```python
import numpy as np
n = 20
n_epochs = 1
sampled_data = np.zeros(n*n_epochs)
for t in range(n_epochs):
    for iteration in range(n):
        ind  = np.random.randint(0, n)
        sampled_data[ind] = 1
prob_sampled_single_epoch = np.sum(sampled_data > 0) / n
prob_never_sampled_single_epoch = (1 - prob_sampled_single_epoch)
print("The probability P(never) that one observation is never sampled during one epoch : ", prob_never_sampled_single_epoch)
```

```
The probability P(never) that one observation is never sampled during one epoch :  0.35
```

DATE / /

h2 (b) $\lim_{n\to\infty} (1+\frac{1}{n})^n = e$

To prove n (large) $P_{never} \simeq 0.3679$

Given eqn similar to $\lim_{n\to\infty} f(n)^{\wedge} g(n)$

for P_{never} we need to find $(1-\frac{1}{n})^n$

$\therefore \lim_{n\to\infty} (1-\frac{1}{n})^n = \lim_{n\to\infty} \left((1+\frac{1}{n})^n\right)^{-1}$

$\therefore \lim_{n\to\infty} (1-\frac{1}{n})^n = e^{-1} = \frac{1}{e}$

$e \simeq 2.718 \quad \therefore 1/e = 1/2.718 \simeq 0.3679$

h2 (d) $Var(\nabla f(x^{(i)}, y^{(i)})) = \sigma^2$, variance of gradient over minibatch

$\rightarrow$ Total size $= n$ Minibatch $= \tilde{n} \leq n$

$\therefore$ variance over minibatch $\Rightarrow \frac{\sigma^2}{\tilde{n}}$

h2 (e) Multiply $\tilde{n}$ by factor of $K \rightarrow (K\tilde{n})$

Variance becomes $\rightarrow \frac{\sigma^2}{K\tilde{n}} = \frac{1}{K}\left(\frac{\sigma^2}{\tilde{n}}\right)$

New minibatch variance decreases by a factor of $1/K$ compared to the original variance when we multiply it by a factor a k to mini batch size

```
# problem 1_f
The authors argue: “When the minibatch size is multiplied by k, multiply the learning rate by k.”

Gradient noise decreases when we use SGD (stochastic gradient descent) while working with mini-batches
as the gradient keeps the weights being updated during each (mini-batch * epochs) iteration.

Now the author argues that when mini batch size is multiplied by a factor of k we need to multiply
the learning rate by the same factor of k.

I feel the claim is incorrect because the gradient noise is proportional to the square root of the mini-batch size.

So, if you multiply the mini-batch size by a factor of k, the gradient noise will increase by a factor of sqrt(k).

There are a few ways to change the learning rate so that the gradient noise stays constant.
Using Learning rate schedulers
Using Exponential decay
Using Regularization techniques, such as L2 regularization, Dropout
Using Stochastic Gradient Descent (SGD) with Minibatch
Using a larger batch size
Using a more robust Optimization Algorithm
```

h_2 (2) XOR Problem

Equation 6.2 → $J(\Theta) = \frac{1}{4} \sum_{x \in X} (f^*(x) - f(x,\theta))^2$

given $f(x;\theta)$ where θ consist of w & b.

$f(x; w, b) = x^T w + b$.

Solve for $w \to w_1$ & w_2 & b

$$\frac{\partial J(\theta)}{\partial w_1} = \frac{1}{4} \sum_{x \in X} \left(f^*(x) - (x_1^T w_1 + x_2^T w_2 + b)\right) \cdot (2) \cdot (-x_1)$$

$$\therefore \frac{\partial J(\theta)}{\partial w_1} = -\frac{1}{2} x_1 \sum_{x \in X} \left(f^*(x) - (x_1 w_1 + x_2 w_2 + b)\right) \quad \text{--- (1)}$$

Similarly for w_2 & b we get

$$\frac{\partial J(\theta)}{\partial w_2} = -\frac{1}{2} x_2 \sum_{x \in X} \left(f^*(x) - (x_1 w_1 + x_2 w_2 + b)\right) \quad \text{--- (2)}$$

$$\frac{\partial J(\theta)}{\partial b} = -\frac{1}{2} \sum_{x \in X} \left(f^*(x) - (x_1 w_1 + x_2 w_2 + b)\right) \quad \text{--- (3)}$$

Solving eq (1) for XOR inputs

when $x_1 = 0, x_2 = 0$ $f^*(x) = 0$ we get 0

when $x_1 = 0, x_2 = 1$ $f^*(x) = 1$ we get 0

when $x_1 = 1, x_2 = 0$ $f^*(x) = 1$ we get

$-\frac{1}{2} x_1 \left(f^*(x) - (x_1 w_1 + x_2 w_2 + b)\right)$

$= -\frac{1}{2}(1)\left[1 - (w_1 + b)\right] = -\frac{(1 - (w_1 + b))}{2}$

DATE / /

When $x_1 = 1$, $x_2 = 1$, $f^*(x) = 0$ we get

$$\frac{-1}{2}(1)\left[0 - (w_1 + w_2 + b)\right] = \frac{(w_1 + w_2 + b)}{2}$$

Summation to zero yields

$$0 + 0 + -\frac{(1 - (w_1 + b))}{2} + \frac{1}{2}(w_1 + w_2 + b) = 0$$

$$-1 + w_1 + b + w_1 + w_2 + b = 0$$

$$2b + 2w_1 + w_2 = 1 \quad \ldots \text{(4)}$$

Similarly solving for eqn (2)

When $x_2 = 0$, $x_1 = 0$, $f^*(x) = 0$

we get 0

when $x_2 = 0$, $x_1 = 1$, $f^*(x) = 1$

we get 0

when $x_2 = 1$, $x_1 = 0$, $f^*(x) = 1$ we get

$$\frac{-1}{2}(1)\left[1 - (w_2 + b)\right] = \frac{-1}{2}\left[1 - w_2 - b\right]$$

when $x_2 = 1$, $x_1 = 1$, $f^*(x) = 0$ we get

$$\frac{-1}{2}(1)\left[0 - (w_1 + w_2 + b)\right] = \frac{-1}{2}\left[-w_1 - w_2 - b\right]$$

summation to zero yields

$$0 = 0 + 0 + \left(\frac{-1}{2}\right)(1 - w_2 - b) + \left(\frac{-1}{2}\right)(-w_1 - w_2 - b)$$

$$0 = -1 + w_2 + b + w_1 + w_2 + b$$

$$0 = 2b + 2w_2 + w_1 - 1 \quad \ldots \text{(5)}$$

So $2b + 2w_2 + w_1 = 1$

Similarly solving for eqn (3)

when $x_1 = 0, x_2 = 0, f^*(x) = 0$ we get

$\frac{-1}{2}[0 - (b)] = +\frac{b}{2}$

when $x_1 = 0, x_2 = 1, f^*(x) = 1$ we get

$\frac{-1}{2}[1 - (w_2 + b)] = \frac{-1}{2}(1 - w_2 - b)$ ...

when $x_1 = 1, x_2 = 0, f^*(x) = 1$, we get

$\frac{-1}{2}[1 - (w_1 + b)] = \frac{-1}{2}[1 - w_1 - b]$ ...

when $x_1 = 1, x_2 = 1, f^*(x) = 0$ we get

$\frac{-1}{2}[0 - (w_1 + w_2 + b)] = \frac{-1}{2}[-w_1 - w_2 - b]$ ...

Summation to Zero yields

$$\left(+\frac{b}{2}\right) + \left(\frac{-1}{2}(1 - w_2 - b)\right) + \left(\frac{-1}{2}(1 - w_1 - b)\right) + \left(\frac{-1}{2}(-w_1 - w_2 - b)\right) = 0$$

$\frac{-1}{2}\ \ b + b - 1 + w_2 + b - 1 + w_1 + b + w_1 + w_2 + b = 0$

$4b - 2 + 2w_1 + 2w_2 = 0$

Divide by 2 on both sides

$2b - 1 + w_1 + w_2 = 0$

$\Rightarrow 2b = (1 - w_1 - w_2)$ .... (6)

