Basic Text Classification with Naive Bayes

In the mini-project, you'll learn the basics of text analysis using a subset of movie reviews from the rotten tomatoes database. You'll also use a fundamental technique in Bayesian inference, called Naive Bayes. This mini-project is based on Lab 10 of Harvard's CS109 class. Please free to go to the original lab for additional exercises and solutions.

In [1]:

```
%matplotlib inline
import numpy as np
import scipy as sp
import matplotlib as mpl
import matplotlib.cm as cm
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from six.moves import range

# Setup Pandas
pd.set_option('display.width', 500)
pd.set_option('display.max_columns', 100)
pd.set_option('display.notebook_repr_html', True)

# Setup Seaborn
sns.set_style("whitegrid")
sns.set_context("poster")
```

Table of Contents

Rotten Tomatoes Dataset

In [2]:

```
critics = pd.read_csv('C:/critics.csv')
#let's drop rows with missing quotes
critics = critics[~critics.quote.isnull()]
critics.head()
```

Out[2]:

	critic	fresh	imdb	publication	quote	review_date	rtid	title
1	Derek Adams	fresh	114709	Time Out	So ingenious in concept, design and execution ...	2009-10-04	9559	Toy story
2	Richard Corliss	fresh	114709	TIME Magazine	The year's most inventive comedy.	2008-08-31	9559	Toy story
3	David Ansen	fresh	114709	Newsweek	A winning animated feature that has something ...	2008-08-18	9559	Toy story
4	Leonard Klady	fresh	114709	Variety	The film sports a provocative and appealing st...	2008-06-09	9559	Toy story
5	Jonathan Rosenbaum	fresh	114709	Chicago Reader	An entertaining computer-generated, hyperreali...	2008-03-10	9559	Toy story

Explore

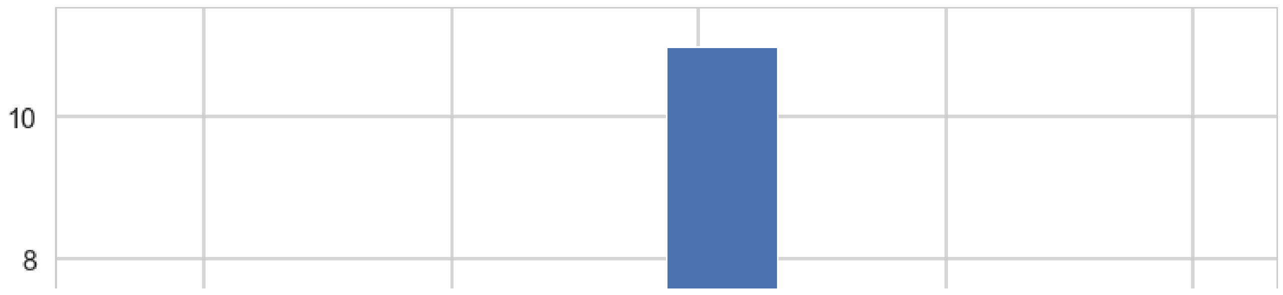
In [3]:

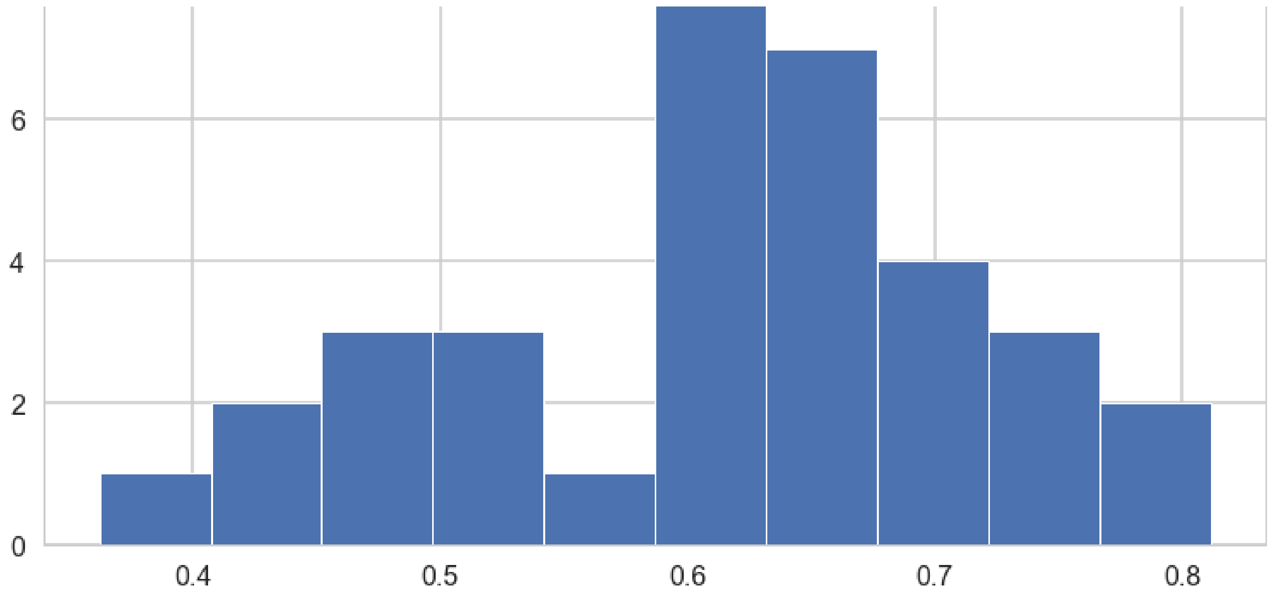
```
n_reviews = len(critics)
n_movies = critics.rtid.unique().size
n_critics = critics.critic.unique().size


print("Number of reviews: {:d}".format(n_reviews))
print("Number of critics: {:d}".format(n_critics))
print("Number of movies:  {:d}".format(n_movies))
```

```
Number of reviews: 15561
Number of critics: 623
Number of movies:  1921
```

In [4]:

```
df = critics.copy()
df['fresh'] = df.fresh == 'fresh'
grp = df.groupby('critic')
counts = grp.critic.count()  # number of reviews by each critic
means = grp.fresh.mean()     # average freshness for each critic

means[counts > 100].hist(bins=10, edgecolor='w', lw=1)
plt.xlabel = "Average Rating per critic"
plt.ylabel = "Number of Critics"
plt.yticks([0, 2, 4, 6, 8, 10]);
```





Exercise Set I

Exercise: Look at the histogram above. Tell a story about the average ratings per critic. What shape does the distribution look like? What is interesting about the distribution? What might explain these interesting things?

ANSWER

- Majority of the critics voted between 60% to 70%. It's interesting to see the dip around 55%. Very few critics seem to have given an average rating of 55% to 60%. Overall, the shape suggests of a normal distribution to quite an extent excpet for the dip around 55%.

