Edward: Unsupervised Learning of Finite Mixture Models

Michael L. Thompson

May 8, 2017, rev. Sept. 9, 2017

Contents

Preface

This is an R implementation of the Edward tutorial "Unsupervised Learning" (jupyter notebook).

See the Appendix for instructions on setting up R to run Python, TensorFlow, and Edward.

Michael L. Thompson

Setup

Load the R packages we need.
Note that in all that follows, the R code explicitly specifies integer constants as being long integers by appending "`L`" to the number. This is necessary for proper interfacing with Python using the `reticulate` package in R.

```
library( magrittr )
library( tidyverse )
library( reticulate )
library( tensorflow )

library( mvtnorm )

use_virtualenv("tensorflow")
```

Unsupervised Learning

In unsupervised learning, the task is to infer hidden structure from unlabeled data, comprised of training examples $\{x_n\}$.

We demonstrate with an example in Edward. A webpage version is available at http://edwardlib.org/tutorials/unsupervised.

Load the Python modules we need. Be sure that each of Python modules has first been installed (e.g., using `pip3` at the operating system/shell command prompt).

```
ed   <- import( module = "edward" )
np   <- import( module = "numpy" )
six  <- import( module = "six" )

# Assign the Edward models used in the script.
#from edward.models import
Categorical  <- ed$models$Categorical
Dirichlet    <- ed$models$Dirichlet
Empirical    <- ed$models$Empirical
InverseGamma <- ed$models$InverseGamma
Normal       <- ed$models$Normal
MultivariateNormalDiag <- ed$models$MultivariateNormalDiag
ParamMixture <- ed$models$ParamMixture
```

