MA5832 Assessment 1

Nikki Fitzherbert 13848336

17/05/2020

1 Implementation in R

Question 1 Part A

Let λ be the average number of people waiting in the queue and x the actual number of people waiting in the queue.
Then the probability density function is: $f(x) = \frac{1}{x!}\lambda^x e^{-\lambda}$
And the corresponding cumulative probability function is:

$$F(x;\lambda) = \frac{1}{x!}\lambda^x e^{-\lambda} = p(X < x) = \sum_{y=0}^{x} \frac{1}{y!}\lambda^y e^{-\lambda}$$

Therefore,

$$\begin{aligned}
p(X \geq x;\lambda) &= 1 - p(X < x;\lambda) \\
p(X \geq 8;3) &= 1 - p(X < 8;3) \\
&= 1 - [p(x=0) + p(x=1) + \cdots + p(x=8)] \\
&= 1 - \left[\left(\frac{1}{0!}3^0 e^{-0}\right) + \left(\frac{1}{1!}3^1 e^{-1}\right) + \cdots + \left(\frac{1}{8!}3^8 e^{-8}\right)\right] \\
&= 0.01191 \\
&\approx 1.19\%
\end{aligned}$$

Question 1 Part B

If it is assumed that the number of international students in MA5832 (X) follows a binomial distribution where p equals the percentage of international students and n is the total number of students in MA5832, then the probability density function is:

$$f(x) = \binom{n}{k} p^k (1-p)^{n-k}$$

And the corresponding cumulative probability function is:

$$F(k;n,p) = p(X \leq k) = \sum_{i=0}^{\lfloor k \rfloor} \binom{n}{i} p^i (1-p)^{n-i}$$

Therefore,

$$\begin{aligned}
p(X \leq 5; 20, 0.15) &= \sum_{i=0}^{\lfloor 5 \rfloor} \binom{20}{i} 0.15^i (1-0.15)^{20-i} \\
&= [p(k=0) + p(k=1) + \cdots + p(k=5)] \\
&= \left[\left(\binom{20}{0} \times 0.15^0 \times (1-0.15)^{(20-0)} \right) + \cdots + \left(\binom{20}{0} \times 0.15^5 \times (1-0.15)^{(20-5} \right. \right. \\
&= \left[\left(\binom{20}{0} \times 0.15^0 \times 0.85^{20} \right) + \cdots + \left(\binom{20}{0} \times 0.15^5 \times 0.85^{15} \right) \right] \\
&= 0.932692 \\
&\approx 93.27\%
\end{aligned}$$

Question 2 Part A

If the optimal solution of β has the following form: $\hat{\beta} = (X'X)^{-1}X'Y$, then $\hat{\beta}$ for the `marketing` dataset in the `datarium` package can be calculated using R as follows:

```
X <- cbind(1, marketing$youtube, marketing$newspaper, marketing$facebook)
Y <- marketing$sales

beta_hat <- solve(t(X) %*% X) %*% t(X) %*% Y
```

The optimal solution of the standard deviation for β has the following form:

$$s.d.(\hat{\beta}) = s^2(X'X)^{-1} \; where \; s^2 = \frac{1}{n-4} \sum_{i=1}^{n} (Y_i - X_i\hat{\beta})^2, \; i = 1, 2, \ldots, n$$

so $s.d.(\hat{\beta})$ can be calculated using the equation below:

```
sig_sq <- sum((Y - X %*% beta_hat)^2) / (nrow(X) - 4)
st_dev <- sqrt(sig_sq * solve(t(X) %*% X))
```

Question 2 Part B

The manual calculation of the optimal values of $\hat{\beta}$ and $s.d.(\hat{\beta})$ using the R code from Part A are as follows.

Note that the expression for the standard deviation produces a square matrix with the relevant values on the diagonal and `NaNs` everywhere else, so only the diagonal values are presented in the table alongside their respective coefficient values.

	beta_hat	st_dev
(Intercept)	3.5266672	0.3742899
youtube	0.0457646	0.0013949
newspaper	-0.0010375	0.0058710
facebook	0.1885300	0.0086112

The R code to estimate a linear regression model using the `lm()` function and the corresponding results for the same dataset is:

```
lm_mod <- lm(sales ~., data = marketing)
summary(lm_mod)$coef[,1]
summary(lm_mod)$coef[,2]
```

	beta_hat	st_error
(Intercept)	3.5266672	0.3742899
youtube	0.0457646	0.0013949
facebook	0.1885300	0.0086112
newspaper	-0.0010375	0.0058710

As is expected and clearly visible from the two sets of tables above, the results from the manual calculation of $\widehat{\beta}$ and $s.d(\widehat{\beta})$ is identical to that using the normal `lm()` function in R. This is because the `lm()` function in R calls `lm.fit()`, which by default uses those same equations to estimate the optimal coefficient values and corresponding standard errors for the input dataset.

