**CS5785**
**Applied Machine Learning**
HW3
Due Date: 11/13/2016

**Siyadong Xiong**
sx225@cornell.edu
Teammates: N/A

# 1 Programming Exercises

## 1.1 Sentiment Analysis of online reviews

### 1.1.1 Load and Split Labeled Sentences Data

We loaded and parsed the labeled reviews from yelp, amazon and imdb using the following code. (As usual, we assumed that common Python scientific libraries were already imported). We used the `stratify` parameter in split function so that the labeled review ratio in training and test set will not change.

```
dt = np.dtype([('text','object'), ('label', 'int16')])

amazon = np.loadtxt("data/amazon_cells_labelled.txt",
   dtype=dt, delimiter='\t', comments="THISISCOMMENT")
yelp = np.loadtxt("data/yelp_labelled.txt", dtype=dt,
   delimiter='\t', comments="THISISCOMMENT")
imdb = np.loadtxt("data/imdb_labelled.txt", dtype=dt,
   delimiter='\t', comments="THISISCOMMENT")

train_list = []
test_list = []
test_ratio = 0.2

for d in [amazon, yelp, imdb]:
    tr, te = train_test_split(d, test_size=test_ratio,
       stratify=d['label'])
    train_list.append(tr)
    test_list.append(te)

train = np.row_stack((x.reshape(-1, 1) for x in train_list))
test = np.row_stack((x.reshape(-1, 1) for x in test_list))

```

```
22 pos_ratio = (train[train['label'] == 1].shape[0] +
23             test[test['label'] == 1].shape[0]) /
                  float(train.shape[0] + test.shape[0])
24 print "Positive label ratio is %f" % pos_ratio
```

We split the data set with test set ratio of 0.2 and 2400 reviews for training and 600 reviews for testing. The output of above code indicated that labels are exactly balanced.

```
1 Positive label ratio is 0.500000
```

### 1.1.2 Code Design

Here we designed two classes to extract features from sentences. As the only difference between Bag-of-Words and N-grams is `tokenize()`, we only need to overrride it when implementing N-grams.

| Class | Methods |
|---|---|
| `BagOfWordsVectorizer` | `fit(X):` |
| `(object)` | `fit_transform(X, normalize)` |
| | `transform(X, normalize)` |
| | `tokenize(X)` |
| | `l1_normalize(X)` |
| | `l2_normalize(X)` |
| `NGramVectorizer` | `tokenize(self, X)` |
| `(BagOfWordsVectorizer)` | |

Table 1: Class design.

### 1.1.3 Preprocessing and Tokenize

One important thing in sentiment analysis is sentence preprocessing.

- Firstly, we lowercased the sentence as letter cases rarely affect the language semantics. We split the whole sentences into words with non-alphabetic characters.

- For each remaining word, we use `nltk` to lemmatize all the words such that we reduced the complexity.

- Then we filtered out stop words and stripped punctuation as stop words usually have high frequency in a sentence while contain rather little information.

- Finally, we specifically replaced certain words containing apostrophes with pre-defined words.

Code is as follows:

```
    def __init__(self):
        self.func_lemmatize = WordNetLemmatizer().lemmatize

        self.STRIP_CHARS = string.punctuation + ' '
        self.PUNCTUATIONS = set(string.punctuation)
        self.STOP_WORDS = set(stopwords.words('english'))
        self.APOSTROPHE_DICT = {##..see notebook file..##}
        self._vocabulary = None
        self._lookup_voc = None

    def tokenize(self, X):
        return map(self._tokenize_and_clean, X)

    def _strip_and_lemmatize(self, word, chars):
        word = str(self.func_lemmatize(word))

        if word in self.APOSTROPHE_DICT:
            word = self.APOSTROPHE_DICT[word]

        return word.translate(None, self.STRIP_CHARS)

    def _tokenize_and_clean(self, sentence):
        _ = [self._strip_and_lemmatize(s, self.STRIP_CHARS)
             for s in re.split(r"[^a-z\']", sentence.lower())
             if (s not in self.STOP_WORDS and len(s) > 0)]

        return filter(lambda s: len(s) > 0 and
                              s not in self.STOP_WORDS and
                              s not in self.PUNCTUATIONS, _)
```

### 1.1.4 Bag-of-Words and Bag-of-Ngrams

As mentioned, the only difference between N-grams and Bag-of-Words approaches to extract features is the tokenizing function. For the former one, we need to override the `tokenize` function:

```
    def tokenize(self, X):
```

```
        _ = super(NGramVectorizer, self).tokenize(X)
        return map(self._gen_ngram, _)

    def _gen_ngram(self, tokens):
        r = []
        for i in range(0, len(tokens) - self.N + 1):
            r.append(" ".join(tokens[i:i+self.N]))
        return r
```

After tokenizing sentences, we could build the bag of words. Firstly, we iterate all tokens in the training set to generate vocabulary and word look-up table. We can only see training set at this time or it may include more information from testing set and result in overfit. Once we built the vocabulary, we can count up the occurrence of words in each sentence and get the bag of words feature vectors.

```
    def fit(self, X):
        self.fit_transform(X)
        return self

    def fit_transform(self, X, normalize='l2'):
        # 1. build vocabulary
        tokens = self.tokenize(X)
        _ = set(np.concatenate(tokens))
        self._lookup_voc = {}
        self._vocabulary = {}
        for i, x in enumerate(_):
            self._lookup_voc[i] = x
            self._vocabulary[x] = i
        self._voc_len = len(_)

        # 2. generate X
        _ = np.array(map(self._bag_of_words, tokens))

        if normalize is None:
            return _
        elif normalize == 'l2':
            return l2_normalize(_)
        else:
            return l1_normalize(_)

```

```
    def transform(self, X, normalize='l2'):
        tokens = self.tokenize(X)
        if self._vocabulary is not None:
            _ = np.array(map(self._bag_of_words, tokens))

            if normalize is None:
                return _
            elif normalize == 'l2':
                return l2_normalize(_)
            else:
                return l1_normalize(_)
        else:
            _ = None
            return _
```

We randomly chose two sentences from training set and demonstrated their bag of words feature vectors:

```
[[0 0 0 ..., 0 0 0]
 [0 0 0 ..., 0 0 0]]
bag of words vector dimensions: 4099
nonzero idx: array([1390, 2124, 2329])
nonzero idx: array([2045, 2854, 3481])
```

### 1.1.5 Postprocessing: L1 and L2 normalization

We considered three post processing approaches: 1). no normalization; 2). l1-norm normalization; 3). l2-norm normalization. Geometrically speaking, l2-norm forces all vectors into a sphere and l1-norm forces them into the inner square of the sphere (both in high dimension space). In this way, information of short sentences and long sentences are forces similar or the same. For word count feature, however, it also makes sense to use directly the feature vector as long reviews are supposed to contain more energy than short ones. Hence, no normalization is also an option.

```
    def l1_normalize(X):
        return np.apply_along_axis(lambda x: _l1(x), 1, X)

    def l2_normalize(X):
        return np.apply_along_axis(lambda x: _l2(x), 1, X)

```

```
    def _l1(x):
        _l = len(x.nonzero()[0])
        if _l > 0:
            return x / float(_l)
        else:
            return np.zeros((len(x),))

    def _l2(x):
        _l = len(x.nonzero()[0])
        if _l > 0:
            return x / float(np.sqrt(np.dot(x, x.T)))
        else:
            return np.zeros((len(x),))
```

### 1.1.6 Training Set Clustering

We implemented brute force k-means algorithm from scratch.

```
import random
class KMeans():
    def __init__(self, max_iter=1000):
        self.max_iter = max_iter

    def fit(self, X, k):
        self.X = X
        self.N, self.p = X.shape
        self.k = k

        _ = set([])
        while len(_) < k:
            _.add(random.randint(0, self.N - 1))
        self.center = np.copy(X[tuple(_), :])
        assert self.center.shape == (self.k, self.p)

        self.status = np.zeros((self.N,))

        converge = False
        iter_count = 0
        while not converge and iter_count < self.max_iter:
            converge = True
```

```
            _status = np.apply_along_axis(self._get_center_idx,
                1, X)

            if not np.array_equal(self.status, _status):
                converge = False
                self._update_center(_status)
                self.status = _status

            iter_count += 1
        self.iter_count = iter_count

    def _get_center_idx(self, x):
        return np.argmin(np.sum((self.center - x) ** 2,
            axis=1), axis=0)

    def _update_center(self, status):
        for i in range(self.k):
            _center = np.mean(self.X[status == i], axis=0)
            self.center[i] = _center

    def get_center(self):
        return self.center # shape: k*p

    def get_status(self):
        return self.status

    def get_cluster(self, k):
        return self.X[self.status == k]
```

For bag-of-words model, we compare three normalization methods using the confusion matrix of label v.s. cluster.

```
kmeans = KMeans()
bowv = BagOfWordsVectorizer() # or NGramVectorizer(2)
train_bag = bowv.fit_transform(train['text'][:, 0],
    normalize="l1")
kmeans.fit(train_bag, 2)
train_center = kmeans.get_center()
train_status = kmeans.get_status()
cluster0 = train['label'][train_status == 0, 0]
cluster1 = train['label'][train_status == 1, 0]
```

```
print np.sum(cluster0)
print cluster0.shape[0] - np.sum(cluster0)
print np.sum(cluster1)
print cluster1.shape[0] - np.sum(cluster1)
```

| no | Cluster0 | Cluster1 |
|---|---|---|
| Pos | 542 | 658 |
| Neg | 536 | 664 |

| l1 | Cluster0 | Cluster1 |
|---|---|---|
| Pos | 68 | 1132 |
| Neg | 1 | 1199 |

| l2 | Cluster0 | Cluster1 |
|---|---|---|
| Pos | 229 | 971 |
| Neg | 254 | 946 |