The Vector Space Model and a Search Engine

All the diagrams here are snipped from *Introduction to Information Retrieval* by Manning et. al. which is a great resource on text processing. For additional information on text mining and natural language processing, see *Foundations of Statistical Natural Language Processing* by Manning and Schutze.

Also check out Python packages `nltk`, `spaCy`, `pattern`, and their associated resources. Also see `word2vec`.

Let us define the vector derived from document d by $\bar V(d)$. What does this mean? Each document is treated as a vector containing information about the words contained in it. Each vector has the same length and each entry "slot" in the vector contains some kind of data about the words that appear in the document such as presence/absence (1/0), count (an integer) or some other statistic. Each vector has the same length because each document shared the same vocabulary across the full collection of documents -- this collection is called a *corpus*.

To define the vocabulary, we take a union of all words we have seen in all documents. We then just associate an array index with them. So "hello" may be at index 5 and "world" at index 99.

Suppose we have the following corpus:

`A Fox one day spied a beautiful bunch of ripe grapes hanging from a vine trained along the branches of a tree. The grapes seemed ready to burst with juice, and the Fox's mouth watered as he gazed longingly at them.`

Suppose we treat each sentence as a document d. The vocabulary (often called the *lexicon*) is the following:

$V = \{$ `a, along, and, as, at, beautiful, branches, bunch, burst, day, fox, fox's, from, gazed, grapes, hanging, he, juice, longingly, mouth, of, one, ready, ripe, seemed, spied, the, them, to, trained, tree, vine, watered, with`$\}$

Then the document

`A Fox one day spied a beautiful bunch of ripe grapes hanging from a vine trained along the branches of a tree`

may be represented as the following sparse vector of word counts:

$$\bar{V}(d) = (4, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 2, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0)$$

or more succinctly as

```
[(0, 4), (1, 1), (5, 1), (6, 1), (7, 1), (9, 1), (10, 1), (12, 1), (14, 1), (15, 1), (20, 2), (21, 1), (23, 1), (26, 1), (30, 1), (31, 1)]
```

along with a dictionary

```
{
    0: a, 1: along, 5: beautiful, 6: branches, 7: bunch, 9: day, 10: fox, 12: from, 14: grapes,
    15: hanging, 19: mouth, 20: of, 21: one, 23: ripe, 24: seemed, 25: spied, 26: the,
    30: tree, 31: vine,
}
```

Then, a set of documents becomes, in the usual `sklearn` style, a sparse matrix with rows being sparse arrays representing documents and columns representing the features/words in the vocabulary.

Notice that this representation loses the relative ordering of the terms in the document. That is "cat ate rat" and "rat ate cat" are the same. Thus, this representation is also known as the Bag-Of-Words representation.

Here is another example, from the book quoted above, although the matrix is transposed here so that documents are columns:

Such a matrix is also catted a Term-Document Matrix. Here, the terms being indexed could be stemmed before indexing; for instance, `jealous` and `jealousy` after stemming are the same feature. One could also make use of other "Natural Language Processing" transformations in constructing the vocabulary. We could use Lemmatization, which reduces words to lemmas: work, constructing the vocabulary. We could use Lemmatization, which reduces words to lemmas: work, working, worked would all reduce to work. We could remove "stopwords" from our vocabulary, such as common words like "the". We could look for particular parts of speech, such as adjectives. This is often done in Sentiment Analysis. And so on. It all depends on our application.

From the book:

> The standard way of quantifying the similarity between two documents d_1 and d_2 is to compute the cosine similarity of their vector representations $\bar{V}(d_1)$ and $\bar{V}(d_2)$:

$$S_{12} = \frac{\bar{V}(d_1) \cdot \bar{V}(d_2)}{|\bar{V}(d_1)| \times |\bar{V}(d_2)|}$$

> There is a far more compelling reason to represent documents as vectors: we can also view a query as a vector. Consider the query q = jealous gossip. This query turns into the unit vector $\bar{V}(q)$ = (0, 0.707, 0.707) on the three coordinates below.

> The key idea now: to assign to each document d a score equal to the dot product:

$$\bar{V}(q) \cdot \bar{V}(d)$$

Then we can use this simple Vector Model as a Search engine.

In Code

In [5]:

```
from sklearn.feature_extraction.text import CountVectorizer

text = ['Hop on pop', 'Hop off pop', 'Hop Hop hop']
print("Original text is\n{}".format('\n'.join(text)))

vectorizer = CountVectorizer(min_df=0)

# call `fit` to build the vocabulary
vectorizer.fit(text)

# call `transform` to convert text to a bag of words
x = vectorizer.transform(text)

# CountVectorizer uses a sparse array to save memory, but it's easier in th
is assignment to
# convert back to a "normal" numpy array
x = x.toarray()

print("")
print("Transformed text vector is \n{}".format(x))

# `get_feature_names` tracks which word is associated with each column of t
```

```
he transformed x
print("")
print("Words for each feature:")
print(vectorizer.get_feature_names())

# Notice that the bag of words treatment doesn't preserve information about the *order* of words,
# just their frequency
```

```
Original text is
Hop on pop
Hop off pop
Hop Hop hop

Transformed text vector is
[[1 0 1 1]
 [1 1 0 1]
 [3 0 0 0]]

Words for each feature:
['hop', 'off', 'on', 'pop']
```

In [6]:

```
def make_xy(critics, vectorizer=None):
    #Your code here
    if vectorizer is None:
        vectorizer = CountVectorizer()
    X = vectorizer.fit_transform(critics.quote)
    X = X.tocsc()  # some versions of sklearn return COO format
    y = (critics.fresh == 'fresh').values.astype(np.int)
    return X, y
X, y = make_xy(critics)
X,y
```

Out[6]:

```
(<15561x22417 sparse matrix of type '<class 'numpy.int64'>'
  with 272265 stored elements in Compressed Sparse Column format>,
 array([1, 1, 1, ..., 1, 1, 1]))
```

Naive Bayes

From Bayes' Theorem, we have that

$$P(c \vert f) = \frac{P(c \cap f)}{P(f)}$$

where c represents a *class* or category, and f represents a feature vector, such as $\bar V(d)$ as above. **We are computing the probability that a document (or whatever we are classifying) belongs to category c given the features in the document.** $P(f)$ is really just a normalization constant, so the literature usually writes Bayes' Theorem in context of Naive Bayes as

$$P(c \vert f) \propto P(f \vert c) P(c)$$

$P(c)$ is called the *prior* and is simply the probability of seeing class c. But what is $P(f \vert c)$? This is the probability that we see feature set f given that this document is actually in class c. This is called the *likelihood* and comes from the data. One of the major assumptions of the Naive Bayes model is that *likelihood* and comes from the data. One of the major assumptions of the Naive Bayes model is that the features are *conditionally independent* given the class. While the presence of a particular discriminative word may uniquely identify the document as being part of class c and thus violate general feature independence, conditional independence means that the presence of that term is independent of all the other words that appear *within that class*. This is a very important distinction. Recall that if two events are independent, then:

$$P(A \cap B) = P(A) \cdot P(B)$$

Thus, conditional independence implies

$$P(f|c) = \prod_i P(f_i|c)$$

where f_i is an individual feature (a word in this example).

