In [ ]:
```
#Group 8 - Wk3 - TFIDF and Ngrams
#Group Member: Athena Zhang, Pratheek Praveen Kumar, Weifeng Li, Wenke Yu, Ziq
iao Wei
```

Import packages

Make sure you installed ***sklearn***, ***matplotlib*** and ***numpy*** if you use your local machine

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
import sklearn
from sklearn import datasets
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import confusion_matrix, precision_score, precision_recal
l_curve, recall_score, f1_score
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
from sklearn.metrics.pairwise import cosine_similarity
```

Prepare dataset (same as last time)

The 20 newsgroups text dataset: [Details (https://scikit-learn.org/0.19/datasets/twenty_newsgroups.html)](https://scikit-learn.org/0.19/datasets/twenty_newsgroups.html)

In [2]:
```
dataset = sklearn.datasets.fetch_20newsgroups()
class Dataset:
  def __init__(self, dataset, start_idx, end_idx):
    self.data = dataset.data[start_idx:end_idx]
    self.labels = dataset.target[start_idx:end_idx]
    self.vecs = None


def split_dataset(dataset, train_rate=0.7):
  data_size = len(dataset.data)
  train_last_idx = int(train_rate * data_size)
  train = Dataset(dataset, 0, train_last_idx)
  test = Dataset(dataset, train_last_idx, data_size)
  return train, test

train, test = split_dataset(dataset)
print('train data size:', len(train.data))
print('test data size:', len(test.data))
```

```
train data size: 7919
test data size: 3395
```

NGrams

Convert a list of text documents to a matrix of token frequencies ([Details (http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html)](http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html))

In [3]:
```
unigram = CountVectorizer(ngram_range=(1,1))
bigram = CountVectorizer(ngram_range=(2,2))
trigram = CountVectorizer(ngram_range=(3,3))
#fourgram = CountVectorizer(ngram_range=(4,4))
combined = CountVectorizer(ngram_range=(1,3))
vectorizers = [unigram, bigram, trigram, combined]
print("Fitting vectorizers")
[vectorizer.fit(train.data) for vectorizer in vectorizers]
```

```
Fitting vectorizers
```

Out[3]:
```
[CountVectorizer(analyzer='word', binary=False, decode_error='strict',
                 dtype=<class 'numpy.int64'>, encoding='utf-8', input='conten
t',
                 lowercase=True, max_df=1.0, max_features=None, min_df=1,
                 ngram_range=(1, 1), preprocessor=None, stop_words=None,
                 strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
                 tokenizer=None, vocabulary=None),
 CountVectorizer(analyzer='word', binary=False, decode_error='strict',
                 dtype=<class 'numpy.int64'>, encoding='utf-8', input='conten
t',
                 lowercase=True, max_df=1.0, max_features=None, min_df=1,
                 ngram_range=(2, 2), preprocessor=None, stop_words=None,
                 strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
                 tokenizer=None, vocabulary=None),
 CountVectorizer(analyzer='word', binary=False, decode_error='strict',
                 dtype=<class 'numpy.int64'>, encoding='utf-8', input='conten
t',
                 lowercase=True, max_df=1.0, max_features=None, min_df=1,
                 ngram_range=(3, 3), preprocessor=None, stop_words=None,
                 strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
                 tokenizer=None, vocabulary=None),
 CountVectorizer(analyzer='word', binary=False, decode_error='strict',
                 dtype=<class 'numpy.int64'>, encoding='utf-8', input='conten
t',
                 lowercase=True, max_df=1.0, max_features=None, min_df=1,
                 ngram_range=(1, 3), preprocessor=None, stop_words=None,
                 strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
                 tokenizer=None, vocabulary=None)]
```

See vocabulary size

Q: Which one has the largest vocabulary size, **unigram**, **bigram**, **trigram**, or **combined**?

In [4]:
```
for vectorizer in vectorizers:
  print('Vocabulary Size:', len(vectorizer.vocabulary_))
```