2 Optimisation

Question 3 Part A

The following part details the five main steps required to implement a classical gradient descent algorithm for a multiple linear regression model:

Step 1: Determine default values for the learning rate (α), convergence threshold and maximum number of iterations the algorithm will run through, and initialise values for the intercept and the three predictors: `youtube`, `newspaper` and `facebook`.

Step 2: Calculate the partial derivates with respect to the predictor variables (β_1 to β_3) and the intercept (β_0) for the mean squared error loss function: $\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - X_i\beta)^2$:

$$
\begin{aligned}
D_{\beta_j} &= \frac{1}{n}\sum_{i=1}^{n} 2(Y_i - (\beta_1 X_{1i} + \beta_2 X_{2i} + \beta_3 X_{3i} + \beta_0))(-X_{ji}) \\
&= \frac{1}{n}\sum_{i=1}^{n} 2(Y_i - X_i\beta)(-X_{ji}),\ where\ j = 1, 2, 3
\end{aligned}
$$

$$D_{\beta_0} = \frac{-2}{n}\sum_{i=0}^{n}(Y_i - (\beta_1 X_{1i} + \beta_2 X_{2i} + \beta_3 X_{3i} + \beta_0))$$
$$= \frac{-2}{n}\sum_{i=0}^{n}(Y_i - X_i\beta)$$

These partial derivatives are also the gradients of the linear regression.

Step 4: Update the values of D_{β_j} and D_{β_0} according to the following equation:
$\beta_{j(new)} = \beta_{j(current)} - \alpha D_{\beta_j},\ where\ j = 0, 1, 2, 3$

Note that α is the learning rate or step size. This parameter determines the size of the 'step' the algorithm takes each iteration for each coefficient in the linear regression model.

Step 5: Continue to update the values of D_{β_j} and D_{β_0} until the loss function is very small (or ideally zero), or the number of iterations reaches the maximum number specified. 'Very small' means that the algorithm will stop when the difference between subsequent values of the loss function is less than a specified threshold.

Question 3 Part B

The R code that will implement the classical gradient descent procedure described in Part A above is:

```
# getting the data
data <- marketing[,c(1,3,2,4)]

# setting up the gradient descent algorithm
grad_desc <- function(X, Y, p0, step_size, max_iter, conv_thresh) {
  p <- matrix(0, nrow = max_iter, ncol = length(p0))
  gradient <- matrix(0, nrow = max_iter, ncol = length(p0))
  p[1,] <- p0
  for(i in 1:(max_iter-1)) {
    yhat <- p[i,1] + as.matrix(X[,2:4]) %*% p[i,2:4]
    gradient[i,] <- -2*colMeans(X*(Y - yhat))
    p[i+1,] <- p[i,] - step_size*gradient[i,]
    if(i > 1 & all(abs(gradient[i,]) < conv_thresh)) {
      i = i - 1
      break;
    }
  }
  return(list("i" = i, "p" = p, "g" = gradient))
}

# defining the initial data inputs and model coefficients
X <- cbind(1, data[,c(1:3)])
```

```
Y <- data$sales
p0 <- c(3.0, 0.0, 0.0, 0.0)

# running the algorithm and obtaining the model coefficient results
gd_results <- grad_desc(X, Y, p0, step_size = 0.000003, max_iter = 10000,
                        conv_thresh = 0.01)
gd_results$p[gd_results$i,]
```

Question 3 Part C

The R code in Part B above produced the results in the first column of the table below:

	cgd estimates	beta_hat
(Intercept)	3.0065401	3.5266672
youtube	0.0469484	0.0457646
newspaper	0.0015954	-0.0010375
facebook	0.1935015	0.1885300

Whilst these results are very similar what was obtained through manual estimation of the coefficients (equation $\widehat{\beta} = (X'X)^{-1}X'Y$) in Question 2 Part A (the second column of the table above), they are not identical. It could be purely coincidental, but the coefficient results from the classical gradient descent algorithm are all slightly larger in magnitude than the manual estimates with the exception of the intercept term.

Classical gradient descent is an iterative procedure, and the final values of the coefficients depend on four main parameters: the initial coefficient vector (p_0), the step size, the maximum number of iterations allowed, and the convergence threshold in addition to the input dataset. In contrast, the manual estimation only uses the input dataset and will always produce the optimum solution for a linear regression model of that data.

Question 4

Classical gradient descent (also known as 'batch gradient descent') is an iterative optimisation algorithm that attempts to find the local minimum for a given function by moving in the direction of steepest descent as defined by the negative of the gradient. In the linear regression case that effectively translates to minimising the model's loss function (usually the mean squared errors) to find the line of best fit. The classical gradient descent algorithm works by starting with an initial set of model parameters (such as the intercept and a coefficient for each predictor in linear regression), determining the gradient for each parameter based on a specified loss function, and updating the value of each parameter until the gradient is as close to zero as possible. The size of the step taken over each iteration is determined by the learning rate α and the point at which the algorithm stops by the convergence criterion. Visually, classical gradient descent is akin to standing on top of a mountain and following a path to the valley below by only moving in the direction of the steepest downward slope in the immediate vicinity.

The classical gradient descent algorithm is ideal in situations when an optimal solution cannot be found analytically such as via linear algebra. However, it does have its disadvantages in that it may not always achieve convergence, can be computationally expensive, and can only be used to find a local minimum:

1. The learning rate needs to be carefully chosen. If it is too large then the algorithm may look one or more possible solutions before achieving convergence (or not converge at all). Similarly, if the learning rate is too small then the algorithm may require a very large number of iterations to achieve convergence, which could be very expensive computationally.
2. The algorithm iterates over the entire dataset each time, which makes it computationally expensive for very large datasets.
3. The algorithm is a first-order algorithm, which means that it only uses the first-derivative of the loss function to update the model parameters. This means that it may end up at a local minimum instead of the global minimum if there are many minimal points, or at a saddle-point where the gradient is zero but not an optimal point. In terms of our mountain-top analogy, the first is akin to ending up at a smaller valley half-way down the mountain instead of the valley floor and the second is akin to ending up at a flat area half-way down, not knowing that the ground continues to descend several metres away.

Stochastic gradient descent (also known as 'incremental gradient descent') is a varient on the classical gradient descent algorithm. Conceptually it is almost identical to classical gradient descent, which means that it can have some of the same issues with regards to convergence and settling on a local minima or saddle-point rather than the global minimum. However, the stochastic approach has perhaps two major advantages over the classical approach that can make it a better algorithm to use for certain situations.

1. The algorithm only iterates over a sub-sample of the input dataset, which means that it often gets to the minimum much faster than the classical approach and is much less computationally expensive for very large datasets.
2. The stochastic nature of the algorithm means that it can escape shallow local minima and saddle-points more easily. (The flipside to this is that the path to the optimum can be much noisier, which means that the loss curve will jump around more on its way to convergence compared to the smooth curve exhibited by classical gradient descent).

Note that it is possible that whilst the stochastic approach may get to the minimum much faster than the classical approach, it may never actually converge due to its random element and continue to oscillate around the optimal solution. In practice these oscillations will generally be close enough to the minimum that they are a reasonable approximation of the true solution.

Newton's method uses a slightly different approach to gradient descent in order to solve optimisation problems. It is uses the second-derivative to approximate the roots of a function (which can be used to locate local minima) instead of attempting to locate local minima directly as gradient descent algorithms do. Returning to the mountain-top analogy, it is akin to looking out across the horizon such that the line of sight is tangential to the surface, finding the lowest point, jumping to that point and repeating until the valley floor is reached.

In Newton's method, model parameters are updated using the Hessian matrix instead of the learning rate. This means that the algorithm typically converges extremely quickly; much faster than even stochastic gradient descent, and therefore requires far fewer iterations to get close to the minimum. However, the speed at which Newton's method convergence needs to be weighed up against a couple of other factors that can determine whether it is the most appropriate algorithm in a specific situation.

1. One iteration of Newton's method can be more expensive than one iteration of gradient descent as it requires the calculation and inversion of a *n-by-n* Hessian matrix. If a dataset is too large then it may not be even possible to use Newton's method as the computation would likely be prohibitively expensive.
2. The loss function needs to be convex and the starting parameters need to be initialised close to the minimum. If either is not the case then the algorithm may not converge to a solution as the Hessian matrix will not be positive definite in the first case and could become ill-conditioned in the second.