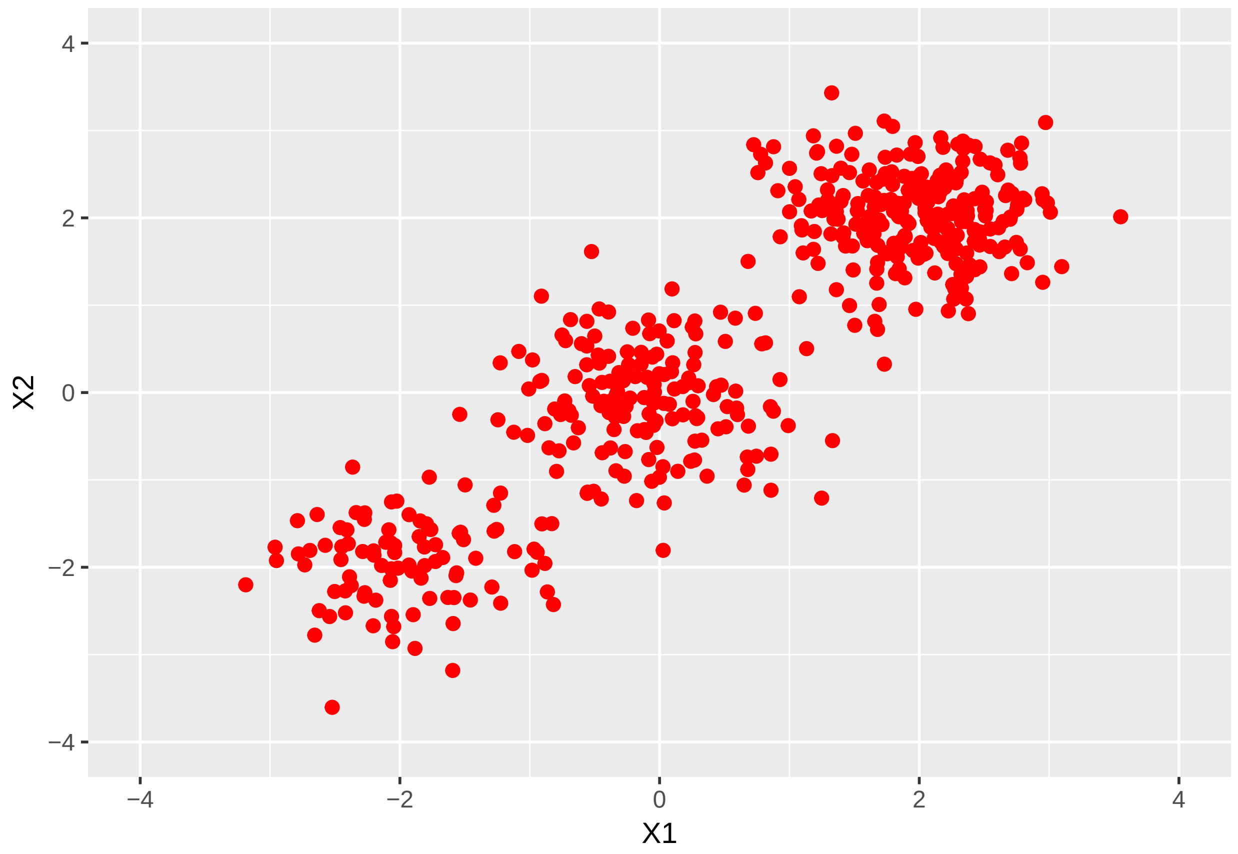
Data

Use a simulated data set of 2-dimensional data points $\mathbf{x}_n \in \mathbb{R}^2$.

```
build_toy_dataset <- function(
  N    = 500L,
  D    = 2L,
  K    = 2L,
  p_k  = seq_len(K) %>% {(.)/sum(.)},
  mus  = t( sapply( 2*seq_len(K) - K - 1, rep, D ) ),
  stds = matrix( 0.3, nrow = K, ncol = D )
){
  # p_k  <- np$array( c(0.4, 0.6) )
  # mus  <- list(c(1, 1), c(-1, -1))
  # stds <- matrix(rep(0.1,4), nrow=2)
  # x    <- np$zeros(c(N, 2L), dtype=np$float32)
  # for( n in seq_len(N)){
  #   k      <- np$argmax(np$random$multinomial(1, p_k))
  #   x[n, ] <- np$random$multivariate_normal(mus[[k]], np$diag(stds[k,]))
  # }

  k_i  <- sample.int(length(p_k), N, prob = p_k, replace = TRUE)
  n_k  <- table( k_i )

  x    <- do.call(
    rbind,
    lapply(
      seq_along(p_k),
      function(k) {
        rmvnorm( n_k[k], mean = mus[k,], sigma = diag( stds[k,] ))
      }
    )
  )
  return( x )
}

N <- 500L  # number of data points
K <- 3L  # number of components
D <- 2L  # dimensionality of data

tf$reset_default_graph()
ed$set_seed( 42L )

x_train <- build_toy_dataset( N = N, K = K, D = D )
```

We visualize the generated data points.

```
x_train %>% as_data_frame() %>% setNames( c("X1", "X2" ) ) %>%
{
  ggplot(.,aes(x=X1,y=X2)) +
    geom_point(fill='red',color='red',shape=21,size=2) +
    lims( x= (K+1)*c(-1,1), y= (K+1)*c(-1,1) )
} %>% print()
```



Model

A mixture model is a model typically used for clustering. It assigns a mixture component to each data point, and this mixture component determines the distribution that the data point is generated from. A mixture of Gaussians uses Gaussian distributions to generate this data (Bishop, 2006).

For a set of N data points, the likelihood of each observation $\mathbf{x}_n$ is

$$p(\mathbf{x}_n \mid \pi, \mu, \sigma) = \sum_{k=1}^{K} \pi_k \, \text{Normal}(\mathbf{x}_n \mid \mu_k, \sigma_k).$$

The latent variable π is a K-dimensional probability vector which mixes individual Gaussian distributions, each characterized by a mean μ_k and standard deviation σ_k.

Define the prior on $\pi \in [0, 1]$ such that $\sum_{k=1}^{K} \pi_k = 1$ to be

$$p(\pi) = \text{Dirichlet}(\pi \mid \alpha \mathbf{1}_K)$$

for fixed $\alpha = 1$. Define the prior on each component $\mu_k \in \mathbb{R}^D$ to be

$$p(\mu_k) = \text{Normal}(\mu_k \mid \mathbf{0}, \mathbf{I}).$$

Define the prior on each component $\sigma_k^2 \in \mathbb{R}^D$ to be

$$p(\sigma_k^2) = \text{InverseGamma}(\sigma_k^2 \mid a, b).$$

We build two versions of the model in Edward: one jointly with the mixture assignments $c_n \in \{0, \ldots, K-1\}$ as latent variables, and another with them summed out.

