Homework 3

By: Daniel Speiser and Gideon Glass

November 10^{th}, 2015

Contents

1 Part 1: Sentiment Analysis of Online Reviews **3**
1.1 Summary 3
1.2 Data Set 3
1.3 Solution Steps 3

2 Part 2: EM Algorithm and Implementation **20**
2.1 Summary 20
2.2 Solution Steps 20

3 Part 3: Hand Written Problems **28**
3.1 EM Algorithm Derivation 28
3.2 Procrustes Problem 30
3.3 Classical Scaling Problem 32

1 Part 1: Sentiment Analysis of Online Reviews

1.1 Summary

The goal was to perform sentiment analysis of online Amazon, Yelp, and IMDB reviews. We did this using several methods, including Logistic Regression, K-Means, and GMM/EM clustering of features extracted by bags of words and n-grams models, as well as with PCA dimensionally reduced features.

1.2 Data Set

The sentiment analysis data set contained 3 text files:

- Yelp reviews
- Amazon reviews
- IMDB reviews

1.3 Solution Steps

A. The data was downloaded as 3 text files, and was parsed/processed according to the readme.txt file using the Pandas library. The number of positive and negative (ignoring non-labeled) reviews is balanced, with each file containing 500 positive and negative reviews respectively.

```
def get_label_ratio (data_set):
  label_0 = sum(data_set['Label'] == 0)
  label_1 = sum(data_set['Label'] == 1)
  return (label_0, label_1)

# Print ratio of different labels per data set
for k, data in parsed_data.items():
  label_0, label_1 = get_label_ratio(data)
  print k.upper()
  print "Label 0: ", label_0
  print "Label 1: ", label_1, "\n\n"

AMAZON
Label 0: 500
Label 1: 500


IMDB
Label 0: 500
Label 1: 500


YELP
Label 0: 500
Label 1: 500
```

Figure 1: Label Ratios

B. We used the following strategies for pre-processing:

- Lowercase all words
- Strip punctuation
- Strip stop words

Ideally, we would have also implemented a spell check since it greatly reduces noise (human error), and would allow us to attain a more accurate analysis.

```
# B. Preprocessing
def preprocess (data):
  stopwords = set(["the", "and", "or", "a"]) # add some more later!
  for i in range(len(data)):
    data[i] = re.sub("[^a-zA-Z]", " ", data[i])
    data[i] = data[i].lower().strip()
    temp = [word for word in data[i].split() if word not in stopwords]
    data[i] = " ".join(temp)
  return data

for k, data in parsed_data.items():
  parsed_data[k]['Sentence'] = preprocess(data['Sentence'].values)
```

Figure 2: Pre-Processing Function

C. Split data into training and testing sets: use the first 400 rows from each data set as training, and the last 100 as testing. This sums up to 2400 rows for training, and 600 for testing.

```
# C. Split training and testing data
def split_data (data):
  zeros = data.loc[data['Label'] == 0]
  ones = data.loc[data['Label'] == 1]
  train, test = pd.concat([zeros[:400], ones[:400]]), pd.concat([zeros[400:], ones[400:]])
  return (train, test)

train_dict, test_dict = {}, {}
for k, data in parsed_data.items():
  train_dict[k], test_dict[k] = split_data(data)

train = pd.concat(train_dict, ignore_index=True)
test = pd.concat(test_dict, ignore_index=True)
```

Figure 3: Split Training and Testing Data

D. For the bag of words model, we extracted every unique word found within the training set, and created a list containing counts corresponding to each word. The reason the bag of words model is created only from the words

within the training set (not including the words in the test set), is because we want to avoid overfitting.

We then did a second pass over the entire data set, and created feature vectors based on our bag of words model. Two random feature vectors can be found in figure 5 below.

```
# D. Bag of words
def get_bag (data):
  bag = []
  for sentence in data:
    for word in sentence.split():
      bag.append(word)
  return np.unique(bag)

bag = get_bag(train['Sentence'])

def get_feature_vectors (bag, data, n=1):
    features, labels = [], []
    for i in range(len(data)):
        f = []
        s = n_grams(data['Sentence'][i], n)
        l = data['Label'][i]
        for word in bag:
            f.append(s.count(word))
        features.append(f)
        labels.append(l)
    return pd.DataFrame({"feature": features, "label": labels}, dtype="float64")

train_features = get_feature_vectors(bag, train)
test_features = get_feature_vectors(bag, test)
```

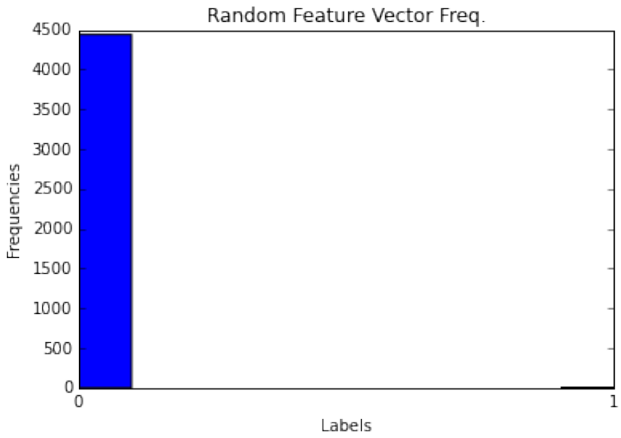
Figure 4: Bag of Words and Feature Extraction

```
# print 2 random feature vectors from training set
two_rand = random.sample(train_features['feature'].values, 2)
print "Random Training Sample Vector 1: \n", two_rand[0]
print "\nRandom Training Sample Vector 2: \n", two_rand[1]

def print_rand_features (data):
  plt.title("Random Feature Vector Freq.")
  plt.xlabel("Labels")
  plt.ylabel("Frequencies")
  plt.hist(data, range=[0,np.max(data)])
  plt.xticks(range(0,np.max(data)+1))
  plt.show()
  return

for fv in two_rand:
  print_rand_features(fv)
```

Figure 5: Get Two Randomly Sampled Feature Vectors



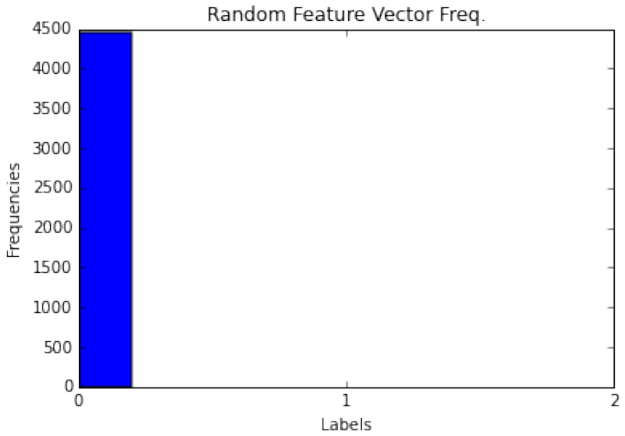


Figure 6: Distribution from two randomly sampled feature vectors. The data is sparse containing mostly zeros.

E. Since the vast majority of the words found in our bag of words model do not appear in each of the reviews, we need to perform some normalization and post-processing on the resulting feature vectors before classifying them. Our strategy was to use L2 normalization. We used L2 normalization because it is the best suited and the most accurate approach for sparse data sets like this one.

```
# E. Postprocessing
def post_process (data):
  norm = []
  for features in data:
    mapped = map(float, features)
    norm.append(pre.normalize(mapped, norm='l2').flatten())
  return norm

train_features_post = zip(post_process(train_features['feature'].values), train_features['label'])
test_features_post = zip(post_process(test_features['feature'].values), test_features['label'])

train_df = pd.DataFrame(train_features_post, columns=["feature", "label"])
test_df = pd.DataFrame(test_features_post, columns=["feature", "label"])
```

Figure 7: Post-Processing Normalization

F. After post-processing the feature vectors, we performed K-means clustering. The assumption was that the positive and negative reviews would create two somewhat distinct groups, and by performing k=2 clustering, we would be able to separate these clusters and return the 2 resulting positive and negative centroids. This allows for us to predict other data sets, by approximating its centroid's similarity or dissimilarity to the training set's centroids.