Table 2: Confusion matrix for different normalizations of Bag-of-word model

| no | Cluster0 | Cluster1 |
|---|---|---|
| Pos | 0 | 1200 |
| Neg | 1 | 1199 |

| l1 | Cluster0 | Cluster1 |
|---|---|---|
| Pos | 1 | 1199 |
| Neg | 0 | 1200 |

| l2 | Cluster0 | Cluster1 |
|---|---|---|
| Pos | 235 | 965 |
| Neg | 188 | 1012 |

Table 3: Confusion matrix for different normalizations of 2-grams model

For bag-of-words, as Table.1.1.6 indicated, when there was no normalization, the cluster result is similar to random guess. It makes sense because kmeans algorithm needs to compute distance while we didn't perform normalization; when using l1 normalization, data was likely to be clustered into one; when using l2 normalization, which is smoother than l1 and differentiable, data was relatively not that likely to be clustered into one.

For 2-Grams model, as Table.1.1.6 indicated, it is meaningless to use clustering to extract more information.

For l2 normalization of bag-of-words model, We inspected two cluster centers:

```
cluster0 = [bowv._lookup_voc[_idx] for _idx in
    train_center[0].nonzero()[0]]
cluster1 = [bowv._lookup_voc[_idx] for _idx in
    train_center[1].nonzero()[0]]
```

unfortunately it is hard to tell the difference between the words of these two cluster centers.

### 1.1.7 Sentiment prediction

```
from sklearn.linear_model import LogisticRegression as LR
import sklearn.metrics as me
print '-- bag-of-words --'
for n in [None, 'l1', 'l2']:
    bowv = BagOfWordsVectorizer()
    train_bag = bowv.fit_transform(train['text'][:,0],
        normalize=n)
    test_bag = bowv.transform(test['text'][:,0], normalize=n)

    cls = LR()
    cls.fit(train_bag, train['label'][:,0])
    _pred = cls.predict(test_bag)
    print n, len((_pred == test['label'][:,0]).nonzero()[0]) /
        float(test_bag.shape[0])
    print me.confusion_matrix(test['label'][:,0], _pred)
print '-- N-gram --'

for n in [None, 'l1', 'l2']:
    bowv = NGramVectorizer(2)
    train_bag = bowv.fit_transform(train['text'][:,0],
        normalize=n)
    test_bag = bowv.transform(test['text'][:,0], normalize=n)

    cls = LR()
    cls.fit(train_bag, train['label'][:,0])
    _pred = cls.predict(test_bag)
```

```
24      print n, len((_pred == test['label'][:,0]).nonzero()[0]) /
            float(test_bag.shape[0])
25      print me.confusion_matrix(test['label'][:,0], _pred)
```

Confusion matrice of classification based on different normalization methods are demonstrated as follows. 2-grams method is worse than bag-of-words one because the way 2-grams extracts features will make the feature vector more sparse.

```
1  -- bag-of-words --
2  None 0.796666666667
3  [[238  62]
4   [ 60 240]]
5  l1 0.716666666667
6  [[214  86]
7   [ 84 216]]
8  l2 0.766666666667
9  [[227  73]
10  [ 67 233]]
11 -- N-gram --
12 None 0.731666666667
13 [[221  79]
14  [ 82 218]]
15 l1 0.706666666667
16 [[217  83]
17  [ 93 207]]
18 l2 0.72
19 [[220  80]
20  [ 88 212]]
```

We inspected the none-normalization of bag-of-words method,

```
1  bowv = BagOfWordsVectorizer()
2  train_bag = bowv.fit_transform(train['text'][:,0],
       normalize=None)
3  test_bag = bowv.transform(test['text'][:,0], normalize=None)
4
5  cls = LR()
6  cls.fit(train_bag, train['label'][:,0])
7  coef = cls.coef_[0]
8  idx = np.argsort(coef)[::-1]
9
```

```
print '-good-'
for i in idx[::-1][:20]:
    print '%s; ' % bowv._lookup_voc[i],
print '\n-bad-'
for i in idx[:20]:
    print '%s; ' % bowv._lookup_voc[i],
```

and got the top 20 words that weights most positive and most negative, respectively.

```
-good-
great; love; excellent; nice; best; loved; delicious; good;
    awesome; wonderful; perfect; comfortable; fantastic;
    happier; amazing; easy; friendly; liked; beautiful; happy;

-bad-
bad; poor; not; worst; disappointment; terrible; awful;
    disappointing; stupid; donot; avoid; waste; rude; isnot;
    horrible; plot; dont; piece; bland; wasnt;
```

For 2-grams method, similar words/grams are:

```
-good-
work great; the best; a great; very good; i love; love this;
    great phone; all the; is great; you wont; wa good; i
    liked; would recommend; to go; loved it; so good; the
    price; work fine; loved the; better than;

-bad-
the worst; would not; not good; piece of; waste of; wa
    terrible; very disappointed; doe not; donot work; so bad;
    the battery; at all; not work; i not; very bad; wa not;
    just donot; very disappointing; the script; it just;
```