The joint version includes an explicit latent variable for the mixture assignments. We implement this with the `ParamMixture` random variable; it takes as input the mixing probabilities, the components' parameters, and the distribution of the components. It is the distribution of the mixture conditional on mixture assignments. (Note we can also write this separately by first building a `Categorical` random variable for `z` and then building `x`; `ParamMixture` avoids requiring `tf.gather` which is slightly more efficient.)

```
p_k     <- Dirichlet( tf$ones( K ) )
mu      <- Normal(tf$zeros(D), tf$ones(D), sample_shape=K)
sigmasq <- InverseGamma(tf$ones(D), tf$ones(D), sample_shape=K)
x       <- ParamMixture(
  p_k,
  dict('loc' = mu, 'scale_diag' = tf$sqrt(sigmasq) ),
  MultivariateNormalDiag,
  sample_shape = N
)
z       <- x$cat
```

The collapsed version marginalizes out the mixture assignments. We implement this with the `Mixture` random variable; it takes as input a Categorical distribution and a list of individual distribution components. It is the distribution of the mixture summing out the mixture assignments.

```
""""
pi = Dirichlet(tf.ones(K))
mu = Normal(tf.zeros(D), tf.ones(D), sample_shape=K)
sigma = InverseGamma(tf.ones(D), tf.ones(D), sample_shape=K)
cat = Categorical(probs=pi, sample_shape=N)
components = [
    MultivariateNormalDiag(mu[k], sigma[k], sample_shape=N)
    for k in range(K)]
x = Mixture(cat=cat, components=components)
""""
```

We will use the joint version in this analysis.

Inference

Each distribution in the model is written with conjugate priors, so we can use Gibbs sampling. It performs Markov chain Monte Carlo by iterating over draws from the complete conditionals of each distribution, i.e., each distribution conditional on a previously drawn value. First we set up Empirical random variables which will approximate the posteriors using the collection of samples.

```
J   <- 500L  # number of MCMC samples
qpi      <- Empirical( tf$Variable( tf$ones(  c(J, K) ) / (1.0*K) ) )
qmu      <- Empirical( tf$Variable( tf$zeros( c(J, K, D) ) ) )
qsigmasq <- Empirical( tf$Variable( tf$ones(  c(J, K, D) ) ) )
qz       <- Empirical( tf$Variable( tf$zeros( c(J, N), dtype=tf$int32 ) ) )
```

Run Gibbs sampling. We write the training loop explicitly, so that we can track the cluster means as the sampler progresses.

```
inference <- ed$Gibbs(
  dict( p_k = qpi, mu = qmu, sigmasq = qsigmasq, z = qz ),
  data = dict( x = x_train )
)
inference$initialize( n_print = 50L )

sess = ed$get_session()
tf$global_variables_initializer()$run()

t_ph = tf$placeholder(tf$int32, c())
running_cluster_means = tf$reduce_mean(
  tf$slice( qmu$params, begin = rep(0L,3), size = c(t_ph,K,D)),
  0L
)

for( i in seq_len(inference$n_iter)){
  info_dict = inference$update()
  #inference$print_progress(info_dict)
  t = info_dict[['t']]
  if (t %% inference$n_print == 0){
    cat( sprintf( "\nIteration %d, Inferred cluster means:\n", t ) )
    running_cluster_means %>%
      sess$run( dict( t_ph = t - 1) ) %>%
      apply(
        1,
        function(x) paste(sprintf("\t%.2f",x),collapse="")
      ) %>%
      paste( collapse="\n" ) %>%
      cat()
  }
}
```

```
##
## Iteration 50, Inferred cluster means:
##  -0.13   -0.11
##  -1.55   -1.59
##  1.89    1.97
## Iteration 100, Inferred cluster means:
##  -0.11   -0.09
##  -1.75   -1.77
##  1.93    2.01
## Iteration 150, Inferred cluster means:
##  -0.11   -0.08
##  -1.82   -1.83
##  1.94    2.02
## Iteration 200, Inferred cluster means:
##  -0.10   -0.09
##  -1.85   -1.86
##  1.95    2.03
## Iteration 250, Inferred cluster means:
##  -0.10   -0.08
##  -1.87   -1.88
##  1.95    2.03
## Iteration 300, Inferred cluster means:
##  -0.10   -0.08
##  -1.88   -1.89
##  1.95    2.03
## Iteration 350, Inferred cluster means:
##  -0.10   -0.08
##  -1.89   -1.90
##  1.95    2.04
## Iteration 400, Inferred cluster means:
##  -0.10   -0.08
##  -1.90   -1.90
##  1.95    2.04
## Iteration 450, Inferred cluster means:
##  -0.10   -0.08
##  -1.90   -1.91
##  1.95    2.04
## Iteration 500, Inferred cluster means:
##  -0.10   -0.08
##  -1.91   -1.91
##  1.96    2.04
```

Criticism

We visualize the predicted memberships of each data point. We pick the cluster assignment which produces the highest posterior predictive density for each data point.