Our k-means algorithm converged most frequently to clusters of approx-

imatelu 665 and 1735 feature vectors in size respectively. This averaged out at about a 51% and 53% 'purity' accuracy rating. With different initializations of the centroids, we were able to attain upwards of 75% accuracy for the smaller cluster, of about 180 feature vectors in size (the larger cluster still remained at approximately 52%.

```
class KMeans:

  def __init__(self, k_clusters = 2, max_iterations = 1000):
    self.k_clusters = k_clusters
    self.max_iterations = max_iterations
    return

  def fit(self, X):
    self.X = X
    centroids = self.get_initial_centroids(X)
   old_centroids = []

   iteration = 0
   while not np.array_equal(centroids, old_centroids) and iteration < self.max_iterations:
     old_centroids = centroids.copy() # must copy list, not assign
     # cluster points to nearest centroid
     clusters, labels = self.clusters(centroids)
     # update centroid
     centroids = self.update_centroids(centroids, clusters)
      iteration += 1
   return centroids, clusters, labels

  def get_initial_centroids(self, X):
    centroids = []
    for i in range(0, self.k_clusters):
      centroids.append(X['feature'][np.random.randint(0, len(X))])
    return np.array(centroids)

  # creates clusters of points nearest to centroids
  def clusters (self, centroids):
   clusters = [[] for i in range(self.k_clusters)]
   labels = [[] for i in range(self.k_clusters)]
   for idx, x in self.X.iterrows():
     min_dist = sys.maxint
     kth_idx = -1
      for idx2, centroid in enumerate(centroids):
         dist = spatial.distance.euclidean(x['feature'], centroid)
        if dist < min_dist:
          min_dist = dist
          kth_idx = idx2
     clusters[kth_idx].append(x['feature'])
     labels[kth_idx].append(x['label'])
    return clusters, labels

  def update_centroids(self, centroids, clusters):
    for idx, cluster in enumerate(clusters):
        if cluster == []:
            raise Exception('Empty cluster, try different centroid initialization')
        centroids[idx] = np.array(cluster).sum(axis=0) / float(len(cluster))
    return centroids

# using k-means on training set
cents, clusts, labs = KMeans(k_clusters = 2, max_iterations = 300).fit(train_df)
```

Figure 8: K-Means Class

```
for idx, cent in enumerate(cents):
  print "Centroid {0}:\n{1}".format(idx, cent)

Centroid 0:
[  1.48903627e-04  9.11844769e-05  0.00000000e+00 ...,  2.17972591e-04
   1.07090737e-04  1.22953124e-04]
Centroid 1:
[ 0.          0.          0.00035391 ...,  0.          0.          0.        ]
```

Figure 9: K-Means Centroids of k = 2

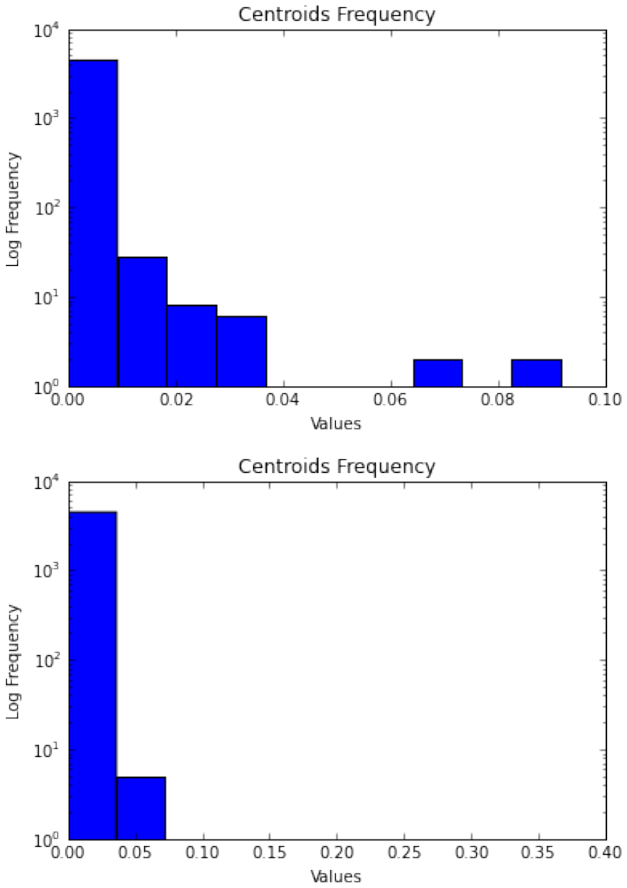


Figure 10: Log Frequencies of the centroids.

```
# accuracy was calculated using the 'purity' method, not NMI or RI
def get_km_accuracy(label_set):
    for idx, labels in enumerate(label_set):
        counts = np.bincount(labels)
        cluster_label = np.argmax(counts)
        print "Number of elements in cluster {0}: {1}".format(idx, len(labels))
        print "Accuracy (Purity) of cluster {0} with label {1}: {2}".format(
         idx, cluster_label, (labels == cluster_label).sum() / float(len(labels)))
    return

get_km_accuracy(labs)

Number of elements in cluster 0: 1734
Accuracy (Purity) of cluster 0 with label 1: 0.512687427912
Number of elements in cluster 1: 666
Accuracy (Purity) of cluster 1 with label 0: 0.533033033033
```

Figure 11: K-Means Accuracy

G. After using K-means, we trained a logistic regression model. Using cross-validation, we achieved an accuracy rating of approximately 82%. The most important words (both positive and negative) in deciding the sentiment of the reviews and for attaining this accuracy can be seen below.

```
# G. Logistic Regression
def logistic_regression (x_train, y_train, x_test, y_test, bag):
    # Fit model
    lr = lm.LogisticRegression(random_state=1).fit(x_train, y_train)

    # Cross validation
    scores = cv.cross_val_score(lr, x_train, y_train, cv=15)
    print "Cross validation: ", scores.mean()

    # Prediction and scoring
    predictions = lr.predict(x_test)
    accuracy = (predictions == y_test).sum() / float(len(y_test))
    print "Prediction accuracy: ", accuracy

    # Confusion matrix
    print "CONFUSION MATRIX:"
    print metrics.confusion_matrix(y_test, predictions)

    # most important weights/words
    weights = [(idx, val) for idx, val in enumerate(lr.coef_[0])]
    weights.sort(key=lambda tup: tup[1], reverse=True)
    num_important_words = 10
    important_positive, important_negative = [], []
    for i in range(num_important_words):
        important_positive.append(bag[weights[i][0]])
        important_negative.append(bag[weights[len(weights) - i - 1][0]])

    print "Top 10 Positive Important Words:\n", important_positive
    print "Top 10 Negative Important Words:\n", important_negative
    return

# G. Using Logistic Regression
x_train = train_df['feature'].values.tolist()
x_test = test_df['feature'].values.tolist()
y_train = train_df['label']
y_test = test_df['label']
logistic_regression(x_train, y_train, x_test, y_test, bag)

Cross validation: 0.796666666667
Prediction accuracy: 0.821666666667
Top 10 Positive Important Words:
['great', 'good', 'love', 'excellent', 'nice', 'best', 'delicious', 'works', 'amazing', 'loved']
Top 10 Negative Important Words:
['not', 'bad', 'poor', 't', 'worst', 'terrible', 'awful', 'no', 'disappointing', 'horrible']
```

Figure 12: Logistic Regression Accuracy and Most Significant Words

H. Instead of the bag of words model, here we used a similar model called N-grams. The idea of n-grams is akin in the sense that we create a dictionary of unique 'words', however in this case the 'words' are actually groups of unique n consecutive words. For example, the sentence "Alice fell down the rabbit hole" with a simple bag of words would give a bag of ['Alice', 'fell',

'down', 'the', 'rabbit', 'hole'], but in an n = 2 n-grams model, it would be ['Alice fell', 'fell down', 'down the', 'the rabbit', 'rabbit hole']. The results (as expected since the sentiment sentences are so short) were slightly worse than using the bag of words alone. The accuracy for Logistic Regression is 69.83% and the 'purity' of the K-means for cluster 1 is 50.73% and for cluster 2 is 55.72%

```
def n_grams (data, n):
  bag = []
  for sentence in data:
    s = sentence.split()
    for i in range(len(s)-n+1):
       bag.append(tuple(s[i:i+n]))
  return set(bag)

# Create bag of bi-grams (n=2)
bi_grams = np.unique(np.concatenate([n_grams(sentence, 2) for sentence in train['Sentence']]))

# Collect feature vectors for each sentence in data
bg_train_features = get_feature_vectors(bi_grams, train, 2)
bg_test_features = get_feature_vectors(bi_grams, test, 2)

# Print 2 random feature vectors from training set
bg_two_rand = random.sample(bg_train_features['feature'].values, 2)
print "Random Training Sample Vector 1: \n", bg_two_rand[0]
print "\nRandom Training Sample Vector 2: \n", bg_two_rand[1]

# Show distribution frequencies (log scale)
for fv in bg_two_rand:
    print_rand_features(fv)
```