### 1.1.8 PCA dimension reduction

We implemented PCA algorithm using SVD decomposition.

```
from numpy.linalg import svd
class PCA(object):
    def fit_transform(self, X, q):
```

```
        self.u = np.mean(X, axis=0)
        self._X = X - self.u
        self.V = svd(self._X)[-1]
        return self._X.dot(self.V[:q, :].T)

    def transform(self, X, q):
        return (X - self.u).dot(self.V[:q, :].T)
```

We use PCA to reduce dimensions and then run clustering and classifications.

```
for q in [10, 50, 100]:
    print '-- q = %d -- ' % q
    for n in [None, 'l1', 'l2']:
        bowv = BagOfWordsVectorizer()
        pca = PCA()
        train_bag =
            pca.fit_transform(bowv.fit_transform(train['text'][:,0],
            normalize=n), q)
        test_bag =
            pca.transform(bowv.transform(test['text'][:,0],
            normalize=n), q)

        cls = LR()
        cls.fit(train_bag, train['label'][:,0])
        _pred = cls.predict(test_bag)
        print n, len((_pred ==
            test['label'][:,0]).nonzero()[0]) /
            float(test_bag.shape[0])
        print me.confusion_matrix(test['label'][:,0], _pred)
        kmeans = KMeans()
      kmeans.fit(train_bag, 2)
       train_center = kmeans.get_center()
      train_status = kmeans.get_status()
       cluster0 = train['label'][train_status == 0, 0]
      cluster1 = train['label'][train_status == 1, 0]
       print np.sum(cluster0),
      print cluster0.shape[0] - np.sum(cluster0)
       print np.sum(cluster1),
      print cluster1.shape[0] - np.sum(cluster1)
```

Choosing reduced dimensions at 10, 50, 100, 200, and 500, respectively, we obtained classification results:

```
 -- Bag-of-words --
-- q = 10 --
None 0.595
[[186 114]
 [129 171]]
l1 0.605
[[219  81]
 [156 144]]
l2 0.596666666667
[[188 112]
 [130 170]]
-- q = 50 --
None 0.706666666667
[[216  84]
 [ 92 208]]
l1 0.685
[[211  89]
 [100 200]]
l2 0.701666666667
[[209  91]
 [ 88 212]]
-- q = 100 --
None 0.731666666667
[[224  76]
 [ 85 215]]
l1 0.728333333333
[[221  79]
 [ 84 216]]
l2 0.731666666667
[[218  82]
 [ 79 221]]
 -- q = 200 --
None 0.768333333333
[[232  68]
 [ 71 229]]
l1 0.736666666667
[[223  77]
 [ 81 219]]
```

```
l2 0.755
[[225  75]
 [ 72 228]]
-- q = 500 --
None 0.798333333333
[[241  59]
 [ 62 238]]
l1 0.746666666667
[[224  76]
 [ 76 224]]
l2 0.775
[[231  69]
 [ 66 234]]
```

and cluster results are:

```
-- KMeans --
cluster0 490 465
cluster1 710 735
--
cluster0 915 866
cluster1 285 334
--
cluster0 708 731
cluster1 492 469
--
cluster0 299 285
cluster1 901 915
--
cluster0 177 205
cluster1 1023 995
--
```

### 1.1.9 Analysis

Compared bag-of-words with 2-grams, the former performed better as the latter induced more sparsity. PCA helps to reduce redundant dimensions as we noticed that when set reduced dimensions to 500, the results were almost identical.

From the inspections of coefficients in the LR model, we obtained the most important words that people used to express their feelings. Particular

words were mentioned in last section.

## 1.2 GMM Model and EM Algorithm

### 1.2.1 EM for K-Means

K-means is a special case of EM algorithm setting the probability of hidden data (cluster) given input $y_i$ and model parameters $\theta$, i.e. responsibilities $\gamma_i^z$ to binary values 0 and 1. Hence, we have:

1. E-step: compute responsibility $\gamma_i^z$, which indicates where data $i$ belongs to cluster $z$ or not.

$$\gamma_i^z = \begin{cases} 1, & when\ z = \underset{z}{\operatorname{argmin}} \|y_i - \mu_z\| \\ 0, & other \end{cases} \tag{1}$$

2. M-step: update new cluster centroids with computed responsibilities.

$$\mu^z = \frac{\sum_{i=1}^{N} \gamma_i^z y_i}{\sum_{i=1}^{N} \gamma_i^z} \tag{2}$$