```
Vocabulary Size: 107212
Vocabulary Size: 825425
Vocabulary Size: 1514525
Vocabulary Size: 2447162
```

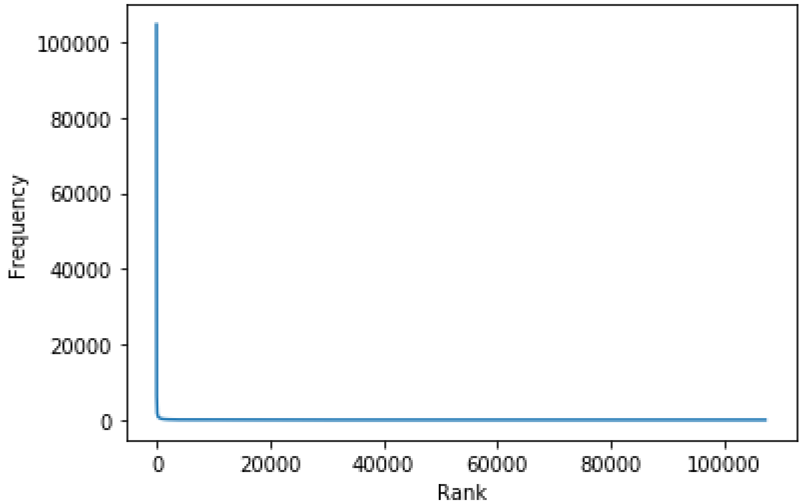
See vocabulary distribution

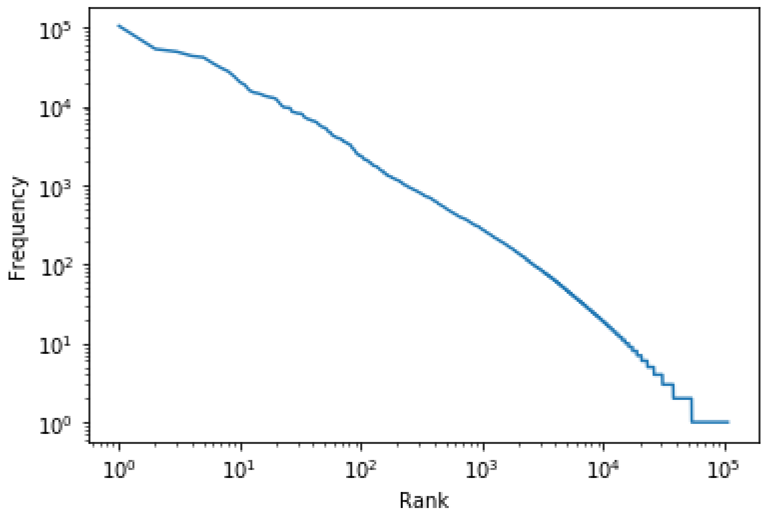
Q: Do you remember the name of the law?

In [5]:
```
def show_distribution(vectorizer, train):
  vecs = vectorizer.transform(train.data)
  sum_mat = np.sum(vecs, axis=0)
  freqs = np.sort(sum_mat).T[::-1]
  plt.plot(list(range(1, sum_mat.shape[1] + 1)), freqs)
  plt.xlabel('Rank')
  plt.ylabel('Frequency')
  plt.show()

  plt.loglog(list(range(1, sum_mat.shape[1] + 1)), freqs)
  plt.xlabel('Rank')
  plt.ylabel('Frequency')
  plt.show()
```

In [6]:
```
show_distribution(unigram, train) # try bigram, trigram and combined as well
```





Convert word (ngram) to index and vice versa

In [7]:
```
def ngram2idx(ngram, vocab_dict):
  index = vocab_dict[ngram] if ngram in vocab_dict.keys() else 'Not Found'
  print(ngram, ' -> ', index)


def idx2ngram(index, vocabs):
  ngram = vocabs[index] if 0 <= index < len(vocabs) else 'Not Found'
  print(index, ' -> ', ngram)
```

In [8]:
```
vectorizer = unigram # change to bigram or trigram
vocab_dict = vectorizer.vocabulary_
vocabs = vectorizer.get_feature_names()

ngram2idx('we are', vocab_dict)
idx2ngram(783807, vocabs)

ngram2idx('to microsoft', vocab_dict)
idx2ngram(736413, vocabs)
```

```
we are  ->  Not Found
783807  ->  Not Found
to microsoft  ->  Not Found
736413  ->  Not Found
```

Convert sentence to vector

In [9]:
```
def sentence2vec(sentence, vectorizer):
  vec = vectorizer.transform([sentence])
  vocabs = vectorizer.get_feature_names()
  print('\"', sentence, '\" -> ')
  print(vec)
  for idx in vec.indices:
    print(idx, vocabs[idx])
  print()
```

In [10]:
```
for vectorizer in vectorizers:
  sentence2vec('We are going to microsoft', vectorizer)
```

```
" We are going to microsoft " ->
  (0, 23184)	1
  (0, 48917)	1
  (0, 67270)	1
  (0, 95302)	1
  (0, 101951)	1
23184 are
48917 going
67270 microsoft
95302 to
101951 we

" We are going to microsoft " ->
  (0, 109903)	1
  (0, 328698)	1
  (0, 736413)	1
  (0, 783807)	1
109903 are going
328698 going to
736413 to microsoft
783807 we are

" We are going to microsoft " ->
  (0, 171354)	1
  (0, 1424292)	1
171354 are going to
1424292 we are going

" We are going to microsoft " ->
  (0, 300895)	1
  (0, 304432)	1
  (0, 304443)	1
  (0, 919583)	1
  (0, 919995)	1
  (0, 1336821)	1
  (0, 2147049)	1
  (0, 2166280)	1
  (0, 2309753)	1
  (0, 2309963)	1
  (0, 2310052)	1
300895 are
304432 are going
304443 are going to
919583 going
919995 going to
1336821 microsoft
2147049 to
2166280 to microsoft
2309753 we
2309963 we are
2310052 we are going
```

TFIDF Weights

Tutorial with unigram

Convert a collection of raw documents to a matrix of TF-IDF features. (Details (https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html))

In [11]:
```
example_docs=[
       "One Cent, Two Cents, Old Cent, New Cent: All About Money (Cat in the H
at's Learning Library",
       "Inside Your Outside: All About the Human Body (Cat in the Hat's Learni
ng Library)",
       "Oh, The Things You Can Do That Are Good for You: All About Staying Hea
lthy (Cat in the Hat's Learning Library)",
       "On Beyond Bugs: All About Insects (Cat in the Hat's Learning Library)"
,
       "There's No Place Like Space: All About Our Solar System (Cat in the Ha
t's Learning Library)"
      ]
```

In [12]:
```
bow =  CountVectorizer()
bow.fit(example_docs)
```

Out[12]:
```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
                dtype=<class 'numpy.int64'>, encoding='utf-8', input='conten
t',
                lowercase=True, max_df=1.0, max_features=None, min_df=1,
                ngram_range=(1, 1), preprocessor=None, stop_words=None,
                strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
                tokenizer=None, vocabulary=None)
```

In [13]:
```
bowbigram = CountVectorizer(ngram_range=(1,2))
bowbigram.fit(example_docs)
```

Out[13]:
```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
                dtype=<class 'numpy.int64'>, encoding='utf-8', input='conten
t',
                lowercase=True, max_df=1.0, max_features=None, min_df=1,
                ngram_range=(1, 2), preprocessor=None, stop_words=None,
                strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
                tokenizer=None, vocabulary=None)
```

In [14]:
```
Y = bow.transform(example_docs)
Y1 = bowbigram.transform(example_docs)
```

In [15]:
```
print("Unigram\n")
print(Y.toarray())
print("Unigram+Bigram\n")
print(Y1.toarray())
```