Figure 13: N-Grams Model

```
# Accuracy
get_km_accuracy(bg_labs)
Number of elements in cluster 0: 2129
Accuracy (Purity) of cluster 0 with label 1: 0.50728041334
Number of elements in cluster 1: 271
Accuracy (Purity) of cluster 1 with label 0: 0.557195571956
```

Figure 14: K-Means Accuracy and Most Significant Words on Bigram

```
# Logistic regression on bigram data
bg_x_train = bg_train_df['feature'].values.tolist()
bg_x_test = bg_test_df['feature'].values.tolist()
bg_y_train = bg_train_df['label']
bg_y_test = bg_test_df['label']
logistic_regression(bg_x_train, bg_y_train, bg_x_test, bg_y_test, bi_grams)

Cross validation: 0.702083333333
Prediction accuracy: 0.698333333333
CONFUSION MATRIX:
[[194 106]
 [ 75 225]]
Top 10 Positive Important Words:
['very good', 'works great', 'is great', 'i love', 'is good', 'love this',
'great phone', 'is very', 'an excellent', 'highly recommend']
Top 10 Negative Important Words:
['didn t', 'don t', 'would not', 'do not', 'not good', 'piece of', 'very disappointed',
'doesn t', 'was terrible', 'waste of']
```

Figure 15: Logistic Regression Accuracy and Most Significant Words on Bigram

I. Similar to part (E): Since the vast majority of the words found in our bag of words model do not appear in each of the reviews, there is a large amount of redundancy. In order to remove some of this redundancy, we implemented PCA. PCA returns the dimensionally reduced feature vectors, 'truncating' in a sense from 4000+ in cardinality to 10, 50, and 100. After attaining the transformed (dimensionally reduced) vectors, we once again ran them through K-Means and Logistic Regression. You can find the centroids and accuracy and weights KM/LR below.

```
# I. PCA
def PCA(features, dimension):
  features -= np.array(features).mean(axis=0)
  u, s, v = np.linalg.svd(np.asmatrix(features))
  return features.dot(np.array(v[:dimension]).T), np.array(v)
```

Figure 17: PCA Implementation

```
dimensions = [10, 50, 100]
train_dfs_truncated = {}
test_dfs_truncated = {}
for dim in dimensions:
  train_dfs_truncated[dim] =
        pd.DataFrame(zip(PCA(x_train, dim), train_features['label']), columns=["feature", "label"])
  test_dfs_truncated[dim] =
        pd.DataFrame(zip(PCA(x_test, dim), test_features['label']), columns=["feature", "label"])
```

Figure 18: Attaining Dimensionally Reduced Features

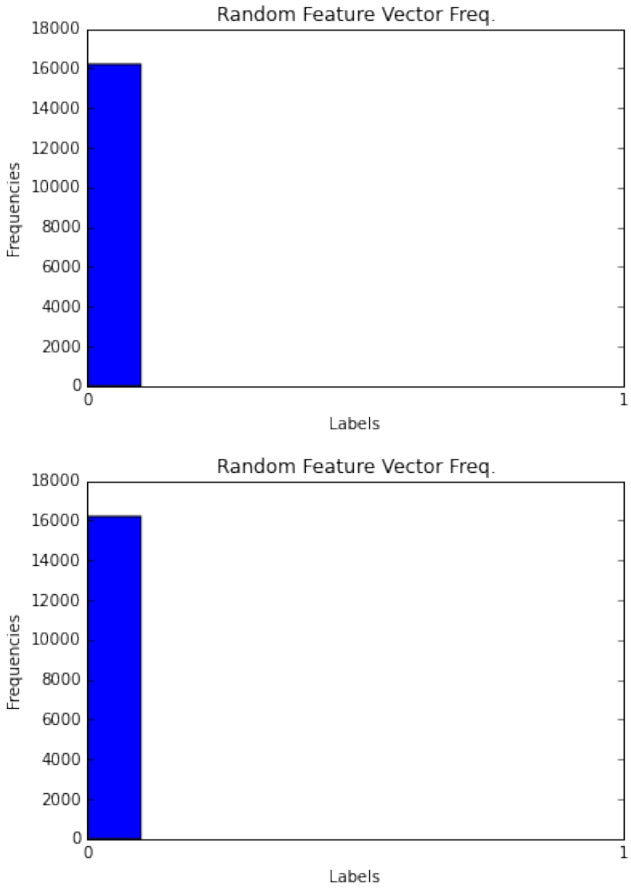


Figure 16: Frequencies of the Centroids of N-Grams.

```
for dim in train_dfs_truncated:
  cents_post_pca, clusts_post_pca, labs_post_pca =
      KMeans(k_clusters = 2, max_iterations = 300).fit(train_dfs_truncated[dim])
  print "K-Means for features of dimension: ", dim
  for idx, cent in enumerate(cents_post_pca):
    print "Centroid {0}:\n{1}".format(idx, cent)
  get_km_accuracy(labs_post_pca)
  print "\n"
```

Figure 19: Determining Accuracy of Dimensionally Reduced Features

```
K-Means for features of dimension: 10
Centroid 0:
[ 0.0495359 -0.02953208 0.02849827 -0.07664226 0.0772723
  0.08254926 -0.1688762 0.06698647 0.05351177 -0.0319536 ]
Centroid 1:
[-0.01143136 0.0068151 -0.00657652 0.01768667 -0.01783207
 -0.01904983 0.03897143 -0.01545842 -0.01234887 0.00737391 ]
Number of elements in cluster 0: 450
Accuracy for cluster 0 with label 0: 0.546666666667
Number of elements in cluster 1: 1950
Accuracy for cluster 1 with label 1: 0.510769230769
```

Figure 20: K-Means Accuracy for Features of Length 10

```
K-Means for features of dimension: 50
Centroid 0:
[  2.33451889e-01 -9.48260162e-02 -1.66303839e-02 1.55797027e-02
  -2.61674284e-02 3.54106282e-02  2.07071187e-02  1.67414802e-02
   1.34516207e-02 -5.93636544e-03 2.85694110e-03 5.69229331e-03
  -1.50157156e-03 2.01590915e-03  1.69539848e-03 -1.24397023e-03
  -3.86187692e-03 1.82187469e-03 -2.04843960e-03 3.60431664e-03
  -8.42079090e-04 1.39121471e-03 -1.71212503e-03 -1.20148639e-03
   4.47382814e-05 1.06648111e-03  2.84893837e-03 -2.64589317e-03
   7.92212628e-04 2.43487782e-04 -2.16747604e-03 1.25104262e-03
  -6.50117065e-04 -1.99350338e-04 1.30007893e-03 -3.93293581e-04
  -1.34478844e-03 1.44695904e-03 -1.20928513e-03 -1.11702416e-03
  -1.14071072e-03 1.02441308e-03  8.78880933e-04  1.12040207e-03
  -3.70525462e-05 5.79754894e-04 -6.73820264e-04 -4.93332896e-04
  -7.64843569e-04 -1.33559670e-03 ]
Centroid 1:
[ -8.96649125e-02 3.64210650e-02  6.38744847e-03 -5.98389964e-03
   1.00504656e-02 -1.36006219e-02 -7.95325319e-03 -6.43011869e-03
  -5.16653946e-03 2.28005732e-03 -1.09730264e-03 -2.18631335e-03
   5.76728177e-04 -7.74276523e-04 -6.51173809e-04 4.77787875e-04
   1.48328145e-03 -6.99751177e-04 7.86770921e-04 -1.38435691e-03
   3.23428301e-04 -5.34341981e-04 6.57598193e-04 4.61470553e-04
  -1.71832154e-05 -4.09617312e-04 -1.09422893e-03 1.01624271e-03
  -3.04275439e-04 -9.35195288e-05 8.32490798e-04 -4.80504259e-04
   2.49698942e-04  7.65670849e-05 -4.99338275e-04 1.51057396e-04
   5.16510440e-04 -5.55752433e-04 4.64465915e-04 4.29030041e-04
   4.38127646e-04 -3.93459696e-04 -3.37563265e-04 -4.30327438e-04
   1.42312548e-05 -2.22674025e-04 2.58802939e-04 1.89480801e-04
   2.93763447e-04  5.12980049e-04 ]
Number of elements in cluster 0: 666
Accuracy for cluster 0 with label 0: 0.533033033033
Number of elements in cluster 1: 1734
Accuracy for cluster 1 with label 1: 0.512687427912
```

