BST 263 HOMEWORK 6

Jenny Wang (71401898)

April 28, 2020

1/7 late days used for this homework; 2/7 late days used in total; 5/7 late days remaining

Question 1.1)

Given:

$\mathbb{P}(Y = 1) = \eta$ and $\mathbb{P}(Y = 0) = 1 - \eta$

For one observation, we can obtain the conditional probability $\mathbb{P}(Y = 1|X)$ using Bayes' rule:

$$\begin{aligned}\mathbb{P}(Y = 1|X) &= \frac{\mathbb{P}(X|Y = 1)\mathbb{P}(Y = 1)}{\mathbb{P}(X|Y = 1)\mathbb{P}(Y = 1) + \mathbb{P}(X|Y = 0)\mathbb{P}(Y = 0)} \\ &= \frac{p(X|Y = 1)\eta}{p(X|Y = 1)\eta + p(X|Y = 0)(1 - \eta)} \\ &= \frac{p_1(x)\eta}{p_1(x)\eta + p_2(x)(1 - \eta)}\end{aligned}$$

Question 1.2)

For n iid observations, we can find the joint probabilities from Bayes' rule and substitute in the PDFs. From Bayes' rule:

$\mathbb{P}(X_i, Y_i) = \mathbb{P}(X_i|Y_i)\mathbb{P}(Y_i)$

The PDFs are:

$p(x_i|y_i) = p_1(x_i)^{y_i} p_2(x_i)^{1-y_i}$ and $p(y_i) = \eta^{y_i}(1 - \eta)^{1-y_i}$

For any ith observation,

$$\begin{aligned}p(x_i, y_i) &= p(x_i|y_i)p(y_i) \\ &= p_1(x_i)^{y_i} p_2(x_i)^{1-y_i} \eta^{y_i}(1 - \eta)^{1-y_i}\end{aligned}$$

The likelihood and the log-likelihood equations are:

$$
\begin{aligned}
L(\eta, p_1, p_2) &= \prod_{i=1}^{n} \Big[p_1(x_i)^{y_i} p_2(x_i)^{1-y_i} \eta^{y_i} (1-\eta)^{1-y_i} \Big] \\
l(\eta, p_1, p_2) &= \sum_{i=1}^{n} \log \Big[p_1(x_i)^{y_i} p_2(x_i)^{1-y_i} \eta^{y_i} (1-\eta)^{1-y_i} \Big] \\
&= \sum_{i=1}^{n} \Big[y_i \log p_1(x_i) + (1-y_i) \log p_2(x_i) + y_i \log \eta + (1-y_i) \log(1-\eta) \Big] \\
&= \sum_{i=1}^{n} \Big[y_i [log\eta + \log p_1(x_i)] + (1-y_i) [\log(1-\eta) + \log p_2(x_i)] \Big] \\
&= \sum_{i=1}^{n} \Big[y_i \log[\eta p_1(x_i)] + (1-y_i) \log[(1-\eta) p_2(x_i)] \Big]
\end{aligned}
$$

Question 1.3)

E-step

For any ith observation, we want to find $\hat{\eta}_i^{(t+1)}$ from $\hat{\eta}^{(t)}$. This is a conditional probability we can calculate using Bayes' rule:

$$
\begin{aligned}
\hat{\eta}_i^{(t+1)} &= \frac{p(X_i|Y_i=1)\hat{\eta}^{(t)}}{p(X_i|Y_i=1)\hat{\eta}^{(t)} + p(X_i|Y_i=0)(1-\hat{\eta}^{(t)})} \\
&= \frac{p_1(x_i)\hat{\eta}^{(t)}}{p_1(x_i)\hat{\eta}^{(t)} + p_2(x_i)(1-\hat{\eta}^{(t)})}
\end{aligned}
$$

We are updating the conditional probability $\mathbb{P}(Y_i|X_i)$, *ie.* the probability of X_i being in each of the $k=2$ mixtures given the value of X_i, which remains unchanged throughout the iterations. We are trying to answer the question: is each X_i more likely to come from mixture 1 or mixture 2? The densities p_1 and p_2 and their respective parameters also remain unchanged for all observations throughout all time steps. The E-step is basically the same as Question 1.1), with the exception that we are estimating $\mathbb{P}(Y=1) = \eta$ using $\hat{\eta}^{(t)}$ (calculated from the M-step of the previous iteration), as we do not know what the true η is.