To make a classification, we then choose the class c such that $P(c|f)$ is maximal.

There is a small caveat when computing these probabilities. For floating point underflow we change the product into a sum by going into log space. This is called the LogSumExp trick. So:

$$\log P(f|c) = \sum_i \log P(f_i|c)$$

There is another caveat. What if we see a term that didn't exist in the training data? This means that $P(f_i|c) = 0$ for that term, and thus $P(f|c) = \prod_i P(f_i|c) = 0$, which doesn't help us at all. Instead of using zeros, we add a small negligible value called α to each count. This is called Laplace Smoothing.

$$P(f_i|c) = \frac{N_{ic} + \alpha}{N_c + \alpha N_i}$$

where N_{ic} is the number of times feature i was seen in class c, N_c is the number of times class c was seen and N_i is the number of times feature i was seen globally. α is sometimes called a regularization parameter.

Multinomial Naive Bayes and Other Likelihood Functions

Since we are modeling word counts, we are using variation of Naive Bayes called Multinomial Naive Bayes. This is because the likelihood function actually takes the form of the multinomial distribution.

$$P(f|c) = \frac{\left(\sum_i f_i\right)!}{\prod_i f_i!} \prod_{f_i} P(f_i|c)^{f_i} \propto \prod_i P(f_i|c)$$

where the nasty term out front is absorbed as a normalization constant such that probabilities sum to 1.

There are many other variations of Naive Bayes, all which depend on what type of value f_i takes. If f_i is continuous, we may be able to use *Gaussian Naive Bayes*. First compute the mean and variance for each class c. Then the likelihood, $P(f|c)$ is given as follows

$$\frac{1}{\sqrt{2\pi\sigma_c^2}} \quad -\frac{\left(v-\mu_c\right)^2}{2\sigma_c^2}$$

$P(f_i = v | c) =$ e

Exercise Set II

Exercise: Implement a simple Naive Bayes classifier:

1. split the data set into a training and test set
2. Use `scikit-learn`'s `MultinomialNB()` classifier with default parameters.
3. train the classifier over the training set and test on the test set
4. print the accuracy scores for both the training and the test sets

What do you notice? Is this a good classifier? If not, why not?

In [7]:

```
#your turn
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

In [8]:

```
from sklearn.naive_bayes import MultinomialNB
NaiveBayes = MultinomialNB()
```

In [9]:

```
NaiveBayes.fit(X_train, y_train)
```

Out[9]:

```
MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True)
```

In [10]:

```
print("Accuracy(train-set) is " +str(NaiveBayes.score(X_train, y_train)))
print("Accuracy(test-set) is " + str(NaiveBayes.score(X_test, y_test)))
```

```
Accuracy(train-set) is 0.92116538132
Accuracy(test-set) is 0.782832176818
```

ANSWER

- Test set does not fair as good as the train set.

Picking Hyperparameters for Naive Bayes and Text Maintenance

We need to know what value to use for α, and we also need to know which words to include in the vocabulary. As mentioned earlier, some words are obvious stopwords. Other words appear so infrequently that they serve as noise, and other words in addition to stopwords appear so frequently that they may also serve as noise.

First, let's find an appropriate value for `min_df` for the `CountVectorizer`. `min_df` can be either an integer or a float/decimal. If it is an integer, `min_df` represents the minimum number of documents a word must appear in for it to be included in the vocabulary. If it is a float, it represents the minimum *percentage* of documents a word must appear in to be included in the vocabulary. From the documentation:

> min_df: When building the vocabulary ignore terms that have a document frequency strictly lower than the given threshold. This value is also called cut-off in the literature. If float, the parameter represents a proportion of documents, integer absolute counts. This parameter is ignored if vocabulary is not None.

Exercise Set III

Exercise: Construct the cumulative distribution of document frequencies (df). The x-axis is a document count x_i and the y-axis is the percentage of words that appear less than x_i times. For example, at $x = 5$, plot a point representing the percentage or number of words that appear in 5 or fewer documents.

Exercise: Look for the point at which the curve begins climbing steeply. This may be a good value for `min_df`. If we were interested in also picking `max_df`, we would likely pick the value where the curve starts to plateau. What value did you choose?

In [11]:

```
# Your turn.
doc_cnt = X.sum(axis=0)
doc_cnt
```

Out[11]:

```
matrix([[ 8,   1, 10, ...,   3,   1,   1]], dtype=int64)
```

In [12]:

```
word_cnt = {}
max_word_cnt = doc_cnt.max()
for n in range(max_word_cnt + 1):
    word_cnt[n] = np.sum(doc_cnt <= n)
word_cnt
```

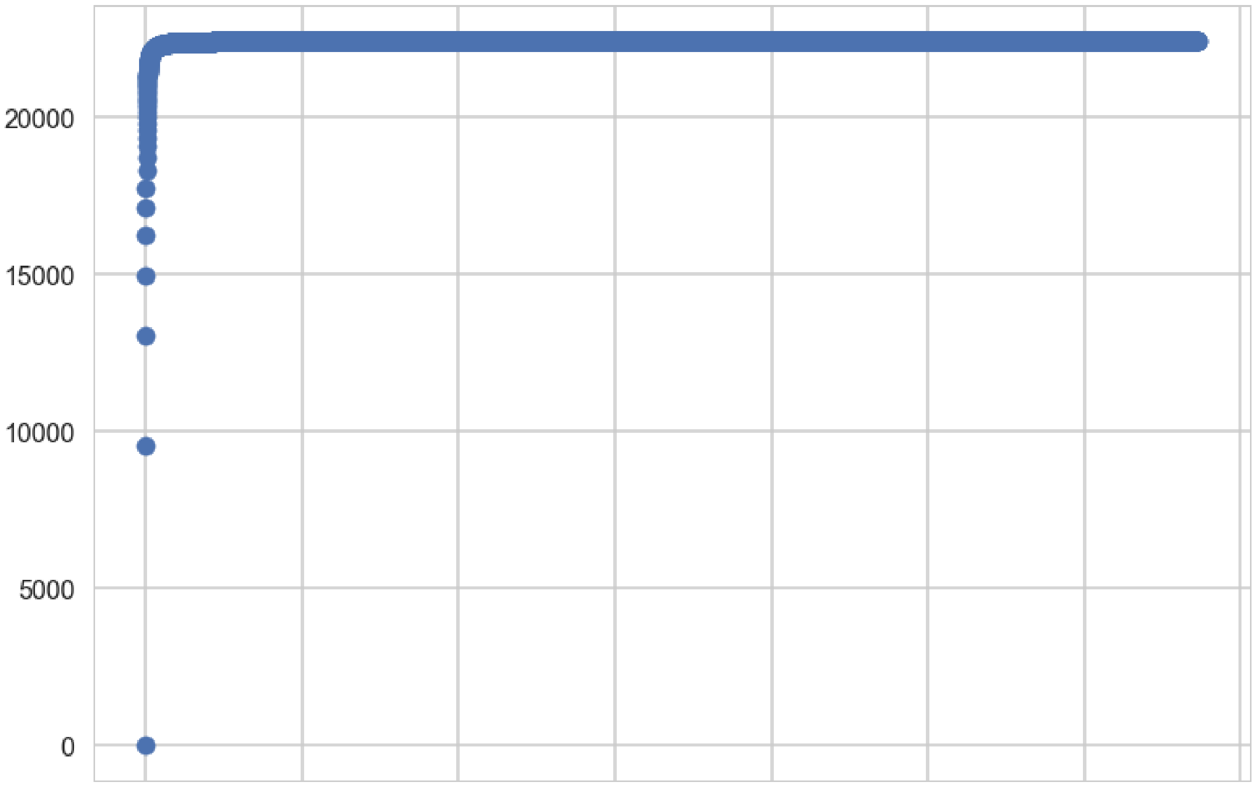
Out[12]:

```
{0: 0,
 1: 9552,
 2: 13038,
 3: 14954,
 4: 16213,
 5: 17109,
 6: 17743,
 7: 18275,
 8: 18690,
 9: 19044,
 10: 19338,
    11: 19573,
    12: 19785,
    13: 19970,
    14: 20146,
    15: 20281,
    16: 20385,
    17: 20491,
    18: 20581,
    19: 20676,
    20: 20753,
    21: 20836,
    22: 20911,
    23: 20968,
    24: 21016,
    25: 21069,
    26: 21122,
    27: 21164,
    28: 21212,
    29: 21256,
    30: 21301,
    31: 21336,
    32: 21364,
    33: 21398,
    34: 21434,
    35: 21460,
    36: 21487,
    37: 21508,
    38: 21526,
    39: 21551,
    40: 21570,
    41: 21590,
    42: 21613,
    43: 21631,
    44: 21652,
    45: 21669,
    46: 21686,
    47: 21706,
    48: 21716,
    49: 21733,
    50: 21745,
    51: 21752,
    52: 21765,
    53: 21785,
    54: 21791,
    55: 21801,
    56: 21818,
    57: 21832,
    58: 21842,
    59: 21852,
    60: 21865,
    61: 21876,
    62: 21886,
    63: 21897,
    64: 21906,
    65: 21910,
    66: 21922,
    67: 21934,
    68: 21941,
    69: 21944,
    70: 21947,
    71: 21953,
    71: 21955,
    72: 21958,
    73: 21964,
    74: 21970,
    75: 21975,
    76: 21986,
    77: 21997,
    78: 21999,
    79: 22003,
    80: 22014,
    81: 22019,
    82: 22024,
    83: 22029,
    84: 22036,
    85: 22040,
    86: 22045,
    87: 22048,
    88: 22050,
    89: 22054,
    90: 22057,
    91: 22061,
    92: 22062,
    93: 22070,
    94: 22075,
    95: 22080,
    96: 22084,
    97: 22087,
    98: 22089,
    99: 22092,
    100: 22097,
    101: 22101,
    102: 22103,
    103: 22106,
    104: 22108,
    105: 22109,
    106: 22117,
    107: 22121,
    108: 22123,
    109: 22123,
    110: 22126,
    111: 22128,
    112: 22134,
    113: 22138,
    114: 22138,
    115: 22138,
    116: 22146,
    117: 22148,
    118: 22151,
    119: 22152,
    120: 22152,
    121: 22154,
    122: 22155,
    123: 22158,
    124: 22159,
    125: 22159,
    126: 22160,
    127: 22162,
    128: 22166,
    129: 22166,
    130: 22167,
    131: 22168,
    132: 22168,
    133: 22170,
    134: 22173,
    135: 22174,
    136: 22174,
    137: 22176,
    138: 22180,
    139: 22180,
    140: 22181,
    141: 22184,
    142: 22188,
    143: 22188,
    144: 22191,
    145: 22193,
    146: 22196,
    147: 22199,
    148: 22199,
    149: 22200,
    150: 22201,
    151: 22202,
    152: 22202,
    153: 22203,
    154: 22204,
    155: 22204,
    156: 22207,
    157: 22209,
    158: 22211,
    159: 22213,
    160: 22215,
    161: 22217,
    162: 22218,
    163: 22218,
    164: 22218,
    165: 22218,
    166: 22221,
    167: 22221,
    168: 22221,
    169: 22222,
    170: 22224,
    171: 22224,
    172: 22226,
    173: 22226,
    174: 22226,
    175: 22227,
    176: 22228,
    177: 22231,
    178: 22232,
    179: 22233,
    180: 22233,
    181: 22233,
    182: 22234,
    183: 22235,
    184: 22236,
    185: 22236,
    186: 22239,
    187: 22239,
    188: 22240,
    189: 22241,
    190: 22241,
    191: 22242,
    192: 22244,
    193: 22244,
    194: 22244,
    195: 22245,
    196: 22247,
    197: 22247,
    198: 22248,
    199: 22249,
    200: 22250,
    201: 22252,
    202: 22253,
    203: 22253,
    204: 22253,
    205: 22254,
    206: 22254,
    207: 22255,
    208: 22257,
    209: 22258,
    210: 22259,
    211: 22259,
    212: 22260,
    213: 22261,
    214: 22263,
    215: 22263,
    216: 22263,
    217: 22263,
    218: 22263,
    219: 22265,
    220: 22265,
    221: 22266,
    222: 22267,
    223: 22268,
    224: 22268,
    225: 22269,
    226: 22270,
    227: 22273,
    228: 22273,
    229: 22273,
    230: 22273,
    231: 22275,
    232: 22277,
    233: 22278,
    234: 22279,
    235: 22279,
    236: 22280,
    237: 22282,
    238: 22282,
    239: 22283,
    240: 22283,
    241: 22283,
    242: 22284,
    243: 22284,
    244: 22287,
    245: 22287,
    246: 22287,
    247: 22287,
    248: 22288,
    249: 22289,
    250: 22289,
    251: 22289,
    252: 22292,
    253: 22292,
    254: 22293,
    255: 22293,
    256: 22294,
    257: 22297,
    258: 22300,
    259: 22300,
    260: 22301,
    261: 22302,
    262: 22302,
    263: 22303,
    264: 22304,
    265: 22304,
    266: 22305,
    267: 22305,
    268: 22305,
    269: 22305,
    270: 22305,
    271: 22306,
    272: 22307,
    273: 22308,
    274: 22309,
    275: 22309,
    276: 22310,
    277: 22310,
    278: 22311,
    279: 22312,
    280: 22312,
    281: 22312,
    282: 22313,
    283: 22313,
    284: 22315,
    285: 22315,
    286: 22315,
    287: 22316,
    288: 22317,
    289: 22317,
    290: 22318,
    291: 22318,
    292: 22318,
    293: 22319,
    294: 22319,
    295: 22320,
    296: 22322,
    297: 22322,
    298: 22322,
    299: 22322,
    300: 22322,
    301: 22323,
    302: 22323,
    303: 22323,