Figure 21: K-Means Accuracy for Features of Length 50

```
K-Means for features of dimension: 100
Centroid 0:
[ -1.84689242e-01 -1.78211387e-01 1.01323766e-01 7.29991324e-02
  -2.63022236e-02 6.47328981e-02 7.80942063e-03 2.74497262e-02
  -2.29242855e-03 -1.35005505e-03 -2.71192559e-03 -8.28274023e-04
  -6.40414585e-03 9.28243948e-03 6.16942542e-03 1.02692320e-03
  -6.87645296e-03 1.60872386e-03 -2.79266092e-04 1.38346133e-03
   3.47052530e-04 -7.60313665e-04 6.19541974e-04 -4.88604441e-04
   3.33512899e-03 2.14831593e-03 2.00663767e-03 -2.92599928e-03
   1.06904302e-03 1.59303478e-03 -3.24099511e-03 -1.60862836e-04
   1.36202607e-04 -1.25584954e-03 -1.55630925e-03 -1.44493455e-04
   8.44393089e-04 -1.71378820e-03 -5.98897070e-04 -1.17734922e-03
  -1.98150406e-03 1.31661907e-03 7.41849395e-04 -8.35495796e-04
  -5.41763376e-04 8.77390982e-05 -4.26449929e-04 -4.56304183e-04
   8.39067011e-04 -8.83704580e-04 3.86160165e-04 7.11436070e-04
   5.74719985e-04 -3.79440408e-04 3.46689609e-04 -2.62696641e-04
   7.36146475e-04 3.01444743e-04 1.16263054e-04 4.11628165e-04
  -5.03755500e-04 2.76243456e-04 -5.45824309e-06 -4.94994644e-04
   2.58165863e-04 -2.46302740e-04 -8.93759461e-04 2.48889336e-03
  -2.90302880e-04 -8.82353528e-04 -7.63765284e-04 2.58681525e-04
  -1.42039990e-03 4.24894710e-04 5.81686918e-04 -1.47730043e-03
   7.01373039e-04 -4.77131959e-04 -3.63047257e-04 -3.45338293e-04
   8.26549598e-04 1.28615637e-05 9.25966633e-04 -9.26524135e-04
   1.29973434e-05 6.88815892e-04 6.84108802e-04 -5.65368108e-04
  -6.61801484e-04 -5.48938175e-04 3.32125891e-04 7.66460740e-05
   5.03641669e-04 -2.63619636e-04 -3.12830562e-04 -4.55243484e-04
  -3.95171442e-04 -1.51589817e-04 -3.13391379e-04 -1.27284451e-04 ]
Centroid 1:
[  4.76242699e-02 4.59538797e-02 -2.61275119e-02 -1.88236757e-02
   6.78233437e-03 -1.66921310e-02 -2.01374997e-03 -7.07823129e-03
   5.91129374e-04 3.48127402e-04 6.99301568e-04 2.13580094e-04
   1.65138352e-03 -2.39358502e-03 -1.59085813e-03 -2.64804097e-04
   1.77317341e-03 -4.14828164e-04 7.20120110e-05 -3.56741600e-04
  -8.94915330e-05 1.96055725e-04 -1.59756107e-04 1.25992340e-04
  -8.60001814e-04 -5.53968259e-04 -5.17434871e-04 7.54502959e-04
  -2.75665181e-04 -4.10782552e-04 8.35728299e-04 4.14803538e-05
  -3.51214270e-05 3.23835415e-04 4.01312449e-04 3.72593186e-05
  -2.17736583e-04 4.41920228e-04 1.54432578e-04 3.03593195e-04
   5.10953877e-04 -3.39505547e-04 -1.91294498e-04 2.15442312e-04
   1.39699990e-04 -2.26245473e-05 1.09965076e-04 1.17663343e-04
  -2.16363192e-04 2.27873508e-04 -9.95758916e-05 -1.83452068e-04
  -1.48198235e-04 9.78431241e-05 -8.93979495e-05 6.77393855e-05
  -1.89823934e-04 -7.77310345e-05 -2.99797811e-05 -1.06143112e-04
   1.29899217e-04 -7.12325893e-05 1.40747149e-06 1.27640128e-04
  -6.65710716e-05 6.35120274e-05 2.30466276e-04 -6.41790112e-04
   7.48579754e-05 2.27525124e-04 1.96945765e-04 -6.67040410e-05
   3.66266641e-04 -1.09564045e-04 -1.49994740e-04 3.80939104e-04
  -1.80857199e-04 1.23034027e-04 9.36159594e-05 8.90494968e-05
  -2.13135431e-04 -3.31650386e-06 -2.38771270e-04 2.38915028e-04
  -3.35151621e-06 -1.77619192e-04 -1.76405414e-04 1.45786745e-04
   1.70653213e-04 1.41550096e-04 -8.56425254e-05 -1.97640820e-05
  -1.29869864e-04 6.79773903e-05 8.06670002e-05 1.17389829e-04
   1.01899554e-04 3.90891981e-05 8.08116134e-05 3.28217766e-05 ]
Number of elements in cluster 0: 492
Accuracy for cluster 0 with label 1: 0.573170731707
Number of elements in cluster 1: 1908
Accuracy for cluster 1 with label 0: 0.518867924528
```


Figure 22: K-Means Accuracy for Features of Length 100

J. According to the above results, our logistic regression classifier with the bag of words model on full length feature vectors was the most accurate. This was as expected since the length of the reviews were rather small, and the accuracy of our sentimental analysis was dependent on the usage of specific, repetitive words. The K-means algorithm is does not pick up on these repetitive words as precisely, and rather computes the noisy distances of thousands of other rarely used words.

2 Part 2: EM Algorithm and Implementation

2.1 Summary

The parameters of a Gaussian Mixture Model (GMM) can be estimated using the Expectation-Maximum (EM) algorithm. The goal of this assignment was to implement the EM algorithm to fit the Gaussian Mixtures from the Old Faithful data set.

2.2 Solution Steps

A. GMM and K-Means are closely related - it is a "soft" version of K-means. Both algorithms have a similar work flow: initialize, assign, re-factor. In K-Means, σ is set to 0, making assignment of weights "hard" (i.e. each point belongs to one cluster or the other). The "soft" version for GMM computes responsibilities which are shared between the two clusters (i.e. an observation can be 0.4 to one cluster and 0.6 to another).

   K-Means:

   (a) Pick random initial parameters

   (b) Assign labels to each cluster based on means (similar to E-Step)

   (c) Update parameters (centroids) based on the previous step (similar to M-Step)

   GMM:

   (a) Pick random initial parameters

   (b) Assign responsibilities to each cluster based on means (similar to k-means assignment)

   (c) Update parameters (mean, covariance, weights) based on the previous step (similar to k-means update)

B. The data set was downloaded, and then parsed using the pandas Python library. The 272 observations consist of two variables - eruption time and waiting time between eruptions (both measured in minutes). Each variable was linearly scaled between [0,1] to account for varying scale of the original observations. The scaled observations are plotted in Fig.18 below.

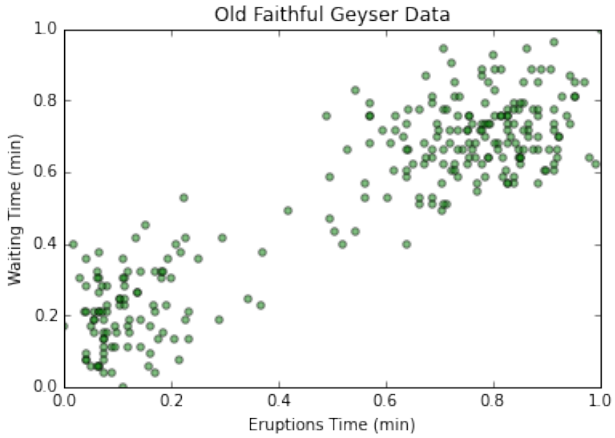


Figure 23: Linearly Scaled Observations from Old Faithful Data Set

C. The EM algorithm is used to fit a GMM. Essentially, the algorithm is broken down into 4 steps:

   (a) Initialization: initial parameters (mean, covariance, weights) are chosen. In this assignment, we chose the mean (μ) by randomly selecting two points from the data set. The covariance was assumed to be spherical ($\sigma^2 I$) with σ chosen randomly as well. For the initial weights (π), the two modes were initialized with an even 0.5 each. (See Fig. 19)

   (b) E-Step: For each observation in the data set, compute the "responsibility" (or weights) for each point respectively. This is the measure of weights that each point owes to the two Gaussian modes. Unlike k-means, observations are not assigned to one or the other cluster. Rather, each point is partially "responsible" to each cluster with varying degree of pull or weight. (See Fig. 20)

   (c) M-Step: Using the computed responsibilities, we can compute a weighted mean and variances that can be used in the next iteration. (See Fig. 20)

   (d) Repeat the E-Step and the M-Step until convergence. In our code, we measured convergence by computing the log likelihood of the data. For each iteration, if the new log likelihood is within some tolerance threshold condition (we set to 0.001) of the previous log likelihood value, then we finish iteration. To prevent an infinite loop, we set a max iteration value (e.g. 100 iterations). (See Fig. 20)

```
def initialize_params (self, X):
   n, p = X.shape # dimensions
   mu = X[np.random.choice(n, self.k, False)]
   sigma = [random.random()*np.identity(p) for i in range(self.k)]
   pi = [0.5, 0.5]
   return [mu, sigma, pi]
```

Figure 24: Initialization of GMM

The entire EM algorithm was run for 50 trials. Two Guassians are plotted from a sample trial (Fig.25).

