STA414/2104: Practice Problems 6

March, 2018

These practice problems are not for credit. Students may complete independently, in groups, or however they like. Treat these as representative of what might be on the tests.

Questions involving coding may be completed in any language. If you would like help regarding your language of choice from the course team, use R or Python. There will not be any *code*-related questions on the tests, but you will be asked about comuptational algorithms.

1. For the random effects model described in lecture,

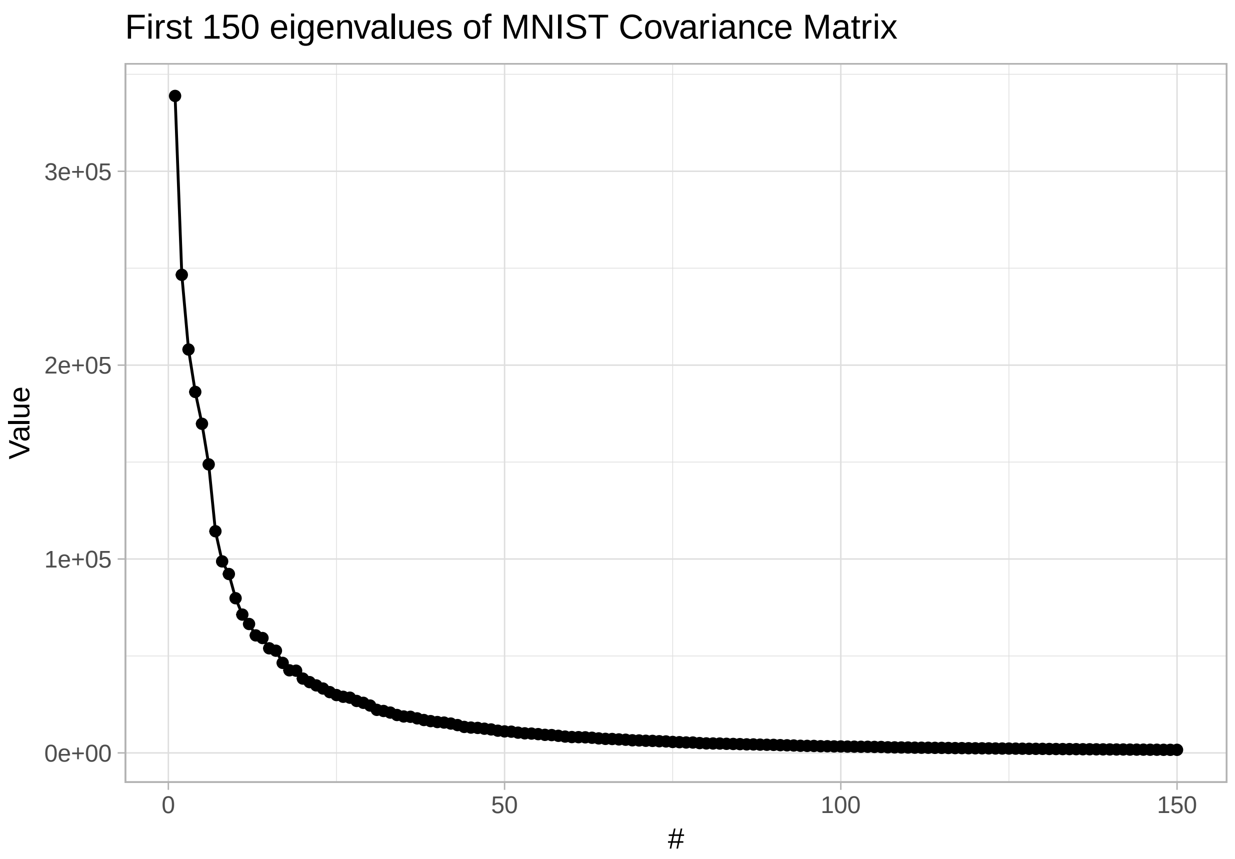
$$\begin{aligned} t_{kt} &= \mathbf{x}_{kt}'\mathbf{w} + b_k + \epsilon_{kt} \\ \mathbf{b} &= (b_1, \ldots, b_K) \sim N(\mathbf{0}, \sigma_b^2 \mathbf{I}) \\ \boldsymbol{\epsilon} &= (\epsilon_{11}, \ldots, \epsilon_{KT}) \sim N(\mathbf{0}, \sigma_\epsilon^2 \mathbf{I}) \\ \mathbf{b} &\perp \boldsymbol{\epsilon} \end{aligned}$$

   (a) Show $Cov(t_{ki}, t_{kj}) = \sigma_b^2$ using the rules for covariance directly (see a few slides back where we did this for the uncorrlated linear model).

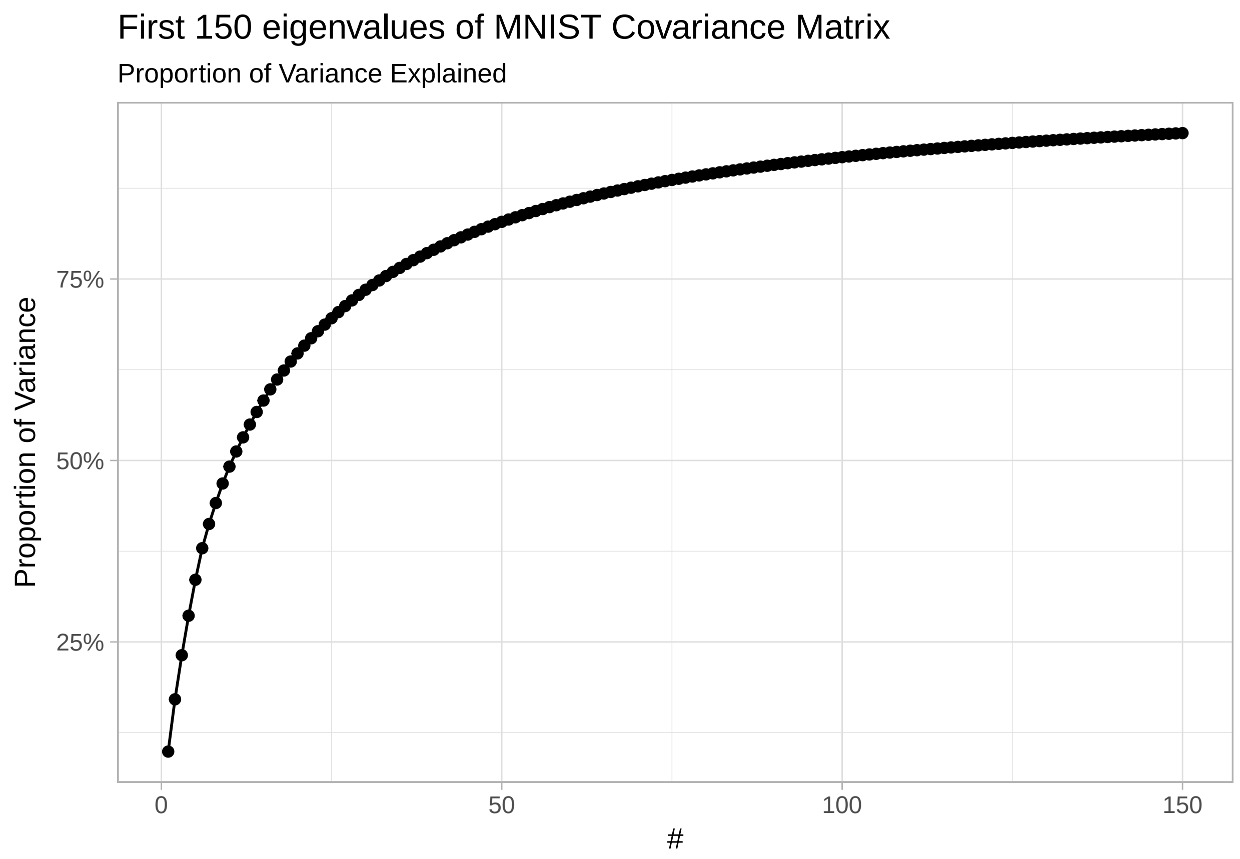
   (b) Derive an explicit formula for Σ_k as stated in lecture (note, the answer doesn't actually depend on k). Verify that this agrees with your answer to part (a).

2. Consider the PCA demo discussed in class. My code for that is below. I actually lost the code I used to read in the MNIST dataset, which is super embarassing since I am usually good about following best practices when it comes to documentation and keeping of code. Anyways, I will show you how to do the importing in a couple lectures- for now, just load the `mnist.Rdata` file posted on the course website, which will load the `mnist_save` object into the global environment.

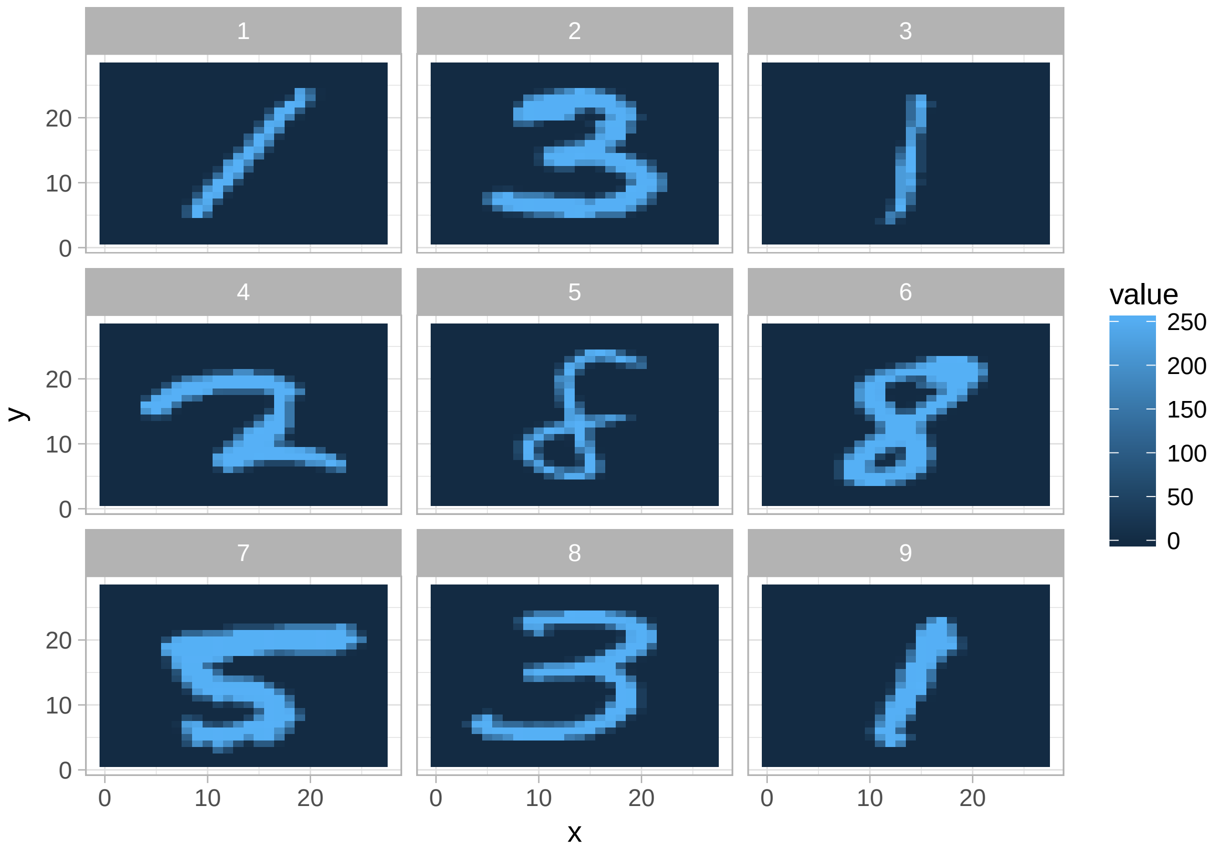
```r
suppressMessages({
  suppressWarnings({
    library(dplyr)
    library(tidyr)
    library(ggplot2)
  })
})
# Load the MNIST data
load(file="mnist.Rdata")

# Recreate those eigenvalue plots- this is just for your own interest
eigenvalues <- eigen(cov(mnist_save))$values

data_frame(x = 1:length(eigenvalues),y = eigenvalues) %>%
  filter(x <= 150) %>%
  ggplot(aes(x=x,y=y)) +
  theme_light() +
  geom_point(colour="black") +
  geom_line()  +
  labs(title="First 150 eigenvalues of MNIST Covariance Matrix",
       x="#",
       y="Value")
```



