Document Classification: Out Of Place vs. Cosine Distance Metrics

Dylan Visher

***Abstract*—In this paper, we compare two different n-gram based methods for supervised content-based document classification. The two methods differ primarily in their distance metric. We compare the "out-of-place" distance metric proposed by Cavnar[?] with the cosine distance metric proposed by Damashek.[?] We will show that the cosine measure is a more accurate supervised, content-based n-gram distance measure for the categorization of English texts into literary certain genres. This paper shows that Damashek's treatment of documents as multidimensional vectors yields greater accuracy and precision than that of Cavnar's more naive, but still effective, distance metric. We also show that Damashek's is much better (relative to Cavnar) when trained on a sparse dataset.**

I. Introduction

Document Classification is a fundamental task in document processing, and appears in many places in our lives such as spam filtering, language identification, genre classification, etc. With the rise of the Internet and the proliferation of unclassified texts in the form of articles, blogs, etc. It is important to be able to classify documents into a genre for search filtering or for finding 'similar' texts. Thus, it is extremely important to be able to determine whether a document is of one category like romance or if it is of another category religion as these two categories could draw very different reader groups. Also, document classification necessitates high accuracy, because misclassifications of genres like romance novels as religious texts would probably lead to that romance novel going undiscovered by its intended audience.

In this paper, we compare two different supervised, content-based document classification schemes that compare n-gram profiles for texts against "average" n-gram profiles for a genre. We look at the "out-of-place" distance metric Cavnar proposes[?] as well as the cosine distance metric Damashek proposes.[?] We will analyze the differences in their methodology, and the differences in their results. We will compare these two methods using the Brown Corpus as defined by the NLTK Python package. Ultimately we show that, though they are comparable in terms of accuracy, Damashek's accuracy is on average higher. In addition, though they are similar when training on a large training set, with a reduced training set, Damashek proves to be a much more accurate method.

II. Research Question

This paper attempts to compare of the pros and cons of an "out-of-place" distance metric as defined by Cavnar in his paper[?] with the pros and cons of a "cosine-distance" metric as defined by Damashek in his paper.[?] Specifically we will see whether the cosine measure or the out-of-place measure provides a more useful character-based n-gram document classification distance metric for categorizing texts belonging to a certain genre. We will also show that Damashek improves in accuracy, relative to Cavnar, when trained on a shorter dataset.

III. Methodology

A. Cavnar

1) Overview: For Cavnar we create an n-gram profile for each genre and for each document we are attempting to classify. These n-gram profiles map n-grams of length 1 to 5 with their associated frequencies generated from the given texts. A text is classified by comparing its n-gram profile with each genre's n-gram profile using the "out-of-place" distance metric. The 'closest' genre to the sample text is chosen to be that text's genre.

2) N-Gram Creation: N-Grams are generated on a word-by-word basis. First the words are tokenized, then punctuation is removed. Then n-grams are generated from each word, and then are added to the global n-gram count. Cavnar in his paper suggests padding the string with a single leading space, and padding the end of the word such that the each n-gram, of any size can contain just the final character followed by n-1 spaces. For this paper, however, we depart from this exact method because it gives an unjustified increased weight to the end of words, resulting in decreased accuracy in text-categorization. We modify his exact method by padding the beginning of words such that each n-gram, or any size can be of just the starting character preceded by spaces. This means that the n-gram profile generated by each word is symmetric, giving an equal weighting to both the start and end of the word. When tested with both options, this change significantly increases the performance of the Cavnar method.

3) Out-Of-Place Distance Metric: The out of place distance metric compares the n-gram profile of the sample text to be categorized and the n-gram profile of the template text. It takes in two n-gram profiles sorted in order of decreasing frequency. Cavnar ignores the first 300 n-grams because they seem to indicate language more than features. In fact, when we tested with and without the first 300 n-grams, we verified his findings but with a nuance, it seems that including the first 300 n-grams just increases the noise of the comparison, thus reducing the comparison effectiveness. After the two texts are aligned based off frequency, and the leading 300 n-grams are discarded, the metric works by summing the absolute value of the differences of the indices for each key in each profile. This is essentially saying, how far out of place from the template profile (what

we know to be the true average for the genre) is each n-gram generated by sample text. Cavnar does not explicitly address the problem of n-grams that are in the sample, but not in the template. To deal with this issue, when n-grams not in template are encountered we give them a frequency of 0, pushing them to the very end of the vector they are not found in. We also found, by changing this cost, Cavnar receives radically different classification. Ultimately since adding them with a frequency of 0 seemed to be the most mathematically sound thing to do, that is what we opted for.