3. Iterate E-step and M-step until $\gamma_i^z$ does not change or reach maximum iteration times.

### 1.2.2 GMM model on Old Faithful Geyser Dataset

We implemented a GMM model class that supported components initialization by random and by passing into parameters. Here we only supported the sphere covariance matrix.

```
1  from scipy.stats import multivariate_normal as mn
2
3  class GMM(object):
4      def __init__(self, n_components, max_iter=500):
5          self.n_components = n_components
6          self.max_iter = 500
7          pass
8
9      def fit(self, X, persist=False, **kwargs):
10         self.N, self.p = X.shape
```

```
11
12          if len(kwargs) > 0:
13              self.means = kwargs['means']
14              self.covs = kwargs['cov']
15              self.prior = kwargs['prior']
16          else:
17              # nc * p
18              _ = np.arange(self.N)
19              np.random.shuffle(_)
20              _idx = _[:self.n_components]
21              self.means = np.copy(X[_idx, :])
22
23              # nc * p * p
24              _X = X - np.mean(X, axis=0)
25              _cov = np.dot(_X.T, _X) * np.identity(self.p) /
                    float(self.N)
26              self.covs = np.repeat(_cov[np.newaxis, :, :],
                    self.n_components, axis=0)
27
28              # nc * 1
29              self.prior = np.full((self.n_components, 1), 1.0 /
                    self.n_components, dtype=float)
30
31
32          if persist:
33              self.all_means = [self.means, ]
34              self.all_covs = [self.means, ]
35              self.all_prior = [self.prior, ]
36
37          self.prob = np.zeros((self.N,))
38          self.resp = np.zeros((self.n_components, self.N))
39          for i in xrange(self.max_iter):
40              resp, self.prob = self._e_step(X)
41
42              if np.allclose(resp, self.resp):
43                  self.iter_times = i
44                  break
45
46              _prior, _means, _covs = self._m_step(X, resp)
47
48              if persist:
```

```
49                 self.all_means.append(_means)
50                 self.all_covs.append(_covs)
51
52             self.resp = resp
53             self.prior = _prior
54             self.means = _means
55             self.covs = _covs
56
57         if not hasattr(self, "iter_times"):
58             self.iter_times = self.max_iter
59
60     def _e_step(self, X):
61         """E-step of EM algorithm
62
63         Return:
64         ------
65         gamma_s : responsibilities for each components and
               data point
66         prob: probability density, shape (N,)
67         """
68
69         # shape: nc * N: phis_ij - phi_j for component i
70         phi_s = np.array([mn.pdf(X, self.means[i],
               self.covs[i])
71                           for i in range(self.n_components)])
72
73         # shape: nc * N / 1 * N = nc * N
74         gamma_s = (self.prior * phi_s) / np.dot(self.prior.T,
               phi_s)
75
76         assert np.allclose(np.sum(gamma_s, axis=0),
               np.ones(self.N))
77
78         # TODO
79         return gamma_s, np.dot(phi_s.T, self.prior)
80
81     def _m_step(self, X, gamma_s):
82         """M-Step of EM algorithm
83
84         Update object's attributes: prior, means, and covs
85         """
```

```
86
87          _prior = (np.sum(gamma_s, axis=1) / self.N)[:,
                np.newaxis]
88
89          _gamma_sum = np.sum(gamma_s, axis=1) # nc*1
90
91          # shape: nc * p
92          _means = np.dot(gamma_s, X) / _gamma_sum[:,
                np.newaxis] # nc * p / nc
93
94          # shape: nc * p * p
95          _covs = np.empty((self.n_components, self.p, self.p))
96          for _i in range(self.n_components):
97              _diff = X - _means[_i]
98              _gamma = gamma_s[_i]
99              assert _gamma.shape == (self.N,)
100             _covs[_i, :, :] = np.dot(_gamma * _diff.T, _diff) *
                    np.identity(self.p) / _gamma_sum[_i]
101
102         return _prior, _means, _covs
103
104     def _predict_prob(self, X):
105         """Predict density given input X
106
107         Paramter:
108         ----
109         X : input with shape N*p
110
111         Returns:
112         ---
113         p : predicted density
114         """
115         if (self.iter_times < self.max_iter):
116             _, prob = self._e_step(X)
117             return prob
118         else:
119             print "Coefficients may not converge!"
120
121     def predict(self, X):
122         if (self.iter_times < self.max_iter):
123             _, prob = self._e_step(X)
```

```
124             return prob
125         else:
126             print "Coefficients may not converge!"
```

Line.42 indicated the termination condition of the EM algorithm: when responsibilities differences are below the threshold, the algorithm converges so we can terminate the iteration. It is noteworthy that the responsibilities mean that the probability that given input belongs to a certain component, and GMM is a generative model that tries to get the information that how the data generates, this termination condition makes sense.