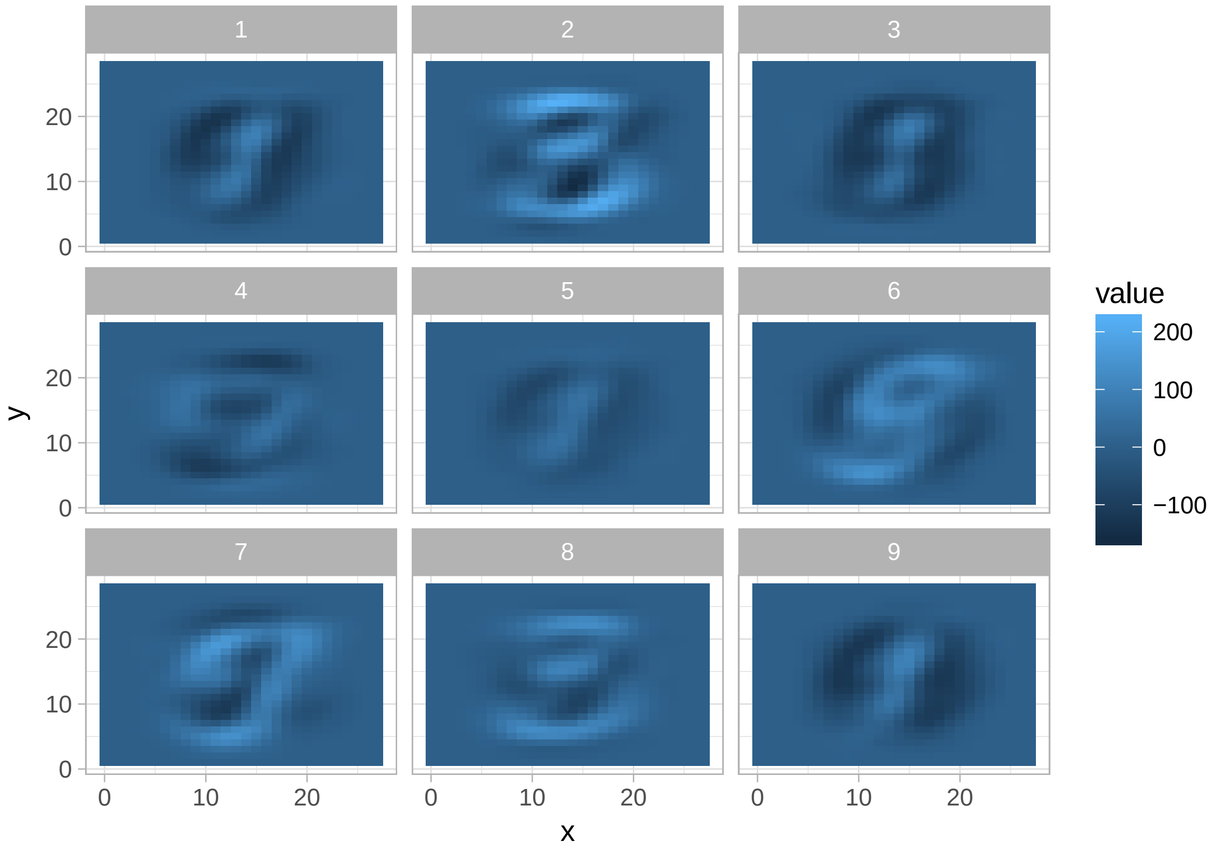
```
data_frame(x = 1:length(eigenvalues),y = eigenvalues,prop=cumsum(y)/sum(y)) %>%
  filter(x <= 150) %>%
  ggplot(aes(x=x,y=prop)) +
  theme_light() +
  geom_point(colour="black") +
  geom_line() +
  labs(title="First 150 eigenvalues of MNIST Covariance Matrix",
       subtitle="Proportion of Variance Explained",
       x="#",
       y="Proportion of Variance") +
  scale_y_continuous(labels=scales::percent_format())
```



```r
# Perform the PCA, and reconstruct the original data matrix using only 5 components
pcaobject <- prcomp(mnist_save)
pcarecon <- pcaobject$x[ ,1:5] %*% t(pcaobject$rotation[ ,1:5])

# Plot the original
pixels_gathered <- mnist_save %>%
  mutate(instance = row_number()) %>%
  filter(instance <= 9) %>%
  gather(pixel, value, -instance) %>%
  tidyr::extract(pixel, "pixel", "(\\d+)", convert = TRUE) %>%
  mutate(pixel = pixel - 2,
         x = pixel %% 28,
         y = 28 - pixel %/% 28)



pixels_gathered %>%
  ggplot(aes(x=x,y=y,fill=value)) +
  theme_light() +
  geom_tile() +
  facet_wrap(~instance)
```



```
# Plot the reconstruction
pixels_gathered <- as_data_frame(pcarecon) %>%
  mutate(instance = row_number()) %>%
  filter(instance <= 9) %>%
  gather(pixel, value, -instance) %>%
  tidyr::extract(pixel, "pixel", "(\\d+)", convert = TRUE) %>%
  mutate(pixel = pixel - 2,
         x = pixel %% 28,
         y = 28 - pixel %/% 28)



pixels_gathered %>%
  ggplot(aes(x=x,y=y,fill=value)) +
  theme_light() +
  geom_tile() +
facet_wrap(~instance)
```



