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 [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"
```

In [4]:
```
others=read_texts("../data/haiku/long_so_others", "../data/haiku/OthersDat
```

In [5]:

```
# don't change anything within this code block

def run_all(haiku, others, feature_function):

    X, Y, featurize_vocab=feature_function(haiku, others)
    kmeans = KMeans(n_clusters=2, random_state=0).fit(X)
    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 [8]:

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

# 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 withi

        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 s
    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]
    # print(X)
    return X, Y, feature_vocab
```

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

In [9]:

```
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 [20]:

```
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()

# approach 1: using TF-IDF word vectors
def fancy_featurize_all_1(haiku, others):

    def tfIdf_featurize(poem, feature_vocab,matrix):

        # featurize text using tf-idf

        feats={}

        tokens=nltk.word_tokenize(poem)

        for token in tokens:
            if token not in feature_vocab:
                feature_vocab[token]=len(feature_vocab)
            if(token in matrix.columns):
                feats[feature_vocab[token]]=matrix[token]
        return feats

    feature_vocab={}
    data=[]
    Y=[]
    vec = TfidfVectorizer(use_idf=False, norm='l1')
    all_data = haiku + others
    all_data = ["".join(x) for x in all_data]
    all_data = [x.replace('\n',' ') for x in all_data]
    all_data = [x.replace('\t',' ') for x in all_data]
    all_data = [x.lower() for x in all_data]
    matrix = vec.fit_transform(all_data)
    matrix = pd.DataFrame(matrix.toarray(), columns=vec.get_feature_names(

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

    # since the data above has all haiku ordered before non-haiku, let's s
    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].mean()
    return X, Y, feature_vocab
```

In [21]:

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

```
0.023 NMI
```

In [22]:

```
from sentence_transformers import SentenceTransformer
import numpy as np

# approach 2: using BERT sentence transformers
def fancy_featurize_all_2(haiku, others):

    def tfIdf_featurize(poem, feature_vocab,matrix):

        # featurize text using BERT Sentence Transformers
        feats={}
        feats = matrix.encode(poem)
        return feats

    feature_vocab={}
    data=[]
    Y=[]
    vec = TfidfVectorizer(use_idf=False, norm='l1')
    all_data = haiku + others
    all_data = ["".join(x) for x in all_data]
    all_data = [x.replace('\n',' ') for x in all_data]
    all_data = [x.replace('\t',' ') for x in all_data]
    all_data = [x.lower() for x in all_data]
    matrix = SentenceTransformer('sentence-transformers/all-distilroberta-v

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

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

    X = np.zeros((len(data), 768), dtype=object)

    for idx,feats in enumerate(data):
        X[idx]=feats
    return X, Y, feature_vocab

run_all(haiku, others, fancy_featurize_all_2)
```

```
0.123 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.

The method I have employed is the BERT Sentence Transformer, which seems to work better between the 2 methods I have implemented (TF-IDF and BERT Sentence Transformers). The BERT Sentence Transformer has a bi-directional architecture, i.e., it learns context from left to right, and from right to left, and therefore tends to learn more context than a naive vectorization method such as TF-IDF. What differentiates a haiku from a regular poem (in most cases) is also the imagery and romantic descriptions of nature. Using Sentence Transformers, we are able to learn the context and style of the poem, and are able to differentiate a haiku from a poem more efficiently.

In [ ]: