In "Literary Pattern Recognition", Long and So train a classifier to differentiate haiku poems from non-haiku poems, and find that many features help do so. In class, we've discussed the importance of representation--how you *describe* a text computationally influences the kinds of things you are able to do with it. While Long and So explore description in the context of classification, in this homework, you'll see how well you can design features that can differentiate these two classes *without* any supervision. Are you able to featurize a collection of poems such that two clusters (haiku/non-haiku) emerge when using KMeans clustering, with the text representation as your only degree of freedom?

In [1]:
```
import csv, os, re
import nltk
from scipy import sparse
from sklearn.cluster import KMeans
from sklearn import metrics
import math
from collections import Counter
import random
```

In [113…
```
import sys, operator, math, nltk
from collections import Counter
```

In [2]:
```
def read_texts(path, metadata, filepath_col):
    data=[]
    with open(metadata, encoding="utf-8") as file:
        csv_reader = csv.reader(file)
        next(csv_reader)
        for cols in csv_reader:
            poem_path=os.path.join(path, cols[filepath_col])
            if os.path.exists(poem_path):
                with open(poem_path, encoding="utf-8") as poem_file:
                    poem=poem_file.read()
                    data.append(poem)
    return data
```

Here we'll use data originally released on Github to support "Literary Pattern Recognition": https://github.com/hoytlong/PatternRecognition

In [3]:
```
haiku=read_texts("../data/haiku/long_so_haiku", "../data/haiku/Haikus.csv", 4)
```

In [4]:
```
others=read_texts("../data/haiku/long_so_others", "../data/haiku/OthersData.csv", 5)
```

In [9]:
```
# don't change anything within this code block

def run_all(haiku, others, feature_function):

    #use thefunction to get the X(feature set position), y(output truth), (feature voca
    X, Y, featurize_vocab=feature_function(haiku, others)
    #use kmeans to cluster X
    kmeans = KMeans(n_clusters=2, random_state=0).fit(X)
```

```
    #calculate the distance between the labeled k means based on the feature function a
    nmi=metrics.normalized_mutual_info_score(Y, kmeans.labels_)
    print("%.3f NMI" % nmi)
```

As one example, let's take a simple featurization and represent each poem by a binary indicator of the dictionary word types it contains. "To be or not to be", for example, would be represented as {"to": 1, "be": 1, "or": 1, "not": 1}

In [39]:

```
# This function takes in a list of haiku poems and non-haiku poems, and returns:

# X (sparse matrix, with poems as rows and features as columns)
# Y (list of poem labels, with 1=haiku and 0=non-haiku)
# feature_vocab (dict mapping feature name to feature ID)

def unigram_featurize_all(haiku, others):

    def unigram_featurize(poem, feature_vocab):

        # featurize text by just noting the binary presence of words within it

        feats={}

        tokens=nltk.word_tokenize(poem.lower())
        for token in tokens:
            if token not in feature_vocab:
                feature_vocab[token]=len(feature_vocab)
            feats[feature_vocab[token]]=1
        return feats

    feature_vocab={}
    data=[]
    Y=[]

    for poem in haiku:
        feats=unigram_featurize(poem, feature_vocab)
        data.append(feats)
        Y.append(1)
    for poem in others:
        feats=unigram_featurize(poem, feature_vocab)
        data.append(feats)
        Y.append(0)

    # since the data above has all haiku ordered before non-haiku, let's shuffle them
    temp = list(zip(data, Y))
    random.shuffle(temp)
    data, Y = zip(*temp)

    # we'll use a sparse representation since our features are sparse
    X=sparse.lil_matrix((len(data), len(feature_vocab)))

    for idx,feats in enumerate(data):
        for f in feats:
            X[idx,f]=feats[f]

    return X, Y, feature_vocab
```

This method yields an NMI of ~0.07 (with some variability due to the randomness of KMeans)

In [11]:
```
import nltk
nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\cheny\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Out[11]:
```
True
```

In [12]:
```
run_all(haiku, others, unigram_featurize_all)
```

```
0.073 NMI
```

Q1: Copy the `unigram_featurize_all` code above and adapt it to create your own featurization method named `fancy_featurize_all`. You may use whatever information you like to represent these poems for the purposes of clustering them into two categories, but you must use the KMeans clustering (with 2 clusters) as defined in `run_all`. Use your own understanding of haiku, or read the Long and So article above for other ideas. Are you able to improve over an NMI of 0.07?

In [248…
```
def fancy_featurize_log(haiku, others):

    # your code here

    #innitialize token list
    haiku_tokens = []
    other_tokens = []

    #remove punctuaiton from string
    def remove_punc(text):