4) Cavnar Pseudocode:

```
function CavnarDistance(template, sample map[string] double) double
    tsorted := Sort(template, by=value, reverse=True)
    ssorted := Sort(sample, by=value, reverse=True)
    for k in sample not in template do
        tsorted.append((k, 0))
    distance := 0
    for i := range ssorted do
        j := find(ssorted[i], tsorted)
        distance += abs(i-j)
    return distance
function GenerateNGrams(category string)
                        (ngrams map[string]double)
    ngrams = nil
    words := tokenize(string)
    for w := range words do
        // Pad the word on either side
        word = " "*len( LongestNgram - 1 ) + word +
               " "*len( LongestNgram - 1)
        stringsoffset := make([]string)
        wordgrams := All Ngrams of length
                     SmallestNgram to LongestNgram
        for g in wordgrams do
            if g is space then
                continue
            else if g in ngrams then
                ngrams[g]++
            else
                ngrams[g] = 0
    return ngrams
function CategorizeText(string smp)
    // Generate Profiles for Each Genre
    Templates := map[string]double
    str := Append(AllTextsIn(c))
    for c := range categories do
        TemplateProfiles[c] := GenerateNGram(str)
    sample := GenerateNGrams[smp]
    minsofar := infinity
    category string := "None"
    for c := range categories do
        d := CavnarDistance(TemplateProfiles[c], sample)
        if d < minsofar then
            minsofar := d
            category := c
    return category
```

B. Damashek

1) Overview: For the purpose of brevity we will be focusing on the differences between Damashek and Cavnar. The main difference is how the distance metric is computed. Damashek employs a cosine distance between the two $absolute$ frequency profiles of the template and the sample. Damashek sees the n-gram profiles as vectors in multidimensional space for which we can compute distances using classical vector distances.

2) N-Gram Creation: Unlike Cavnar, Damashek only looks at n-grams of length 5 (vs 1-5). Damashek also does not pad the word with spaces on either side.

3) Cosine Distance Metric: Damashek views at the template and sample profiles as vectors in n dimensional space. Thus he normalizes the template and sample to have the same n-gram keys, defaulting ones not in there to 0, and sorting them in the same key order. Then he turns the relative frequency vectors into absolute frequency vector thus projecting them onto two different hyperplanes in the same n dimensional space. Then he compares the two by using a normalized cosine distance.

4) Catagorizing a Text:

```
function DamashekDistance(template, sample map[string]double) double
    tsorted := Sort(template, by=key)
    ssorted := Sort(sample, by=key)
    for k in sample not in template do
        tsorted.append((k, 0))
    for k in template not in sample do
        tsorted.append((k, 0))
    tfreqs := [tsorted[1]/sum(tsorted)]
    sfreqs := [ssorted[1]/sum(ssorted)]
    mu1 := len(tfreqs)
    mu2 := len(sfreqs)
    numerator := sum([(i-mu1)*(j - mu2) for i in tfreqs for j in sfreqs])
    xxs := sum([(x-mu1)**2 for x in tfreqs])
    yys := sum([(y-mu2)**2 for y in sfreqs])
    return numerator / (sqrt(xxs * yys)
function GenerateNGrams(category string) (ngrams map[string]double)
    ngrams = nil
    words := tokenize(string)
    for w := range words do
        // Pad the word on either side
        stringsoffset := make([]string) wordgrams := All grams of size 5 in w
        for g in wordgrams do
            if g in ngrams then
                ngrams[g]++
            else
                ngrams[g] = 0
    return ngrams
function CategorizeText(string smp)
    // Generate Profiles for Each Genre
    Templates := map[string]double
```