To do this, we first draw a sample from the posterior and calculate a a $N \times K$ matrix of log-likelihoods, one for each data point $\mathbf{x}_n$ and cluster assignment k. We perform this averaged over 100 posterior samples.

```
# Calculate likelihood for each data point and cluster assignment,
# averaged over many posterior samples. ``x_post`` has shape (N, 100, K, D).
n_sample       <- 100L
mu_sample      <- qmu$sample( n_sample )
sigmasq_sample <- qsigmasq$sample( n_sample )
x_post         <- Normal(
  loc   = tf$ones(list(N, 1L, 1L, 1L)) * mu_sample,
  scale = tf$ones(list(N, 1L, 1L, 1L)) * tf$sqrt(sigmasq_sample)
)
```

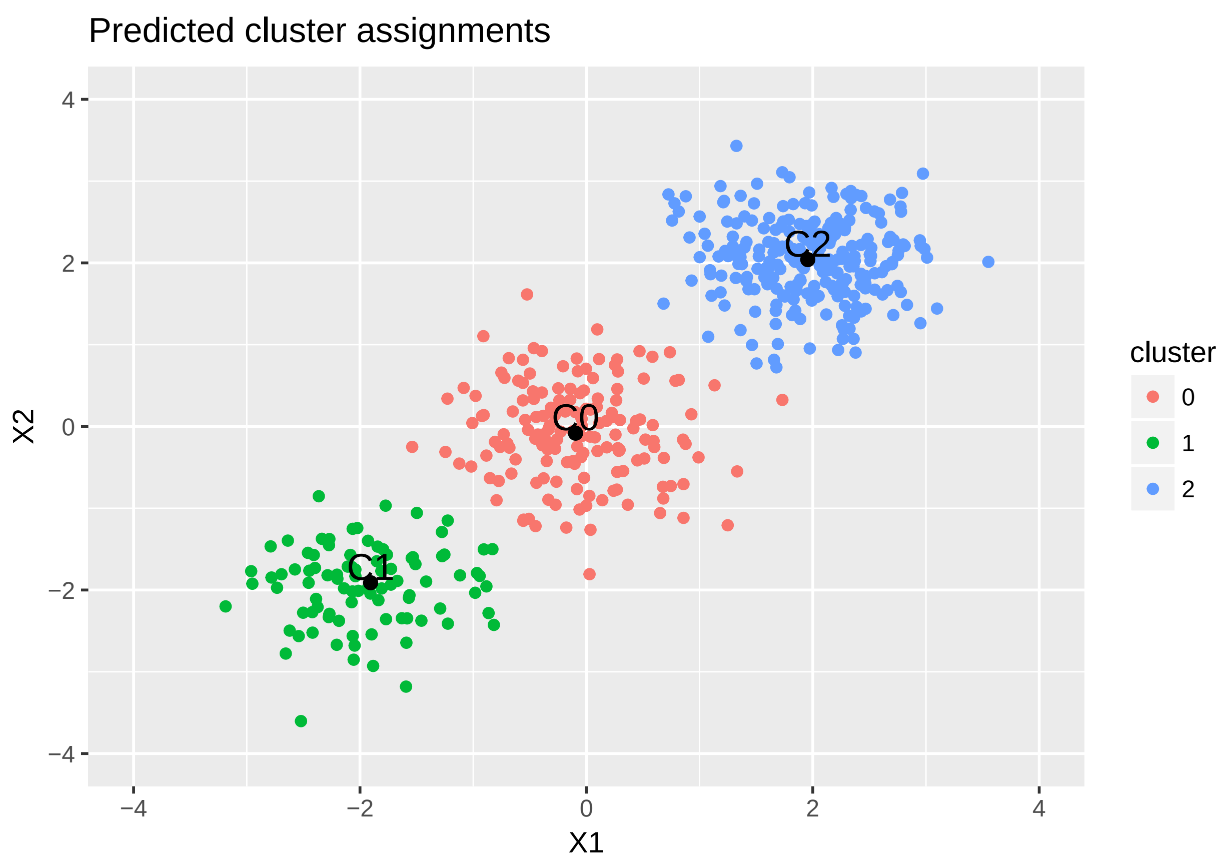
```r
x_broadcasted = tf$tile(
  tf$reshape(
    tf$cast(x_train,dtype = tf$float32),
    list(N, 1L, 1L, D)
  ),
  list(1L, n_sample, K, 1L)
)

# Sum over latent dimension, then average over posterior samples.
# ``log_liks`` ends up with shape (N, K).
log_liks <- x_post$log_prob( x_broadcasted )
log_liks <- tf$reduce_sum( log_liks, 3L )
log_liks <- tf$reduce_mean( log_liks, 1L )
```