```
Unigram

[[1 1 0 0 0 0 0 1 3 1 0 0 0 1 0 0 1 0 0 1 1 0 1 1 0 0 1 0 1 0 0 0 0 0 0 0
  0 1 0 0 1 0 0]
 [1 1 0 0 1 0 0 1 0 0 0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
  0 2 0 0 0 0 1]
 [1 1 1 0 0 0 1 1 0 0 1 1 1 1 1 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0
  1 2 0 1 0 2 0]
 [1 1 0 1 0 1 0 1 0 0 0 0 0 1 0 0 1 1 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
  0 1 0 0 0 0 0]
 [1 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 1 1 1 0 0 1 0 0 0 0 1 0 1 1 1 0 1
  0 1 1 0 0 0 0]]
Unigram+Bigram

[[1 0 1 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 3 1 1 1 1 1 0 0 0 0 0 0 1 1 0 0
  0 0 1 1 0 0 0 0 1 1 1 0 0 1 1 1 1 0 0 0 0 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0]
 [1 0 0 0 0 1 1 1 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
  1 1 1 1 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0
  0 0 0 0 0 0 0 2 1 1 0 0 0 0 0 0 0 0 0 0 1 1]
 [1 0 0 0 1 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
  0 0 1 1 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  0 1 1 0 0 1 1 2 1 0 1 0 0 1 1 0 0 2 1 1 0 0]
 [1 1 0 0 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
  0 0 1 1 1 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [1 0 0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
  0 0 1 1 0 0 0 0 1 1 1 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 1 1 0 0 1 1 1 1 1
  1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 0 0 0 0 0 0 0]]
```

In [16]:
```
tfidf = TfidfVectorizer()
tfidf.fit(example_docs)
```

Out[16]:
```
TfidfVectorizer(analyzer='word', binary=False, decode_error='strict',
                dtype=<class 'numpy.float64'>, encoding='utf-8',
                input='content', lowercase=True, max_df=1.0, max_features=Non
e,
                min_df=1, ngram_range=(1, 1), norm='l2', preprocessor=None,
                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 [17]:
```
X = tfidf.transform(example_docs)
print(X.toarray())
```

```
[[0.1161985  0.1161985  0.         0.         0.         0.
  0.         0.1161985  0.73156679 0.2438556  0.         0.
  0.         0.1161985  0.         0.         0.1161985  0.
  0.         0.1161985  0.1161985  0.         0.2438556  0.2438556
  0.         0.         0.2438556  0.         0.2438556  0.
  0.         0.         0.         0.         0.         0.
  0.         0.1161985  0.         0.         0.2438556  0.
  0.        ]
 [0.17402264 0.17402264 0.         0.         0.36520606 0.
  0.         0.17402264 0.         0.         0.         0.
  0.         0.17402264 0.         0.36520606 0.17402264 0.
  0.36520606 0.17402264 0.17402264 0.         0.         0.
  0.         0.         0.         0.         0.         0.
  0.36520606 0.         0.         0.         0.         0.
  0.         0.34804529 0.         0.         0.         0.
  0.36520606]
 [0.11731593 0.11731593 0.24620066 0.         0.         0.
  0.24620066 0.11731593 0.         0.         0.24620066 0.24620066
  0.24620066 0.11731593 0.24620066 0.         0.11731593 0.
  0.         0.11731593 0.11731593 0.         0.         0.
  0.         0.24620066 0.         0.         0.         0.
  0.         0.         0.         0.         0.24620066 0.
  0.24620066 0.23463186 0.         0.24620066 0.         0.49240131
  0.        ]
 [0.19757794 0.19757794 0.         0.4146395  0.         0.4146395
  0.         0.19757794 0.         0.         0.         0.
  0.         0.19757794 0.         0.         0.19757794 0.4146395
  0.         0.19757794 0.19757794 0.         0.         0.
  0.         0.         0.         0.4146395  0.         0.
  0.         0.         0.         0.         0.         0.
  0.         0.19757794 0.         0.         0.         0.
  0.        ]
 [0.15208639 0.15208639 0.         0.         0.         0.
  0.         0.15208639 0.         0.         0.         0.
  0.         0.15208639 0.         0.         0.15208639 0.
  0.         0.15208639 0.15208639 0.31917038 0.         0.
  0.31917038 0.         0.         0.         0.         0.31917038
  0.         0.31917038 0.31917038 0.31917038 0.         0.31917038
  0.         0.15208639 0.31917038 0.         0.         0.
  0.        ]]
```

In [18]:
```
bog =  CountVectorizer()
bog.fit(example_docs)
```

Out[18]:
```
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
                dtype=<class 'numpy.int64'>, encoding='utf-8', input='conten
t',
                lowercase=True, max_df=1.0, max_features=None, min_df=1,
                ngram_range=(1, 1), preprocessor=None, stop_words=None,
                strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
                tokenizer=None, vocabulary=None)
```

In [19]:
```
Y = bog.transform(example_docs)
```

In [20]:
```
print(Y.toarray())
```

```
[[1 1 0 0 0 0 0 1 3 1 0 0 0 1 0 0 1 0 0 1 1 0 1 1 0 0 1 0 1 0 0 0 0 0 0 0
  0 1 0 0 1 0 0]
 [1 1 0 0 1 0 0 1 0 0 0 0 0 1 0 1 1 0 1 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
  0 2 0 0 0 0 1]
 [1 1 1 0 0 0 1 1 0 0 1 1 1 1 1 0 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0
  1 2 0 1 0 2 0]
 [1 1 0 1 0 1 0 1 0 0 0 0 0 1 0 0 1 1 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
  0 1 0 0 0 0 0]
 [1 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 1 1 1 0 0 1 0 0 0 0 1 0 1 1 1 0 1
  0 1 1 0 0 0 0]]
```

In [21]:
```
print(cosine_similarity(X))
```

```
[[1.         0.18199053 0.12268742 0.18366608 0.14137768]
 [0.18199053 1.         0.22457191 0.30944732 0.23819829]
 [0.12268742 0.22457191 1.         0.20861136 0.16057941]
 [0.18366608 0.30944732 0.20861136 1.         0.24039133]
 [0.14137768 0.23819829 0.16057941 0.24039133 1.        ]]
```

In [22]:
```
print(cosine_similarity(Y))
```

```
[[1.         0.46915743 0.37532595 0.48154341 0.41702883]
 [0.46915743 1.         0.55       0.64951905 0.5625    ]
 [0.37532595 0.55       1.         0.51961524 0.45      ]
 [0.48154341 0.64951905 0.51961524 1.         0.57735027]
 [0.41702883 0.5625     0.45       0.57735027 1.        ]]
```

Create TFIDF vectors for newsgroup articles

In [23]:
```
tfidf = TfidfVectorizer()
tfidf.fit(train.data)
```

Out[23]:
```
TfidfVectorizer(analyzer='word', binary=False, decode_error='strict',
                dtype=<class 'numpy.float64'>, encoding='utf-8',
                input='content', lowercase=True, max_df=1.0, max_features=Non
e,
                min_df=1, ngram_range=(1, 1), norm='l2', preprocessor=None,
                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 [24]:
```
print('the', tfidf.idf_[tfidf.vocabulary_['the']])
print('man', tfidf.idf_[tfidf.vocabulary_['man']])
print('microsoft', tfidf.idf_[tfidf.vocabulary_['microsoft']])
```

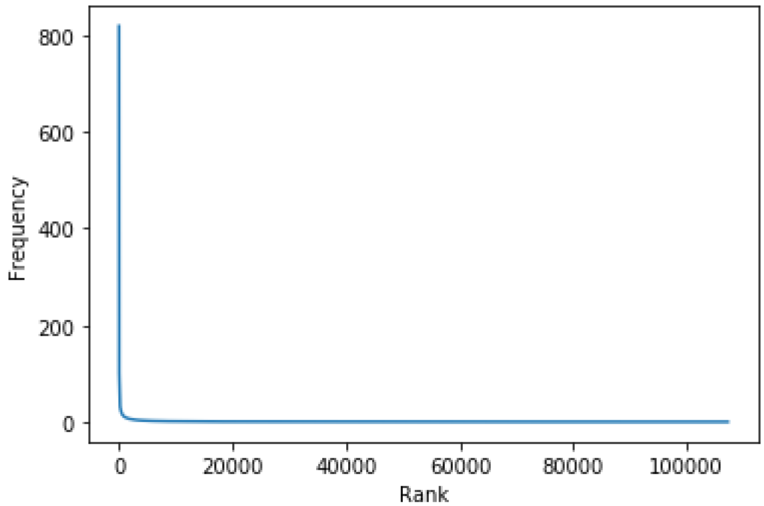
```
the 1.0693987980198238
man 3.7645403890569527
microsoft 5.214972550010716
```

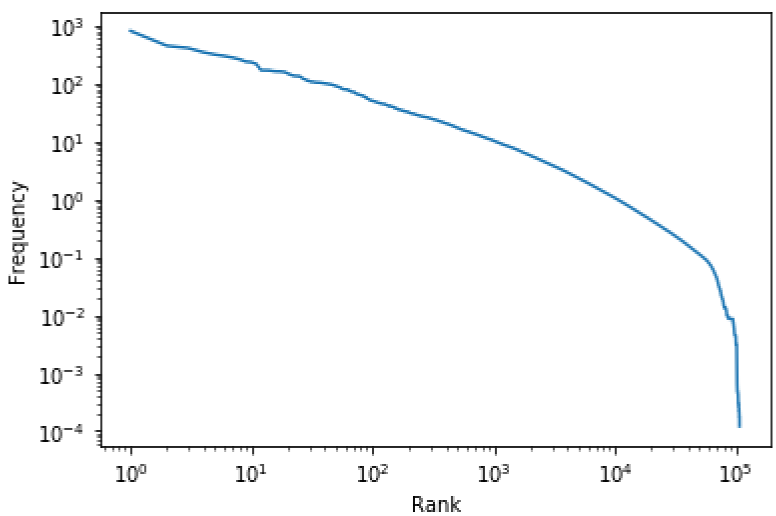
In [25]:
```
print('Vocabulary Size:', len(tfidf.vocabulary_))
sentence2vec('We are going to microsoft', tfidf)
sentence2vec('We are going to microsoft', unigram)
```