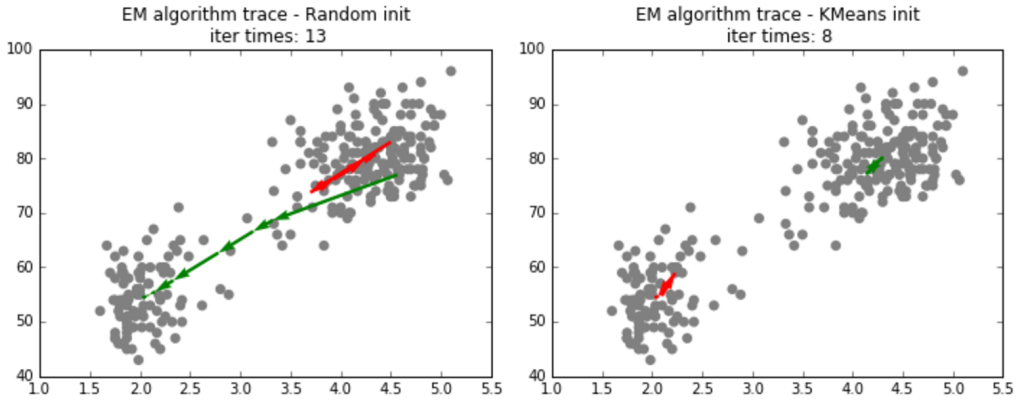


Figure 1: Components comparison between two initialization methods: left).Random; right).K-Means.

The components initialization can be random guess and given by K-Means algorithm. The insight of EM algorithms indicates that the pre-clustering by K-Means would help to accelerate the convergence. As shown in Fig.1 (gray dots indicate data points) , we could see the trace of components centroids, which demonstrated apparent difference between these two methods.

We also statistically analyzed iteration of two different initialization methods by training GMM model 100 times. Fig.2 demonstrates the comparison between iterations of two methods. K-Means initialization reduced the iterations although the improvement is slight since random initialization method neither need too many iterations. The reason should be the data set itself are easy to separate.

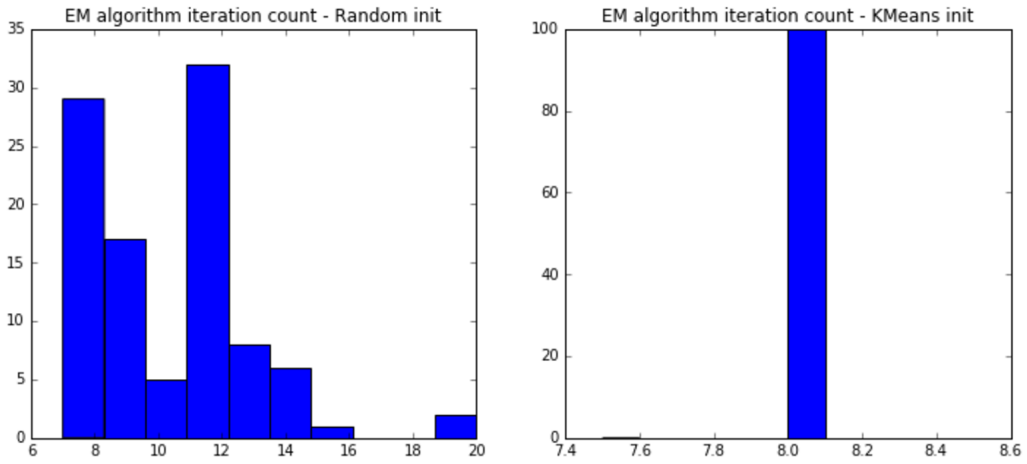


Figure 2: Iteration comparison between two initialization methods: left).Random; right).K-Means.

# 2 Written Exercises

## HTF 14.2

**(1)** We denoted hidden data $\gamma_i^k$ to indicate data point $i$ comes from distribution $g_k$:

$$\gamma_i^k = \begin{cases} 1, & x_i \text{ comes from } g_k \\ 0, & \text{other,} \end{cases} \tag{3}$$

so complete data will be

$$(x_i, \gamma_i^k), i = 1, .., N; j = 1..K.$$

The likelihood of data is

$$\begin{aligned} P(x, \gamma|\theta) &= \prod_{i=1}^{N} P(x_i, \gamma_i^1, .., \gamma_i^K|\theta) \\ &= \prod_{i=1}^{N}\prod_{k=1}^{K} [\pi_k g_k(x_i)]^{\gamma_i^k} \\ &= \prod_{k=1}^{K} \pi_k^{n_k} \prod_{i=1}^{N} (g_k(x_i))^{\gamma_i^k} \\ &= \prod_{k=1}^{K} \pi_k^{n_k} \prod_{i=1}^{N} \left[\frac{1}{\sqrt{2\pi}|\mathbf{L}|^{\frac{1}{2}}\sigma} \exp\left(-\frac{(x_i-\mu_k)^2}{2\mathbf{L}\sigma^2}\right)\right]^{\gamma_i^k}, \end{aligned} \tag{4}$$

where $n_k = \sum_{i=1}^{N} \gamma_i^k$ that satisfies $\sum_{i=1}^{K} n_k = N$, and $\theta$ is distribution parameters $\{\mu_1, \mu_2, .., \mu_K, \sigma\}$.