    304: 22323,
    305: 22324,
    306: 22325,
    307: 22325,
    308: 22325,
    309: 22325,
    310: 22325,
    311: 22325,
    312: 22325,
    313: 22325,
    314: 22325,
    315: 22325,
    316: 22325,
    317: 22325,
    318: 22325,
    319: 22325,
    320: 22326,
    321: 22326,
    322: 22327,
    323: 22327,
    324: 22328,
    325: 22328,
    326: 22328,
    327: 22328,
    328: 22328,
    329: 22328,
    330: 22329,
    331: 22329,
    332: 22329,
    333: 22329,
    334: 22329,
    335: 22329,
    336: 22330,
    337: 22330,
    338: 22330,
    339: 22330,
    340: 22330,
    341: 22331,
    342: 22331,
    343: 22332,
    344: 22333,
    345: 22333,
    346: 22333,
    347: 22333,
    348: 22333,
    349: 22333,
    350: 22333,
    351: 22333,
    352: 22333,
    353: 22333,
    354: 22334,
    355: 22334,
    356: 22334,
    357: 22334,
    358: 22334,
    359: 22334,
    360: 22334,
    361: 22334,
    362: 22334,
    363: 22335,
    364: 22336,
    365: 22336,
    366: 22336,
    367: 22337,
    368: 22337,
    369: 22338,
    370: 22339,
    371: 22339,
    372: 22339,
    373: 22339,
    374: 22339,
    375: 22339,
    376: 22339,
    377: 22339,
    378: 22339,
    379: 22339,
    380: 22339,
    381: 22339,
    382: 22339,
    383: 22340,
    384: 22340,
    385: 22340,
    386: 22341,
    387: 22342,
    388: 22343,
    389: 22343,
    390: 22343,
    391: 22343,
    392: 22343,
    393: 22343,
    394: 22343,
    395: 22343,
    396: 22344,
    397: 22344,
    398: 22344,
    399: 22345,
    400: 22345,
    401: 22345,
    402: 22345,
    403: 22345,
    404: 22345,
    405: 22345,
    406: 22345,
    407: 22345,
    408: 22345,
    409: 22345,
    410: 22345,
    411: 22345,
    412: 22345,
    413: 22346,
    414: 22346,
    415: 22347,
    416: 22347,
    417: 22348,
    418: 22348,
    419: 22348,
    420: 22348,
    421: 22348,
    422: 22349,
    423: 22349,
    424: 22349,
    425: 22349,
    426: 22349,
    427: 22349,
    428: 22349,
    429: 22349,
    430: 22349,
    431: 22349,
    432: 22349,
    433: 22350,
    434: 22351,
    435: 22351,
    436: 22351,
    437: 22351,
    438: 22353,
    439: 22354,
    440: 22354,
    441: 22354,
    442: 22354,
    443: 22355,
    444: 22355,
    445: 22355,
    446: 22355,
    447: 22355,
    448: 22355,
    449: 22355,
    450: 22355,
    451: 22355,
    452: 22355,
    453: 22355,
    454: 22355,
    455: 22355,
    456: 22355,
    457: 22355,
    458: 22355,
    459: 22355,
    460: 22355,
    461: 22355,
    462: 22356,
    463: 22356,
    464: 22356,
    465: 22356,
    466: 22356,
    467: 22356,
    468: 22356,
    469: 22357,
    470: 22357,
    471: 22357,
    472: 22358,
    473: 22358,
    474: 22358,
    475: 22358,
    476: 22358,
    477: 22358,
    478: 22359,
    479: 22359,
    480: 22359,
    481: 22359,
    482: 22359,
    483: 22359,
    484: 22359,
    485: 22359,
    486: 22359,
    487: 22359,
    488: 22359,
    489: 22359,
    490: 22359,
    491: 22359,
    492: 22359,
    493: 22359,
    494: 22359,
    495: 22359,
    496: 22360,
    497: 22360,
    498: 22360,
    499: 22360,
    500: 22360,
    501: 22360,
    502: 22360,
    503: 22360,
    504: 22361,
    505: 22361,
    506: 22361,
    507: 22362,
    508: 22362,
    509: 22362,
    510: 22362,
    511: 22362,
    512: 22362,
    513: 22362,
    514: 22362,
    515: 22362,
    516: 22362,
    517: 22362,
    518: 22362,
    519: 22362,
    520: 22362,
    521: 22362,
    522: 22362,
    523: 22362,
    524: 22362,
    525: 22362,
    526: 22362,
    527: 22362,
    528: 22362,
    529: 22362,
    530: 22362,
    531: 22363,
    532: 22363,
    533: 22363,
    534: 22363,
    535: 22363,
    536: 22363,
    537: 22363,
    538: 22363,
    539: 22363,
    540: 22364,
    541: 22364,
    542: 22364,
    543: 22364,
    544: 22364,
    545: 22366,
    546: 22366,
    547: 22366,
    548: 22366,
    549: 22366,
    550: 22366,
    551: 22366,
    552: 22366,
552: 22366,
553: 22366,
554: 22366,
555: 22366,
556: 22366,
557: 22366,
558: 22366,
559: 22366,
560: 22366,
561: 22366,
562: 22366,
563: 22366,
564: 22366,
565: 22366,
566: 22366,
567: 22366,
568: 22366,
569: 22366,
570: 22366,
571: 22366,
572: 22367,
573: 22367,
574: 22367,
575: 22367,
576: 22367,
577: 22367,
578: 22367,
579: 22367,
580: 22367,
581: 22367,
582: 22367,
583: 22367,
584: 22367,
585: 22367,
586: 22367,
587: 22367,
588: 22367,
589: 22367,
590: 22367,
591: 22368,
592: 22368,
593: 22368,
594: 22368,
595: 22368,
596: 22368,
597: 22368,
598: 22368,
599: 22369,
600: 22369,
601: 22369,
602: 22369,
603: 22369,
604: 22369,
605: 22369,
606: 22369,
607: 22369,
608: 22369,
609: 22369,
610: 22369,
611: 22369,
612: 22369
```