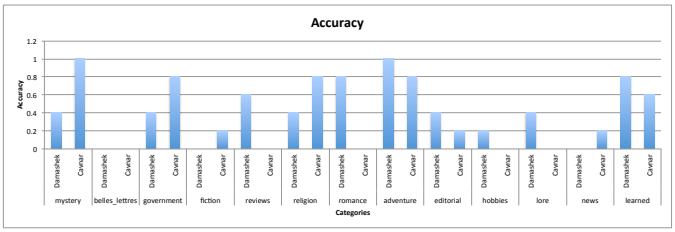

Fig. 1. Accuracy/Recall of Damashek vs. Cavnar By Category

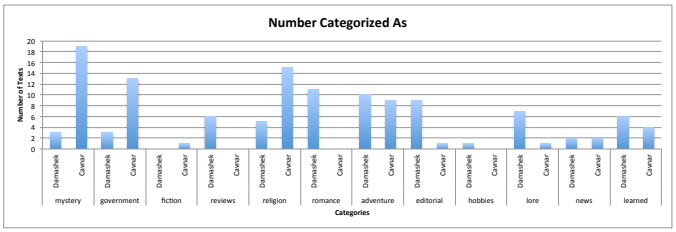

Fig. 2. Number Categorized As Genre Belles-Lettres Removed

```
    str := AppendAllTextsIn(c)
    for c := range categories do
        TemplateProfiles[c] := GenerateNGrams(str)
    sample := GenerateNGrams[smp]
    minsofar := infinity
    category string := "None"
    for c := range categories do
        d := DamashekDistance(TemplateProfiles[c], sample)
        if d < minsofar then
            minsofar := d
            category := c
    return category
```

IV. Results

In this experiment we trained both Cavnar and Damashek's algorithm on the first 10 documents in each category of python's NLTK Brown Corpus[1]. We then tested both Cavnar and Damashek's algorithm on next 5 documents. We then compared the algorithm's outputs against the know category of each text to find the overall Accuracy and Precision, Recall, and F-measure for each category. On the given test set the overall accuracy of Cavnar is .354 and the overall accuracy of Damashek is .415.

In the genre by genre analysis of the accuracies, the recalls, both methods performs similarly. Cavnar performs significantly better classifying mysteries and religion and government. Whereas Damashek performs better categorizing reviews, romance, adventure, lore, and learned. Both techniques have categories where they excel and both have categories where they fail almost entirely. Some categories like belles lettres, fiction, hobbies, and news seem to be pathological cases for both techniques with accuracy rates of 20% or less (essentially random). Both methods perform worse than random on classifying belles lettres. Since no text was classified as belles lettres, the category is removed or the following figures.

Figure 2 shows the number of texts that were categorized as being in a certain category. In many categories there is dramatic over classification (mystery, government, religion and adventure for Cavnar). In the ideal case, only 5 texts would be categorized into each genre, since the test set only contains 5 texts in each genre. Damashek experiences much less over categorization as adventure, romance, and editorial are over

[1]All categories in the brown corpus were used other than science fiction and humor

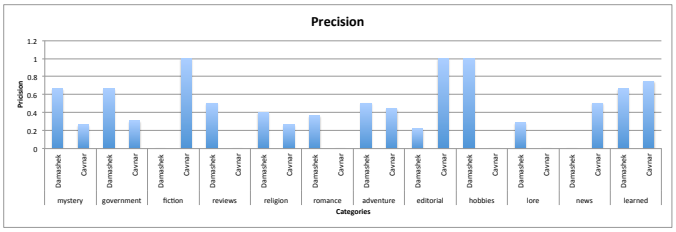

Fig. 3. Precision of Damashek vs. Cavnar Belles-Lettres Removed

classified containing only 2 times the ideal, whereas Cavnar's mystery category is almost 4 times larger than the ideal.

Figure 3 shows the precisions of the two methods. Both methods experience high precision only on the categories that have low accuracy. This indicates that higher accuracy is also accompanied with over classification. The categories with the highest accuracy for Damashek have a precision of about 50%. Whereas Cavnar hovers in the 30% range.

F-Measure is a measure for a more true accuracy which takes into account both recall and precision. Damashek has accuracies ranging mainly from 40% to 100%, if we disclude outliers like news and fiction where he did not classify any. And Cavnar has accuracies generally lower than Damashek, ranging from 20% to 100%. Cavnar only exceeds Damashek's F-Measure in categories that Damashek was unable to properly classify any text, and editorials (which Damshek over-classified). Cavnar has more pathological categories than Damashek (fiction, reviews, romance, hobbies, lore, news vs. fiction, hobbies, news) but even ignoring these categories Damashek does consistently better than Cavnar (with the exception of editorial).