M-step

To find $\hat{\eta}^{(t+1)}$, we find the average of all the $\hat{\eta}_i^{(t+1)}$s, the sum of which represents the complete set of "conditions."

$$
\begin{aligned}
\hat{\eta}^{(t+1)} &= \frac{1}{n} \sum_{i=1}^{n} \hat{\eta}_i^{(t+1)} \\
&= \frac{1}{n} \sum_{i=1}^{n} \frac{p_1(x_i)\hat{\eta}^{(t)}}{p_1(x_i)\hat{\eta}^{(t)} + p_2(x_i)(1-\hat{\eta}^{(t)})}
\end{aligned}
$$

Before we initiate the M-step, we have obtained the conditional probability $\mathbb{P}(Y_i|X_i)$ for every X_i. We are updating $\eta = \mathbb{P}(Y=1)$ for the $(t+1)$th iteration, which we can use as an estimate of the true η to substitute into the log-likelihood found in Question 1.2) and find the MLE of the parameters. Once the algorithm converges, these estimates should be very close to the true parameters.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time
from helpers import plot_digit, plot_cluster_viz, plot_cluster_centers,
plot_cluster_variances, crosstab_with_error_rate
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
from sklearn.decomposition import PCA
from sklearn.ensemble import GradientBoostingRegressor
```

```
Bad key "text.kerning_factor" on line 4 in
/opt/anaconda3/lib/python3.7/site-packages/matplotlib/mpl-data/styleli
b/_classic_test_patch.mplstyle.
You probably need to get an updated matplotlibrc file from
https://github.com/matplotlib/matplotlib/blob/v3.1.3/matplotlibrc.templ
ate
or from the matplotlib source distribution
```

Question 2.1)

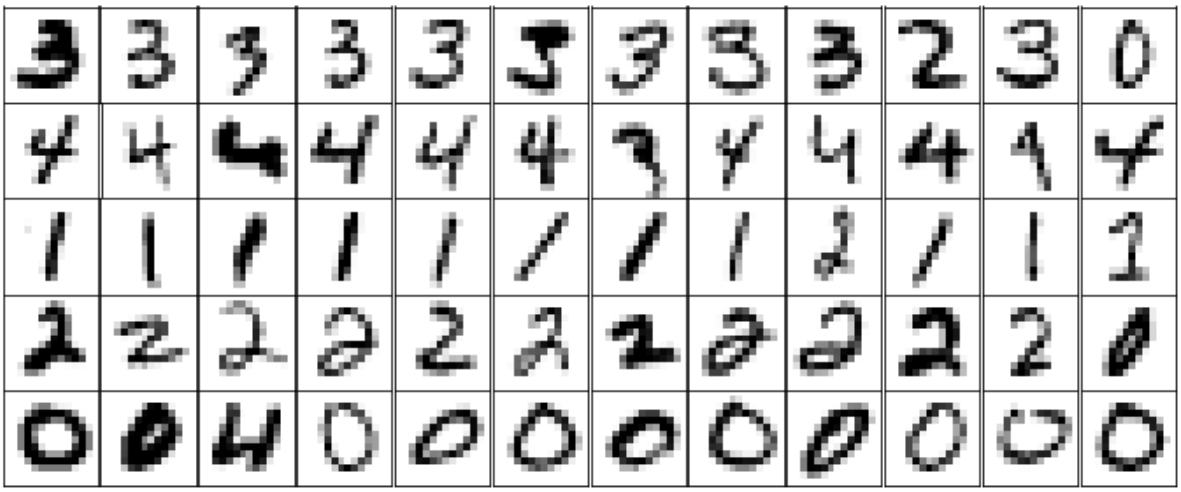
In [2]:
```
# Load data
labels = np.array(pd.read_csv("mnist_labels_preprocessed.csv", header=No
ne)).flatten()
pixels = np.array(pd.read_csv("mnist_pixels_preprocessed.csv", header=No
ne))
```

In [3]:
```
# k-means
## Time taken to run the algorithm
start = time.time()
kmeans1_kpp = KMeans(n_clusters=5, init="k-means++", n_init=1, random_st
ate=0).fit(pixels)
kmeans1_kpp_time = time.time() - start

