LDA

May 10, 2015

```
In [117]: import numpy as np
          import scipy as sp
          import scipy.stats as stats
          import codecs
          import nltk
          import lda
          import sklearn
          import string
          import cPickle as pickle
          import matplotlib.pyplot as plt
          import collections, operator
          import pandas as pd
          import seaborn as sns
          import matplotlib.gridspec as gridspec
          from matplotlib import animation
          from scipy.special import gammaln
          from nltk.corpus import stopwords
          from nltk.stem.porter import *
          from collections import Counter, defaultdict
          from sklearn.preprocessing import normalize
          from sklearn.feature_extraction.text import TfidfTransformer, CountVectorizer
          from collections import defaultdict
          from mpl_toolkits.mplot3d.axes3d import Axes3D
          from matplotlib.ticker import LinearLocator, FormatStrFormatter
          from wordcloud import WordCloud
          plt.style.use("ggplot"); plt.style.use("bmh");
          %matplotlib inline
```

1 AM 207 Final Project

1.0.1 Cole Diamond

1.0.2 Raphael Pestourie

1.0.3 Wei Dai

2 Collapsed Gibbs Sampler for LDA to Classify Books by Thematic Content

3 1. Introduction

LDA is a generative probabilistic model for collections of grouped discrete data. Each group is described as a random mixture over a set of latent topics where each topic is a discrete distribution over the collection's vocabulary. We use Gibbs sampling to sample from the posterior of the distribution described by LDA to extract thematic content from ten classic novels. We train on half of the pages, and perform inference on the remainder. We use nearest neighbor on the queried topic distibution to query the closest match. We were able to correctly label 100% of our test data with the correct title.

4 2. Methodology

4.1 2.1. Pre-processing

- Our first step is to load the data from a folder containing all ten of the classic novels which compose our training corpus

```
In [4]: import codecs
        books = ["beowulf.txt", "divine_comedy.txt", "dracula.txt", "frankenstein.txt", "huck_finn.txt"
        all_docs = []
        for book in books:
            with codecs.open('data/%s'%(book), 'r', encoding='utf-8') as f:
                lines = f.read().splitlines()
                all_docs.append(" ".join(lines))
```

- We remove punctuation and numbers from our books.
- Additionally, we remove stop words, or words that don't have much lexical meaning, ie: "the, is, at, which, on...".

```
In [5]: stemmer = PorterStemmer()

        # def remove_insignificant_words(processed_docs, min_thresh = 9, intra_doc_thresh = .9):
        #     all_tokens = np.unique([item for sublist in processed_docs for item in sublist])
        #     low_freq_words = [k for k, v in Counter(all_tokens).iteritems() if v < min_thresh]
        #     high_freq_words = []
        #     for word in all_tokens:
        #         num_docs_containing_word = np.sum(map(lambda doc: word in doc, processed_docs))
        #         if float(num_docs_containing_word) / len(processed_docs) >= intra_doc_thresh:
        #             high_freq_words.append(word)
        #     words_to_remove = set(low_freq_words + high_freq_words)
        #     return map(lambda doc_tokens: [w for w in doc_tokens if w not in words_to_remove], proces

        def stem_tokens(tokens, stemmer):
            stemmed = []
            for item in tokens:
                stemmed.append(stemmer.stem(item))
            return stemmed

        def tokenize_and_remove_grammar_numbers_stopwords(doc):
            doc = doc.lower()
            no_punctuation = re.sub(r'[^a-zA-Z\s]','',doc)
            tokens = nltk.word_tokenize(no_punctuation)
            filtered = [w for w in tokens if not w in stopwords.words('english')]
            #stemmed = stem_tokens(filtered, stemmer)
            #return stemmed
            return filtered

        processed_docs = np.array(map(tokenize_and_remove_grammar_numbers_stopwords, all_docs))
        #processed_docs = remove_insignificant_words(processed_docs, all_tokens)
```

```
In [43]: processed_docs[0][500:510]
```

```
Out[43]: [u'tread',
          u'warrior',
          u'mail',
          u'viii',
          u'english',
          u'translations',
          u'beowulf',
          u'professor',
          u'garnett',
          u'alone']
```

```
In [570]: np.save("temp_data/processed_docs.npy", processed_docs)
```

```
In [45]: processed_docs = np.load("temp_data/processed_docs.npy")
```

4.2 2.2 Build vocabulary

```
In [7]: vocab = np.unique(np.hstack(processed_docs.flat))
        vocab_dict = {}
        inv_vocab_dict = {}
        for idx, w in enumerate(vocab):
            vocab_dict[w] = idx
            inv_vocab_dict[idx] = w
```

```
In [8]: vocab[np.random.choice(vocab.size, 10)]
```

```
Out[8]: array([u'required', u'ethiop', u'thundercloven', u'sheetshelm', u'unferth',
               u'hushmoney', u'portray', u'running', u'harveys', u'moore'],
              dtype='<U69')
```

4.3 2.3 Map Docs to Vocab

- We now translate our documents into the language of numbers, allowing us to perform operations on our data

```
In [9]: docs_as_nums = map(lambda doc: [vocab_dict[w] for w in doc], processed_docs)
        docs_as_nums[0][:10]
```

```
Out[9]: [39038, 22075, 15318, 4485, 15318, 53645, 1932, 1940, 10849, 1375]
```

4.4 2.4 Remove Low Frequency Words and Words that Appear Across >= 90% of Documents

- We remove words that will contribute very little to the signal we use to distinguish documents

```
In [10]: def freq_map(doc):
             out = np.zeros(vocab.size, dtype=np.int32)
             for w in doc:
                 out[w] += 1
             return out
```

```
In [44]: count_mat =np.array(map(freq_map, np.array(docs_as_nums)), dtype=np.int32)
         low_freq_words = np.where(np.sum(count_mat != 0, axis=0) < 2)
         high_freq_words = np.where(np.sum(count_mat > 0, axis=0) >= .9*count_mat.shape[0])
         words_to_remove = np.unique(np.append(low_freq_words, high_freq_words))
```

```
In [45]: docs_as_nums = map(lambda doc: [word for word in doc if word not in words_to_remove], docs_as_r
```

```
In [46]: count_mat =np.array(map(freq_map, np.array(docs_as_nums)), dtype=np.int32)
```

```
In [47]: np.save("temp_data/docs_as_nums.npy", np.array(docs_as_nums))
         docs_as_nums = np.load("temp_data/docs_as_nums.npy")
```

4.5 2.5 Build Training and Test Set

- We split each of the books in half to use a training data and as test data, respectively.

```
In [48]: test_docs, train_docs = [], []
         for doc in docs_as_nums:
             test_docs.append(np.array(doc[0:len(doc)/2]))
             train_docs.append(np.array(doc[len(doc)/2:]))
         test_docs, train_docs = np.array(test_docs), np.array(train_docs)
```

```
In [49]: test_docs
```

```
Out[49]: array([array([56863,  1728, 16395, ..., 53486, 29188,  3112]),
                array([22076, 14172,  9563, ..., 50365, 13691, 50589]),
                array([ 3030,  8565, 39104, ...,  2742, 31975, 33093]),
                array([22076, 30986, 44646, ..., 17849, 19401, 16044]),
                array([  649, 18404, 52126, ..., 37781, 32764, 28716]),
                array([13364, 55565, 56864, ..., 21574, 43539, 41253]),
                array([22076,   649, 44675, ..., 11543, 36011, 16842]),
                array([ 8731,  8138, 13368, ..., 27002, 39068, 32912]),
                array([41306, 37635, 56864, ..., 36256, 13035, 52297]),
                array([52350, 27031, 56864, ...,  9509, 15737, 19274])], dtype=object)
```

4.6 2.6 Build a Count Matrix

A count matrix is built by setting each row equal to the number of times a vocabulary word is used in a document. The count matrix has dimensions (num_docs x size_of_vocab). We need the count matrix because our LDA function will take it as an input.

```
In [50]: train_count_mat = np.array(map(freq_map, train_docs), dtype=np.int32)
         test_count_mat = np.array(map(freq_map, test_docs), dtype=np.int32)
```

```
In [51]: train_count_mat
```

```
Out[51]: array([[0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0],
                ...,
                [0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0]], dtype=int32)
```

```
In [52]: np.save("temp_data/train_count_mat.npy", train_count_mat)
         np.save("temp_data/test_count_mat.npy", test_count_mat)
```

```
In [53]: train_count_mat = np.load("temp_data/train_count_mat.npy")
         test_count_mat = np.load("temp_data/test_count_mat.npy")
```

4.7 3. LDA with Gibbs Sampling

LDA is a generative probabilistic model for collections of grouped discrete data. Each group is described as a random mixture over a set of latent topics where each topic is a discrete distribution over the collection's vocabulary. Algorithm 1 delineates how we can draw from the posterior of the LDA model using Gibbs Sampling

We define the following parameters whose relationship is described by the plate notation in Figure 1.

- α is the parameter of the Dirichlet prior on the per-document topic distributions,
- β is the parameter of the Dirichlet prior on the per-topic word distribution,
- θ_i is the topic distribution for document i,
- ϕ_k is the word distribution for topic k,
- z_{ij} is the topic for the jth word in document i, and
- w_{ij} is the specific word.