```
Vocabulary Size: 107212
" We are going to microsoft " ->
  (0, 101951)	0.34417196761665414
  (0, 95302)	0.1622831702359858
  (0, 67270)	0.765634619765414
  (0, 48917)	0.4590925435788398
  (0, 23184)	0.24134517772688163
101951 we
95302 to
67270 microsoft
48917 going
23184 are

" We are going to microsoft " ->
  (0, 23184)	1
  (0, 48917)	1
  (0, 67270)	1
  (0, 95302)	1
  (0, 101951)	1
23184 are
48917 going
67270 microsoft
95302 to
101951 we
```

In [26]:
```
show_distribution(tfidf, train)
```

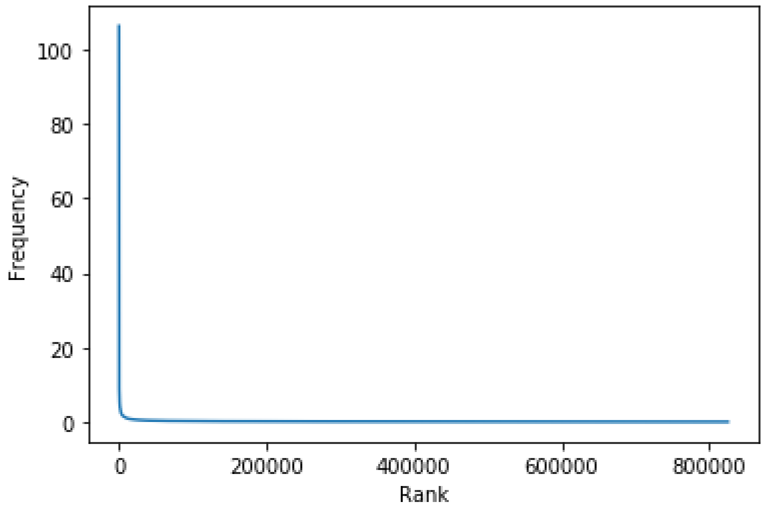


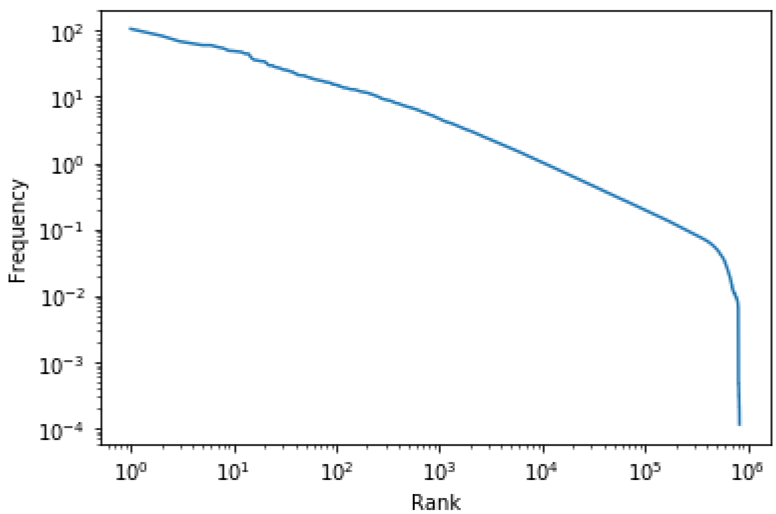


Tutorial with bigram and trigram

In [27]:
```
tfidf_bigram = TfidfVectorizer(ngram_range=(2,2))
tfidf_trigram = TfidfVectorizer(ngram_range=(3,3))
tfidf_combined = TfidfVectorizer(ngram_range=(1,3))
tfidf_bigram.fit(train.data)
tfidf_trigram.fit(train.data)
tfidf_combined.fit(train.data)
tfidf_vectorizers = [tfidf, tfidf_bigram, tfidf_trigram, tfidf_combined]
```

In [28]:
```
show_distribution(tfidf_bigram, train)
```





In [29]:
```
print('Vocabulary Size:', len(tfidf_bigram.vocabulary_))
sentence2vec('We are going to microsoft', tfidf_bigram)
print('Vocabulary Size:', len(tfidf_combined.vocabulary_))
sentence2vec('We are going to microsoft', tfidf_combined)
```

```
Vocabulary Size: 825425
" We are going to microsoft " ->
  (0, 783807)	0.37403683743500654
  (0, 736413)	0.7336470123086184
  (0, 328698)	0.32571753506505674
  (0, 109903)	0.46450682763916384
783807 we are
736413 to microsoft
328698 going to
109903 are going