The trajectories of the means for each iteration of a sample trial was recorded and plotted below (Fig. 26). The figure plots mean against iteration number.

Additionally, the distribution of the total number of iterations needed to converge for all 50 trials was recorded. The results are plotted in a histogram below (Fig. 27).

```
def fit (self, X, mu, sigma, pi, max_iter=100, tolerance=0.01):
    # Dimensions
    n, p = X.shape
    k = self.k

    # Keep track of log likelihood for convergence purposes
    log_likelihood_old, log_likelihood_new = 0, 0

    for i in range(max_iter):
        # E-Step
        resp = np.zeros((k, n))
        for mode in range(k):
            resp[mode] = pi[mode] * mvn(mu[mode], sigma[mode]).pdf(X)
        resp /= resp.sum(0)

        # M-Step
        pi = resp.sum(axis=1) / n
        mu = np.asarray([np.dot(r,X) / r.sum() for r in resp])

        # Sigma implementation adapted from Ref.8
        sigma = np.zeros((k, p, p))
        for j in range(k):
            Y = X - mu[j, :]
            sigma[j] = (resp[j,:,None,None] * mm(Y[:,:,None], Y[:,None,:])).sum(axis=0)
        sigma /= resp.sum(axis=1)[:,None,None]

        # Track trajectory of means against iteration
        self.trajectory.append(mu)

        # Update log likelihood and check for convergence
        log_likelihood_new = np.sum([P * mvn(M, S).pdf(X) for P,M,S in zip(pi, mu, sigma)], axis=0)
        log_likelihood_new = np.log(log_likelihood_new).sum()
        if np.abs(log_likelihood_new - log_likelihood_old) < tolerance:
            break

        # Otherwiae, keep updated value for next iteration
        log_likelihood_old = log_likelihood_new
```

Figure 25: Implementation of EM Algorithm

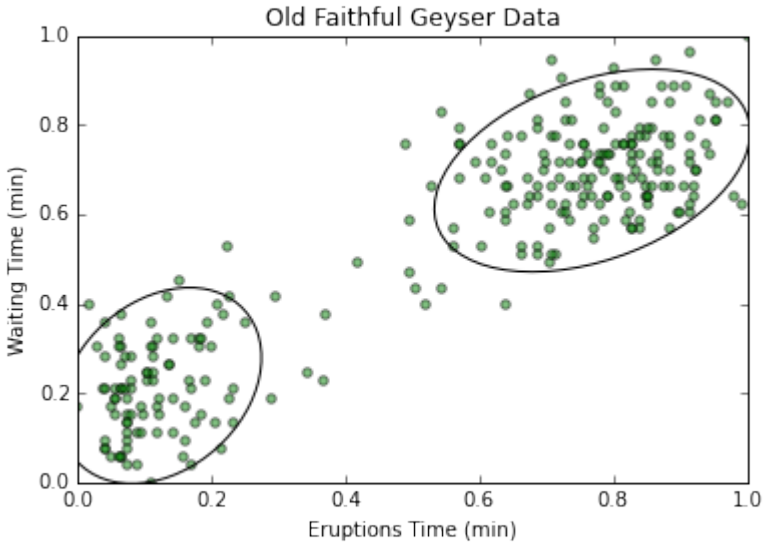


Figure 26: GMM Plotted Over Data Set

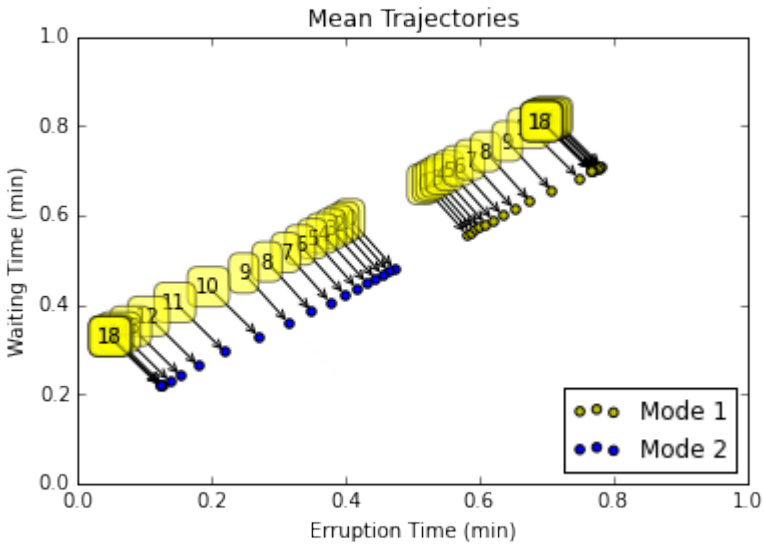


Figure 27: Trajectory of Means from Sample Iteration

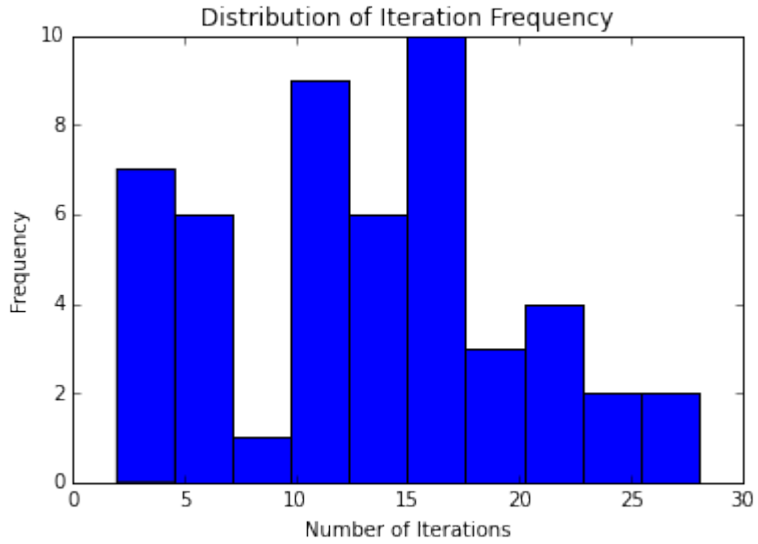


Figure 28: Sample Distribution of Iteration frequencies (50 trials)

D. The above task (GMM and EM) was then repeated - instead of randomly initializing the EM parameters, we ran a k-means algorithms over the data points with K=2 and labeled each point with one of the 2 clusters. Then, we estimated the first guess mean and covariance matrices using max-likelihood over the labeled data points.