```
    612: 22369,
    613: 22369,
    614: 22369,
    615: 22370,
    616: 22370,
    617: 22370,
    618: 22370,
    619: 22370,
    620: 22370,
    621: 22370,
    622: 22370,
    623: 22370,
    624: 22370,
    625: 22370,
    626: 22370,
    627: 22370,
    628: 22370,
    629: 22370,
    630: 22370,
    631: 22370,
    632: 22370,
    633: 22370,
    634: 22370,
    635: 22370,
    636: 22370,
    637: 22370,
    638: 22370,
    639: 22370,
    640: 22370,
    641: 22370,
    642: 22370,
    643: 22370,
    644: 22371,
    645: 22371,
    646: 22371,
    647: 22371,
    648: 22371,
    649: 22371,
    650: 22371,
    651: 22371,
    652: 22371,
    653: 22371,
    654: 22371,
    655: 22371,
    656: 22371,
    657: 22372,
    658: 22372,
    659: 22372,
    660: 22372,
    661: 22372,
    662: 22372,
    663: 22373,
    664: 22373,
    665: 22373,
    666: 22373,
    667: 22373,
    668: 22373,
    669: 22373,
    670: 22373,
    671: 22373,
    672: 22373,
    673: 22373,
    674: 22373,
    675: 22373,
    676: 22373,
    677: 22373,
    678: 22373,
    679: 22373,
    680: 22373,
    681: 22373,
    682: 22373,
    683: 22373,
    684: 22373,
    685: 22373,
    686: 22373,
    687: 22373,
    688: 22374,
    689: 22375,
    690: 22375,
    691: 22376,
    692: 22376,
    693: 22376,
    694: 22376,
    695: 22376,
    696: 22376,
    697: 22376,
    698: 22376,
    699: 22376,
    700: 22376,
    701: 22376,
    702: 22377,
    703: 22378,
    704: 22378,
    705: 22378,
    706: 22378,
    707: 22378,
    708: 22378,
    709: 22378,
    710: 22378,
    711: 22379,
    712: 22379,
    713: 22379,
    714: 22379,
    715: 22379,
    716: 22379,
    717: 22379,
    718: 22379,
    719: 22379,
    720: 22379,
    721: 22379,
    722: 22379,
    723: 22379,
    724: 22379,
    725: 22379,
    726: 22379,
    727: 22379,
    728: 22379,
    729: 22379,
    730: 22379,
    731: 22379,
    732: 22379,
    733: 22379,
    734: 22379,
    735: 22379,
    736: 22379,
    737: 22379,
    738: 22379,
    739: 22379,
    740: 22379,
    741: 22379,
    742: 22379,
    743: 22379,
    744: 22380,
    745: 22380,
    746: 22380,
    747: 22380,
    748: 22380,
    749: 22380,
    750: 22380,
    751: 22380,
    752: 22380,
    753: 22380,
    754: 22380,
    755: 22380,
    756: 22380,
    757: 22380,
    758: 22380,
    759: 22380,
    760: 22380,
    761: 22380,
    762: 22380,
    763: 22380,
    764: 22380,
    765: 22380,
    766: 22380,
    767: 22380,
    768: 22380,
    769: 22380,
    770: 22380,
    771: 22380,
    772: 22380,
    773: 22380,
    774: 22380,
    775: 22380,
    776: 22380,
    777: 22380,
    778: 22380,
    779: 22380,
    780: 22380,
    781: 22380,
    782: 22380,
    783: 22380,
    784: 22380,
    785: 22380,
    786: 22380,
    787: 22380,
    788: 22380,
    789: 22380,
    790: 22381,
    791: 22381,
    792: 22381,
    793: 22381,
    794: 22381,
    795: 22381,
    796: 22381,
    797: 22381,
    798: 22381,
    799: 22381,
    800: 22381,
    801: 22381,
    802: 22381,
    803: 22381,
    804: 22381,
    805: 22381,
    806: 22381,
    807: 22381,
    808: 22381,
    809: 22381,
    810: 22381,
    811: 22381,
    812: 22381,
    813: 22381,
    814: 22381,
    815: 22381,
    816: 22381,
    817: 22381,
    818: 22381,
    819: 22381,
    820: 22381,
    821: 22382,
    822: 22382,
    823: 22382,
    824: 22382,
    825: 22382,
    826: 22382,
    827: 22382,
    828: 22382,
    829: 22382,
    830: 22382,
    831: 22382,
    832: 22382,
    833: 22382,
    834: 22382,
    835: 22382,
    836: 22382,
    837: 22382,
    838: 22382,
    839: 22382,
    840: 22382,
    841: 22382,
    842: 22382,
    843: 22382,
    844: 22382,
    845: 22382,
    846: 22382,
    847: 22382,
    848: 22382,
    849: 22382,
    850: 22382,
    851: 22382,
    852: 22382,
    853: 22382,
    854: 22382,
    855: 22382,
    856: 22382,
    857: 22382,
    858: 22382,
    859: 22382,
    860: 22382,
    861: 22382,
    862: 22382,
    863: 22382,
    864: 22383,
    865: 22383,
    866: 22383,
    867: 22383,
    868: 22383,
    869: 22383,
    870: 22383,
    871: 22383,
    872: 22383,
    873: 22383,
    874: 22383,
    875: 22384,
    876: 22384,
    877: 22384,
    878: 22384,
    879: 22384,
    880: 22384,
    881: 22384,
    882: 22384,
    883: 22384,
    884: 22384,
    885: 22384,
    886: 22384,
    887: 22385,
    888: 22385,
    889: 22385,
    890: 22385,
    891: 22385,
    892: 22385,
    893: 22385,
    894: 22385,
    895: 22385,
    896: 22385,
    897: 22385,
    898: 22385,
    899: 22385,
    900: 22385,
    901: 22385,
    902: 22385,
    903: 22385,
    904: 22385,
    905: 22385,
    906: 22385,
    907: 22385,
    908: 22385,
    909: 22385,
    910: 22385,
    911: 22385,
    912: 22385,
    913: 22385,
    914: 22385,
    915: 22385,
    916: 22385,
    917: 22385,
    918: 22385,
    919: 22385,
    920: 22385,
    921: 22385,
    922: 22385,
    923: 22385,
    924: 22385,
    925: 22385,
    926: 22385,
    927: 22385,
    928: 22385,
    929: 22385,
    930: 22385,
    931: 22385,
    932: 22385,
    933: 22385,
    934: 22385,
    935: 22385,
    936: 22385,
    937: 22385,
    938: 22385,
    939: 22385,
    940: 22385,
    941: 22385,
    942: 22385,
    943: 22385,
    944: 22385,
    945: 22385,
    946: 22385,
    947: 22385,
    948: 22385,
    949: 22385,
    950: 22385,
    951: 22385,
    952: 22385,
    953: 22385,
    954: 22385,
    955: 22385,
    956: 22385,
    957: 22385,
    958: 22385,
    959: 22385,
    960: 22385,
    961: 22385,
    962: 22385,
    963: 22385,
    964: 22385,
    965: 22385,
    966: 22385,
    967: 22385,
    968: 22385,
    969: 22385,
    970: 22385,
    971: 22385,
    972: 22385,
 973: 22385,
 974: 22385,
 975: 22385,
 976: 22385,
 977: 22385,
 978: 22385,
 979: 22385,
 980: 22385,
 981: 22385,
 982: 22385,
 983: 22385,
 984: 22385,
 985: 22385,
 986: 22385,
 987: 22385,
 988: 22385,
 989: 22385,
 990: 22385,
 991: 22385,
 992: 22385,
 993: 22386,
 994: 22386,
 995: 22386,
 996: 22386,
 997: 22386,
 998: 22386,
 999: 22386,
 ...}
```

In [13]:

```
x1= list(word_cnt.keys())
y1 = list(word_cnt.values())

plt.scatter(x = x1, y = y1)
plt.xlabel = ('doc_cnt(xi)')
plt.ylabel=  ("% of words <= xi")
```



ANSWER:

- Curve rises steeply at 0, so the choice of min_diff = 0 seems right. Curve starts to plateau somewhere at 22500. That could have seved as the max_df value.

The parameter α is chosen to be a small value that simply avoids having zeros in the probability computations. This value can sometimes be chosen arbitrarily with domain expertise, but we will use K-fold cross validation. In K-fold cross-validation, we divide the data into K non-overlapping parts. We train on $K-1$ of the folds and test on the remaining fold. We then iterate, so that each fold serves as the test fold exactly once. The function `cv_score` performs the K-fold cross-validation algorithm for us, but we need to pass a function that measures the performance of the algorithm on each fold.

In [14]:

```
from sklearn.model_selection import KFold
def cv_score(clf, X, y, scorefunc):
    result = 0.
    nfold = 5
    for train, test in KFold(nfold).split(X): # split data into train/test
groups, 5 times
        clf.fit(X[train], y[train]) # fit the classifier, passed is as clf.
        result += scorefunc(clf, X[test], y[test]) # evaluate score
function on held-out data
    return result / nfold # average
```

We use the log-likelihood as the score here in `scorefunc`. The higher the log-likelihood, the better. Indeed, what we do in `cv_score` above is to implement the cross-validation part of `GridSearchCV`.

The custom scoring function `scorefunc` allows us to use different metrics depending on the decision risk we care about (precision, accuracy, profit etc.) directly on the validation set. You will often find people using `roc_auc`, precision, recall, or `F1-score` as the scoring function.

In [15]:

```
def log_likelihood(clf, x, y):
    prob = clf.predict_log_proba(x)
    rotten = y == 0
    fresh = ~rotten
    return prob[rotten, 0].sum() + prob[fresh, 1].sum()
```

We'll cross-validate over the regularization parameter α.

Let's set up the train and test masks first, and then we can run the cross-validation procedure.

In [16]:

```
from sklearn.model_selection import train_test_split
_, itest = train_test_split(range(critics.shape[0]), train_size=0.7)
mask = np.zeros(critics.shape[0], dtype=np.bool)
mask[itest] = True
```

In [17]:

```
len(critics), len(itest) , len(mask)
```

Out[17]:

```
(15561, 4669, 15561)
```

Exercise Set IV

Exercise: What does using the function `log_likelihood` as the score mean? What are we trying to optimize for?

Exercise: Without writing any code, what do you think would happen if you choose a value of α that is too high?

Exercise: Using the skeleton code below, find the best values of the parameter `alpha`, and use the value of `min_df` you chose in the previous exercise set. Use the `cv_score` function above with the `log_likelihood` function for scoring.

Question :- What does using the function log_likelihood as the score mean? What are we trying to optimize for?

The log_likelihood function calculates the sum of the logs of the predicted probabilities for the categories (rotten/fresh) based on the words found in the document. We are trying to optimize for confidence level(α) in predicting correct probabilities.

Question:- Without writing any code, what do you think would happen if you choose a value of α that is too high?

It's important to choose optimum alpha so that the probability estimate does not exceed 1. If alpha is too high, all the probability estimates will hover around the middle value of 0.5. And if it's too low, then our log likelihoods will have -ve values that will bring down the overall score causing misleading results.

Using the skeleton code below, find the best values of the parameter alpha, and use the value of min_df you chose in the previous exercise set. Use the cv_score function above with the log_likelihood function for scoring.

In [18]:

```
from sklearn.naive_bayes import MultinomialNB

#the grid of parameters to search over
alphas = [.1, 1, 5, 10, 50]
best_min_df = 0 # YOUR TURN: put your value of min_df here.

log_lik = {}

#Find the best value for alpha and min_df, and the best classifier
best_alpha = None
maxscore=-np.inf
for alpha in alphas:
    vectorizer = CountVectorizer(min_df = best_min_df)
    Xthis, ythis = make_xy(critics, vectorizer)
    Xtrainthis = Xthis[mask]
    ytrainthis = ythis[mask]
    # your turn
    clf = MultinomialNB(alpha = alpha)
    clf.fit(Xtrainthis, ytrainthis)
    log_lik[alpha] = log_likelihood(clf, Xtrainthis, ytrainthis)

best_alpha = max(alphas, key=lambda x: log_lik[x])
print("Best alpha value is :" +str(best_alpha))
```

```
Best alpha value is :0.1
```

In [19]:

```
print("alpha: {}".format(best_alpha))
```

```
alpha: 0.1
```

Exercise Set V: Working with the Best Parameters

Exercise: Using the best value of `alpha` you just found, calculate the accuracy on the training and test sets. Is this classifier better? Why (not)?

In [20]:

```
vectorizer = CountVectorizer(min_df=best_min_df)
X, y = make_xy(critics, vectorizer)
xtrain=X[mask]
ytrain=y[mask]
xtest=X[~mask]
ytest=y[~mask]

clf = MultinomialNB(alpha=best_alpha).fit(xtrain, ytrain)

#your turn. Print the accuracy on the test and training dataset
training_accuracy = clf.score(xtrain, ytrain)
test_accuracy = clf.score(xtest, ytest)

print("Accuracy on training data: {:2f}".format(training_accuracy))
print("Accuracy on test data:     {:2f}".format(test_accuracy))
```

```
Accuracy on training data: 0.973656
Accuracy on test data:     0.732097
```

In [21]:

```
from sklearn.metrics import confusion_matrix
print(confusion_matrix(ytest, clf.predict(xtest)))
```

