessor=N
one,
                smooth_idf=True, stop_words=None, strip_accents=None,
                sublinear_tf=False, token_pattern='(?u)\\b\\w\\w+\\b',
                tokenizer=None, use_idf=True, vocabulary=None)
```

In [36]:
```
train_x_tfidf = tfidf_vect.transform(train_x)
test_x_tfidf = tfidf_vect.transform(test_x)
```

In [37]:
```
# fit the training dataset on the NB classifier
nb = naive_bayes.MultinomialNB()
nb.fit(train_x_tfidf, train_y)

# predict the labels on validation dataset
predictions_nb = nb.predict(test_x_tfidf)

# Use accuracy_score function to get the accuracy
print("Naive Bayes Accuracy Score -> ",accuracy_score(predictions_nb, te
st_y)*100)
```

```
Naive Bayes Accuracy Score ->  65.91110039385902
```

In [28]:
```
vectorizer = TfidfVectorizer(max_df=0.5, max_features=10000,
                             min_df=2, use_idf=True, ngram_range=(1,3))
train_x, test_x, train_y, test_y = model_selection.train_test_split(df_n
ona[1], df_nona[0],test_size=0.2)
```

In [31]:
```
encoder = LabelEncoder()
enc_train_y = encoder.fit_transform(train_y)
enc_test_y = encoder.fit_transform(test_y)
```

In [32]:
```
vectorizer.fit(df_nona[1])
```

Out[32]:
```
TfidfVectorizer(analyzer='word', binary=False, decode_error='strict',
                dtype=<class 'numpy.float64'>, encoding='utf-8',
                input='content', lowercase=True, max_df=0.5, max_featur
es=10000,
                min_df=2, ngram_range=(1, 3), norm='l2', preprocessor=N
one,
                smooth_idf=True, stop_words=None, strip_accents=None,
                sublinear_tf=False, token_pattern='(?u)\\b\\w\\w+\\b',
                tokenizer=None, use_idf=True, vocabulary=None)
```

In [33]:
```
train_x_tfidf = vectorizer.transform(train_x)
test_x_tfidf = vectorizer.transform(test_x)
```

In [44]:
```
# fit the training dataset on the NB classifier
nb = naive_bayes.MultinomialNB()
nb.fit(train_x_tfidf, train_y)

# predict the labels on validation dataset
predictions_nb = nb.predict(test_x_tfidf)

# Use accuracy_score function to get the accuracy
print("Naive Bayes Accuracy Score -> ",accuracy_score(predictions_nb, te
st_y)*100)
```

```
Naive Bayes Accuracy Score ->  76.93050193050193
```

So even using NLTK's stopword system, TF-IDF vectors, and N-Grams of up to 3 words, we haven't cracked 80% accuracy

Next I move on to a combination of linear techniques

In [34]:
```
from sklearn.decomposition import TruncatedSVD
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import Normalizer
from sklearn.neighbors import KNeighborsClassifier
```

In [35]:
```
svd = TruncatedSVD(100)
lsa = make_pipeline(svd, Normalizer(copy=False))

# Run SVD on the training data, then project the training data.
train_x_lsa = lsa.fit_transform(train_x_tfidf)

explained_variance = svd.explained_variance_ratio_.sum()
print("Explained variance of the SVD step: %2f %".format(int(explained_v
ariance * 100)))

test_x_lsa = lsa.transform(test_x_tfidf)
```

```
Explained variance of the SVD step: %2f %
```

In [36]:
```
test_x_lsa.shape
```

Out[36]:
```
(12432, 100)
```

So here I'm making a big matrix of the training data's TF-IDF values, and then using Singular Value Decomposition to factor that matrix. This makes it easy to reduce the number of dimensions (since with up to 3 words as a feature, there are a LOT of dimensions).

Now that I've done SVD, I can project my TF-IDF vectors into this reduced vector space and use a simple categorizing algorithm (K-Nearest Neighbor) to train categories.

In [37]:
```
num_cats = len(encoder.classes_)
```

In [38]:
```
# Build a k-NN classifier. Use k = 5 (majority wins), the cosine distanc
e,
# and brute-force calculation of distances.
knn_lsa = KNeighborsClassifier(n_neighbors=num_cats, algorithm='brute',
metric='cosine')
knn_lsa.fit(train_x_lsa, enc_train_y)

# Classify the test vectors.
p = knn_lsa.predict(test_x_lsa)
```

In [57]:
```
# Measure accuracy
numRight = 0;
for i in range(0,len(p)):
    if p[i] == enc_test_y[i]:
        numRight += 1

print("  (%d / %d) correct - %.2f%%" % (numRight, len(enc_test_y), float
(numRight) / float(len(enc_test_y)) * 100.0))
```

```
  (10273 / 12432) correct - 82.63%
```

Now that we've broken into the 80's with a respectable 82.6% accuracy, we can move on to pushing these models up to a production environment.

My next steps are to train a LSTM neural network on these word vectors. I tried to use Word2Vec to create new word vectors but I believe the relatively small size of the data makes this difficult, and that the linear nature of SVD allows these vectors to perform better

However, there is clearly a use of the words that involves order that is important, and it would behoove any data scientist to at least try an LSTM and see if the accuracy could be improved

In [128]:
```
encoder = LabelEncoder()
full_enc_y = encoder.fit_transform(full_train_y)
```

In [133]:
```
encoder.inverse_transform([7, 13, 1, 6, 6])
```

Out[133]:
```
array(['DELETION OF INTEREST', 'RETURNED CHECK', 'BILL', 'DECLARATION',
       'DECLARATION'], dtype=object)
```

In [63]:
```
vectorizer
```

Out[63]:
```
TfidfVectorizer(analyzer='word', binary=False, decode_error='strict',
                dtype=<class 'numpy.float64'>, encoding='utf-8',
                input='content', lowercase=True, max_df=0.5, max_featur
es=10000,
                min_df=2, ngram_range=(1, 3), norm='l2', preprocessor=N
one,
                smooth_idf=True, stop_words=None, strip_accents=None,
                sublinear_tf=False, token_pattern='(?u)\\b\\w\\w+\\b',
                tokenizer=None, use_idf=True, vocabulary=None)
```

In [64]:
```
full_train_x_tfidf = vectorizer.transform(full_train_x)
```

In [65]:
```
svd = TruncatedSVD(100)
lsa = make_pipeline(svd, Normalizer(copy=False))

# Run SVD on the training data, then project the training data.
full_train_x_lsa = lsa.fit_transform(full_train_x_tfidf)
```

In [67]:
```
knn_lsa = KNeighborsClassifier(n_neighbors=num_cats, algorithm='brute',
metric='cosine')
knn_lsa.fit(full_train_x_lsa, full_enc_y)
```

Out[67]:
```
KNeighborsClassifier(algorithm='brute', leaf_size=30, metric='cosine',
                     metric_params=None, n_jobs=None, n_neighbors=14, p
=2,
                     weights='uniform')
```