start = time.time()
kmeans2_kpp = KMeans(n_clusters=5, init="random", n_init=1, random_state
=0).fit(pixels)
kmeans2_kpp_time = time.time() - start

print("The time it takes to run the first algorithm is " + str(kmeans1_k
pp_time) + " seconds.")
print("The time it takes to run the second algorithm is " + str(kmeans2_
kpp_time) + " seconds.")

## Objective function at convergence
print("The objective function at convergence for the first algorithm is
 " + str(kmeans1_kpp.inertia_) + ".")
print("The objective function at convergence for the second algorithm is
" + str(kmeans2_kpp.inertia_) + ".")
```

```
The time it takes to run the first algorithm is 0.43067312240600586 sec
onds.
The time it takes to run the second algorithm is 0.39258384704589844 se
conds.
The objective function at convergence for the first algorithm is 147655
27714.831192.
The objective function at convergence for the second algorithm is 14765
527723.013445.
```

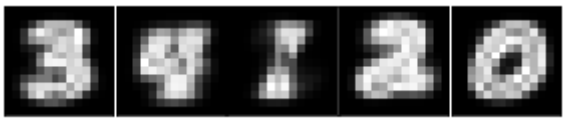
From the objective functions, the *k*-means++ initialization performs slightly better than the random initialization.

In [4]:
```
# Plot 1
plot_cluster_viz(pixels, kmeans1_kpp.labels_, random_state=0)
```



In [5]:
```
# Plot 2
plot_cluster_centers(pixels, kmeans1_kpp.labels_)
```

In [6]:
```
# Plot 3
plot_cluster_variances(pixels, kmeans1_kpp.labels_)
```



In [7]:
```
# Table 1
crosstab_with_error_rate(kmeans1_kpp.labels_, labels)
```

Out[7]:

	True 0	True 1	True 2	True 3	True 4	Assign to	Error rate
Cluster 0	346	38	492	5128	3	Digit 3	0.146329
Cluster 1	109	14	327	147	5559	Digit 4	0.096979
Cluster 2	23	6608	667	392	235	Digit 1	0.166183
Cluster 3	220	82	4402	427	36	Digit 2	0.148055
Cluster 4	5225	0	70	37	9	Digit 0	0.021719

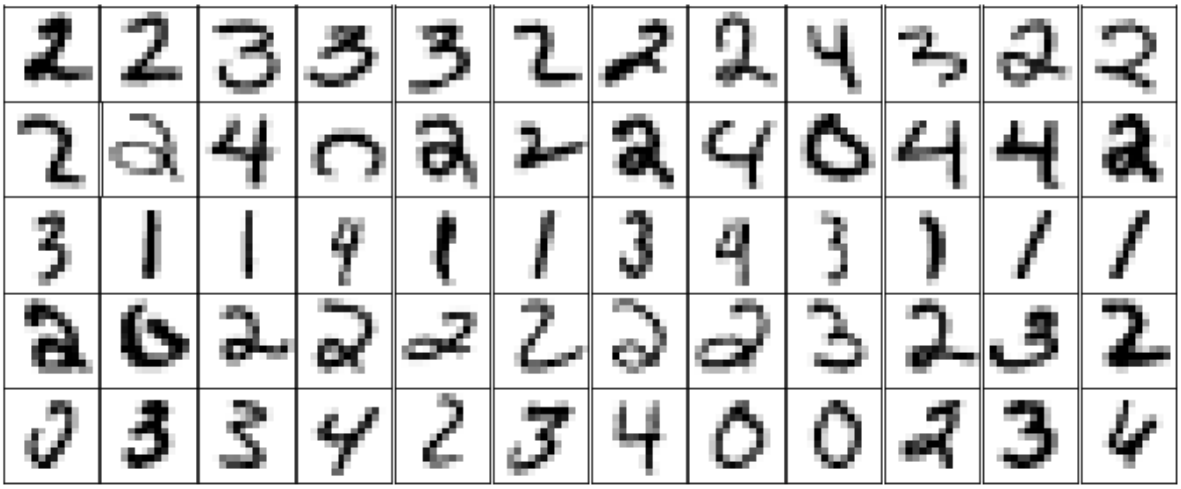
Question 2.2)

In [8]:
```
# EM for Gaussian mixtures
## Time taken to run the algorithm
start = time.time()
em1_kmeans = GaussianMixture(n_components=5, init_params="kmeans", n_ini
t=1, random_state=0).fit(pixels)
em1_kmeans_time = time.time() - start