        # initializing punctuations string
        punc = '''!()-[]{};:'"\,,<>./?@#$%^&*_~'''

        #remove punc if see one
        for i in text:
            if i in punc:
                text = text.replace(i, '')

        return text


    #generate token list from poems
    def token_list(text, remove, token_list):
        for poem in text:
            poem = remove(poem)
            t = nltk.word_tokenize(poem.lower())

            for j in t:
                token_list.append(j)

        return token_list


    token_list(haiku, remove_punc, haiku_tokens)
    token_list(others, remove_punc, other_tokens)


    #log dictionary
```

```
def logodds_with_uninformative_prior(one_tokens, two_tokens):

    def get_counter_from_list(tokens):
        counter=Counter()
        for token in tokens:
            counter[token]+=1
        return counter


    oneCounter=get_counter_from_list(one_tokens)
    twoCounter=get_counter_from_list(two_tokens)

    vocab=dict(oneCounter)
    vocab.update(dict(twoCounter))
    oneSum=sum(oneCounter.values())
    twoSum=sum(twoCounter.values())

    ranks={}
    alpha=0.01
    alphaV=len(vocab)*alpha

    for word in vocab:

        log_odds_ratio=math.log( (oneCounter[word] + alpha) / (oneSum+alphaV-oneCou
        variance=1./(oneCounter[word] + alpha) + 1./(twoCounter[word] + alpha)

        ranks[word]=log_odds_ratio/math.sqrt(variance)

    sorted_x = sorted(ranks.items(), key=operator.itemgetter(1), reverse=True)

    dictionary = {}

    for a, b in sorted_x:
        dictionary.setdefault(a, []).append(b)

    return dictionary



#getting feature based on log dictionary
def log_featurize(poem, feature_vocab, log_dic):

    # featurize text by just noting the binary presence of words within it

    feats={}

    poem = remove_punc(poem)

    tokens=nltk.word_tokenize(poem.lower())


    for token in tokens:

        if token not in feature_vocab:
            feature_vocab[token] = len(token)

        feats[feature_vocab[token]] = log_dic[token][0]

    return feats

feature_vocab={}
```

```
    data=[]
    Y=[]

    log_dic = logodds_with_uninformative_prior(haiku_tokens, other_tokens)

    for poem in haiku:
        #get the features
        feats = log_featurize(poem, feature_vocab, log_dic)

        data.append(feats)
        Y.append(1)

    for poem in others:
        feats = log_featurize(poem, feature_vocab, log_dic)
        data.append(feats)
        Y.append(0)

    # since the data above has all haiku ordered before non-haiku, let's shuffle them
    temp = list(zip(data, Y))
    random.shuffle(temp)
    data, Y = zip(*temp)

    # we'll use a sparse representation since our features are sparse
    X=sparse.lil_matrix((len(data), len(feature_vocab)))

    for idx,feats in enumerate(data):
        for f in feats:
            X[idx,f]=feats[f]

    return X, Y, feature_vocab
```

In [253…

```
def fancy_featurize_all_1(haiku, others):

    # your code here

        #remove punctuaiton from string
    def remove_punc(text):

        # initializing punctuations string
        punc = '''!()-[]{};:'"\,,<>./?@#$%^&*_~'''

        #remove punc if see one
        for i in text:
            if i in punc:
                text = text.replace(i, '')

        return text

    def unigram_featurize(poem, feature_vocab):

        # featurize text by just noting the binary presence of words within it

        feats={}
        tokens=nltk.word_tokenize(poem.lower())


        for token in tokens:

            if token not in feature_vocab:
```

```
                feature_vocab[token] = len(token)

            feats[feature_vocab[token]]=1

        return feats

    feature_vocab={}
    data=[]
    Y=[]

    for poem in haiku:
        #get the features
        poem = remove_punc(poem)
        feats=unigram_featurize(poem, feature_vocab)

        data.append(feats)
        Y.append(1)

    for poem in others:
        poem = remove_punc(poem)
        feats=unigram_featurize(poem, feature_vocab)
        data.append(feats)
        Y.append(0)

    # since the data above has all haiku ordered before non-haiku, let's shuffle them
    temp = list(zip(data, Y))
    random.shuffle(temp)
    data, Y = zip(*temp)

    # we'll use a sparse representation since our features are sparse
    X=sparse.lil_matrix((len(data), len(feature_vocab)))

    for idx,feats in enumerate(data):
        for f in feats:
            X[idx,f]=feats[f]

    return X, Y, feature_vocab
```

In [254...

```
run_all(haiku, others, fancy_featurize_all_1)
```

```
0.106 NMI
```

In [249...

```
run_all(haiku, others, fancy_featurize_log)
```

```
0.014 NMI
```

Q2: Describe your method for featurization in 100 words and why you expect it to be able to separate haiku poems from non-haiku poems in this data.

1. Remove the punctuations in the source file.
2. Using the token size in the featueset dictionary for each token, account for the token size instead of incrimenting length of the dictionary, the score got better around 0.014.
3. Also tried to Use loggodds ratio to pick out the feature set for haiku tokens and the other's token, account for more appropriate sizing, however the score got way worse, around 0.014.

In [ ]: