CS771A Assignment 2

- Yash Gupta (190997)

Q1. Gradient Descent

In [ ]:
```
# importing libraries
import numpy as np
import time as t
```

In [ ]:
```
# defining the gradient descent function
def gradient_descent(gradient, init_, learn_rate, n_iter=50, tol=1e-06):
    x = init_
    for _ in range(n_iter):
        delta = -learn_rate * gradient(x)
        if np.all(np.abs(delta) <= tol):
            break
        x += delta
    return np.round(x, 3) # Use np.round() instead of round() to round a nump
```

Q1. (a)

Use this function to find minima for (i) $x^2 + 3x + 4$ and (ii) $x^4 - 3x^2 + 2x$.

First, we'll have to find the gradient of the expressions in (i) and (ii). As these are expressions in one variable, their gradients are simply there derivatives wrt. x. They can be given as follows:
(i) $2x + 3$
(ii) $4x^3 - 6x + 2$

Now that we have the gradients, we can find the minima as follows:

(i)

In [ ]:
```
min_x1 = gradient_descent(gradient=lambda x: 2 * x + 3, init_=0.0, learn_rate
print('Point of minima:', min_x1)
min_y1 = min_x1 ** 2 + 3 * min_x1 + 4
print('Minima:', min_y1)
```

```
Point of minima: -1.5
Minima: 1.75
```

(ii)

In [ ]:
```
min_x2 = gradient_descent(gradient=lambda x: 4 * (x ** 3) - 6 * x + 2, init_=
print('Point of minima:', min_x2)
min_y2 = min_x2 ** 4 - 3 * (min_x2 ** 2) + 2 * min_x2
print('Minima:', min_y2)
```

```
Point of minima: -1.366
Minima: -4.848076206064
```

Q1. (b)

Write a gradient function to calculate gradients for a linear regression $y = ax + b$

The loss function (L_2 loss) for a linear regression $y = ax + b$, say $L(a, b)$, will be:

$$L(a,b) = \sum_{n=1}^{N} (y_n - ax_n - b)^2$$

where N is the number of data points.

The gradient for this loss function will be:

$$\left[\frac{\partial L}{\partial a}, \frac{\partial L}{\partial b}\right] = \left[-2\sum_{n=1}^{N} x_n(y_n - ax_n - b), -2\sum_{n=1}^{N}(y_n - ax_n - b)\right]$$

Hence, the gradient function can be written as follows:

In [ ]:
```
def gradient_lr(params):
    grad = np.array([0.0, 0.0])
    for Xn, yn in zip(X, y): # X is the features and y is the labels both of
        grad[0] += -2 * Xn * (yn - params[0] * Xn - params[1])
        grad[1] += -2 * (yn - params[0] * Xn - params[1])
    grad /= len(X) # so that grad is not very large and doesn't depend on the
    return grad
```

Q1. (c)

Generate artificial data for this regression according to the following protocol and use gradient descent to find the optimal parameters relating X with y.

In [ ]:
```
np.random.seed(0)
X = 2.5 * np.random.randn(10000) + 1.5 # array of 10000 values with mean = 1.
res = 1.5 * np.random.randn(10000) # generate 10000 residual terms
y = 2 + 0.3 * X + res # actual values of y
```

Running gradient descent on this data:

In [ ]:
```
params_init = np.array([0.0, 0.0])
tic = t.process_time()
params = gradient_descent(gradient_lr, init_=params_init, learn_rate=0.05, n_
toc = t.process_time()
time_vanilla_gd = toc - tic
a = params[0]
b = params[1]
print('a =', a)
print('b =', b)
print('Time taken:', time_vanilla_gd, 'seconds')
```

```
a = 0.295
b = 2.023
Time taken: 3.325558632 seconds
```

Q1. (d)

Implement minibatch stochastic gradient descent using the code base you have developed so far.

We can define the minibatch stochastic gradient descent function by creating a minibatch in every iteration of the gradient descent and passing the minibatch to the gradient function as follows:

In [ ]:

```
# defining the minibatch stochastic gradient descent function
def minibatch_sgd(gradient, init_, learn_rate, n_iter=50, tol=1e-06, batch_si
    x = init_
    for _ in range(n_iter):
        idx = np.random.choice(np.arange(len(X)), batch_size, replace=False)
        X_batch = X[idx]
        y_batch = y[idx]
        delta = -learn_rate * gradient(x, X_batch, y_batch)
        if np.all(np.abs(delta) <= tol):
            break
        x += delta
    return np.round(x, 3)
```

We will also need to define a gradient function which takes the minibatches as inputs as follows:

In [ ]:

```
def gradient_lr_sgd(params, X_batch, y_batch):
    grad = np.array([0.0, 0.0])
    for Xn, yn in zip(X_batch, y_batch):
        grad[0] += -2 * Xn * (yn - params[0] * Xn - params[1])
        grad[1] += -2 * (yn - params[0] * Xn - params[1])
    grad /= len(X_batch) # so that grad is not very large and doesn't depend
    return grad
```

Running minibatch stochastic gradient descent on the data:

In [ ]:

```
params_sgd_init = np.array([0.0, 0.0])
tic = t.process_time()
params_sgd = minibatch_sgd(gradient_lr_sgd, init_=params_sgd_init, learn_rate
toc = t.process_time()
time_minibatch_sgd = toc - tic
a = params[0]
b = params[1]
print('a =', a)
print('b =', b)
print('Time taken:', time_minibatch_sgd, 'seconds')
```

```
a = 0.295
b = 2.023
Time taken: 0.4765957890000001 seconds
```

Q1. (e)

Does SGD do better or worse in terms of time performance on our data? Is there an optimal minibatch size that works best? Quantify and interpret your findings.

Let's also find the time taken by SGD using a single training example per iteration.

In [ ]:

```
params_sgd_init_ = np.array([0.0, 0.0])
tic = t.process_time()
params_sgd = minibatch_sgd(gradient_lr_sgd, init_=params_sgd_init_, learn_rat
toc = t.process_time()
```

```
time_sgd = toc - tic
a = params[0]
b = params[1]
print('a =', a)
print('b =', b)
print('Time taken:', time_sgd, 'seconds')
```

```
a = 0.295
b = 2.023
Time taken: 0.2541064430000004 seconds
```

Let's look at the times taken by vanilla gradient descent and minibatch SGD.

In [ ]:
```
print('Time taken by vanilla gradient descent:', time_vanilla_gd)
print('Time taken by SGD:', time_sgd)
print('Time taken by minibatch SGD:', time_minibatch_sgd)
```

```
Time taken by vanilla gradient descent: 3.325558632
Time taken by SGD: 0.2541064430000004
Time taken by minibatch SGD: 0.4765957890000001
```

As we can see, SGD and minibatch SGD do much better in terms of time performance than vanilla gradient descent.

Vanilla gradient descent uses all the data in each iteration whereas SGD and minibatch SGD use only a part of the data in each iteration, which reduces the time of each iteration and hence take less time then vanilla gradient descent.

We found that the times taken by the gradient descents were as follows:
$t_{\text{SGD}} < t_{\text{minibatch SGD}} < t_{\text{vanilla gradient descent}}$

So, SGD appears to be working the fastest. However, it should be noted that as SGD uses only a single data point in each iteration, the updates would be unstable and it may take more number of iterations for SGD to converge, or, may have a higher loss as compared to minibatch SGD. But for same number of iterations, SGD would take the least time as it uses only a single data point in each iteration

Now, let's find the optimal minibatch size that works best. We will consider both the time taken as well as the loss.

First, let's define a function to find the L_2 loss:

In [ ]:
```
def l2_loss(params):
    loss = 0.0
    for i in range(len(X)):
        loss += (y[i] - params[0] * X[i] - params[1]) ** 2
    return loss
```

Now, let's iterate over some minibatch sizes and see which works the best.

In [ ]:
```
params_sgd_init = np.array([0.0, 0.0])
min_time = float('inf')
min_loss = float('inf')
min_time_batch_size = 0
min_loss_batch_size = 0
for batch_size in range(125, 1001, 125):
    # print('Current batch size:', batch_size)
    tic = t.process_time()
    params_sgd = minibatch_sgd(gradient_lr_sgd, init_=params_sgd_init, learn_
    toc = t.process_time()
    time_sgd = toc - tic
    loss_sgd = l2_loss(params_sgd)
    if time_sgd < min_time:
        min_time = time_sgd
        min_time_batch_size = batch_size
    if loss_sgd < min_loss:
        min_loss = loss_sgd
        min_loss_batch_size = batch_size
    print(f'For a batch size of {batch_size}: time taken = {round(time_sgd, 3
print()
print('The least time taking minibatch size is', min_time_batch_size, 'and it
print('The least loss minibatch size is', min_loss_batch_size, 'and it gives
```

```
For a batch size of 125: time taken = 0.281 seconds and L2 loss = 22625.428
For a batch size of 250: time taken = 0.332 seconds and L2 loss = 22321.531
For a batch size of 375: time taken = 0.532 seconds and L2 loss = 22214.284
For a batch size of 500: time taken = 0.657 seconds and L2 loss = 22374.676
For a batch size of 625: time taken = 0.771 seconds and L2 loss = 22228.737
For a batch size of 750: time taken = 0.906 seconds and L2 loss = 22485.522
For a batch size of 875: time taken = 1.04 seconds and L2 loss = 22256.803
For a batch size of 1000: time taken = 1.185 seconds and L2 loss = 22189.813

The least time taking minibatch size is 125 and it takes 0.2811508429999998 s
econds
The least loss minibatch size is 1000 and it gives a loss of 22189.8131663953
94
```

As we can see, the time taken increases as the batch size increases but the minimum loss can be seen with a batch size of 375.

As the time taken for a batch size of 375 is also pretty low, we can say that this is the optimal minibatch size that works best.

Q2. Bayesian network

Q2. (i)

Calculate the probability that someone has both cold and a fever

Let C represent cold and F represent fever. So, we need to find $P(C \cap F)$.

$$P(C \cap F) = P(F|C)P(C) = 0.307 \times 0.02 = 0.00614$$

Hence, the probability that someone has both cold and a fever is 0.00614

Q2. (ii)

Calculate the probability that someone who has a cough has a cold

Let X represent cough, C represent cold, L represent lung disease and S represent smokes. So, we need to find $P(C|X)$.

$$P(C|X) = \frac{P(X|C)P(C)}{P(X)}$$

Now,

$$P(X|C) = P\big(X|(C \cap L)\big)P(L|C) + P\big(X|(C \cap \overline{L})\big)P(\overline{L}|C)$$

Since C and L are independent, $P(L|C) = P(L)$ and $P(\overline{L}|C) = P(\overline{L})$. So,

$$P(X|C) = P\big(X|(C \cap L)\big)P(L) + P\big(X|(C \cap \overline{L})\big)P(\overline{L})$$

Now,

$$P(L) = P(L|S)P(S) + P(L|\overline{S})P(\overline{S}) = 0.1009 \times 0.2 + 0.001 \times 0.8 = 0.02098$$

and

$$P(\overline{L}) = 1 - P(L) = 1 - 0.02098 = 0.97902$$

Hence,

$$P(X|C) = 0.7525 \times 0.02098 + 0.505 \times 0.97902 = 0.51019255$$

Now,

$$P(X) = P\big(X|(L \cap C)\big)P(L \cap C) + P\big(X|(L \cap \overline{C})\big)P(L \cap \overline{C}) + P\big(X|(\overline{L} \cap C)\big)P($$

Again, since L and C are independent, $P(L \cap C) = P(L)P(C)$,
$P(L \cap \overline{C}) = P(L)P(\overline{C})$, $P(\overline{L} \cap C) = P(\overline{L})P(C)$ and $P(\overline{L} \cap \overline{C}) = P(\overline{L})P(\overline{C})$
Hence,

$$P(X) = P\big(X|(L \cap C)\big)P(L)P(C) + P\big(X|(L \cap \overline{C})\big)P(L)P(\overline{C}) + P\big(X|(\overline{L} \cap C)\big)$$

$$= 0.7525 \times 0.02098 \times 0.02 + 0.505 \times 0.02098 \times 0.98 + 0.505 \times 0.97902 \times 0.02 +$$

Hence,

$$P(C|X) = \frac{0.51019255 \times 0.02}{0.030181249} = 0.33808577637$$

Hence, the probability that someone who has a cough has a cold is 0.33808577637

Q3. Derive the MLE for the parameters of a k-sided multinomial distribution.

Suppose there are k categories. Let us do n independent trials, such that each trial leads to the success of exactly one of the k categories. Let p_i be the probability of success and X_i be the number of successes of category i for $i = 1, \ldots, k$.
Then, the likelihood of the k-sided multinomial distribution with parameters n and p_i for $i = 1, \ldots, k$ is:

$$f(\mathbf{x}; n, \mathbf{p}) = f(x_1, \ldots, x_k; n, p_1, \ldots, p_k) = n! \prod_{i=1}^{k} \frac{p_i^{x_i}}{x_i!}$$

where $\sum_{i=1}^{k} x_i = n$ and $\sum_{i=1}^{k} p_i = 1$.

We will write $f(\mathbf{x}; n, \mathbf{p})$ as $f(\mathbf{p})$ for brevity.

Now, the log-likelihood will be:

$$LL(\mathbf{p}) = \log f(\mathbf{p}) = \log\left(n! \prod_{i=1}^{k} \frac{p_i^{x_i}}{x_i!}\right)$$

$$= \log n! + \log \prod_{i=1}^{k} \frac{p_i^{x_i}}{x_i!}$$

$$= \log n! + \sum_{i=1}^{k} \log \frac{p_i^{x_i}}{x_i!}$$

$$= \log n! + \sum_{i=1}^{k} x_i \log p_i - \sum_{i=1}^{k} \log x_i!$$

We need to maximize $LL(\mathbf{p})$ with the constraint $\sum_{i=1}^{k} p_i = 1$.

Hence, we can define the lagrangian as:

$$l(\mathbf{p}, \lambda) = LL(\mathbf{p}) + \lambda\left(1 - \sum_{i=1}^{k} p_i\right)$$

where λ is the lagrange multiplier.

Now, to find \argmax$_{\mathbf{p}} l(\mathbf{p}, \lambda)$,

$$\frac{\partial}{\partial p_i} l(\mathbf{p}, \lambda) = 0$$

for $i = 1, \ldots, k$

$$\frac{\partial}{\partial p_i} l(\mathbf{p}, \lambda) = \frac{\partial}{\partial p_i} LL(\mathbf{p}) + \frac{\partial}{\partial p_i} \lambda\left(1 - \sum_{i=1}^{k} p_i\right)$$

$$= \frac{\partial}{\partial p_i} \sum_{i=1}^{k} x_i \log p_i - \lambda \frac{\partial}{\partial p_i} \sum_{i=1}^{k} p_i$$

$$= \frac{x_i}{p_i} - \lambda$$

Since, $\frac{\partial}{\partial p_i} l(\mathbf{p}, \lambda) = 0$,

$$\frac{x_i}{p_i} - \lambda = 0$$

$$\Rightarrow p_i = \frac{x_i}{\lambda}$$

Now, since $\sum_{i=1}^{k} p_i = 1$,

$$\sum_{i=1}^{k} \frac{x_i}{\lambda} = 1$$

$$\Rightarrow \lambda = \sum_{i=1}^{k} x_i = n$$

Hence,

$$p_i = \frac{x_i}{n}$$

In other words,

$$\mathbf{p} = \left(\frac{x_1}{n}, \ldots, \frac{x_k}{n}\right)$$

is the MLE for the parameters of a k-sided multinomial distribution.