Therefore, the log-likelihood can be denoted by

$$\begin{aligned} l(\gamma;\theta) &= \log P(x,\gamma|\theta) \\ &= \sum_{k=1}^{K} \left\{ n_k \log \pi_k + \sum_{i=1}^{N} \gamma_i^k \left[ \log \frac{1}{\sqrt{2\pi}|\mathbf{L}|^{\frac{1}{2}}\sigma} - \frac{1}{2\mathbf{L}\sigma^2} (x_i - \mu_k)^2 \right] \right\} \end{aligned} \tag{5}$$

**(2) EM algorithm**

1. E-step: compute the condition expectation of log-likelihood given x and current $\theta^{(i)}$

$$Q(\theta, \theta^{(i)}) = E\left[ l(\gamma;\theta)|x, \theta^{(i)} \right]. \tag{6}$$

We denote the $E(\gamma_i^k|x,\theta)$ as $\hat{\gamma_i^k}$ given according to Bayes rules by:

$$\begin{aligned} \hat{\gamma_i^k} &= P(\gamma_i^k = 1|x,\theta) \\ &= \frac{P(\gamma_i^k = 1, x_i|\theta)}{\sum_{k=1}^{K} P(\gamma_i^k = 1, x_i|\theta)} \\ &= \frac{P(x_i|\gamma_i^k = 1,\theta)P(\gamma_i^k = 1|\theta)}{\sum_{k=1}^{K} P(x_i|\gamma_i^k = 1,\theta)P(\gamma_i^k = 1|\theta)} \\ &= \frac{\pi_k g_k(x_i)}{\sum_{k=1}^{K} \pi_k g_k(x_i)}. \end{aligned} \tag{7}$$

Hence, as $n_k = \sum_{i=1}^{N} \gamma_i^k = \sum_{i=1}^{N} \hat{\gamma_i^k}$, Eq.6 can be further derived as:

$$Q(\theta, \theta^{(i)}) = \sum_{k=1}^{K} \left\{ n_k \log \pi_k + + \sum_{i=1}^{N} \hat{\gamma_i^k} \left[ \log \frac{1}{\sqrt{2\pi}|\mathbf{L}|^{\frac{1}{2}}\sigma} - \frac{1}{2\mathbf{L}\sigma^2} (x_i - \mu_k)^2 \right] \right\}. \tag{8}$$

2. M-step: we need to compute $\theta^{(i+1)}$ for next iteration which is given by

$$\theta^{(i+1)} = \underset{\theta}{\operatorname{argmax}}\, Q(\theta, \theta^{(i)}), \tag{9}$$

which leads to

$$\begin{aligned}\mu_k &= \arg\min_{\mu_k} \frac{\partial Q(\theta, \theta^{(i)})}{\partial \mu_k} = 0 \\ &= \frac{\sum_{i=1}^{N} \hat{\gamma_i^k} x_i}{\sum_{i=1}^{N} \hat{\gamma_i^k}};\end{aligned} \tag{10}$$

$$\begin{aligned}\sigma^2 &= \arg\min_{\sigma^2} \frac{\partial Q(\theta, \theta^{(i)})}{\partial \sigma^2} = 0 \\ &= \frac{\sum_{k=1}^{K}\sum_{i=1}^{N} \hat{\gamma_i^k}(x_i - \mu_k)^2}{2\mathbf{L}^2 \sum_{k=1}^{K}\sum_{i=1}^{N} \hat{\gamma_i^k}};\end{aligned} \tag{11}$$

$$\begin{aligned}\pi_k &= \arg\min_{\pi_k} \frac{\partial Q(\theta, \theta^{(i)})}{\partial \pi_k} = 0 \\ &= \frac{n_k}{N} = \frac{\sum_{i=1}^{N} \hat{\gamma_i^k}}{N}.\end{aligned} \tag{12}$$

**(3)** $\sigma \to 0$ When $\sigma \to 0$, $g_k(x)$ is similar to Dirac function given by $\delta(\mu_k)$. This is similar to those Sec.1.2.1 covered.

## HTF 14.8

The problem is

$$\min_{\mu, \mathbf{R}} \|\mathbf{X}_2 - \mathbf{X}_1\mathbf{R} - \mathbf{1}\mu_T\|_F, \tag{13}$$

the objective of which can be derived as

$$\begin{aligned}&tr\left\{\left[(\mathbf{X}_2 - \mathbf{X}_1\mathbf{R}) - \mathbf{1}\mu^T\right]^T \left[(\mathbf{X}_2 - \mathbf{X}_1\mathbf{R}) - \mathbf{1}\mu^T\right]\right\} \\ = tr&\left\{(\mathbf{X}_2 - \mathbf{X}_1\mathbf{R})^2 - N(x_2\mu^T + \mu x_2^T) + N(\mathbf{R}^T x_1 \mu^T + \mu x_1^T \mathbf{R}) + N\mu\mu^T\right\}\end{aligned} \tag{14}$$

given that $Tr$ is a linear operator and that $x_2^T\mathbf{1} = Nx_2$ and its similar equations. Setting the differentiate on $\mu$ to 0, we get

$$\begin{aligned}\mu &= \frac{1}{N}\left(\mathbf{X}_2^T - \mathbf{R}\mathbf{X}_1^T\right)\mathbf{1} \\ &= \bar{x}_2 - \mathbf{R}\bar{x}_1.\end{aligned} \tag{15}$$

Substituting $\mu$ in Eq.13 with Eq.15, we obtain new objective function as

$$\begin{aligned}
\hat{\mathbf{R}} &= \underset{\mathbf{R}}{\operatorname{argmin}} \|\widetilde{\mathbf{X}}_2 - \widetilde{\mathbf{X}}_1\mathbf{R}\|_F \\
&= \underset{\mathbf{R}}{\operatorname{argmin}} \left\langle \mathbf{R}^T\widetilde{\mathbf{X}}_1^T - \widetilde{\mathbf{X}}_2^T, \mathbf{R}^T\widetilde{\mathbf{X}}_1^T - \widetilde{\mathbf{X}}_2^T \right\rangle \\
&= \underset{\mathbf{R}}{\operatorname{argmin}} \|\widetilde{\mathbf{X}}_1\|_F + \|\widetilde{\mathbf{X}}_2\|_F - 2\left\langle \mathbf{R}^T\widetilde{\mathbf{X}}_1^T, \widetilde{\mathbf{X}}_2^T \right\rangle \\
&= \underset{\mathbf{R}}{\operatorname{argmax}} \left\langle \mathbf{R}^T, \widetilde{\mathbf{X}}_2^T\widetilde{\mathbf{X}}_1 \right\rangle = \underset{\mathbf{R}}{\operatorname{argmax}} \left\langle \mathbf{R}, \mathbf{U}\mathbf{D}\mathbf{V}^T \right\rangle \\
&= \underset{\mathbf{R}}{\operatorname{argmax}} \left\langle \mathbf{V}^T\mathbf{R}^T\mathbf{U}, \mathbf{D} \right\rangle \\
&= \mathbf{U}\left( \underset{\mathbf{R}}{\operatorname{argmax}} \langle \mathbf{R}, \mathbf{D} \rangle \right)\mathbf{V}^T \\
&= \mathbf{U}\mathbf{V}^T,
\end{aligned} \tag{16}$$

where inner product $\langle A, B\rangle = AB^T$.

Given scaling factor $\beta$, the problem changes toward

$$\min_{\beta,\mathbf{R}} \|\mathbf{X}_2 - \beta\mathbf{X}_1\mathbf{R}\|_F, \tag{17}$$

We derived the objective function as

$$\|\mathbf{X}_2 - \beta\mathbf{X}_1\mathbf{R}\|_F = \|\mathbf{X}_2\|_F + \beta^2\|\mathbf{X}_1\|_F - 2\beta \left\langle \mathbf{X}_2, \mathbf{X}_1\mathbf{R} \right\rangle, \tag{18}$$

and set the differentiate to $\beta$ to 0, and then can get the expression as

$$\beta\|\mathbf{X}_1\|_F = \left\langle \mathbf{X}_2, \mathbf{X}_1\mathbf{R} \right\rangle. \tag{19}$$

Since $\mathbf{R}$ is linear transform on the inner product space $\langle \mathbf{X}_2, \mathbf{X}_1 \rangle$, the sum of eigenvalues do not change, which can be written as

$$tr\left(\langle \mathbf{X}_2, \mathbf{X}_1\mathbf{R} \rangle\right) = tr(\mathbf{D}), \tag{20}$$

and leads to $\beta = \frac{tr(\mathbf{D})}{\|\mathbf{X}_1\|_F}$.

## HTF 14.11

The problem is to minimize

$$S_C(z_1, z_2, ..., z_N) = \sum_{i,j} (s_{ij} - \langle z_i, z_j \rangle)^2, \tag{21}$$

where $s_{ij} = \langle x_i, x_j \rangle$. Here, to simplify, we already subtracted the means for $x$ and $z$. Recall that for 0 mean situation, we have

$$\|x_i - x_j\|^2 = -2x_i^T x_j, \tag{22}$$

so we could use $\mathbf{S}$ to estimate $x$. Since $\mathbf{S}$ is symmetric, which indicates that its eigenvectors are orthogonal, we eigendecompose $\mathbf{S}$ as

$$\mathbf{S} = \mathbf{E}_k \mathbf{D}_k \mathbf{D}_k^T \mathbf{E}_k^T, \tag{23}$$

which indicates that the estimations of $x$, i.e. $z$, are rows of $\mathbf{E}_k \mathbf{D}_k$.

# 3 Attachments Details

`hw3-setiment_analysis.ipynb` and `hw3-em.ipynb` in `hw3_sidxiong.zip`

# References

[1] *http://nlp.stanford.edu/IR-book/html/htmledition/determining-the-vocabulary-of-terms-1.html*