- First, let's define our conditional distribution

```
In [54]: def conditional_dist(alpha, beta, nwt, nd, nt, d, w):
             """
             Compute the conditional distribution
             """
             W = nwt.shape[0]
             p_z =  (ndt[d,:] + alpha) * ((nwt[w,:] + beta) / (nt + beta * W))
             # normalization
             p_z /= np.sum(p_z)
             return p_z
```

- We'll also need the log likelihood to verify that our model is converging

```
In [55]: def log_likelihood(alpha, beta, nwt, ndt, n_topics):
             """
             Compute the likelihood that the model generated the data.
             """
             W = nwt.shape[0]
             n_docs = ndt.shape[0]
             likelihood = 0

             for t in xrange(n_topics):
                 likelihood += log_multinomial_beta(nwt[:,t]+beta) - log_multinomial_beta(beta, W)

             for d in xrange(n_docs):
                 likelihood += log_multinomial_beta(ndt[d,:]+alpha) - log_multinomial_beta(alpha, n_topi

             return likelihood

         def log_multinomial_beta(alpha, K=None):
             """
             Logarithm of the multinomial beta function.
             """
             if K is None:
                 return np.sum(gammaln(alpha)) - gammaln(np.sum(alpha))
             else:
                 return K * gammaln(alpha) - gammaln(K*alpha)
```

- Since our input is a count matrix, we need to recover our document by multiplying the token by its frequency and combining (in any order since we have a bag of words assumption)

```
In [56]: def word_indices(arr):
             """
             Transform a row of the count matrix into a document by replicating the token by its freque
             """
             for idx in arr.nonzero()[0]:
                 for i in xrange(int(arr[idx])):
                     yield idx
```

- To perform LDA with Gibbs Sampling we need to initialize z randomly and initialize our counters.
- We set the number of topics to 1000.

```
In [57]: n_topics = 15
         alpha = .1 # prior weight of topic k in a document; few topics per document
         beta = 0.05 # prior weight of word w in a topic; few words per topic
         n_docs, W = train_count_mat.shape
         # number of times document m and topic z co-occur
         ndt = np.zeros((n_docs, n_topics))
         # number of times word w and topic z co-occur
         nwt = np.zeros((W, n_topics))
         nd = np.zeros(n_docs)
         nt = np.zeros(n_topics)
         iters = 25
         topics = defaultdict(dict)
         delta_topics = []
         delta_doc_topics = defaultdict(list)
         likelihoods = []

         for d in xrange(n_docs):
             # i is a number between 0 and doc_length-1
             # w is a number between 0 and W-1
             for i, w in enumerate(word_indices(train_count_mat[d, :])):
                 # choose an arbitrary topic as first topic for word i
                 t = np.random.randint(n_topics)
                 ndt[d,t] += 1
                 nd[d] += 1
                 nwt[w,t] += 1
                 nt[t] += 1
                 topics[d][i] = t
```

- Now, we do Gibbs sampling for 25 iterations

```
In [58]: # for each iteration
         for it in xrange(iters):
             delta_topics_iteration = 0
             # for each doc
             for d in xrange(n_docs):
                 delta_doc_topics_iteration = 0
                 # for each word
                 for i, w in enumerate(word_indices(train_count_mat[d, :])):
                     # get topic of mth document, ith word
                     t = topics[d][i]
                     # decrement counters
                     ndt[d,t] -= 1; nd[d] -= 1; nwt[w,t] -= 1; nt[t] -= 1

                     p_z = conditional_dist(alpha, beta, nwt, nd, nt, d, w)
```

```
                t = np.random.multinomial(1,p_z).argmax()

                # increment counters
                ndt[d,t] += 1; nd[d] += 1; nwt[w,t] += 1; nt[t] += 1;
                # increment convergence counter if the value for topic changes
                if topics[d][i] != t:
                    delta_doc_topics_iteration += 1
                    delta_topics_iteration += 1

                topics[d][i] = t

            delta_doc_topics[d].append(delta_doc_topics_iteration)


        print "-"*50, "\n Iteration", it+1, "\n", "-"*50, "\n"
        likelihood = log_likelihood(alpha, beta, nwt, ndt, n_topics)
        print "Likelihood", likelihood
        likelihoods.append(likelihood)
        print "Delta topics", delta_topics_iteration, "\n"
        delta_topics.append(delta_topics_iteration)
```

```
--------------------------------------------------
 Iteration 1
--------------------------------------------------

Likelihood -1509346.52104
Delta topics 111125

--------------------------------------------------
 Iteration 2
--------------------------------------------------

Likelihood -1461240.90892
Delta topics 92657

--------------------------------------------------
 Iteration 3
--------------------------------------------------

Likelihood -1431243.70243
Delta topics 83071

--------------------------------------------------
 Iteration 4
--------------------------------------------------

Likelihood -1403205.95332
Delta topics 75952

--------------------------------------------------
 Iteration 5
--------------------------------------------------
```

```
Likelihood -1375892.45689
Delta topics 69177

--------------------------------------------------
 Iteration 6
--------------------------------------------------

Likelihood -1352426.21036
Delta topics 62897

--------------------------------------------------
 Iteration 7
--------------------------------------------------

Likelihood -1333683.48167
Delta topics 57406

--------------------------------------------------
 Iteration 8
--------------------------------------------------

Likelihood -1318235.17813
Delta topics 52978

--------------------------------------------------
 Iteration 9
--------------------------------------------------

Likelihood -1307237.20132
Delta topics 49550

--------------------------------------------------
 Iteration 10
--------------------------------------------------

Likelihood -1297412.91813
Delta topics 47382

--------------------------------------------------
 Iteration 11
--------------------------------------------------

Likelihood -1290103.32715
Delta topics 45165

--------------------------------------------------
 Iteration 12
--------------------------------------------------

Likelihood -1284317.27549
Delta topics 43606

--------------------------------------------------
 Iteration 13
```

```
--------------------------------------------------

Likelihood -1279485.59909
Delta topics 42267

--------------------------------------------------
 Iteration 14
--------------------------------------------------

Likelihood -1275789.80175
Delta topics 41323

--------------------------------------------------
 Iteration 15
--------------------------------------------------

Likelihood -1271979.01508
Delta topics 40635

--------------------------------------------------
 Iteration 16
--------------------------------------------------

Likelihood -1269073.96222
Delta topics 39917

--------------------------------------------------
 Iteration 17
--------------------------------------------------

Likelihood -1266772.80354
Delta topics 39140

--------------------------------------------------
 Iteration 18
--------------------------------------------------

Likelihood -1264857.20525
Delta topics 38478

--------------------------------------------------
 Iteration 19
--------------------------------------------------

Likelihood -1263315.6469
Delta topics 38290

--------------------------------------------------
 Iteration 20
--------------------------------------------------

Likelihood -1261075.70762
Delta topics 38180
```

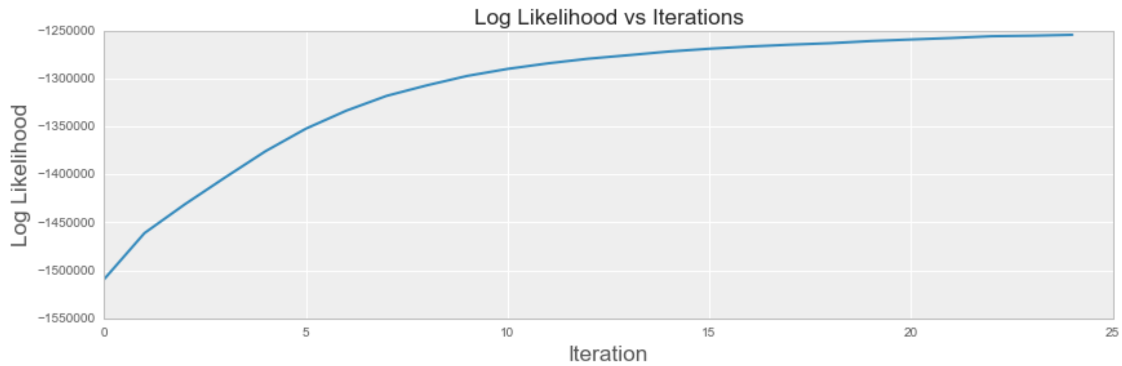
```
--------------------------------------------------
 Iteration 21
--------------------------------------------------

Likelihood -1259446.66802
Delta topics 37530

--------------------------------------------------
 Iteration 22
--------------------------------------------------

Likelihood -1257919.33876
Delta topics 37167

--------------------------------------------------
 Iteration 23
--------------------------------------------------

Likelihood -1256020.67474
Delta topics 36837

--------------------------------------------------
 Iteration 24
--------------------------------------------------

Likelihood -1255487.62725
Delta topics 36557

--------------------------------------------------
 Iteration 25
--------------------------------------------------

Likelihood -1254576.01859
Delta topics 36666
```

5 4. Analysis

5.0.1 4.1 Log Likelihood

We verify that the likelihood that our model generated the data increases over ever iteration. For convergence, we want to see a plateau, such that we are seeing diminishing gains in our log likelihood. As the graph below illustrates, this is exactly the case.

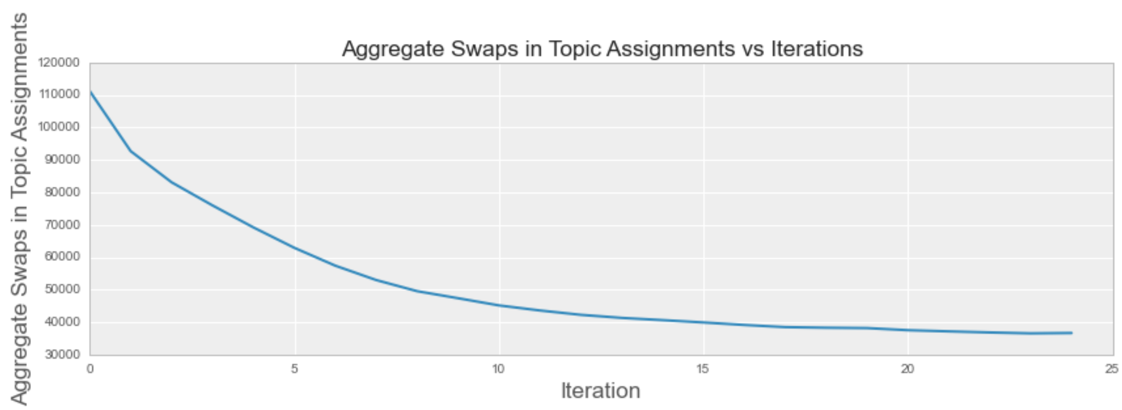
```
In [149]: plt.style.use("ggplot");plt.style.use("bmh");

          ax = plt.figure(figsize=(14,4))
          plt.plot(np.arange(25), likelihoods)
          plt.title("Log Likelihood vs Iterations", fontsize="xx-large")
          plt.xlabel("Iteration", fontsize="xx-large")
          plt.ylabel("Log Likelihood", fontsize="xx-large")
          plt.show()
```



5.0.2 4.2 Aggregate Word-Topic Assignment Swaps

We present a custom statistic to measure the total number of words whose topic assignment changed between iterations. We know that if the algorithm converges, the number of swaps every iteration should level out. The graph below illustrates this trend.

```
In [60]: plt.figure(figsize=(14,4))
         plt.plot(np.arange(iters), delta_topics)
         plt.title("Aggregate Swaps in Topic Assignments vs Iterations", fontsize="xx-large")
         plt.xlabel("Iteration", fontsize="xx-large")
         plt.ylabel("Aggregate Swaps in Topic Assignments", fontsize="xx-large")
         plt.show()
```

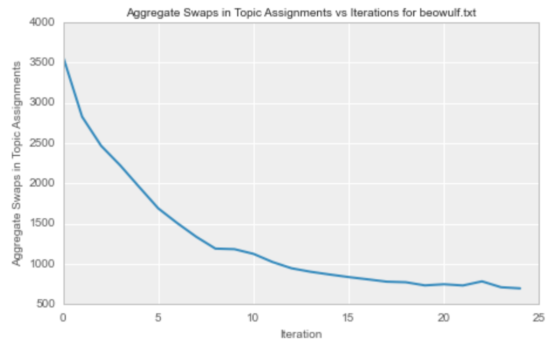


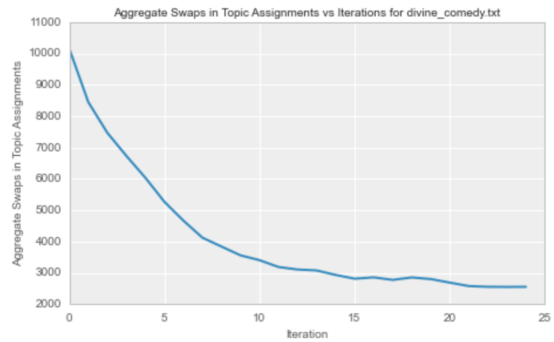
5.0.3 4.3 Aggregate Word-Topic Assignment Swaps per Document

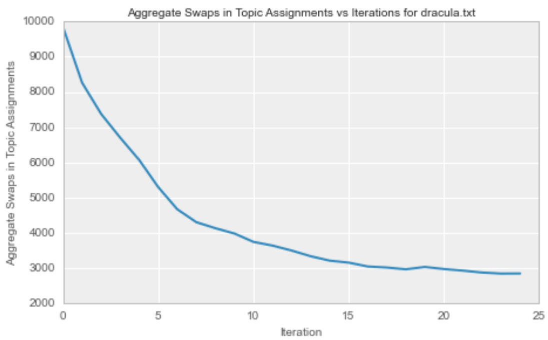
We apply the word-topic assignment swaps to a per-document basis. We should still see that on a document granularity, word-topic assignments should plateau. Each of the ten documents below illustrate this trend

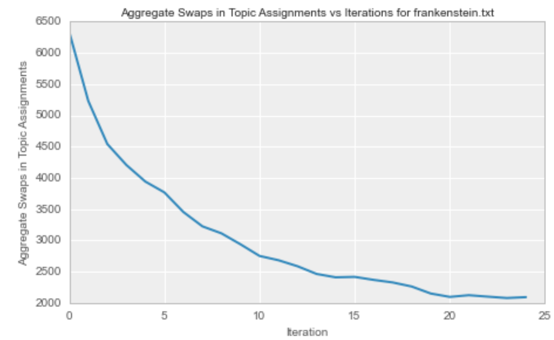
```
In [61]: plt.figure(figsize=(16,25))
         gs = gridspec.GridSpec(5, 2)

         for i in range(len(books)):
             ax = plt.subplot(gs[i])
```

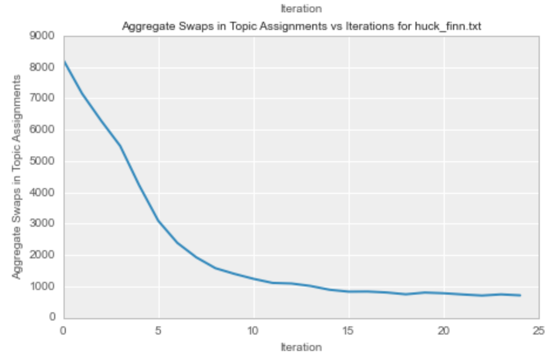
```
    ax.plot(np.arange(iters), delta_doc_topics[i])
    ax.set_title("Aggregate Swaps in Topic Assignments vs Iterations for %s" %(books[i]), font
    ax.set_xlabel("Iteration", fontsize="medium")
    ax.set_ylabel("Aggregate Swaps in Topic Assignments", fontsize="medium")

plt.show()
```

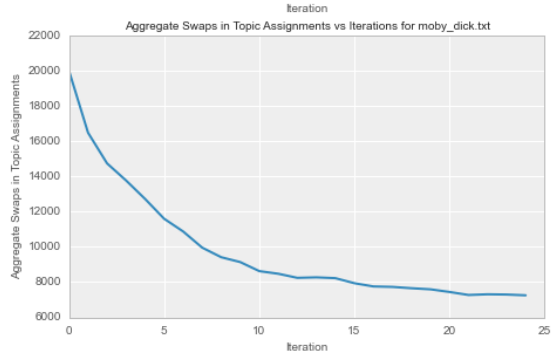

Aggregate Swaps in Topic Assignments vs Iterations for beowulf.txt
Aggregate Swaps in Topic Assignments
Iteration

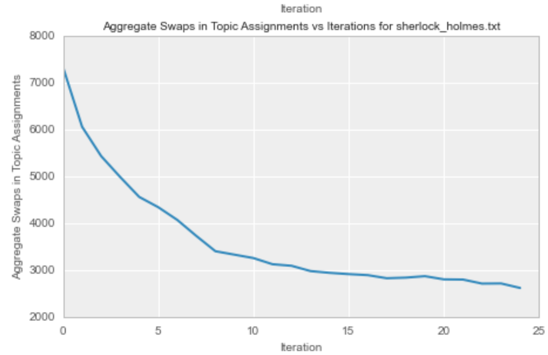

Aggregate Swaps in Topic Assignments vs Iterations for divine_comedy.txt
Aggregate Swaps in Topic Assignments
Iteration

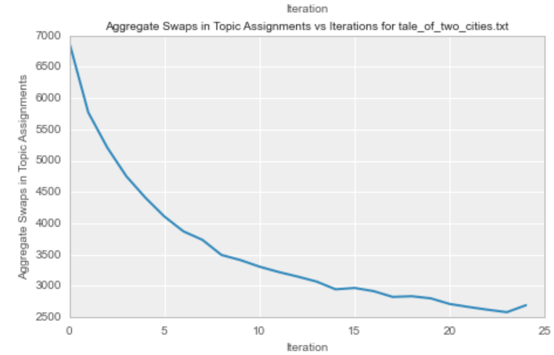

Aggregate Swaps in Topic Assignments vs Iterations for dracula.txt
Aggregate Swaps in Topic Assignments
Iteration

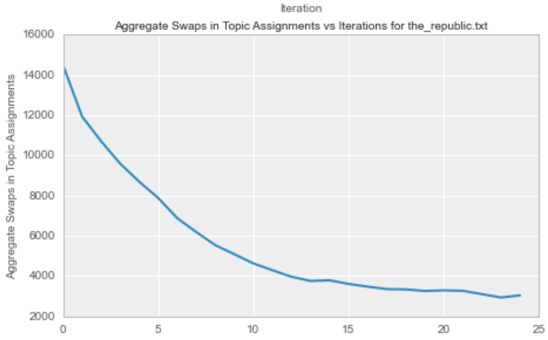

Aggregate Swaps in Topic Assignments vs Iterations for frankenstein.txt
Aggregate Swaps in Topic Assignments
Iteration

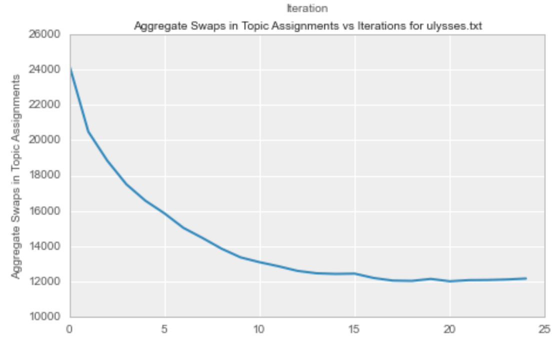

Aggregate Swaps in Topic Assignments vs Iterations for huck_finn.txt
Aggregate Swaps in Topic Assignments
Iteration


Aggregate Swaps in Topic Assignments vs Iterations for moby_dick.txt
Aggregate Swaps in Topic Assignments
Iteration


Aggregate Swaps in Topic Assignments vs Iterations for sherlock_holmes.txt
Aggregate Swaps in Topic Assignments
Iteration


Aggregate Swaps in Topic Assignments vs Iterations for tale_of_two_cities.txt
Aggregate Swaps in Topic Assignments
Iteration

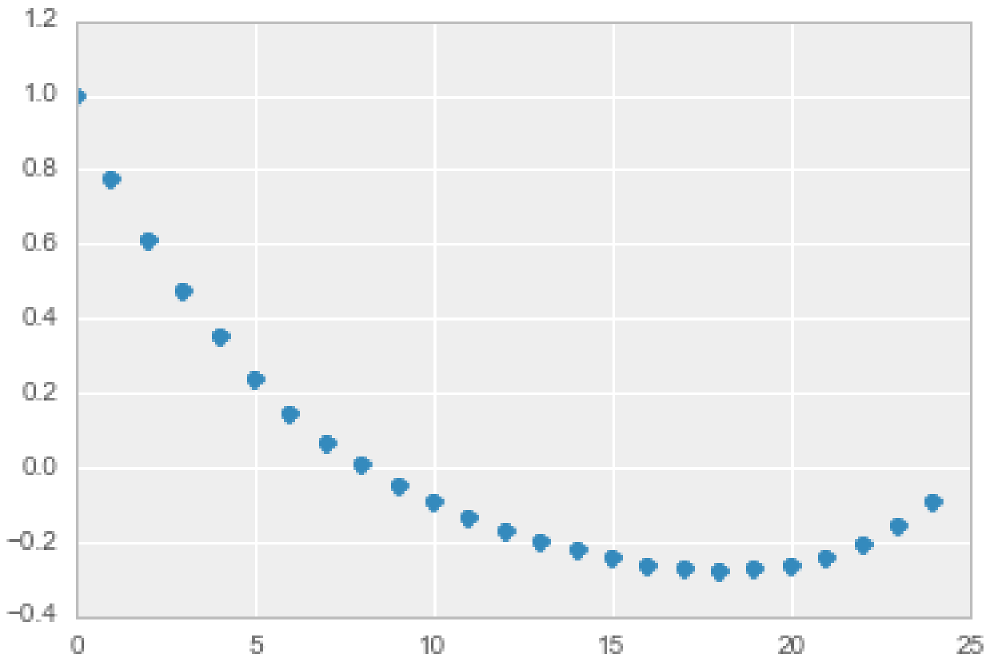

Aggregate Swaps in Topic Assignments vs Iterations for the_republic.txt
Aggregate Swaps in Topic Assignments
Iteration


Aggregate Swaps in Topic Assignments vs Iterations for ulysses.txt
Aggregate Swaps in Topic Assignments
Iteration

5.0.4 4.4 Autocorrelation of Swaps

In [62]:
```
plt.acorr(delta_topics-np.mean(delta_topics),  normed=True, usevlines=False, maxlags=iters-1, l
plt.xlim([0,iters])
```

Out[62]: (0, 25)



5.0.5 4.5 Topics as a Distribution over Words

- One important output of LDA is a matrix of topics where each topic is a distribution over the vocabulary.
- We want to verify that we observe only a few high-mass words per topic since we set our beta parameter to a small number (.5)

In [63]:
```
topic_words = defaultdict(lambda: [])
for d in xrange(n_docs):
        for i, w in enumerate(word_indices(train_count_mat[d, :])):
            t = topics[d][i]
            topic_words[t].append(inv_vocab_dict[w])

# Normalize
for topic in topic_words.keys():
    norm_topic_words = Counter(topic_words[topic])
    total = sum(norm_topic_words.values(), 0.0)
    for key in norm_topic_words:
        norm_topic_words[key] /= total
    topic_words[topic] = norm_topic_words
```

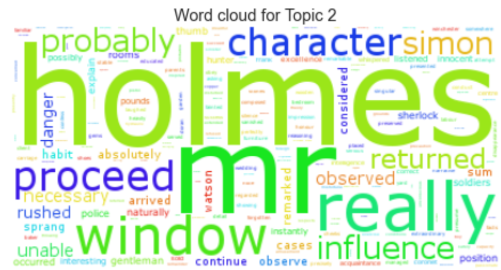
- Let's see what sort of topics LDA discovered. We will choose two topics at random

```
In [64]: for i in np.random.choice(n_topics, 2):
             if topic_words[i]:
                 sorted_topic_words = sorted(topic_words[i].items(), key=operator.itemgetter(1), reverse
                 print "\nMost important words for topic", i
                 for word in sorted_topic_words[:10]:
                     print word[0], word[1]
```

```
Most important words for topic 0
thou 0.0776546939689
thy 0.0375965088956
thee 0.0286449591586
hath 0.0129797471187
spake 0.010518070941
een 0.00805639476334
cried 0.00794450039163
beheld 0.00749692290478
lo 0.00570661295737
doth 0.00559471858566

Most important words for topic 2
holmes 0.0204101280447
mr 0.0155964186002
really 0.0118417252335
proceed 0.00616154808896
influence 0.00548762876673
window 0.00539135457784
character 0.00529508038895
probably 0.00510253201117
observe 0.0048137094445
danger 0.0048137094445
```

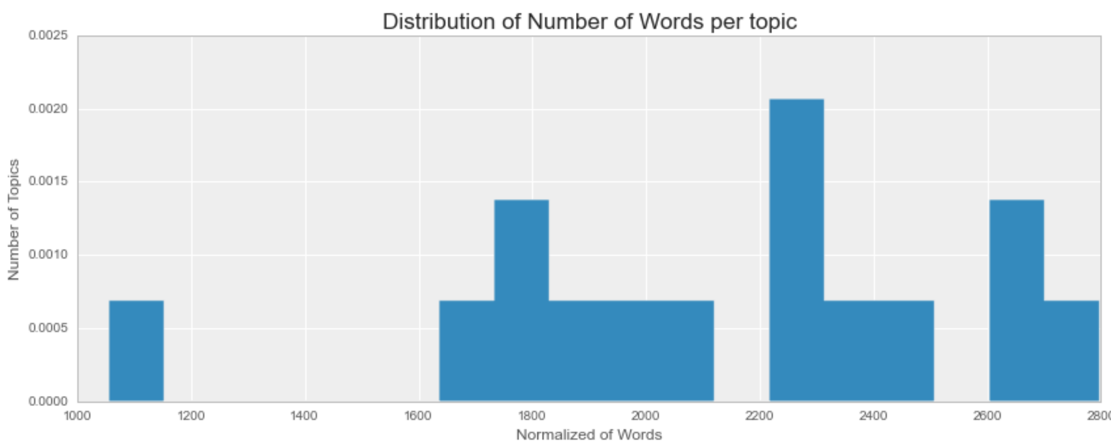
- We can also visualize these topics as **wordclouds**

```
In [65]: plt.figure(figsize=(17,10))
         gs = gridspec.GridSpec(1, 2)

         ax = plt.subplot(gs[0])
         wc = WordCloud(font_path="Verdana.ttf", background_color="white")
         wc.generate(" ".join([ (" " + word[0])*int(1000*word[1]) for word in topic_words[0].items()]))
         ax.imshow(wc)
         plt.axis("off")
         ax.set_title("Word cloud for Topic 47")

         ax = plt.subplot(gs[1])
         wc = WordCloud(font_path="Verdana.ttf", background_color="white")
         wc.generate(" ".join([ (" " + word[0])*int(1000*word[1]) for word in topic_words[2].items()]))
         plt.imshow(wc)
         plt.axis("off")
         ax.set_title("Word cloud for Topic 2")

         plt.show()
```



- Because we set our parameters to ensure sparsity over topics, each topic should be only described by a few words. Let's see a histogram to verify that the sparsity constraint was realized.

```
In [68]: num_words_per_topic = [len(words) for topic, words in topic_words.iteritems()]
         plt.figure(figsize=(14,5))
         plt.hist(num_words_per_topic, bins=18, normed=True, histtype='stepfilled')
         plt.title("Distribution of Number of Words per topic", fontsize="xx-large")
         plt.xlabel("Normalized of Words")
         plt.ylabel("Number of Topics")
         plt.show()
```



5.0.6 4.6 Documents as a Distribution over Topics

- Let's find our topic distributions over the train documents.
- We want to verify that we observe few high-mass topics per document since we set our alpha parameter to a large number (.8)

```
In [69]: train_doc_topic_dist = np.zeros((n_docs, n_topics))
         for d in xrange(n_docs):
             # for each word
             for i, w in enumerate(word_indices(train_count_mat[d, :])):
                 # get topic of mth document, ith word
                 z = topics[d][i]
                 train_doc_topic_dist[d, z] += 1
```

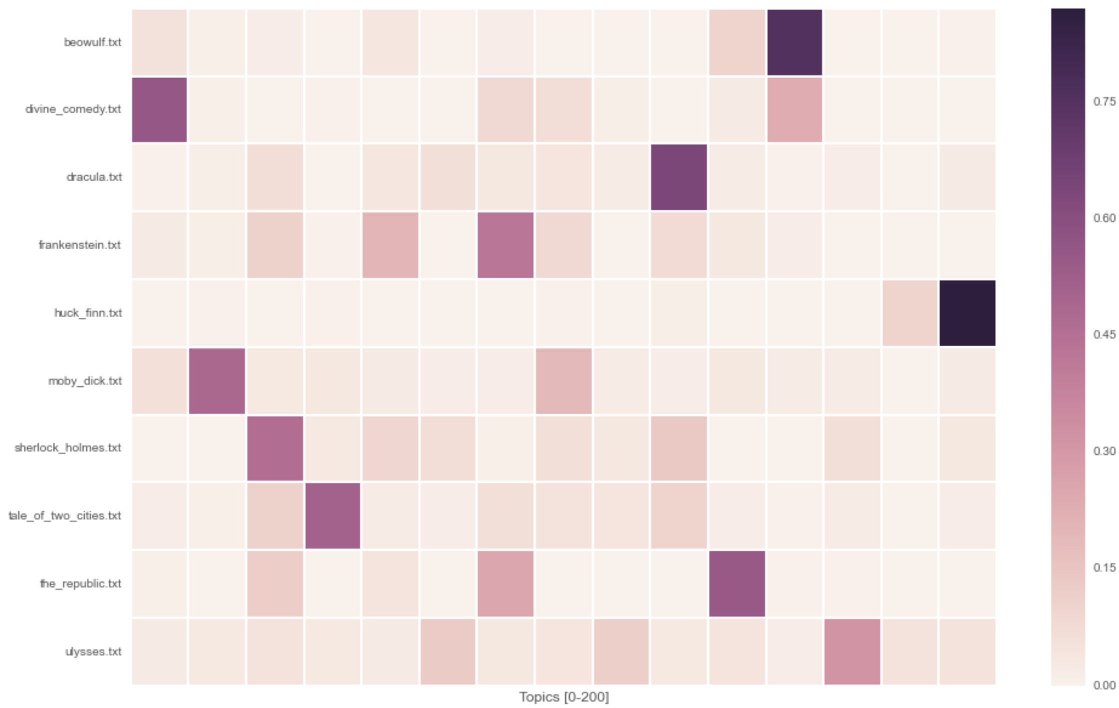
```
# NORMALIZE TOPIC DISTRIBUTION
row_sums = train_doc_topic_dist.sum(axis=1)
train_doc_topic_dist = train_doc_topic_dist / row_sums[:, np.newaxis]
```

In [70]:
```
doc_topic_dist_df = pd.DataFrame(train_doc_topic_dist, columns=(["Topic " + str(i) for i in ra
doc_topic_dist_df
```

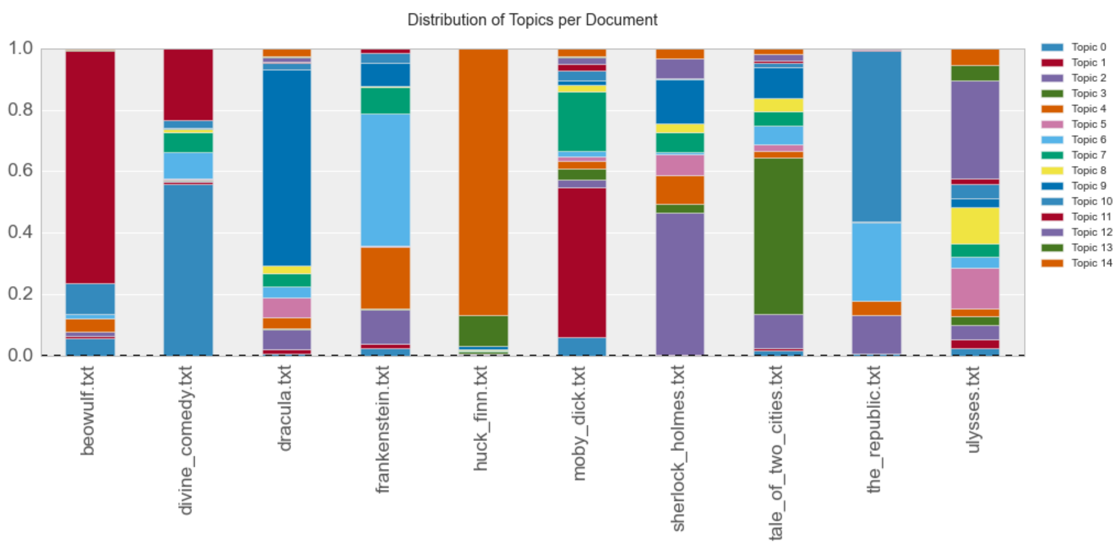
Out[70]:
```
                         Topic 0   Topic 1   Topic 2   Topic 3   Topic 4  \
beowulf.txt             0.055384  0.007986  0.015714  0.000000  0.040443
divine_comedy.txt       0.557911  0.008299  0.002616  0.003969  0.000180
dracula.txt             0.006229  0.012829  0.066654  0.001766  0.038208
frankenstein.txt        0.025014  0.013293  0.110635  0.005432  0.200686
huck_finn.txt           0.000000  0.003541  0.002435  0.005976  0.000000
moby_dick.txt           0.059602  0.487327  0.027262  0.033142  0.024990
sherlock_holmes.txt     0.000000  0.002174  0.461957  0.028502  0.092633
tale_of_two_cities.txt  0.015789  0.008084  0.109511  0.510294  0.022862
the_republic.txt        0.006993  0.000181  0.124299  0.001266  0.046718
ulysses.txt             0.024600  0.027245  0.048718  0.028413  0.024943

                         Topic 5   Topic 6   Topic 7   Topic 8   Topic 9  \
beowulf.txt             0.000000  0.015456  0.000000  0.000000  0.000258
divine_comedy.txt       0.002977  0.084341  0.067743  0.010734  0.003157
dracula.txt             0.063494  0.036627  0.043321  0.023520  0.639212
frankenstein.txt        0.003288  0.429960  0.085620  0.001858  0.076329
huck_finn.txt           0.000553  0.000000  0.003984  0.002435  0.011510
moby_dick.txt           0.016304  0.018620  0.190432  0.022050  0.016972
sherlock_holmes.txt     0.069082  0.009420  0.061957  0.030918  0.142633
tale_of_two_cities.txt  0.019452  0.061261  0.049135  0.042440  0.098901
the_republic.txt        0.000000  0.251733  0.002050  0.000000  0.003135
ulysses.txt             0.132309  0.033773  0.043634  0.118257  0.028620

                        Topic 10  Topic 11  Topic 12  Topic 13  Topic 14
beowulf.txt             0.102009  0.756054  0.000000  0.002834  0.003864
divine_comedy.txt       0.023814  0.233899  0.000271  0.000090  0.000000
dracula.txt             0.022218  0.003533  0.014037  0.001766  0.026587
frankenstein.txt        0.030875  0.014723  0.001715  0.000572  0.000000
huck_finn.txt           0.000111  0.000000  0.000000  0.099712  0.869743
moby_dick.txt           0.031583  0.021204  0.021605  0.002539  0.026371
sherlock_holmes.txt     0.003140  0.000000  0.062923  0.000121  0.034541
tale_of_two_cities.txt  0.016294  0.005179  0.020968  0.000758  0.019073
the_republic.txt        0.553680  0.004280  0.004039  0.001386  0.000241
ulysses.txt             0.047516  0.017316  0.318869  0.051673  0.054113
```

First, we can look at a heatmap of our topics over documents

In [71]:
```
plt.figure(figsize=(16,10))
sns.heatmap(doc_topic_dist_df)
plt.gca().axes.get_xaxis().set_ticks([])
plt.xlabel("Topics [0-200]")
plt.show()
```