start = time.time()
em2_kmeans = GaussianMixture(n_components=5, init_params="random", n_ini
t=1, random_state=0).fit(pixels)
em2_kmeans_time = time.time() - start

print("The time it takes to run the first algorithm is " + str(em1_kmean
s_time) + " seconds.")
print("The time it takes to run the second algorithm is " + str(em2_kmea
ns_time) + " seconds.")

## Per-sample average log-likelihood function
print("The per-sample average log-likelihood function for the first algo
rithm is " + str(em1_kmeans.score(pixels)) + ".")
print("The per-sample average log-likelihood function for the second alg
orithm is " + str(em2_kmeans.score(pixels)) + ".")
```

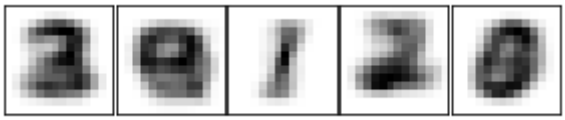
```
The time it takes to run the first algorithm is 59.27602195739746 secon
ds.
The time it takes to run the second algorithm is 43.651488065719604 sec
onds.
The per-sample average log-likelihood function for the first algorithm
is -75.22425534125553.
The per-sample average log-likelihood function for the second algorithm
is -142.97014458679465.
```

From the per-sample average log-likelihood functions, the *k*-means++ initialization performs better than the random initialization.

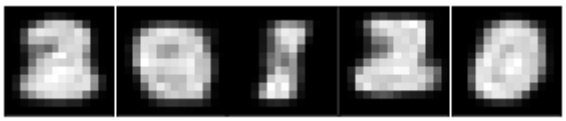
In [9]:
```
# Plot 1
em1_kmeans.labels_ = em1_kmeans.predict(pixels)
plot_cluster_viz(pixels, em1_kmeans.labels_, random_state=0)
```



In [10]:
```
# Plot 2
plot_cluster_centers(pixels, em1_kmeans.labels_)
```



In [11]:
```
# Plot 3
plot_cluster_variances(pixels, em1_kmeans.labels_)
```



In [12]:
```
# Table 1
crosstab_with_error_rate(em1_kmeans.labels_, labels)
```

Out[12]:

	True 0	True 1	True 2	True 3	True 4	Assign to	Error rate
Cluster 0	158	76	2177	1889	514	Digit 2	0.547777
Cluster 1	144	18	111	76	304	Digit 4	0.534456
Cluster 2	120	6077	144	930	665	Digit 1	0.234249
Cluster 3	42	26	637	110	64	Digit 2	0.275313
Cluster 4	5459	545	2889	3126	4295	Digit 0	0.665379

Question 2.3a)

The *k*-means algorithm had more satisfactory results compared to the EM algorithm. Across the board, the *k*-means algorithm had an error rate of less than 20% for all of the clusters. On the other hand, the EM algorithm had error rates as high as 67%. This shows that the *k*-means algorithm gives more correct predictions compared to the EM algorithm. Additionally, from the plots, the *k*-means algorithm visually gives digits with better defined centers and less variance compared to the EM algorithm.

Question 2.3b)

By looking at the cluster centers, the *k*-means algorithm captured most of the digits' traits pretty well. We can see the curves in `3`, `2`, and `0`, as well as where the stroke is supposed to change direction in `3`, `4`, and `2`. It seems to have only picked up the looped `2` (as opposed to the straight `2`) and gives a `1` that is a little blurry. For the EM algorithm, `1` and `2` were predicted correctly without much ambiguity, and EM seems to focus more on the straight `2` as opposed to the looped `2`. The biggest downfall of EM was that `3` was erronerously predicted as `2`. While both `4` and `0` were predicted correctly, they look more like `8`. As such, we may not want to use the EM algorithm for the purposes of this example.

Question 2.3c)

It took 0.4 seconds to run the best *k*-means algorithm and almost 1 minute to run the best EM algorithm. The *k*-means algorithm took a very short time to run and gave us good results, while the EM algorithm took much longer with worse results. Therefore, the quality of the results was worth the amount of time need to run the *k*-means algorithm, but not the EM algorithm.

Question 2.3d)

Taking both the error rates and the runtime into consideration, the *k*-means algorithm outperforms the EM algorithm on both of these fronts. The answer would not change depending on the initialization strategy used, since we used the better of the two strategies to perform our analysis to begin with.

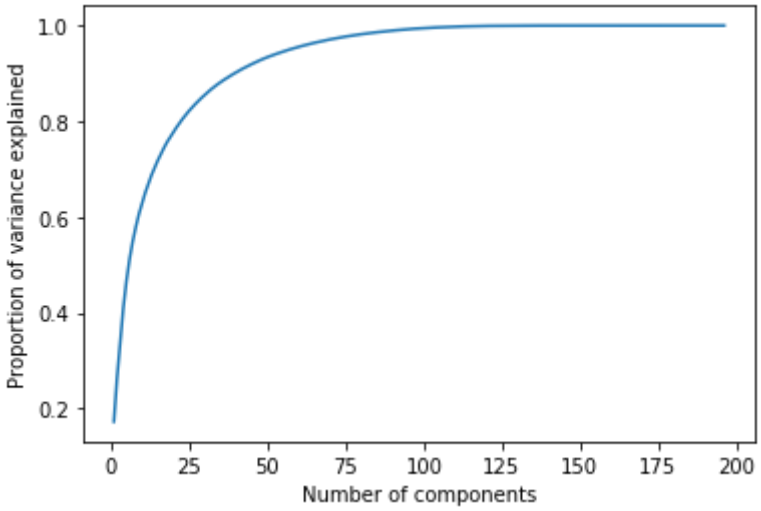
Question 2.3e)

Judging by the high error rates and the excess runtime, mixture models for Gaussian distributions are not good for modeling the data. As alternatives, we could try using *k*-nearest neighbors, naïve Bayes, decision trees, and support vector machines. We could also use hierarchical classification and neural networks.

Question 2.4)

In [13]:
```
# PCA
full_pca = PCA(random_state=0).fit(pixels)

