Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.036 Introduction to Machine Learning - Spring 2017

Project 3: Clustering Census Data

Issued: Monday, 17 April 2017. Due: Friday, 5 May 2017 at 9 pm

Introduction

For this project, you will be finding demographic clusters in census data by building a categorical mixture model and fitting it using the Expectation Maximization (EM) algorithm. Table 1 shows a couple of example entries from the dataset.

In Part 1, you will become familiar with the EM algorithm by modeling a toy dataset using a mixtures of Gaussians model, and comparing it to the K-means algorithm.

In Part 2, you will re-derive and implement the EM algorithm for a mixture model with categorical components. You will then use your model to answer several questions about US demographics.

Prerequisites: Python 3 with the `numpy`, `scipy`, `matplotlib`, and `pandas` packages.

`age`	`citizen`	`income`	`industry`	`jobhunt`
5	0	1	10	2
3	2	5	8	1

Table 1: Example entries from the census dataset. The data is categorical, so for example, `age = 3` means the person is between 20 and 29 years old. As another example, in the table above, the first person was between 40 and 49 years old, was born in the US, had negative income, and was looking for a job in public administration. The meaning of each number can be found in `categories.txt`.

1 K-Means Versus EM (45 Points)

1. (15 points) Your first task is to implement the K-means algorithm on a 2-dimensional toy dataset, which is loaded in `main.py` as `toy_data`. Try different numbers of clusters, with $K = [1, 2, 3, 4, 5]$, and plot the results with the provided function. Since the initialization is random, make sure you try each K value a few times to select the one that minimizes the total cost. Submit the plots of the best solution for each K.

Your next task is to implement the EM algorithm for Gaussian mixture models.

Recall the Gaussian mixture model presented in class, where a model is generated from the weighted sum of several Gaussians with unknown parameters. We can write this as:

$$P(x \mid \mu, \pi, \sigma^2) = \sum_{j=1}^{K} \pi_j N(x; \mu^{(j)}, \sigma_j^2 I)$$

In our notation $N(x; \mu^{(j)}, \sigma_j^2 I)$ denotes a normal distribution with mean $\mu^{(j)}$ and covariance matrix $\sigma_j^2 I$, and π_j denotes the mixing proportion or probability assigned to choosing each Gaussian.

The goal of the EM algorithm is to estimate these unknown parameters by maximizing the *log-likelihood* of the observed data $x^{(1)}, ..., x^{(n)}$:

$$\mathcal{L}(x^{(1)}, ..., x^{(n)}; \mu, \pi, \sigma^2) = \sum_{i=1}^{n} \log \left(\sum_{j=1}^{K} \pi_j N(x^{(i)}; \mu^{(j)}, \sigma^2 I) \right).$$

Starting with some initial guess of the unknown parameters, the algorithm iterates between two steps: *E*xpectation and *M*aximization. The E-step "softly" or fractionally assigns point $x^{(i)}$ to cluster j based on the posterior probability:

$$p(j \mid i) = \frac{\pi_j N(x^{(i)}; \mu^{(j)}, \sigma_j^2 I)}{\sum_{l=1}^{K} \pi_l N(x^{(i)}; \mu^{(l)}, \sigma_l^2 I)}$$

The M-step takes these soft assignments as given and finds new parameter values which maximize the log-likelihood of the weighted dataset. In particular, we update the following variables:

$$\begin{aligned} n_j &= \sum_{i=1}^{n} p(j \mid i), \\ \pi_j &= \frac{n_j}{n}, \\ \mu^{(j)} &= \frac{1}{n_j} \sum_{i=1}^{n} p(j \mid i) x^{(i)}, \\ \sigma_j^2 &= \frac{1}{2n_j} \sum_{i=1}^{n} p(j \mid i) \, || \, x^{(i)} - \mu^{(j)} ||^2 \end{aligned}$$

2. (20 points) Implement the EM algorithm for the Gaussian mixture model described above. To this end, expand the skeleton functions `e_step` and `m_step` found in the `GMM` class provided in `project3.py`. Note that the spec for these methods are in the `MixtureModel` class.
3. To confirm that your implementation is correct, run the test provided in `main.py`. You should also check that your log-likelihood values are monotonically increasing.
4. (5 points) Plot the results of your EM implementation for K = [1, 2, 3, 4, 5]. Since the EM algorithm can also get stuck in a local optimum, you should run the algorithm multiple times with different initializations for each K value. Select the solutions that achieve the highest log-likelihood and submit those plots in your write-up.
5. (5 points) Compare the best K-means and EM solutions for K = [1, 2, 3, 4, 5]. Briefly explain when, how, and why they differ.

2 Clustering Census Data (55 Points)

Now that you've warmed up on the toy data, it's time to use your EM skills on a real dataset! As previously mentioned, we will be learning from the 1990 US census dataset. Unfortunately, bit rot has corrupted several cells. Thus, unlike before when all the data were observed, we now have to take into account missing entries (or not, as the case may be). Appendix A.1 provides an example of how missing data are represented.

Since our goal is to infer demographic clusters from categorical data, we will fit a mixture of categorical distributions. For instance, if the features are age and income, one cluster may assign high probability to young people with low income and another may prefer middle-aged people with high income.

First, let us define some variables and notation:

D **-** the number of features in the dataset. In our case, a feature is a census item like age or citizenship.

K **-** the number of mixture components

π **-** a K-vector of mixing proportions. $\sum_{k=1}^{K} \pi_k = 1$

α **-** a list of length K containing the parameters for each cluster's d categorical distributions. $\alpha_{k,d}$ is a vector that is as long as the number of categories associated with field d. Each $\alpha_{k,d}$ sums to 1. For instance, permissible values for "sex" are male and female, so d for this feature is 2.

$x_d^{(i)}$ **-** the categorical variable for the d^{th} field of data point i

$z^{(i)}$ **-** a categorical variable indicating the mixture component from which $x^{(i)}$ was drawn. This can be thought of as the demographic that this person is part of.

$$\mathcal{L}(\pi, \alpha \mid X, z) = p(X, \mathbf{z} \mid \pi, \alpha) = \prod_{i=1}^{n} p(z^{(i)} \mid \pi) \prod_{d=1}^{D} p(x_d^{(i)} \mid \alpha_{z^{(i)},d})^{[\![x_d^{(i)} \text{ is not missing}]\!]} \tag{1}$$

$$z^{(i)} \sim \text{Cat}(\pi)$$
$$x_d^{(i)} \sim \text{Cat}(\alpha_{z^{(i)},d})$$

1. (2 points) In our model, we have assumed that the features are independent, which is not realistic (for instance, age and income are highly dependent). Explain why this will cause the model to produce overconfident cluster assignments. (Hint: think about how the distance between clusters changes as redundant features are added)

   Having made the independence assumption, show that the simple $(\ldots)^{[\![x_d^{(i)} \text{ is not missing}]\!]}$ expression can be obtained from the fully-observed model by marginalizing out unknown values.

   Having done this, you should be able to appreciate why this approximation is helpful!

2. (4 points) (E-step) Derive the posterior distribution for the latent variable associated to a particular data point, $p(z^{(i)} \mid x^{(i)}, \pi, \alpha)$. Your answer should be in terms of π, α, and $x^{(i)}$.

   With the posterior in hand, we can calculate the expected complete log-likelihood. Note that we work in the log-domain because of its numerical stability and convenience; this is valid since we only want to maximize $E_{\mathbf{z}|X,\alpha,\pi}[L(\alpha, \pi \mid X, \mathbf{z})]$ and log is a monotonic function. The complete calculation is as follows:

$$
\begin{aligned}
J(\alpha, \pi) &= E_{\mathbf{z}|X}[\log L(\alpha, \pi \mid X, \mathbf{z})] \\
&= \sum_{\mathbf{k}} p(\mathbf{z} = \mathbf{k} \mid X) \log p(X, \mathbf{z} \mid \alpha, \pi) \\
&= \sum_{i=1}^{n} \sum_{k=1}^{K} p(z^{(i)} = k \mid x^{(i)}) \log p(z^{(i)} = k \mid \pi) + \\
&\quad \sum_{i=1}^{n} \sum_{k=1}^{K} \sum_{d=1}^{D} p(z^{(i)} = k \mid x^{(i)}) \log p(x_d^{(i)} \mid z^{(i)} = k, \alpha, \pi)^{[\![x_d^{(i)} \text{ is not missing}]\!]}
\end{aligned}
$$