V. Discussion

When looking at the recall for both methods (Figure 1), both methods classify categories with well over 50 percent accuracy. In his paper, Cavnar reports an accuracy of greater

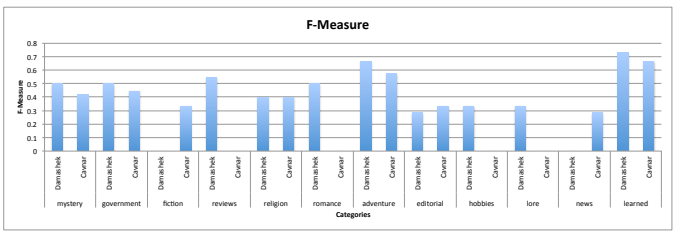

Fig. 4. F-Measure of Damashek vs. Cavnar Belles-Lettres Removed

than 99% in most cases for language detection. However, the task of classifying documents written in the same language into different genres is inherently much more difficult. Genre classification necessitates inner language differentiation, and therefore more fine-grained measures. In addition, we notice that there are a few pathological genres that are responsible for most of the loss of accuracy. Upon analysis of the nature of these categories it becomes apparent why our methods failed on them.

Belles Lettres, the hardest genre for both methods to classify, is defined as fine or beautiful writing and is a genre that encompasses all other genres, poetry, fiction, drama, even essays, the only requisite being that the works are "beautiful". A writing's beauty has nothing to do with the words that an author uses. Humans are able to put a text in multiple categories, whereas our algorithms both seek a single "right" answer. It is unclear to both algorithms whether a "beautiful" fiction piece be classified as belles lettres or fiction. Similarly fiction could be classified more specifically as lore, adventure, some might even say religion should fall under that category. The boundaries between genres is not nearly as clear as classifying a text as french or english. As a text often can have many genres, but most of the time it only has one language.

This reasoning is extended further when viewing other genres such as lore, adventure, romance, and mystery. All of these genres are specifically a subset of fiction. It would be equally valid to classify a romantic fiction document as either romance or fiction. It is evident from the over classification of romance and adventure that Damashek's method sees many texts in the fiction categories as romance or adventure specifically. This is probably because the training vector for fiction determines an "average" fiction work. It is, however, less likely for a work to be the "average" fiction work than to lean in a direction. If writers would write the perfectly "average" fiction novel, there would be nothing novel about it. Works in general categories such as fiction will tend to bias in a certain directions (romance, adventure, lore, ...) rather than being the perfectly average fiction.

Hobbies are often miscategorized as adventure or editorial. This is because many people write about hobbies in editorials and many people have hobbies that involve adventure. People are prone to describing their hobbies with more adventurous vocabulary.

The news genre poses an entirely new problem. Not only can the government, reviews, or editorial genre be classified as news, but news also naturally assumes the language of whatever it is describing. Its vocabulary can change dramatically with depending on subject. If writing about the current crisis in Ukraine words like 'Russia', 'Ukraine', 'Crimea', 'USSR', and many other associated words will occur with a very high frequency, whereas on average these words might fall into noise as celebrity gossip, elections, financial crises, etc. consume most articles. This inconsistency leads to the template vector (the one created during training on the news corpus) to be very sparse and to not contain a large quantity of words that is in not in the sample articles.

Figure 2 and 3, the number categorized in each genre and the precisions, indicate that both these methods higher accuracy is also coupled with higher over classification. But Damashek tends to have much lower over classification coupled with its high accuracy.

VI. Conclusion

Both techniques are comparable with each other achieving similar results on this dataset. Damashek, however, is far less susceptible to pathological categories and over-categorization. Damashek makes much more modest over classifications than Cavnar because the cosine distances of word vectors give a much more specific analysis of categories, that is not as susceptible to words that are not present.

Upon removal of the pathological, umbrella categories (belles lettres, news, fiction, hobbies) the overall accuracy of the methods improve dramatically with Cavnar attaining 49% Accuracy and Damashek 62% accuracy. This improvement indicates that much of the miscategorization of the methods (especially Damashek) has to do with their propensity to favor the more specific. All of these pathological umbrella categories can be categorized as a more specific category as most texts are not average but rather they are nuanced, leaning in a direction.

If we want to improve Damashek further, we can incorporate Cavnar's approach for creating profiles from n-grams of varying length. If Damashek uses n-grams of length 4-5 (keeping the pathological genres in the dataset) it can attains an accuracy of 45.7% on the initial dataset, a significant improvement over the 41.4% accuracy without this modification.