Vocabulary Size: 2447162
" We are going to microsoft " ->
  (0, 2310052)	0.46430036807785535
  (0, 2309963)	0.2594895230780238
  (0, 2309753)	0.15107675883552044
  (0, 2166280)	0.5089704924175021
  (0, 2147049)	0.07123536394490301
  (0, 1336821)	0.33608081915392957
  (0, 919995)	0.225967817533126
  (0, 919583)	0.20152197161710347
  (0, 304443)	0.33233605156914814
  (0, 304432)	0.3222534336381154
  (0, 300895)	0.10594020037149467
2310052 we are going
2309963 we are
2309753 we
2166280 to microsoft
2147049 to
1336821 microsoft
919995 going to
919583 going
304443 are going to
304432 are going
300895 are
```

Excercise 1

1. Find two documents that are very close in the Bag-of-word space but very far apart in the TFIDF space.
2. Find two documents that are very close in the TFIDF space but very far apart in the Bag-of-word space.

In [30]:
```
# Find two documents that are very close in the Bag-of-word space but very far apart in the TFIDF space
tfidf_vectors = tfidf.transform(train.data)
```

In [31]:
```
bow = CountVectorizer()
bow.fit(train.data)
bow_vectors = bow.transform(train.data)
```

In [32]:
```
cos_tfidf = cosine_similarity(tfidf_vectors)
cos_bow = cosine_similarity(bow_vectors)
```

In [33]:
```
diff = cos_bow - cos_tfidf
np.where(diff == np.max(diff))
```

Out[33]:
```
(array([ 524, 2397], dtype=int64), array([2397,  524], dtype=int64))
```

In [34]:
```
# Find two documents that are very close in the TFIDF space but very far apart in the Bag-of-word space.
diff = cos_tfidf - cos_bow
np.where(diff==np.max(diff))
```

Out[34]:
```
(array([ 726, 5339], dtype=int64), array([5339,  726], dtype=int64))
```

Classification with MNB

- function for training and testing given vectorizer, classifier, return eval

In [35]:
```
def classification(vectorizer, model, fit_vect=False):
  if fit_vect:
    vectorizer.fit(train.data)
  train.vecs = vectorizer.transform(train.data)
  test.vecs = vectorizer.transform(test.data)
  model.fit(train.vecs, train.labels)
  train_preds = model.predict(train.vecs)
  train_f1 = f1_score(train.labels, train_preds, average='micro')
  test_preds = model.predict(test.vecs)
  test_f1 = f1_score(test.labels, test_preds, average='micro')
  return train_f1, test_f1
```

MNB with default parameters

In [36]:
```
model = MultinomialNB()
classification(tfidf_trigram, model)
```

Out[36]:
```
(0.9963379214547291, 0.8150220913107512)
```

Let's tune MNB (alpha)

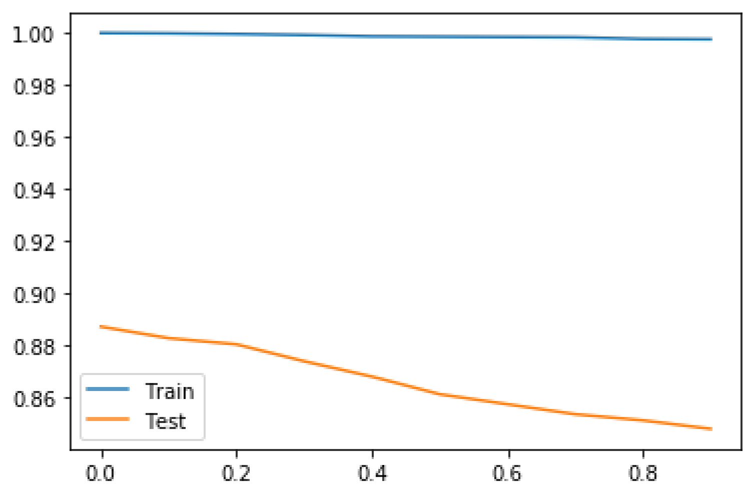
Naive Bayes classifier for multinomial models ([Details (https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html)](https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html))

In [39]:
```
arange = np.arange(0, 1, 0.1)
results = []
for a in arange:
  model = MultinomialNB(alpha=a)
  res = classification(bigram, model)
  results.append(res)
  print(a, '=>', res)

plt.plot(arange, results)
plt.legend(["Train", "Test"])
plt.show()
```

```
C:\Users\student\anaconda3\lib\site-packages\sklearn\naive_bayes.py:507: User
Warning: alpha too small will result in numeric errors, setting alpha = 1.0e-
10
  'setting alpha = %.1e' % _ALPHA_MIN)