# Cumulative sum of the varaicne calculated by each component
cum_exp_var = np.cumsum(full_pca.explained_variance_ratio_)

# Plot the proportion of total variance explained by each component
plt.plot(np.arange(len(cum_exp_var))+1, cum_exp_var)
plt.xlabel("Number of components")
plt.ylabel("Proportion of variance explained")
```

Out[13]:
```
Text(0, 0.5, 'Proportion of variance explained')
```



From the plot above, it is probably sufficient to include the first 60 PCs, as this is where the curve tapers off into a plateau.

In [14]:
```
# Number of PCs it takes to explain x amount of variance
cutoffs = np.flip(np.arange(0.5, 1, 0.05))
cutoffs_df = pd.DataFrame([[min(np.arange(len(cum_exp_var))[cum_exp_var
> c]) + 1 for c in cutoffs]],
                          columns=cutoffs, index=["Number of PCs needed"
])
cutoffs_df.columns.name = "Variance explained"
cutoffs_df

# Variance explained by 60 PCs
print("The variance explained by 60 PCs is: " + str(np.sum(PCA(60, rando
m_state=0).fit(pixels).explained_variance_ratio_)) + ".")
```

```
The variance explained by 60 PCs is: 0.9553591276779209.
```

From the table above, we see that the first 58 PCs would explain 95% of the total variance. Therefore, 60 is a good number as it explains even more than 95%.

Question 2.5)

In [15]:
```
# Final PCA
final_pca = PCA(60, random_state=0).fit(pixels)

# Projections from the first 60 PCs
pixel_X_transform = final_pca.transform(pixels)
print("The new dimensions are " + str(pixel_X_transform.shape) + ".")
```