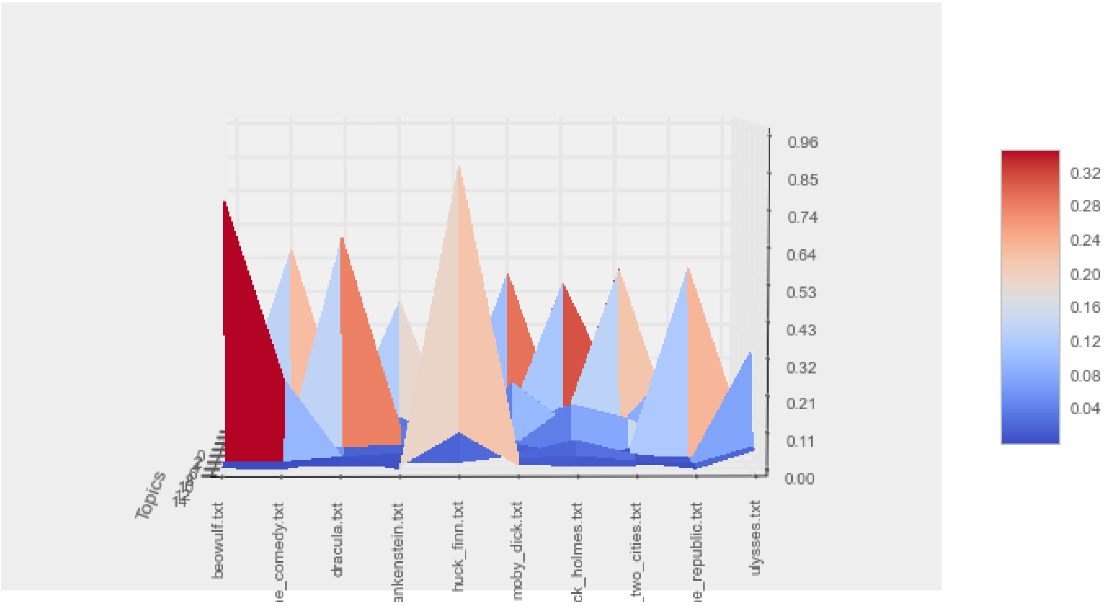

```
In [93]: doc_topic_dist_df.plot(kind='bar', figsize=(16,5), stacked="true", title="Distribution of Topi
         plt.legend(bbox_to_anchor=(1.1,1.05))
         plt.show()
```



5.0.7 4.7 Topic Landscape of Documents

```
In [106]: # Create an init function and the animate functions.
          # Both are explained in the tutorial. Since we are changing
```

```
# the the elevation and azimuth and no objects are really
# changed on the plot we don't have to return anything from
# the init and animate function. (return value is explained
# in the tutorial.
def init():
    # Create a figure and a 3D Axes
    xx,yy = np.meshgrid(np.arange(n_topics),np.arange(n_docs)) # Define a mesh grid in the re
    zz=train_doc_topic_dist
    surf = ax.plot_surface(xx, yy, zz, rstride=1, cstride=1, cmap=plt.cm.coolwarm, linewidth=
    ax.view_init(elev=50., azim=250)
    ax.set_zlim(0.0001, np.max(train_doc_topic_dist)*1.1)
    ax.zaxis.set_major_locator(LinearLocator(10))
    ax.zaxis.set_major_formatter(FormatStrFormatter('%.02f'))
    ax.set_yticklabels(books, rotation='vertical')
    ax.set_xlabel("Topics")
    ax.set_ylabel("")
    fig.colorbar(surf, shrink=0.5, aspect=5)

def animate(i):
    ax.view_init(elev=5., azim=i)

# Animate
fig = plt.figure(figsize=(10,5))
ax = Axes3D(fig)
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=360, interval=20, blit=True)
# Save
anim.save('ipynb_assets/topic_dist_3D.mp4', fps=30, extra_args=['-vcodec', 'libx264'])
```



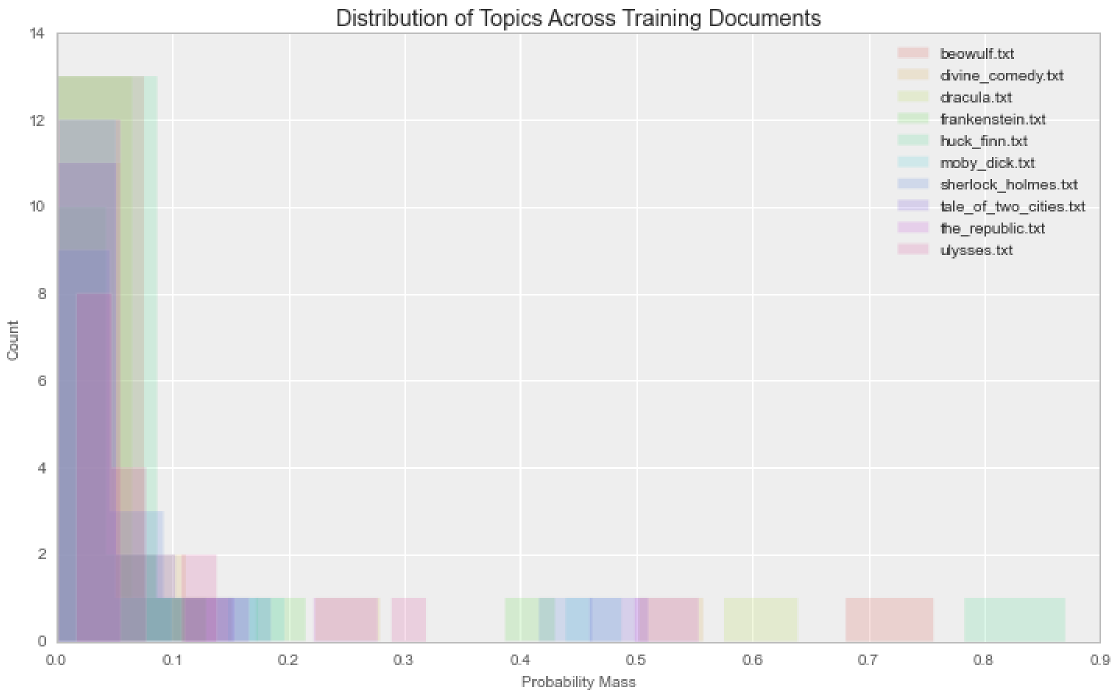

```
In [107]: from IPython.display import HTML
          from base64 import b64encode
          video = open("ipynb_assets/topic_dist_3D.mp4", "rb").read()
```

```
video_encoded = b64encode(video)
video_tag = '<video controls alt="test" src="data:video/x-m4v;base64,{0}">'.format(video_encod
HTML(data=video_tag)
```

Out[107]: <IPython.core.display.HTML object>

5.0.8 4.8 One-Dimensional Histogram of Topics over Documents

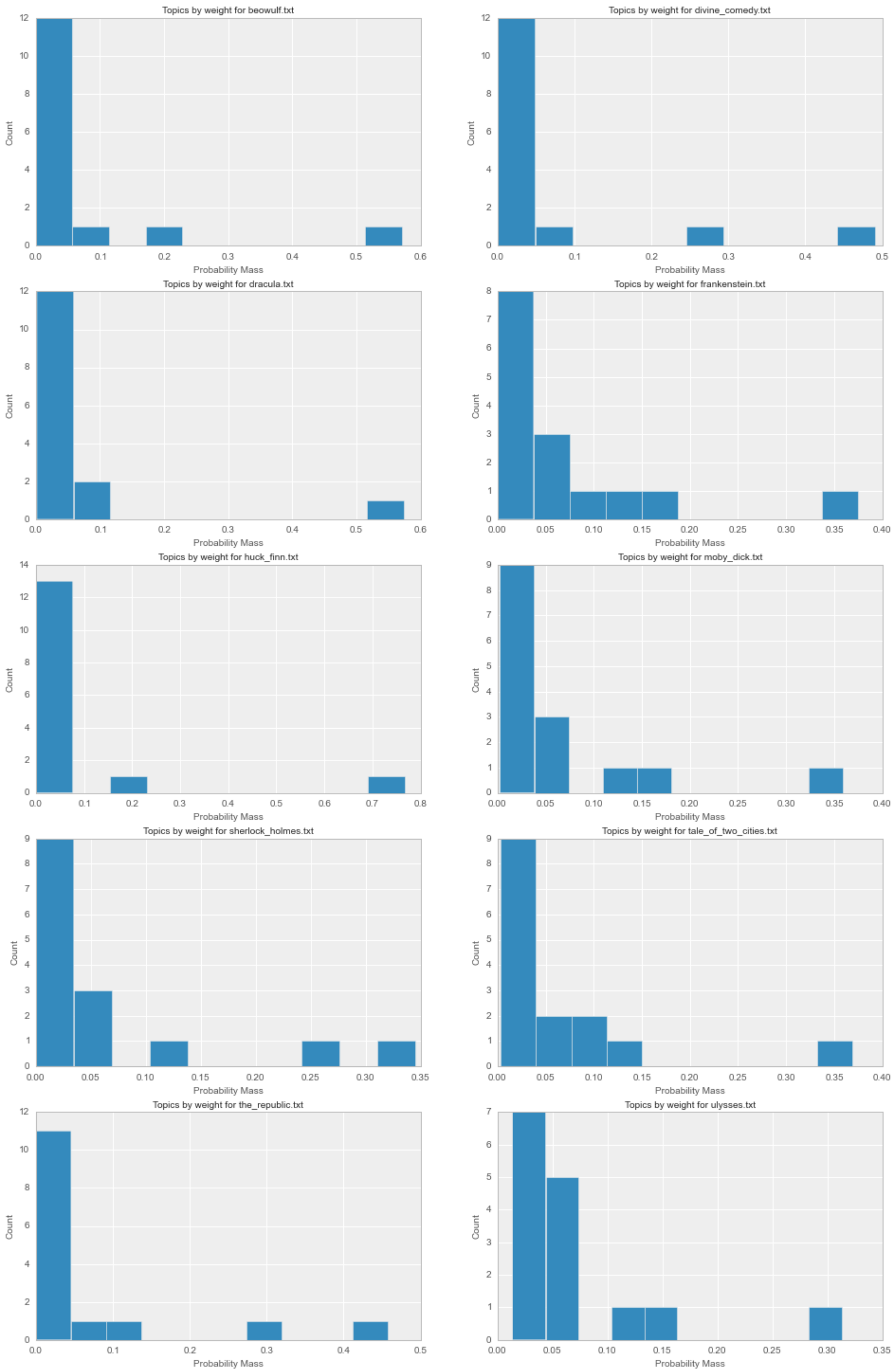
In [85]:
```
plt.figure(figsize=(12,7))
c = sns.color_palette("hls", n_docs)

for i in range(n_docs):
    plt.hist(train_doc_topic_dist[i, :], label=books[i], histtype="stepfilled", alpha=0.2, colo

plt.title("Distribution of Topics Across Training Documents")
plt.xlabel("Probability Mass", fontsize="medium")
plt.ylabel("Count", fontsize="medium")
plt.legend()
plt.show()
```