3. (4 points) (M-step) The goal is to now find the maximum likelihood estimates of the parameters π and α by taking $\arg\min_{\pi,\alpha} J(\alpha, \pi)$. This can be done by minimizing w.r.t. each parameter.
   (a) Write down the ML (maximum likelihood) estimate of π from the weighted data. You should find that it is analogous to the GMM case.
   (b) Perform a similar computation for the ML estimate of α. The update will be different from Part 1 since we're using categorical rather than Gaussian mixture components. As in part (a), your answer should make intuitive sense from a counting perspective.
4. (20 points) Having derived the updates for the EM algorithm, we can proceed to translate it into code. Before you begin, see Appendix A for implementation tips.
   (a) Implement `e_step` for the `CMM` class in `project3.py`.
   (b) Implement `m_step` for the `CMM` class in `project3.py`.
5. (5 points) After verifying your E and M steps by running `test_em_cmm`, uncomment the lines associated with this part and run your EM algorithm on the full census dataset. Given what you know about the EM algorithm, comment on the overall trend of the log-likelihood over the iterations.
6. (10 points) The final step in the training pipeline is to choose the model that best explains the data without overfitting to its peculiarities. We will do this in two ways:
   (a) Start by implementing the `bic` method of the `CMM` class. Note that when training has completed, `self.max_ll` and `self.n_train` will become available.
   (b) Uncomment and run the code in `main.py` that plots the maximized LLs and BICs for the trained models. Based on the graph of the LL, what value of K should you choose? What about for the BIC? Do both results agree?
7. (10 points) The model defined in Equation 1, trained on census data, may be interpreted as identifying K demographics that describe the population. Using the model you just learned, you can answer interesting questions about the data:
   (a) Call `print_clusters` with the model trained with the best choice of K, as determined by the BIC. Qualitatively describe the clusters that your model finds; include several of the key features and their values.

Next, call `print_clusters` with a model trained with a larger or smaller value of K. How do the clusters found by the second model compare to the first? You may want to discuss whether the cluster identities are stable and, if not, what concepts has the second model added or removed?

(b) Now we will explore the qualitative effects of different initializations. Run your best model several times and print the clusters. How stable are the discovered clusters? Are some found more often than others?

(c) Imagine that you trained your model on the census data for each state. How could you use your model to determine how similar two states' populations are?

Bear in mind that your model was trained with *no supervision whatsoever*. Despite having no notion of a person, an age, a country, etc., it was able to figure out what factors "go together" to make a demographic. Not bad! Comparing this to the last project, while you could train a simple neural model to generate the data, the results would not have the straightforward interpretation of probabilities parameterizing categorical distributions. An active area of research is in designing neural generative models that incorporate discrete values.

Submission Instructions

1. Save your writeup in the project 3 folder as `writeup.pdf`.
2. Run `packager.py` to produce `submit.zip`.
3. Upload `submit.zip` to LMOD.

We will not accept your submission in any other format. It is, therefore, strongly recommended that you submit your project in advance of the deadline just in case you run into difficulty.

A Implementation Notes

A.1 The `pandas` package

Pandas is a package for scientific computing in Python that builds upon Numpy to provide a convenient way to manipulate tabular data (via the `DataFrame` and `Series` types).

For this project, you can generally treat the `data` (a Pandas `DataFrame`) as a Numpy matrix. The main difference, however, is in indexing:

```
import numpy as np, pandas as pd
arr = np.random.randn(4, 2)
df = pd.DataFrame(arr)
assert (arr.mean(1) == df.mean(1)).all()    # most numpy ops exist in pandas
assert (arr[:, 0] == df.iloc[:, 0]).all()  # iloc provides a positional indexer
```

The main reason why we're using Pandas is that it gracefully handles missing values, which is relevant in Part 2. Consider the following code example:

```
import numpy as np, pandas as pd
N = 10
data = pd.Series(np.random.randint(4, size=N))  # a particular feature
data[1::4] = np.nan                             # add missing values
zs = np.random.dirichlet([1]*N)                 # soft cluster assignments
ex_counts = (data * zs).sum()
assert not pd.isnull(ex_counts)                 # NaNs are ignored!
```

A.2 General Tips

- Your code will run very slowly if you use too many loops. Most of the operations can and should be vectorized, as in the previous projects.
- To suppress log-likelihood printouts when fitting the mixture models, call `fit` with `verbose=False`.
- By default, trained models (and their maximized LLs) are saved to the `models` directory. This allows you to not have to retrain the models when choosing K or computing the BIC.

A.3 Part 1 Hints

- Part 1 can be done entirely in Numpy - no Pandas necessary!
- When calculating Gaussian probabilities for the E-step, you may want to look at `scipy.stats.{norm, multivariate_normal}`.

A.4 Part 2 Hints

- `alpha` is a list of Numpy arrays in which each entry i corresponds to the parameters of the k categorical distributions for a particular feature.
  `type(alpha[i]) == np.ndarray` and `alpha[i].shape == (k, ds[i])`
- If you are using more than one loop over the entries of `alpha`, you are using too many loops.
- You may find the `pd.get_dummies` function useful for converting categorical values into a form suitable for efficient selection via matrix-matrix multiplication.
- When debugging your algorithm, you may wish to run on a smaller version of the dataset. To do this, run the following command in your terminal and appropriately modify `main.py`.

```
head -n 1000 census_data.csv > tiny_data.csv
```