We then take the arg max along the columns (cluster assignments).

```r
# Choose the cluster with the highest likelihood for each data point.
clusters = tf$argmax(log_liks, 1L)$eval()
```

Plot the data points, colored by their predicted membership.

```r
df_means <- running_cluster_means$eval(dict('t_ph'=t)) %>%
  as_tibble() %>%
  setNames(sprintf("X%d",seq_len(D))) %>%
  mutate(cluster=factor(0:(K-1)), c_name = sprintf("C%s",as.character(cluster)))

x_train %>%
  as_tibble() %>%
  setNames( sprintf("X%d", seq_len(D) ) ) %>%
  mutate( cluster = factor( clusters ) ) %>%
  {
    ggplot(., aes(x = X1, y = X2, color = cluster )) +
      geom_point() +
      lims( x = (K+1)*c(-1,1), y = (K+1)*c(-1,1) ) +
      ggtitle("Predicted cluster assignments") +
      geom_text(
        data      = df_means,
        aes(label = c_name),
        color     = 'black',
        size      = 5,
        nudge_y   = 0.2
      ) +
      geom_point(
        data  = df_means,
        color = 'black',
        size  = 2
      )
  } %>%
  print()
```



The model has correctly clustered the data.

```
sess$close()
```

Remarks: The log-sum-exp trick

For a collapsed mixture model, implementing the log density can be tricky. In general, the log density is

$$\log p(\pi) + \left[\sum_{k=1}^{K} \log p(\mu_k) + \log p(\sigma_k)\right] + \sum_{n=1}^{N} \log p(\mathbf{x}_n \mid \pi, \mu, \sigma),$$

where the likelihood is

$$\sum_{n=1}^{N} \log p(\mathbf{x}_n \mid \pi, \mu, \sigma) = \sum_{n=1}^{N} \log \sum_{k=1}^{K} \pi_k \operatorname{Normal}(\mathbf{x}_n \mid \mu_k, \sigma_k).$$

To prevent numerical instability, we'd like to work on the log-scale,

$$\sum_{n=1}^{N} \log p(\mathbf{x}_n \mid \pi, \mu, \sigma) = \sum_{n=1}^{N} \log \sum_{k=1}^{K} \exp\Big(\log \pi_k + \log \operatorname{Normal}(\mathbf{x}_n \mid \mu_k, \sigma_k)\Big).$$

This expression involves a log sum exp operation, which is numerically unstable as exponentiation will often lead to one value dominating the rest. Therefore we use the log-sum-exp trick. It is based on the identity

$$\begin{aligned}\mathbf{x}_{\max} &= \arg\max \mathbf{x},\\ \log \sum_i \exp(\mathbf{x}_i) &= \log \Big(\exp(\mathbf{x}_{\max}) \sum_i \exp(\mathbf{x}_i - \mathbf{x}_{\max}) \Big)\\ &= \mathbf{x}_{\max} + \log \sum_i \exp(\mathbf{x}_i - \mathbf{x}_{\max}).\end{aligned}$$

Subtracting the maximum value before taking the log-sum-exp leads to more numerically stable output. The `Mixture` random variable implements this trick for calculating the log-density.

Appendix

Assuming R and RStudio already installed, this implementation (under Windows 10) was made possible by installing Python and the TensorFlow and Edward modules, and then the `reticulate` and `tensorflow` packages in R, per the instructions at the following links:

- Python setup:
  - Python: Python, Download the latest version for Windows
  - TensorFlow: Installing TensorFlow on Windows
  - Edward: Edward, Getting Started
  - Also `pip`-ed `numpy` and `jupyter`.
- R setup:
  - Package `reticulate`: R Interface to Python
  - Package `tensorflow`: TensorFlow for R

(*This notebook also uses R packages `magrittr` & `tidyverse`, which are my go-to tools for data science in R.*)

Michael L. Thompson