(a) Implement PCA from scratch. I used the `princomp` function in `R`; you can either do it from scratch on your own based on the lecture slides, or look up the documentation for the `princomp` function.

(b) Solve the optimization problem presented in lecture:

$$\mathbf{u}_j = \text{argmax } Var(\mathbf{u}'\mathbf{x}) = \text{argmax } \mathbf{u}'\mathbf{S}\mathbf{u}$$

$$\text{subject to } \mathbf{u}'\mathbf{u} = 1$$

$$\text{and } \mathbf{u} \perp \mathbf{u}_k \text{ for } k < j$$

where

$$\mathbf{S} = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{x}_i - \bar{\mathbf{x}})'$$

You may use Lagrange multipliers to incorporate the constraints. Recall that $\mathbf{u} \perp \mathbf{v}$ means that $\mathbf{u}'\mathbf{v} = 0$.

(c) Explain why the quantity

$$\frac{\sum_{i=1}^{d} \lambda_i}{tr(\mathbf{S})}$$

is referred to as the “proportion of variance explained” by the first d principal components.

3. Consider the Neural Network code example discussed in class. For the below questions, I would be very interested if you brought your results to office hours or to class, and we can discuss.
   (a) Run my code for $1,000$ iterations. This took about 20 minutes on my 2015 macbook pro, so go make yourself some tea. When you're done,
      (i) Flatten the final parameters into a vector (`np.concatenate` will help you), and then compute its L_2 norm (square root of the sum of the squared values). You won't be able to interpret this number very well, but just make sure it's not ridiculously high (when I did it, I got about 157.4)
      (ii) Make a histogram of the weights. Discuss what you see- are most of them near zero? Are they bell curve shaped, skewed, or some other shape? Multimodal? What is the variance?
   (b) Implement L_2-regularlization on the full parameter vector in the loss function. For a reasonable choice of λ (we have discussed this previously), re-run the code, and re-do the previous question. How does regularlization change the results?
   (c) Modify my code to use *stochastic gradient descent.* You're going to notice a trade-off between speed and stability as you adjust the size of the minibatches. Are you able to make the minibatches small enough that you can run the algorithm on the full $60,000$-image dataset?
   (d) Now for the hardest part: I used autograd to compute the gradient. You should implement it directly. We did most of this in class, so code up your formulas and use them to write your own gradient function. I think the hardest part is keeping the weights structured in a logical way, and reconciling this with the vector-based notation used in class. Discuss
      (i) How computing the gradient manually changes the stability of the gradient descent algorithm (it shouldn't)
      (ii) How computing the gradient manually changes the performance (time per iteration) of the algorithm. It should be much slower than using Autograd. If you show me your results, I can help explain why I think this is the case- but I might be wrong, so you have to show me your results first :)