5.0.9 4.9 Histogram of Topics Over Documents Individually

In [172]:
```
plt.figure(figsize=(16,25))
gs = gridspec.GridSpec(5, 2)

for i in range(len(books)):
    ax = plt.subplot(gs[i])
    ax.hist(train_doc_topic_dist[i, :], log=False)
    ax.set_title("Topics by weight for %s" %(books[i]), fontsize="medium")
```

```
    ax.set_xlabel("Probability Mass", fontsize="medium")
    ax.set_ylabel("Count", fontsize="medium")

plt.show()
```


Topics by weight for beowulf.txt
Topics by weight for divine_comedy.txt
Topics by weight for dracula.txt
Topics by weight for frankenstein.txt
Topics by weight for huck_finn.txt
Topics by weight for moby_dick.txt
Topics by weight for sherlock_holmes.txt
Topics by weight for tale_of_two_cities.txt
Topics by weight for the_republic.txt
Topics by weight for ulysses.txt
Count
Probability Mass

6 5. Prediction

- We want to see if we can use a document's topic distribution as a unique signature for classification
- Our theory is that topics across a book will remain consistent
  - So if we take new, unseen data from one of the books, compute its topic distribution, and compare it to the training data's topic distributions, we can know which book the unseen data came from!
- First, we retrain using our topic dict as a starting point. This will let us use our trained model to infer topics for each of the test documents' words better.
- Next, we take the topic that maximizes the coniditional distribution for each word, just as we did before. We observe the topic distribution across the test documents.
- We do this many times so that we can have means and standard deviations for our predictions!

In [289]:
```
test_doc_topic_dists = []
iters = 20
for i in range(iters):
    n_docs, W = test_count_mat.shape
    # number of times document m and topic z co-occur
    ndt_test = np.zeros((n_docs, n_topics))
    # number of times word w and topic z co-occur
    nwt_test = np.zeros((W, n_topics))
    nd_test = np.zeros(n_docs)
    nt_test = np.zeros(n_topics)
    iters = 3
    topics_test = topics
    likelihoods_test = []

    for d in xrange(n_docs):
        for i, w in enumerate(word_indices(test_count_mat[d, :])):
            t = np.random.randint(n_topics)
            ndt_test[d,t] += 1
            nd_test[d] += 1
            nwt_test[w,t] += 1
            nt_test[t] += 1
            topics_test[d][i] = t

    # for each iteration
    for it in xrange(iters):
        for d in xrange(n_docs):
            for i, w in enumerate(word_indices(test_count_mat[d, :])):
                t = topics_test[d][i]
                ndt_test[d,t] -= 1; nd_test[d] -= 1; nwt_test[w,t] -= 1; nt_test[t] -= 1

                p_z = conditional_dist(alpha, beta, nwt_test, nd_test, nt_test, d, w)
                t = np.random.multinomial(1,p_z).argmax()

                ndt_test[d,t] += 1; nd_test[d] += 1; nwt_test[w,t] += 1; nt_test[t] += 1;
                topics_test[d][i] = t

    ##########################################################################################
    #- Now that we have trained our test model, we observe the topic distribution across the
```

```
    #- We take the topic that maximizes the coniditional distribution, just as we did before.
    ######################################################################################

    test_doc_topic_dist = np.zeros((n_docs, n_topics))
    for d in xrange(n_docs):
        # for each word
        for i, w in enumerate(word_indices(test_count_mat[d, :])):
        # get topic of mth document, ith word
            p_z = conditional_dist(alpha, beta, nwt_test, nd_test, nt_test, d, w)
            z = np.random.multinomial(1,p_z).argmax()
            test_doc_topic_dist[d, z] += 1

    # NORMALIZE TOPIC DISTRIBUTION
    row_sums = test_doc_topic_dist.sum(axis=1) + 0.000001
    test_doc_topic_dist = test_doc_topic_dist / row_sums[:, np.newaxis]
    test_doc_topic_dists.append(test_doc_topic_dist)
```

- We already have computed topic distributions over documents in our analysis, so now we can find the most similar topic distribution simply by computing the frobenius norm!
- Since we may have different votes per iteration, we choose the mode of the prediction for each book

In [291]:
```
mins = []
topic_distribution_norms = np.zeros((iters, n_docs, n_docs))

for k in range(iters):
    for i in xrange(n_docs):
        query_dist = test_doc_topic_dists[k][i, :]
        for j in xrange(n_docs):
            topic_distribution_norms[k, i, j] = np.linalg.norm(train_doc_topic_dist[j, :] - q
    mins.append(np.argmin(topic_distribution_norms[k], axis=1))

predictions = map(int, stats.mode(mins, axis=0)[0][0])
print predictions
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Since the test documents are in order, the indices should correspond to the label, which they do!

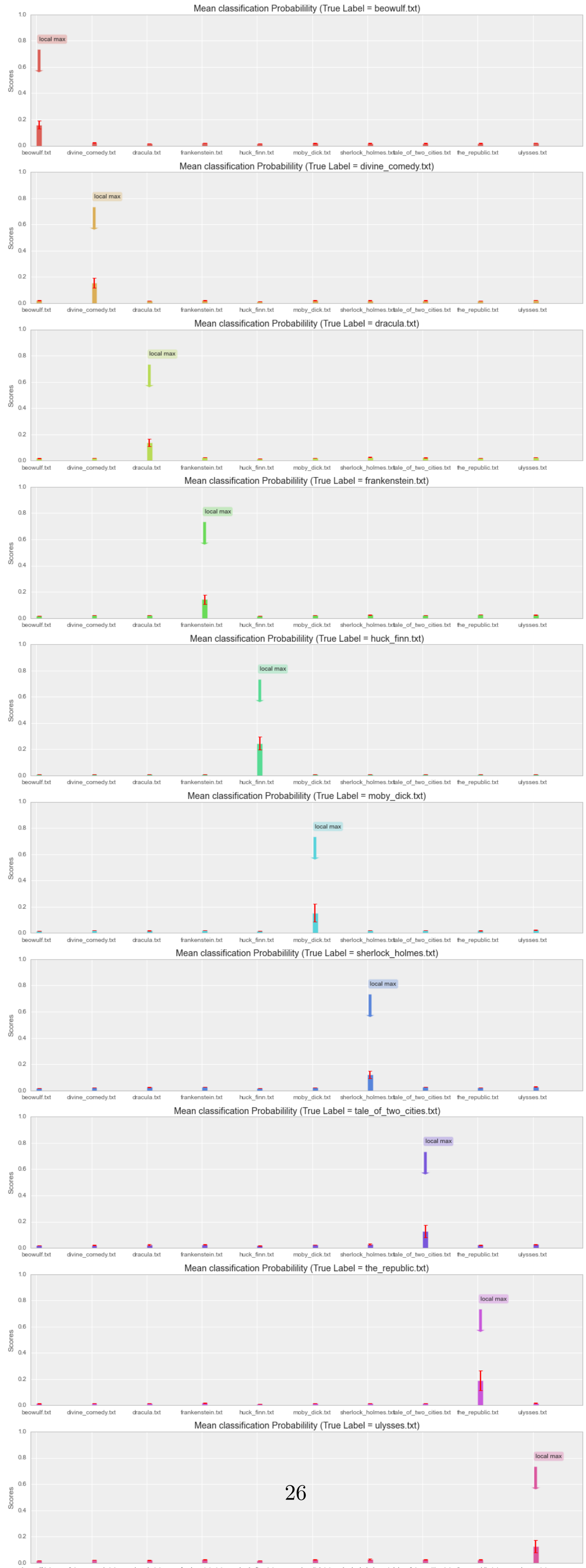
In [292]:
```
"Classification accuracy: %%%0.2f"%( 100*np.mean(np.array(books)[predictions] == np.array(boo
```

Out[292]:
```
'Classification accuracy: %100.00'
```

- We can also get a feel for the posterior by comparing the probabilities for each class prediction across all of the test data

In [294]:
```
#!/usr/bin/env python
# a bar plot with errorbars
import numpy as np
import matplotlib.pyplot as plt

N = len(books)
menMeans = (20, 35, 30, 35, 27)
menStd =   (2, 3, 4, 1, 2)

ind = np.arange(N)  # the x locations for the groups
width = 0.10       # the width of the bars
```

```
c = sns.color_palette("hls", n_docs)
#fig, ax = plt.subplots(figsize=(18,5))
#rects = []

plt.figure(figsize=(16,45))
gs = gridspec.GridSpec(10, 1)

for i in range(len(books)):
    ax = plt.subplot(gs[i])

    # MEAN DISTRIBUTION ACROSS ITERATIONS AND BOOK-TOPIC DICTIONARY FOR A TEST DOCUMENT TO BE
    mean = np.mean((1./topic_distribution_norms[:, i, :])/np.sum((1./topic_distribution_norms
    std = np.std((1./topic_distribution_norms[:, i, :])/np.sum((1./topic_distribution_norms[:

    ax.bar(ind+width, mean, width, color=c[i], label=books[i], yerr=std, error_kw={ 'ecolor':
    ax.set_ylabel('Scores')
    ax.set_ylim([0, 1])
    ax.set_xticks(ind+width)
    ax.set_xticklabels( (books) )

    ax.annotate('local max',
                xy=(i+.15, .5),
                xytext=(i+.15, .8),
                arrowprops=dict(shrink=0.2, headwidth=15, width=5, fc=sns.color_palette('hls'
                #textcoords = 'offset points', ha = 'right', va = 'bottom',
                bbox = dict(boxstyle = 'round,pad=0.3', fc = sns.color_palette('hls', 10)[i],
            )
    ax.set_title('Mean classification Probabilility (True Label = %s)'%(books[i]))


plt.show()
```


Mean classification Probabilility (True Label = beowulf.txt)
local max
Mean classification Probabilility (True Label = divine_comedy.txt)
local max
Mean classification Probabilility (True Label = dracula.txt)
local max
Mean classification Probabilility (True Label = frankenstein.txt)
local max
Mean classification Probabilility (True Label = huck_finn.txt)
local max
Mean classification Probabilility (True Label = moby_dick.txt)
local max
Mean classification Probabilility (True Label = sherlock_holmes.txt)
local max
Mean classification Probabilility (True Label = tale_of_two_cities.txt)
local max
Mean classification Probabilility (True Label = the_republic.txt)
local max
Mean classification Probabilility (True Label = ulysses.txt)
local max
Scores
0.0
0.2
0.4
0.6
0.8
1.0
beowulf.txt
divine_comedy.txt
dracula.txt
frankenstein.txt
huck_finn.txt
moby_dick.txt
sherlock_holmes.txt
tale_of_two_cities.txt
the_republic.txt
ulysses.txt

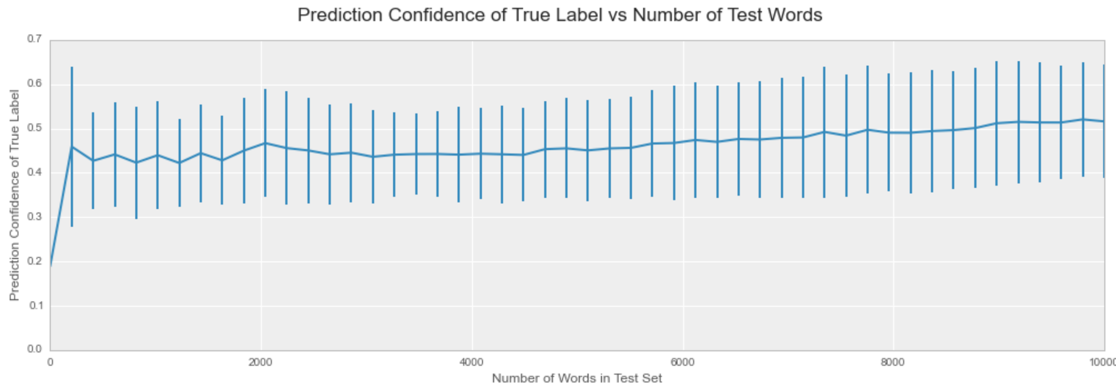
6.1 5.1 "Push it to the Limit"

6.1.1 How much training data do we need to make accurate predictions?

- Currently, we use half of the words in the test document to achieve a perfect classification
- But what is the sensitivity of our model to the number of words in the test data?

In [396]:
```
test_count_mats = []
test_docs = doc[0:len(doc)/2]
accuracy = []; err = []
word_counts = np.linspace(1, 10000, 50)
for num_words in word_counts:
    test_docs = []
    for doc in docs_as_nums:
        test_docs.append(np.array(doc[0:int(num_words)]))
    test_count_mat =np.array(map(freq_map, np.array(test_docs)), dtype=np.int32)

    test_doc_topic_dist = np.zeros((n_docs, n_topics))
    for d in xrange(n_docs):
        # for each word
        for i, w in enumerate(word_indices(test_count_mat[d, :])):
            # get topic of mth document, ith word
            p_z = conditional_dist(alpha, beta, nwt, nd, nt, d, w)
            z = np.random.multinomial(1,p_z).argmax()
            test_doc_topic_dist[d, z] += 1

    # NORMALIZE TOPIC DISTRIBUTION
    row_sums = test_doc_topic_dist.sum(axis=1) + 0.000001
    test_doc_topic_dist = test_doc_topic_dist / row_sums[:, np.newaxis]
    topic_distribution_norms = np.zeros((n_docs, n_docs))

    for i in xrange(n_docs):
        query_dist = test_doc_topic_dist[i, :]
        for j in xrange(n_docs):
            topic_distribution_norms[i, j] = np.linalg.norm(train_doc_topic_dist[j, :] - query

    acc = np.mean( (1./topic_distribution_norms.diagonal()) / (np.sum(1./topic_distribution_n
    std = np.std( (1./topic_distribution_norms.diagonal()) / (np.sum(1./topic_distribution_no
    accuracy.append(acc)
    err.append(std)
```

In [397]:
```
plt.figure(figsize=(17,5))
plt.suptitle("Prediction Confidence of True Label vs Number of Test Words", fontsize="xx-large
plt.errorbar(word_counts, accuracy, yerr=np.array(err))
plt.xlabel("Number of Words in Test Set")
plt.ylabel("Prediction Confidence of True Label")
plt.show()
```



6.2 5.2 Generating Documents

6.2.1 Applying the LDA Generative Model to "create" new pages of The Adventures of Huckleberry Finn!!

According to the generative LDA Model, to generate words from a document:

For s sentences: For n words: 1. Sample a topic index from the topic proportions found in Dracula 2. Sample a word from the Multinomial corresponding to the topic index from 2).

```
In [410]: num_sentences = 20
          num_words = 10
          topic_dist = doc_topic_dist_df.iloc[books.index("huck_finn.txt")].values
          v = np.zeros(train_count_mat.shape[1])

          for s in range(num_sentences):
              for n in range(num_words):
                  z = np.random.multinomial(1,topic_dist).argmax()
                  sorted_topic_words = sorted(topic_words[z].items(), key=operator.itemgetter(1), revers
                  w, p = [w[0] for w in sorted_topic_words], [w[1] for w in sorted_topic_words]
                  idx = np.random.multinomial(1,p).argmax()
                  word = w[idx] if n != 0 else w[idx].capitalize()
                  print word,
              print "."
```

```
Pewter handy funeral im listened kinder warnt average theyve reckon .
Shaming furnaces scared canoe canoe im dont sneak dont aint .
Slip couple tom jane confining spoon mouth stopped blowing flabby .
Erroneously invite maybe theyre endowed disappears desperadoes pretty outwardly tom .
Cant dropped cabin madam susan aint count borrowing hiding picked .
Slick bank id aint lick dug testament charity colonel hit .
Wouldnt fastened doors dont piece forty hide runaway occasionally ive .
Counted tearing objects theyre hed reckon baggage hed minute urge .
Bank m aint counted ive gang crouching tom circus wanted .
Kinder salary breakers youve bridle plate cow wed headway hed .
Kissing bag didnt tom whelps warnt exact dont spoon licking .
Em er whilst laughed yonder congress jumped minute minute hes .
Warnt theres theres learnt breakfast jim stole mary fetch wouldnt .
Fools sale mary theres dat spinning shiver po bag cant .
Raft cool tom dont text dropped didnt nat declaration wouldnt .
```

```
Missus specially toughest ive ive northumberland interruptions pick cavaliers afire .
Hed dozen bidder coffin dont chile preach minute impetus kin .
Duke shes im shows sober troublesome tom dont smiled circumnavigated .
Righti thatand lit dey desperately id couldnt hadnt steamboat hes .
Folks tom ram prettiest ole rope racket orphans runnin honest .
```

6.3 Conclusion

- We train an LDA model on half the pages of ten classic books, the other half is used for testing
- Given the test data and our model, we perform inference on the new text to determine the topic distribution. We compare the queried topic distribution with our training data, and assign it to the closest match.
- Our hypothesis that thematic content would be a good signal for identifying texts was valid
  - We achieved a perfect classification of our query text
- We explored the sensitivity of our model to number of words
- We used the LDA generative model to construct new sentences from a given book
- Future work may use bigrams or n-grams to map to topics, instead of unigrams