   The K-Means algorithm was implemented using the sklearn library (Fig.28).

```
# Repeat 50 trials
k_mu, k_sigma, k_pi, k_total_iterations = [], [], [], []
for trial in range(50):
    # Get initial params from K-Means clustering
    k = cluster.KMeans(2).fit(features)
    k_m = k.cluster_centers_
    k_s = [np.cov(k.predict(features))*np.identity(2) for i in range(2)]
    k_p = [0.5, 0.5]

    k_g = GMM(2)
    k_mu, k_sigma, k_pi = k_g.fit(features, k_m, k_s, k_p)
    k_total_iterations.append(len(k_g.trajectory))
```

Figure 29: Implementation of EM Algorithm

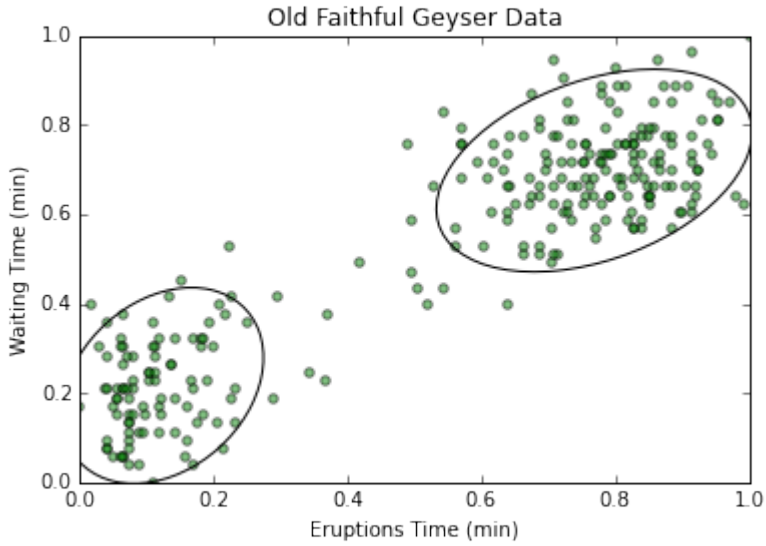


Figure 30: GMM Plotted over Data Set using k-means initialization

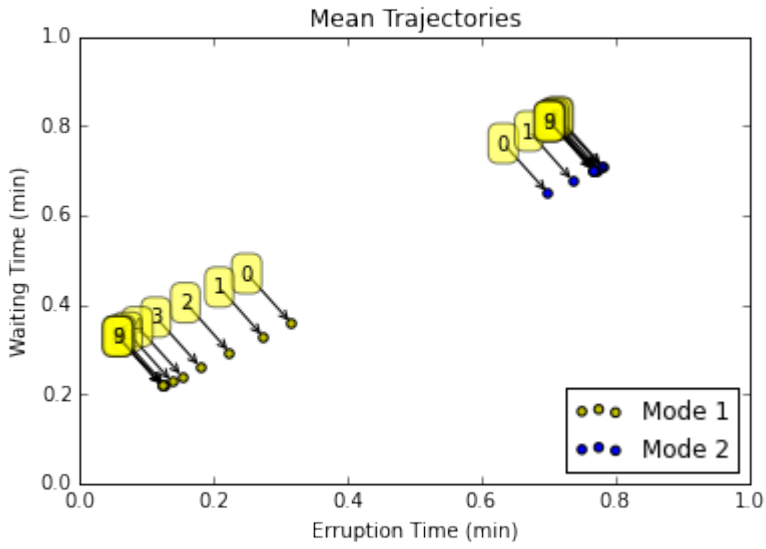


Figure 31: Sample Distribution of Iteration frequencies (50 trials) using k-means initialization

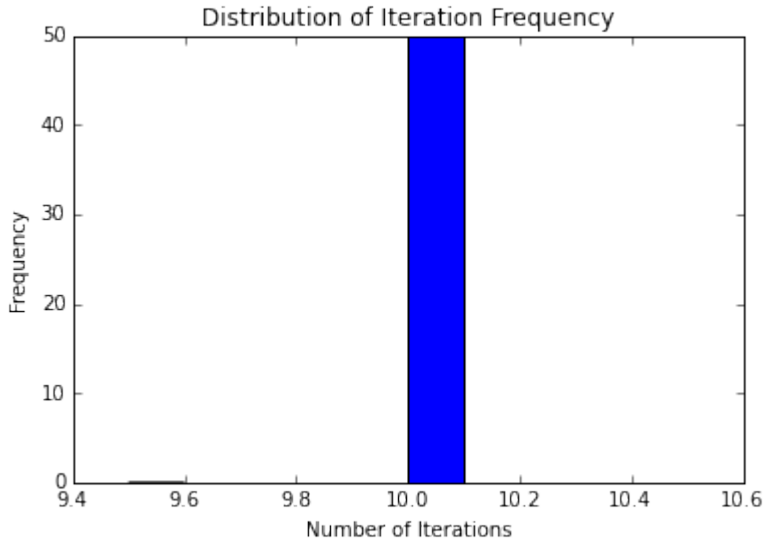


Figure 32: Sample Distribution of Iteration frequencies (50 trials) using k-means initialization

E. Comparing the algorithm performance of the two (random vs. k-means parameter initialization), we see that the range and variance in the number of iterations is greater when the initial parameters are randomized. When the parameters were randomized, it sometimes took close to 30 iterations before convergence. The k-means method was more stable - since k-means already finds a cluster center (and k-means is closely associated with GMM as explained above), the number of iterations until convergence was consistently lower. (In fact, they were consistently 10 iterations, likely due to the fact the the cluster centers from scikit's K-Means library are almost identical each trial).