```
[[2815 1427]
 [1491 5159]]
```

Once again, this classifier does not yield very good result on the test dataset yet it performs well on the training dataset fetching 0.98 accuracy.

Interpretation

What are the strongly predictive features?

We use a neat trick to identify strongly predictive features (i.e. words).

- first, create a data set such that each row has exactly one feature. This is represented by the identity matrix.
- use the trained classifier to make predictions on this matrix
- sort the rows by predicted probabilities, and pick the top and bottom K rows

In [22]:

```
words = np.array(vectorizer.get_feature_names())

x = np.eye(xtest.shape[1])
probs = clf.predict_log_proba(x)[:, 0]
ind = np.argsort(probs)

good_words = words[ind[:10]]
bad_words = words[ind[-10:]]

good_prob = probs[ind[:10]]
bad_prob = probs[ind[-10:]]

print("Good words\t     P(fresh | word)")
for w, p in zip(good_words, good_prob):
    print("{:>20}".format(w), "{:.2f}".format(1 - np.exp(p)))

print("Bad words\t     P(fresh | word)")
for w, p in zip(bad_words, bad_prob):
    print("{:>20}".format(w), "{:.2f}".format(1 - np.exp(p)))
```

```
Good words	     P(fresh | word)
            touching 1.00
           excellent 1.00
               truth 0.99
              inside 0.99
             delight 0.99
              superb 0.99
            physical 0.99
             assured 0.99
            funniest 0.99
         astonishing 0.99
Bad words	     P(fresh | word)
            strictly 0.01
           obnoxious 0.01
                test 0.01
              boring 0.01
         uninvolving 0.01
       disappointing 0.01
                lame 0.01
             numbers 0.01
              cliche 0.01
       unfortunately 0.01
```

Exercise Set VI

Exercise: Why does this method work? What does the probability for each row in the identity matrix represent

The above exercise is an example of *feature selection*. There are many other feature selection methods. A list of feature selection methods available in `sklearn` is here. The most common feature selection technique for text mining is the chi-squared $\left(\chi^2\right)$ method.

Prediction Errors

We can see mis-predictions as well.

In [23]:

```
x, y = make_xy(critics, vectorizer)

prob = clf.predict_proba(x)[:, 0]
predict = clf.predict(x)

bad_rotten = np.argsort(prob[y == 0])[:5]
bad_fresh = np.argsort(prob[y == 1])[-5:]

print("Mis-predicted Rotten quotes")
print('---------------------------')
for row in bad_rotten:
    print(critics[y == 0].quote.iloc[row])
    print("")

print("Mis-predicted Fresh quotes")
print('--------------------------')
for row in bad_fresh:
    print(critics[y == 1].quote.iloc[row])
    print("")
```

```
Mis-predicted Rotten quotes
---------------------------
It must have been an act of great restraint for Sean Combs to resist titlin
g this film, about Chris Wallace, his close friend turned rapper and cultur
al icon, The Notorious B.I.G. -- The Sean Combs Story.

As a depiction of a loving-turbulent relationship between a single mom (Sus
an Sarandon) and her rebellious teenage daughter (Natalie Portman), Wang's
meller is nicely crafted but old-fashioned like Hollywood's weepies of yest
eryear.

Anne Frank Remembered tells the audience very little about Anne, but the fi
lm speaks volumes concerning the problematic aspects of Holocaust represent
ation.

Despite great scenery, the distinctive visual ideas of Mr. Scott (Alien, Bl
ade Runner) and the strong dramatic presence of Mr. Bridges, most of White
Squall remains listless and tame.
```

```
While Leone's vision still has a magnificent sweep, the film finally subsid
es to an emotional core that is sombre, even elegiac, and which centres on
a man who is bent and broken by time, and finally left with nothing but an
impotent sadness.

Mis-predicted Fresh quotes
--------------------------
Even if the plotting (a mistaken identity farce involving that old chestnut
, amnesia brought on by a bump to the head) is square as a square peg. Mado
nna has never found a better fit than the role of Susan.

Jessica Biel in a teacup-rattling '20s period piece? With her lewd pinup gr
in and husky flat voice, she sticks out like a sore starlet in Easy Virtue
-- but that's the whole point in this loosely 'freshened up' version of a N
oel Coward play.

Harmless family fare might be deemed politically correct for its concern fo
r endangered species, but whitewashing of Chinese regime will strike adults
... as somewhat bizarre.

The players, who include Simon Callow, Kristin Scott Thomas, Rowan Atkinson
and Sophie Thompson, exude comedic brightness as they go about their gossip
y, farcical, self-deprecating, sorry-about-that-old-chap, just-being-Englis
h business.

Madonna, making her directorial debut, aims for the romping irreverence of
Richard Lester's 60s comedies, and though she lacks the formal control to p
ull it off, this is a charming mess.
```

Exercise Set VII: Predicting the Freshness for a New Review

Exercise:

- Using your best trained classifier, predict the freshness of the following sentence: *'This movie is not remarkable, touching, or superb in any way'*
- Is the result what you'd expect? Why (not)?

In [24]:

```
#your turn
vector = vectorizer.transform(['This movie is not remarkable, touching, or superb in any way'])
NaiveBayes.predict_proba(vector)
```

Out[24]:

```
array([[  2.09442285e-04,   9.99790558e-01]])
```

Aside: TF-IDF Weighting for Term Importance

TF-IDF stands for

```
Term-Frequency X Inverse Document Frequency.
```

In the standard `CountVectorizer` model above, we used just the term frequency in a document of words in our vocabulary. In TF-IDF, we weight this term frequency by the inverse of its popularity in all documents. For example, if the word "movie" showed up in all the documents, it would not have much predictive value. It could actually be considered a stopword. By weighing its counts by 1 divided by its overall frequency, we downweight it. We can then use this TF-IDF weighted features as inputs to any classifier. **TF-IDF is essentially a measure of term importance, and of how discriminative a word is in a corpus.** There are a variety of nuances involved in computing TF-IDF, mainly involving where to add the smoothing term to avoid division by 0, or log of 0 errors. The formula for TF-IDF in `scikit-learn` differs from that of most textbooks:

$$\mbox{TF-IDF}(t, d) = \mbox{TF}(t, d)\times \mbox{IDF}(t) = n_{td} \log{\left(\frac{\vert D \vert}{\vert d : t \in d \vert} + 1 \right)}$$

where n_{td} is the number of times term t occurs in document d, $\vert D \vert$ is the number of documents, and $\vert d : t \in d \vert$ is the number of documents that contain t

In [25]:

```
# http://scikit-learn.org/dev/modules/feature_extraction.html#text-feature-
extraction
# http://scikit-learn.org/dev/modules/classes.html#text-feature-extraction-
ref
from sklearn.feature_extraction.text import TfidfVectorizer
tfidfvectorizer = TfidfVectorizer(min_df=1, stop_words='english')
Xtfidf=tfidfvectorizer.fit_transform(critics.quote)
```

In [ ]:

```

```