```
The new dimensions are (30596, 60).
```

In [16]:
```
# k_means using initialization 1
kmeans_kpp = KMeans(n_clusters=5, init="k-means++", n_init=1, random_sta
te=0).fit(pixel_X_transform)

# Table 1
crosstab_with_error_rate(kmeans_kpp.labels_, labels)
```

Out[16]:

	True 0	True 1	True 2	True 3	True 4	Assign to	Error rate
Cluster 0	347	39	494	5130	3	Digit 3	0.146848
Cluster 1	109	13	327	147	5558	Digit 4	0.096848
Cluster 2	23	6608	668	390	235	Digit 1	0.166078
Cluster 3	220	82	4399	427	37	Digit 2	0.148306
Cluster 4	5224	0	70	37	9	Digit 0	0.021723

In [17]:
```
# EM for Gaussian mixtures using initialization 1
np.random.seed(263)

em_kmeans = GaussianMixture(n_components=5, init_params="kmeans", n_init
=1, random_state=0).fit(pixel_X_transform)

# Table 1
em_kmeans.labels_ = em_kmeans.predict(pixel_X_transform)
crosstab_with_error_rate(em_kmeans.labels_, labels)
```

Out[17]:

	True 0	True 1	True 2	True 3	True 4	Assign to	Error rate
Cluster 0	6	0	8	5341	0	Digit 3	0.002614
Cluster 1	4	1043	15	33	4812	Digit 4	0.185373
Cluster 2	0	4513	0	1	1	Digit 1	0.000443
Cluster 3	633	1186	5932	752	1028	Digit 2	0.377610
Cluster 4	5280	0	3	4	1	Digit 0	0.001513

Using the first initialization, the error rates for the *k*-means algorithm did not change too much from Question 2.1). However, the error rates for the EM algorithm decreased by a significant amount, and the runtime was reduced as well. Therefore, dimensionality reduction was a good call.

Question 3)

In [18]:
```
# Load data
data = pd.read_csv("baseball.csv")

# Include only data from before 2002
train1 = data[data.Year < 2002]
```

In [19]:
```
# RS ~ SLG + OBP
X1 = np.array(train1[["SLG", "OBP"]])
y1 = np.array(train1["RS"])
regressor1 = GradientBoostingRegressor(random_state=0).fit(X1, y1)
```

In [20]:
```
# Drop NAs
train2 = train1.dropna(subset=["OOBP", "OSLG"])

# RA ~ OOBP + OSLG
X2 = np.array(train2[["OSLG", "OOBP"]])
y2 = np.array(train2["RA"])

regressor2 = GradientBoostingRegressor(random_state=0).fit(X2, y2)
```

In [21]:
```
# Create RD
train1["RD"] = train1["RS"] - train1["RA"]

# W ~ RD
X3 = np.array(train1[["RD"]])
y3 = np.array(train1["W"])

regressor3 = GradientBoostingRegressor(random_state=0).fit(X3, y3)
```

```
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:2: Set
tingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

In [22]:
```
# Use these statistics to predict RS, RA, and W
SLG = 0.430
OBP = 0.349
OOBP = 0.307
OSLG = 0.373

RS_pred = regressor1.predict(np.array([[SLG, OBP]]))
RA_pred = regressor2.predict(np.array([[OSLG, OOBP]]))
RD_pred = RS_pred - RA_pred
W_pred = regressor3.predict(np.array([RD_pred]))

print("The gradient boosting model predicts", int(W_pred), "wins in 200
2.")
```

```
The gradient boosting model predicts 104 wins in 2002.
```

In [23]:
```
# True results
data[(data["Year"]==2002) & (data["Team"]=="OAK")]
```

Out[23]:

	Team	League	Year	RS	RA	W	OBP	SLG	BA	Playoffs	RankSeason	RankPlayo
320	OAK	AL	2002	800	654	103	0.339	0.432	0.261	1	1.0	

In 2002, Oakland won 103 games. The gradient boosting model predicted 104 wins, which is super close. On the other hand, the random forest from the previous homework predicted 98 wins. Therefore, the gradient boosting model performs much better than the random forest.