0.0 => (0.9996211642884203, 0.8871870397643593)
0.1 => (0.9994948857178937, 0.8827687776141384)
0.2 => (0.9992423285768405, 0.8804123711340206)
0.30000000000000004 => (0.9988634928652608, 0.8739322533136966)
0.4 => (0.9983583785831545, 0.8680412371134021)
0.5 => (0.9982321000126277, 0.8612665684830634)
0.6000000000000001 => (0.9981058214421012, 0.8574374079528719)
0.7000000000000001 => (0.9979795428715746, 0.8536082474226804)
0.8 => (0.9973481500189418, 0.8512518409425626)
0.9 => (0.9972218714484152, 0.8480117820324006)
```

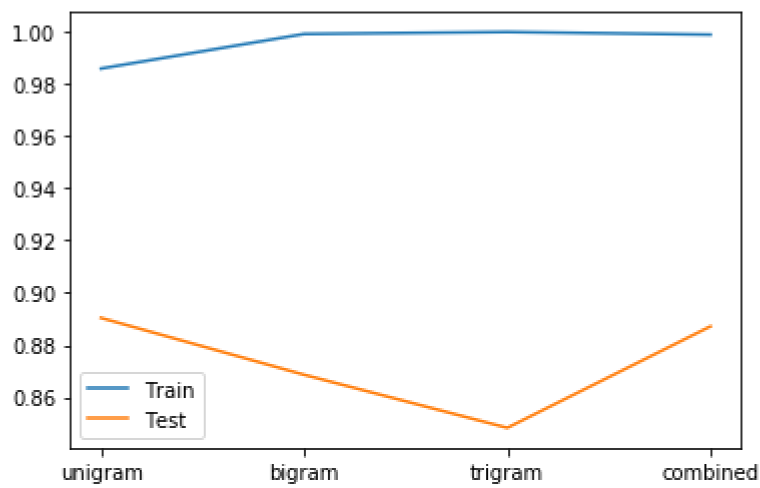


Let's compare different n-grams with a fixed alpha for MNB

In [41]:
```
vectorizer_names = ['unigram', 'bigram', 'trigram', 'combined']
xs = list(range(len(vectorizer_names)))
results = []
for i in range(len(tfidf_vectorizers)):
  model = MultinomialNB(alpha=0.1)
  res = classification(tfidf_vectorizers[i], model)
  results.append(res)
  print(vectorizer_names[i], '=>', res)

plt.plot(xs, results)
plt.xticks(xs, vectorizer_names)
plt.legend(["Train", "Test"])
plt.show()
```

```
unigram => (0.9857305215304962, 0.8904270986745213)
bigram => (0.9989897714357874, 0.8686303387334315)
trigram => (0.9997474428589468, 0.8483063328424153)
combined => (0.9987372142947342, 0.8871870397643593)
```



Excercise2: Let's tune TfidfVectorizer with a fixed alpha for MNB

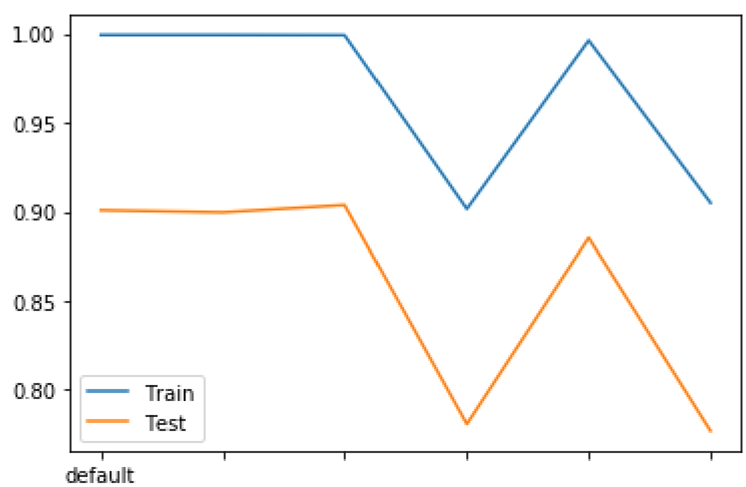
Compare TfidfVectorizers with different parameters (Details (http://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html#sklearn.feature_extraction.
e.g.,

- stop_words ='english'(None by default)
- min_df = 5 (1 by default)
- sublinear_tf = True (False by default)
- use_idf = False (True by default)
- binary = True (False by default)

In [46]:
```
#benchmark: 0.9037
tfidf_vectorizers = [TfidfVectorizer(), TfidfVectorizer(stop_words ="english"
), TfidfVectorizer(sublinear_tf=True, stop_words ="english"),  TfidfVectorizer
(min_df=100, max_df=0.7),
                    TfidfVectorizer(min_df=5, max_df=0.7), TfidfVectorizer(min
_df=100, max_df=0.7, sublinear_tf=True)] # add parameters to each vectorizer
names = ['default', '', '', '', '', ''] # give short names to say what you cha
nged
xs = list(range(len(tfidf_vectorizers)))
results = list()
for i in range(len(tfidf_vectorizers)):
  tfidf_vectorizers[i].fit(train.data)
  model = MultinomialNB(alpha=0.0001) # set a very small value
  res = classification(tfidf_vectorizers[i], model) # we need to set fit_vect=
True, but why?
  results.append(res)
  print(names[i], '=>', res)

plt.plot(xs, results)
plt.xticks(xs, names)
plt.legend(["Train", "Test"])
plt.show()
```

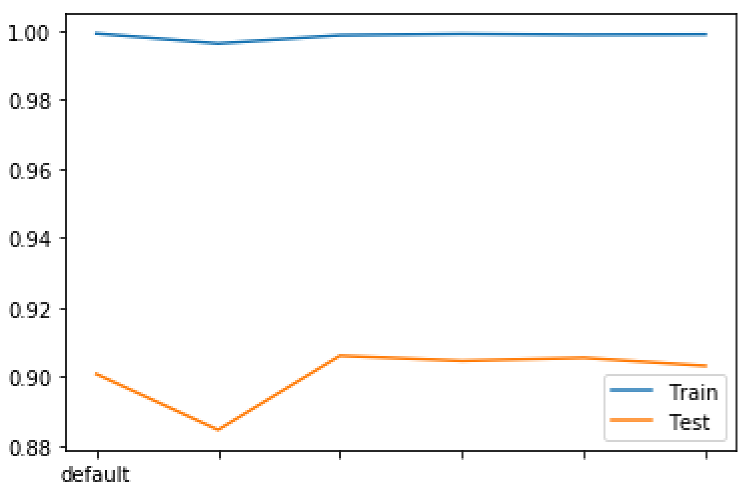
```
default => (0.9992423285768405, 0.9007363770250368)
 => (0.9992423285768405, 0.8995581737849779)
 => (0.999116050006314, 0.9036818851251841)
 => (0.9015027149892664, 0.7808541973490427)
 => (0.9962116428842025, 0.8854197349042711)
 => (0.9049122363934841, 0.7770250368188512)
```



In [66]:
```
tfidf_vectorizers = [TfidfVectorizer(),
                     TfidfVectorizer(min_df=5, max_df=0.7, stop_words ="englis
h"),
                     TfidfVectorizer(use_idf=False),
                     TfidfVectorizer(use_idf=False, sublinear_tf=True),
                     TfidfVectorizer(use_idf=False, stop_words ="english"),
                     TfidfVectorizer(use_idf=False, sublinear_tf=True, stop_wo
rds ="english")] # add parameters to each vectorizer
names = ['default', '', '', '', '', ''] # give short names to say what you cha
nged
xs = list(range(len(tfidf_vectorizers)))
results = list()
for i in range(len(tfidf_vectorizers)):
  tfidf_vectorizers[i].fit(train.data)
  model = MultinomialNB(alpha=0.0001) # set a very small value
  res = classification(tfidf_vectorizers[i], model) # we need to set fit_vect=
True, but why?
  results.append(res)
  print(names[i], '=>', res)

plt.plot(xs, results)
plt.xticks(xs, names)
plt.legend(["Train", "Test"])
plt.show()
```

```
default => (0.9992423285768405, 0.9007363770250368)
 => (0.9963379214547291, 0.8845360824742268)
 => (0.9987372142947342, 0.9060382916053019)
 => (0.999116050006314, 0.9045655375552283)
 => (0.9988634928652608, 0.9054491899852725)
 => (0.9989897714357874, 0.9030927835051547)
```



What parameter did you change to get the best test performance?

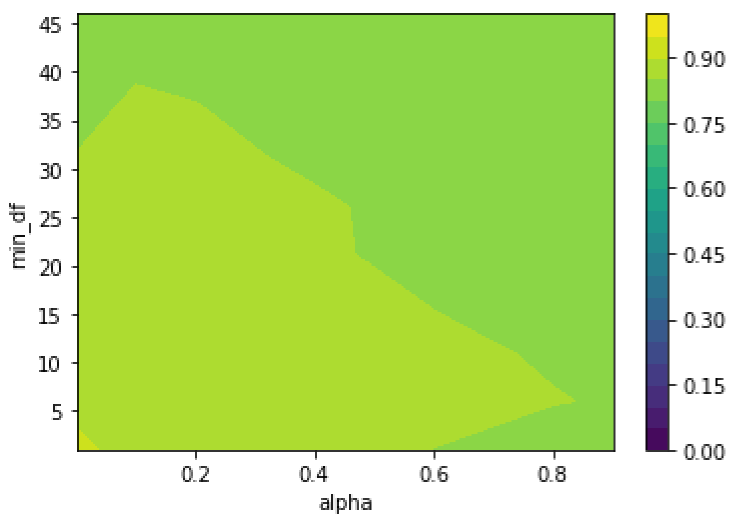
In [ ]:
```
#change use_idf to false and the best result is 0.906
```

[Optional] Let's tune parameters of TfidfVectorizer and MNB simultaneously

This parameter tuning is called ***grid search***

In [51]:
```
min_dfs = np.arange(1, 50, 5)
alphas = np.arange(0.001, 1.0, 0.1)
xx, yy = np.meshgrid(alphas, min_dfs)
f1_score_mat = np.zeros((len(min_dfs), len(alphas)))
for i in range(len(min_dfs)):
  tfidf_vectorizer = TfidfVectorizer(min_df=min_dfs[i])
  for j in range(len(alphas)):
    model = MultinomialNB(alpha=alphas[j])
    res = classification(tfidf_vectorizer, model, fit_vect=True)
    f1_score_mat[i][j] = res[1]

plt.contourf(alphas, min_dfs, f1_score_mat, cmap=plt.cm.viridis, levels=np.ara
nge(0.0,1.05,0.05))
plt.ylabel('min_df')
plt.xlabel('alpha')
plt.colorbar()
plt.show()
```



In [47]:
```
min_dfs = np.arange(0.0, 1.0, 0.1)
alphas = np.arange(0.1, 1.0, 0.1)
xx, yy = np.meshgrid(alphas, min_dfs)
f1_score_mat = np.zeros((len(min_dfs), len(alphas)))
for i in range(len(min_dfs)):
  tfidf_vectorizer = TfidfVectorizer(min_df=min_dfs[i])
  for j in range(len(alphas)):
    model = MultinomialNB(alpha=alphas[j])
    res = classification(tfidf_vectorizer, model, fit_vect=True)
    f1_score_mat[i][j] = res[1]

plt.contourf(alphas, min_dfs, f1_score_mat, cmap=plt.cm.viridis, levels=np.ara
nge(0.0,1.05,0.05))
plt.xlabel('min_df')
plt.ylabel('alpha')
plt.colorbar()
plt.show()
```