Damashek also continues to perform well on sparse training sets. When Damashek is run on n-grams of length 4 and 5 rather than just 5 and trains on only half the items (but still all the genres), it is still able to achieve the 41.5% accuracy rate of the algorithm in the original experiment. Thus Damashek proves to be superior when using a sparse training set. This probably has to do with the cosine distance metric inherently penalizing less than the out of order distance metric for words that were not in the template profile.

Ultimately Damashek offers a distance metric with lower risk for over-classification and pathological categories, which excels at sparse training sets. Damashek's method, however, always favors the more specific categorization whereas Cavnar's sometimes allows a more general classification.

The code used for this experiment can be found on `https://github.com/dyv/ngram_comparison`. The code was written in python 2.7 and uses the nltk package.

APPENDIX

```
#!/usr/bin/python
from __future__ import division
from bisect import bisect_left
from multiprocessing import Pool
import copy
import pickle
import itertools
import string
import sys
import math
import re
from operator import itemgetter
from nltk.corpus import brown # using the brown corpus
from nltk.corpus import stopwords

BAD_CATS = ["science_fiction", "humor"]
CATAGORIES = [x for x in brown.categories() if x not in BAD_CATS]
STOPWORDS = stopwords.words('english')

# DICT: Dictionary of all ngrams [SMALLEST_NGRAM, LARGEST_NGRAM)
# Used in Cavnar's approach
DDICT = {}
CDICT = {}
# LARGEST_NGRAM: Largest size of ngrams to put in DICT non-inclusive
LARGEST_NGRAM = 6
# SMALLEST_NGRAM: Smallest size of ngrams to put in DICT inclusive
SMALLEST_NGRAM = 1
# TRAINING_SET: Maximum number of items (per category) to train on
TRAINING_SET = 5
# COMPARISON_SET: Number of items to check agains
COMPARISON_SET = 5

def categorize_all():
    cos_correct = 0
    oop_correct = 0
    total = 0
    cwrong = []
    owrong = []
    pool = Pool()
    results =  pool.map(categorize_cat, CATAGORIES)
    cats = { x : {} for x in CATAGORIES}
    for res in results:
        # the category itself
        cat = res[0]
        cats[cat] = {}
        # category number classified
        cats[cat]["total"] = res[3]
        # category number correctly classified
        cats[cat]["dright"] = res[1]
        cats[cat]["cright"] = res[2]
        # category number classified incorrectly
        cats[cat]["dwrong"] = res[4]
        #print "DWRONG: " + str(len(cats[cat]["dwrong"]))
        #print "DRIGHT: " + str(res[1])
        # numbers correct at this point
        cats[cat]["cwrong"] = res[5]
```

```
    # category accuracy measures
    if res[3] == 0:
        cats[cat]["daccuracy"] = 1
        cats[cat]["caccuracy"] = 1
        cats[cat]["drecall"] = 1
        cats[cat]["crecall"] = 1
    else:
        cats[cat]["daccuracy"] = res[1]/res[3]
        cats[cat]["caccuracy"] = res[2]/res[3]
        # category recall measures recall is just category by category
        # accuracy
        cats[cat]["drecall"] = res[1]/res[3]
        cats[cat]["crecall"] = res[2]/res[3]
    # add to global counts
    total += res[3]
    cos_correct += res[1]
    oop_correct += res[2]

# compute precision
# true positives vs categorized as belonging to c
for cat in cats:
    dtotal_classified = cats[cat]["dright"]
    ctotal_classified = cats[cat]["cright"]
    for c in cats:
        for w in cats[c]["dwrong"]:
            if w == "NO CATEGORY":
                print "IMPROPERLY CLASSIFIED DAMASHEK: " + c
            if w == cat:
                dtotal_classified += 1
        for w in cats[c]["cwrong"]:
            if w == "NO CATEGORY":
                print "IMPROPERLY CLASSIFIED CAVNAR: " + c
            if w == cat:
                ctotal_classified += 1
    # indicate that all the texts incorrectly categorized
    # to be in this category are in this category
    # print "TOTAL CLASSIFIED: " + cat + ", " + str(ctotal_classified)
    cats[cat]["ccat"] = ctotal_classified
    cats[cat]["dcat"] = dtotal_classified
    if dtotal_classified == 0:
        cats[cat]["dprecision"] = 0