3 Part 3: Hand Written Problems

3.1 EM Algorithm Derivation

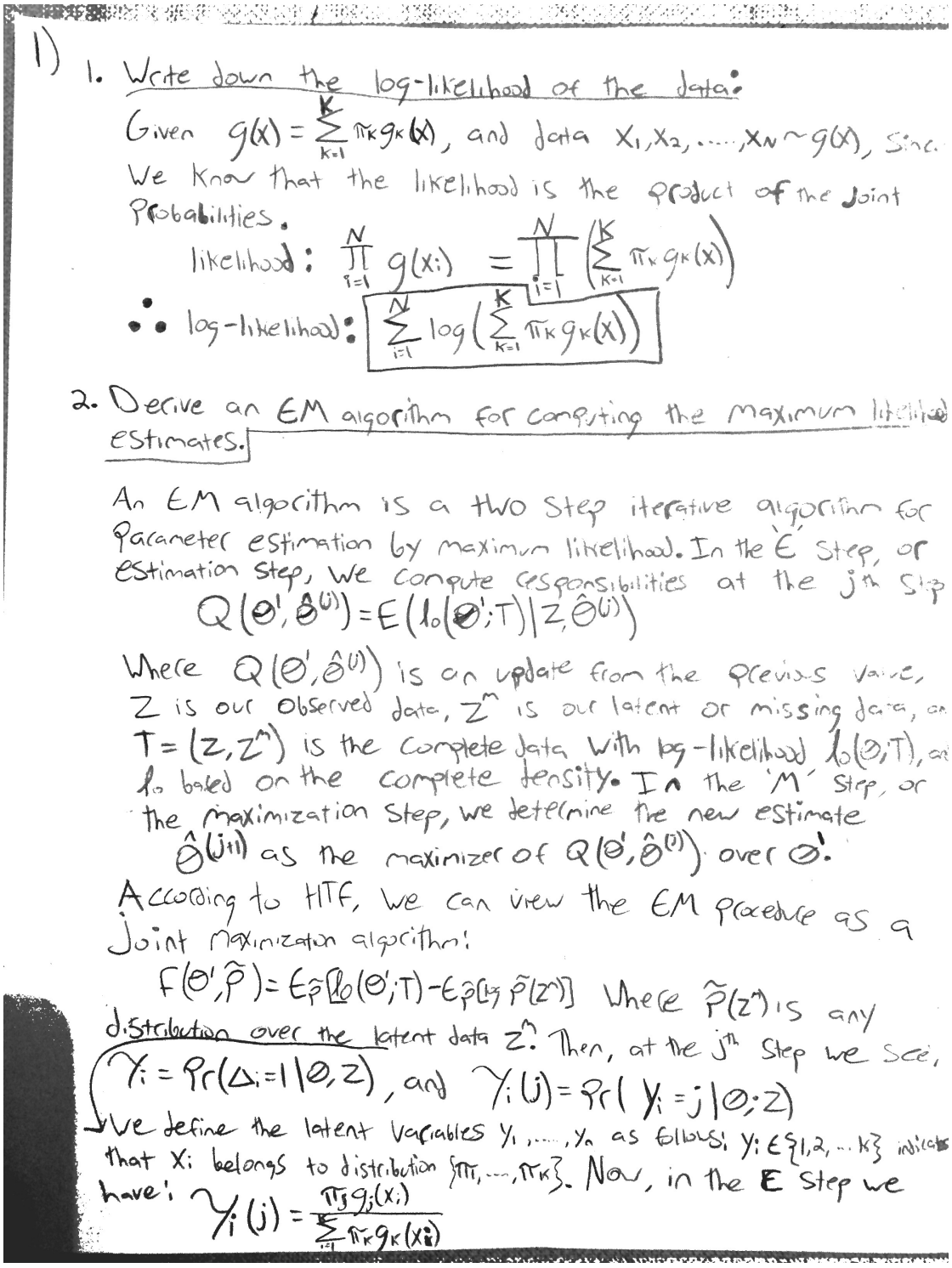


Figure 33: Problem 1.a

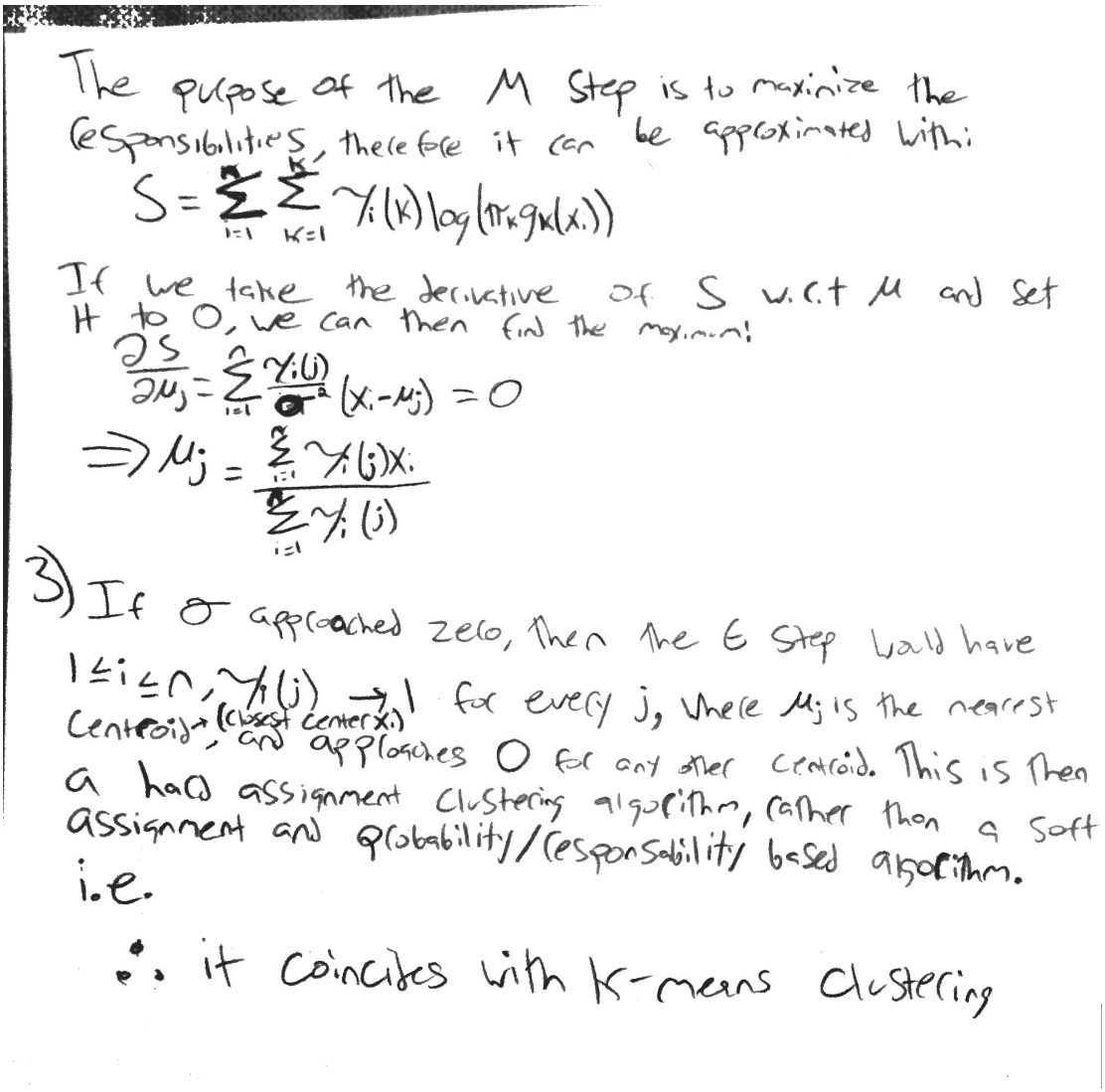


Figure 34: Problem 1.b,1.c

3.2 Procrustes Problem

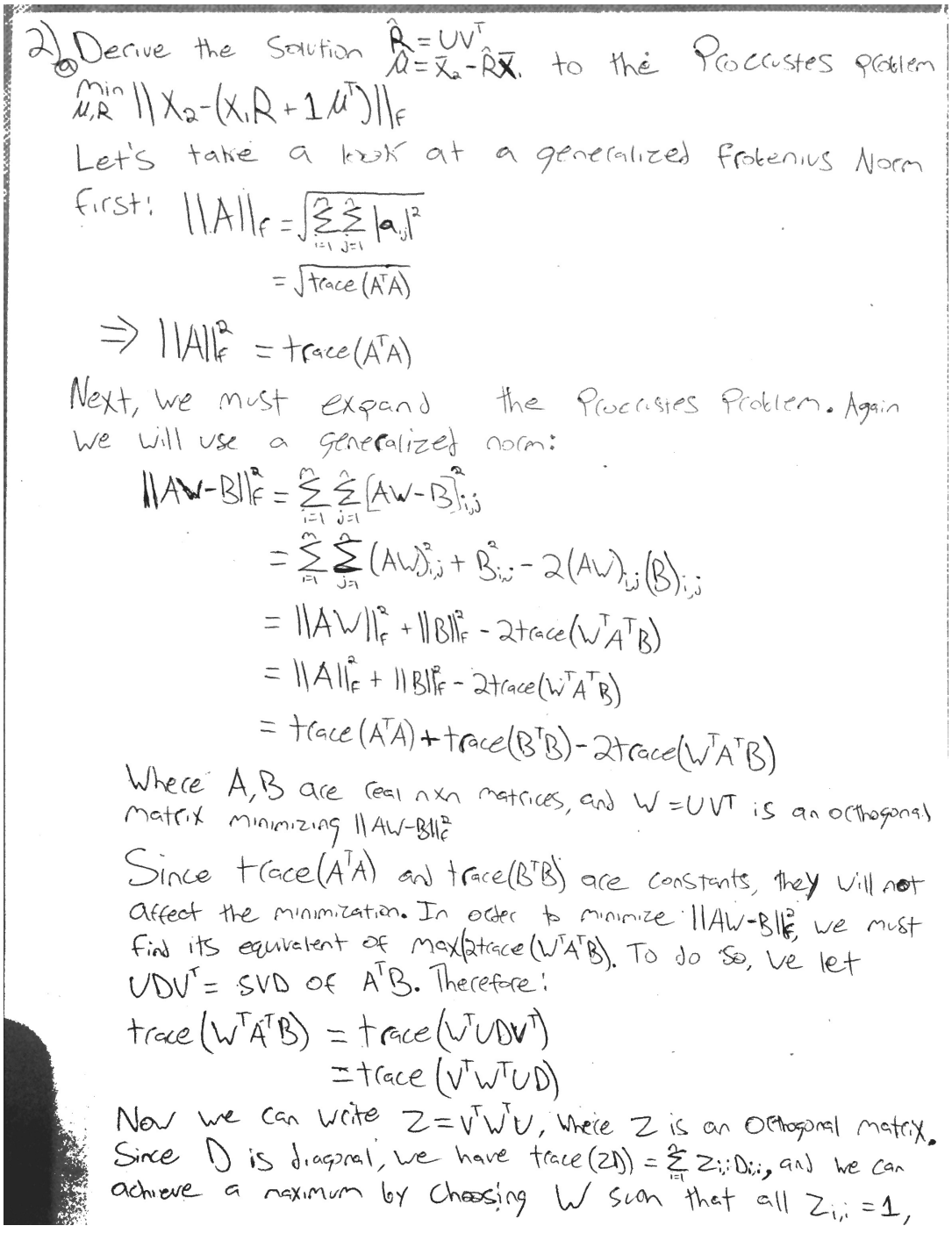


Figure 35: Problem 2.a

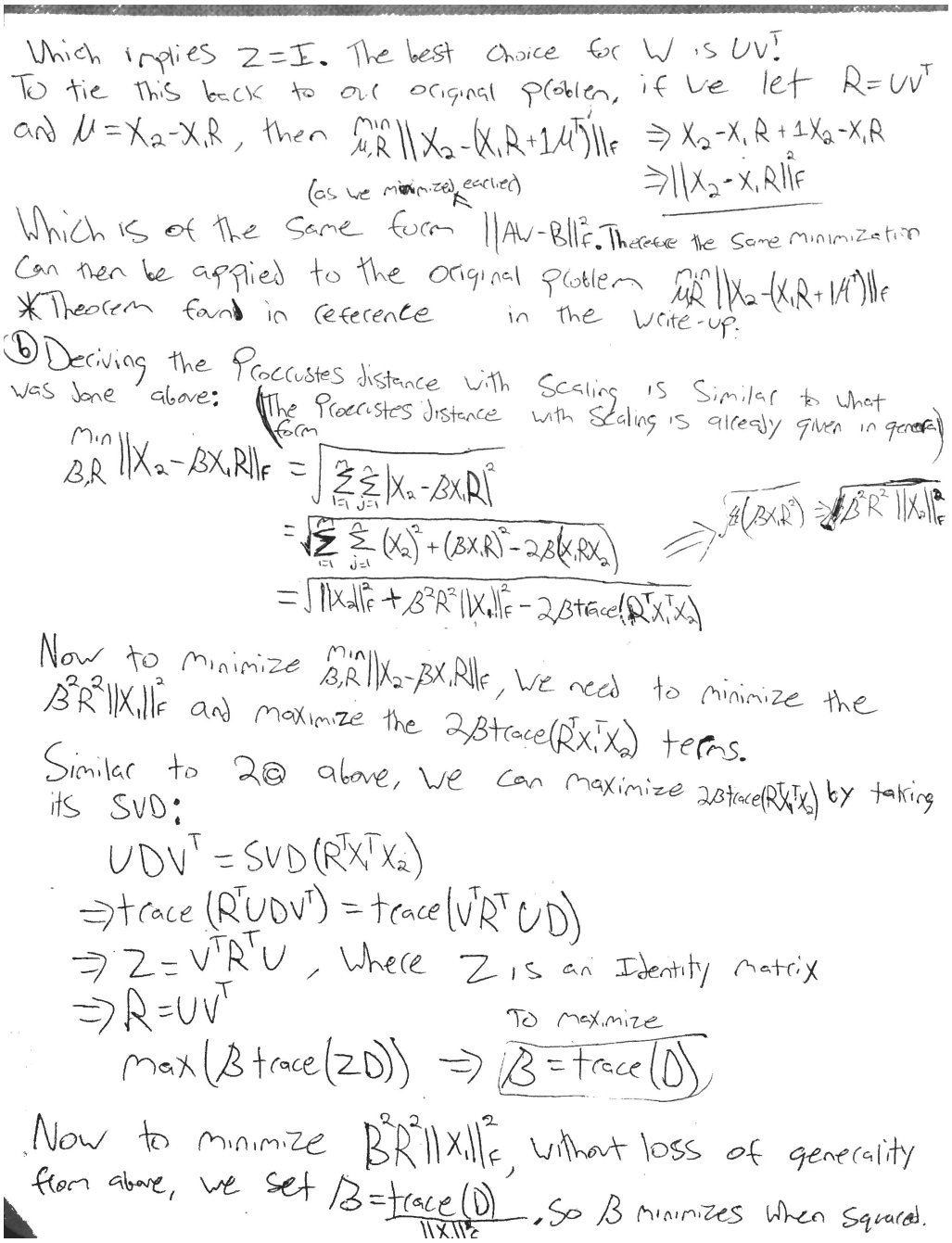


Figure 36: Problem 2.b

3.3 Classical Scaling Problem

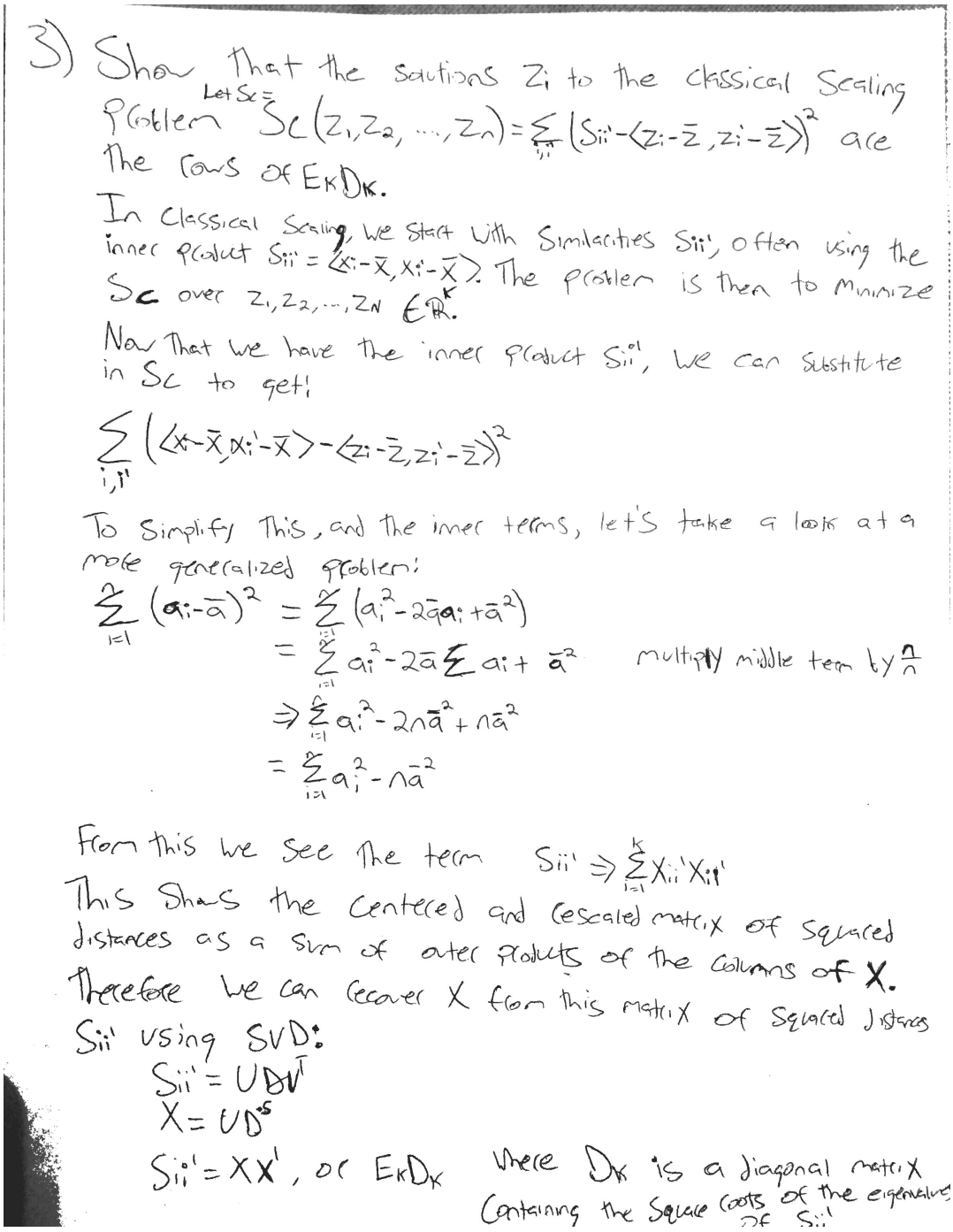


Figure 37: Problem 3

References

[1] HTF: The Elements of Statistical Learning
http://statweb.stanford.edu/ tibs/ElemStatLearn/

[2] The Elements of Statistical Learning Solutions Manual
waxworksmath.com/Authors/G_M/Hastie/WriteUp/weatherwax_epstein_hastie_solutions_manual.pdf

[3] Matplotlib Plotting (Histograms, pixel images, etc):
http://matplotlib.org/

[4] Numpy
http://www.numpy.org/

[5] Pandas Data Parser (read_csv, to_csv):
http://pandas.pydata.org/

[6] Scikit-learn
http://scikit-learn.org/

[7] Scipy