    else:
        cats[cat]["dprecision"] = cats[cat]["dright"]/dtotal_classified
    if ctotal_classified == 0:
        cats[cat]["cprecision"] = 0
    else:
        cats[cat]["cprecision"] = cats[cat]["cright"]/ctotal_classified

#compute fmeasure
for cat in cats:
    dprec = cats[cat]["dprecision"]
    drec = cats[cat]["drecall"]
    cprec = cats[cat]["cprecision"]
    crec = cats[cat]["crecall"]
    if dprec + drec == 0:
        cats[cat]["dfmeasure"] = 0
    else:
        cats[cat]["dfmeasure"] = 2 * (dprec * drec) / (dprec + drec)
```

```
        if cprec + crec == 0:
            cats[cat]["cfmeasure"] = 0
        else:
            cats[cat]["cfmeasure"] = 2 * (cprec * crec) / (cprec + crec)
    print "DAMASHEK HOBBIES: "
    #print cats["hobbies"]["dwrong"]
    log = open("results.csv", "w")
    print >>log, "category, Method, Accuracy, Precision, Recall, Fmeasure, Categorized"
    for cat in cats:
        print >>log, cat + ", Damashek, " + str(cats[cat]["daccuracy"]) + ", " + \
                  str(cats[cat]["dprecision"]) + ", " + str(cats[cat]["drecall"]) + ", " + \
                  str(cats[cat]["dfmeasure"]) + ", " + str(cats[cat]["dcat"])
        print >>log, ", Cavnar, " + str(cats[cat]["caccuracy"]) + ", " + \
                  str(cats[cat]["cprecision"]) + ", " + str(cats[cat]["crecall"]) + ", " + \
                  str(cats[cat]["cfmeasure"]) + ", " + str(cats[cat]["ccat"])
    log.close()
    print "Damashek Accuracy: " + str(cos_correct/total)
    print "Damashec Wrong: " + str(total - cos_correct)
    print "Cavnar Accuracy: " + str(oop_correct/total)
    print "Cavnar Wrong: " + str(total - oop_correct)
    print "Total Classified: " + str(total)

def categorize_cat(category):
    print "categorizing: " + category
    total = 0
    count = 0
    cos_correct = 0
    oop_correct = 0
    shared = 0
    cos_false_negative = []
    oop_false_negative = []
    for text in brown.fileids(category)[10:]:
        # only compare the first five files after the training set
        if count >= COMPARISON_SET:
            count = 0
            break

        # indicate that we have compared one more file
        count += 1
        total += 1

        #categorize this text
        cos, oop = categorize(text)
        # check to see if the cos distance or out of place distance categorized
        # it correctly
        if cos == category:
            # cos distance correctly classified
            cos_correct += 1
        else:
            # cos distance incorrectly classified
            cos_false_negative.append(cos)
        if oop == category:
            # out of place distance correctly classified
            oop_correct += 1
            if cos == category:
                # both classified correctly
                shared += 1
```

```
        else:
            # out of place distance incorrectly classified
            oop_false_negative.append(oop)
    return (category, cos_correct, oop_correct, total, cos_false_negative, oop_false_negative)

def categorize(text):
    dcounts = {}
    dcounts = generate_text(text, dcounts, "damashek")
    ccounts = {}
    ccounts = generate_text(text, ccounts, "cavnar")
    cosine_estimate = "NO CATEGORY"
    cosine_min = sys.maxint
    out_of_place_estimate = "NO CATEGORY"
    out_of_place_min = sys.maxint
    for category in CATAGORIES:
        # use Damashek Dictionary for Cosine Measure
        tcos = cosine_measure(DDICT[category], dcounts)
        if tcos < cosine_min:
            cosine_estimate = category
            cosine_min = tcos
        # use Cavnar Dictionary for Out of Place Measure
        tout = out_of_place_measure(CDICT[category], ccounts)
        if tout < out_of_place_min:
            out_of_place_estimate = category
            out_of_place_min = tout
    return (cosine_estimate, out_of_place_estimate)

def cosine_measure(template, sample):
    """ cosine_measure: compares the difference between template
        and sample dictionaries by comparing the cosine difference """
    # make template and sample have the same keys
    # Then sort in the same order
    for key in template:
        if key not in sample:
            sample[key] = 0

    for key in sample:
        if key not in template:
            template[key] = 0

    tvector = sorted(template.iteritems(), key=itemgetter(0), reverse=True)
    svector = sorted(sample.iteritems(), key=itemgetter(0), reverse=True)
    # turn relative frequencies into frequencies
    # make unit vectors
    ttotal = sum(tup[1] for tup in tvector)
    stotal = sum(tup[1] for tup in svector)
    tfreqs = [tup[1]/ttotal for tup in tvector]
    sfreqs = [tup[1]/stotal for tup in svector]
    return cosine_dist(tfreqs, sfreqs)

def cosine_dist(vect1, vect2):
    mu1 = 1/len(vect1)
    mu2 = 1/len(vect2)
    numerator = 0
    sumsqr1 = 0
    sumsqr2 = 0
```

```
    for i in xrange(0, len(vect1)):
        x = vect1[i]; y = vect2[i]
        numerator += (x) * (y)
        sumsqr1 += x*x
        sumsqr2 += y*y
    dist =  math.acos(numerator/math.sqrt(sumsqr1*sumsqr2))
    return dist

def out_of_place_measure(template, sample):
    """ out_of_place_measure: compares the difference between template
        and sample by means of couting the number of inversions. """
    tvector = sorted(template.iteritems(), key=itemgetter(1), reverse=True)
    tvector = tvector[300:]
    keyindex = {key[0]: index for index, key in enumerate(tvector)}
    svector = sorted(sample.iteritems(), key=itemgetter(1), reverse=True)
    svector = svector[300:]
    dist = 0
    for i in xrange(0, len(svector)):
        if svector[i][0] in keyindex:
            index = keyindex[svector[i][0]]
            dist += abs(index - i)
        else:
            dist += abs(len(tvector))
    return dist

def opendict():
    global DDICT, CDICT
    DDICT = pickle.load(open('ddict.pk1', 'rb'))
    CDICT = pickle.load(open('cdict.pk1', 'rb'))

def generate():
    global CDICT, DDICT
    for category in CATAGORIES:
        # skip the two categories with too little information
        CDICT[category] = generate_corpus(brown.fileids(categories=category), "cavnar")
        DDICT[category] = generate_corpus(brown.fileids(categories=category), "damashek")
    dict_file = open('cdict.pk1', 'wb')
    pickle.dump(CDICT, dict_file)
    dict_file.close()
    dict_file = open('ddict.pk1', 'wb')
    pickle.dump(DDICT, dict_file)
    dict_file.close()

    # sort each dictionary in DICT
    #vdict = [sorted(DICT[d].iteritems(), key=itemgetter(1), reverse=True) for d in DICT]
    #log_file = open('dict.txt', 'w')
    #print >>log_file, vdict
    #log_file.close()
    #print "vdict"
    #print vdict
```

```
PUNCT_STRING = r"(["+string.punctuation+"])+$"
PUNCT = re.compile(PUNCT_STRING)
# where corpus is the 'news' category or something like that
def generate_corpus(corpus, method):
    counts = {}
    max_training_items = 0
    for text in corpus:
        if max_training_items >= TRAINING_SET:
            break
        max_training_items += 1
        counts = generate_text(text, counts, method)
    return counts

def generate_text(text, counts, method):
    for word in brown.words(text):
        if re.match(PUNCT, word):
            continue
        counts = generate_ngrams(word.lower(), counts, method)
    return counts

def generate_ngrams(string, counts, method):
    global SMALLEST_NGRAM, LARGEST_NGRAM
    lower = SMALLEST_NGRAM
    upper = LARGEST_NGRAM
    if method == "damashek":
        lower = LARGEST_NGRAM - 2

    # pad string with upper-1 spaces on either side
    if method == "cavnar":
        string = " "*(upper-1) + string + " "*(upper-1)
    if method == "damashek":
        string = string
    my_strings = []
    # generate all strings
    for i in range(0, upper):
        my_strings.append(string[i:])
    # generate all N-grams (in myrange) for this string
    for i in range(lower, upper):
        strings = my_strings[0:i]
        for gram in itertools.izip(*strings):
            gram = "".join(gram)
            if gram.isspace():
                continue
            elif gram in STOPWORDS:
                continue
            elif gram in counts:
                counts[gram] += 1
            else:
                counts[gram] = 1
    return counts


if __name__ == "__main__":
    generate()
    #opendict()
    categorize_all()
```