http://www.scipy.org/

[8] Scikit GMM Source
https://github.com/scikit-learn/scikit-learn/blob/c957249/sklearn/mixture/gmm.py#L115

[9] EM Algorithm
https://gist.github.com/bistaumanga/6023716

[10] EM Algorithm
http://comnuan.com/cmnn01004/

[11] Bivariate Normal Ellipse Plotting
http://www.nhsilbert.net/source/2014/06/bivariate-normal-ellipse-plotting-in-python/

[12] PCA
http://sebastianraschka.com/Articles/2014_pca_step_by_step.html

[13] K-Means
http://stanford.edu/ cpiech/cs221/handouts/kmeans.html

[14] EM Algorithm Problem
http://www.stat.ucla.edu/ yuille/courses/stat153/emtutorial.pdf
http://ce.sharif.edu/courses/91-92/2/ce725-1/resources/root/Quizes/SPR-Quiz9-sol.pdf

[15] Procrustes Problem
http://mathworld.wolfram.com/FrobeniusNorm.html
http://winvector.github.io/xDrift/orthApprox.pdf

[16] Classical Scaling Problem
http://users.stat.umn.edu/ gary/classes/5401/handouts/34.mds.handout.pdf
http://www.sosmath.com/CBB/viewtopic.php?t=44808