substituting eq^n 6 in eq^n 4 we get

$$(1 - w_1 - w_2) + 2w_1 + w_2 = 1$$

$$w_1 = 0 \quad \cdots 7$$

substituting eq^n 6 in eq^n 5 we get

$$(1 - w_1 - w_2) + 2w_2 + w_1 = 1$$

$$w_2 = 0 \quad \cdots 8$$

substituting eq^n 7 & eq^n 8 in eq^n 6

$$2b = (1 - w_1 - w_2)$$

$$= (1 - 0 - 0)$$

$$= 1$$

$$2b = 1$$

$$\therefore b = \frac{1}{2} = 0.5$$

```python
# Training Linear Regression Model with user defined params + grid search

import numpy as np

X_tr = np.reshape(np.load("age_regression_Xtr.npy"), (-1, 48*48))
y_tr = np.load("age_regression_ytr.npy")
X_te = np.reshape(np.load("age_regression_Xte.npy"), (-1, 48*48))
y_te = np.load("age_regression_yte.npy")
print(X_tr.shape,y_tr.shape,X_te.shape,y_te.shape)

split_80_20: int = int(X_tr.shape[0]*0.8)
print('80 / 20 Split : ',split_80_20)
X_tr_80, y_tr_80 = X_tr[:split_80_20,:], y_tr[:split_80_20]
X_vl_20, y_vl_20 = X_tr[split_80_20:,:], y_tr[split_80_20:]
del X_tr,y_tr
print(X_tr_80.shape,y_tr_80.shape,X_te.shape,y_te.shape,X_vl_20.shape,y_vl_20.shape)

# custom grid params
grid_search_mini_batch_size : list = [160,200,400,640]
grid_search_learning_rate : list = [3e-8,3e-6,3e-5,3e-3]
grid_search_no_of_epochs : list = [16,32,48,64]
grid_search_regularization_val : list = [1e-7,1e-5,1e-3,1e-1]

# custom helper functions
def get_initialize_weights_and_bias_liner_reg(input_dims:int) -> tuple:
  # return np.random.randn(input_dims), 0.1
  return np.zeros(input_dims), 0.1

def get_y_hat_pred_func(x: np.ndarray, w: np.ndarray, b: np.ndarray) -> np.ndarray:
  return np.dot(x, w) + b

def get_unregularized_loss_func(y_pred: np.ndarray, y_true: np.ndarray) -> float:
  return 0.5 * np.mean((y_pred - y_true)**2)

def get_regularized_loss_func(y_pred: np.ndarray, y_true: np.ndarray, w: np.ndarray, alpha: float) -> float:
  return get_unregularized_loss_func(y_pred, y_true) + 0.5 * alpha * np.dot(w.T, w)

# hyperparameter tuning
best_params: dict = {
                'batch_size': 0,
                'learning_rate': 0,
                'num_epochs': 0,
                'regularization_strength': 0,
                'weights': 0,
                'bias': 0,
              }
best_loss: float = X_tr_80.shape[0]
```

```python
best_loss: float = X_tr_80.shape[0]

# save history (log of each step)
result_params: list = list()

# Shuffle training data
shuffling_indices: np.ndarray = np.arange(X_tr_80.shape[0])
np.random.shuffle(shuffling_indices)
X_train_shuffled: np.ndarray = X_tr_80[shuffling_indices]
y_train_shuffled: np.ndarray = y_tr_80[shuffling_indices]

for initial_mini_batch_size in grid_search_mini_batch_size:
    for initial_no_of_epochs in grid_search_no_of_epochs:
        for initial_learning_rate in grid_search_learning_rate:
            for initial_regularization_strength in grid_search_regularization_val:

                # Initialize weights and bias
                w, b = get_initialize_weights_and_bias_liner_reg(input_dims=X_tr_80.shape[1])

                # Training Loop
                for epoch in range(initial_no_of_epochs):

                    for batch_start in range(0, X_tr_80.shape[0], initial_mini_batch_size):

                        # taking mini batches from training data
                        X_mini_batch = X_train_shuffled[batch_start:batch_start+initial_mini_batch_size,:]
                        y_mini_batch = y_train_shuffled[batch_start:batch_start+initial_mini_batch_size]

                        # we compute the y hat predict
                        y_pred_mini = get_y_hat_pred_func(X_mini_batch, w, b)

                        # we compute gradient of w
                        grad_w = np.dot(X_mini_batch.T, (y_pred_mini - y_mini_batch)) / X_mini_batch.shape[0]

                        # apply L2 regularization to learning weights
                        grad_w += initial_regularization_strength * w

                        # update learning weights
                        w -= initial_learning_rate * grad_w

                # compute loss on validation set
                y_val_pred = get_y_hat_pred_func(X_vl_20, w, b)

                # compute and print the loss for monitoring training progress
                unreg_val_loss: float = round(get_unregularized_loss_func(y_val_pred, y_vl_20),4)
                reg_val_loss: float = round(get_regularized_loss_func(y_val_pred, y_vl_20, w, initial_regularization_strength),4)
```

```python
            # print(f"Epoch {epoch + 1}/{initial_no_of_epochs}, UnRegulaized Val Loss: {unreg_val_loss}%, Regulaized Val Loss: {reg_val_loss}%")
            result_params.append({
                    'batch_size': initial_mini_batch_size,
                    'learning_rate': initial_learning_rate,
                    'num_epochs': initial_no_of_epochs,
                    'regularization_strength': initial_regularization_strength,
                    'bias': b,
                    'un_regulaized_loss' : unreg_val_loss,
                    'regulaized_loss' : reg_val_loss,
            })

            # Check if this set of hyperparameters is the best so far
            if unreg_val_loss < best_loss:
                best_loss = unreg_val_loss
                best_params = {
                    'batch_size': initial_mini_batch_size,
                    'learning_rate': initial_learning_rate,
                    'num_epochs': initial_no_of_epochs,
                    'regularization_strength': initial_regularization_strength,
                    'weights': w,
                    'bias': b,
                }
            else:
              continue

    # the best hyperparameters and corresponding loss
    print(f"\nBest Hyperparameters: {best_params}, \nBest Validation Loss: {best_loss}")

    # compute loss on validation set
    y_te_pred = get_y_hat_pred_func(X_te, best_params['weights'], best_params['bias'])

    # compute and display the loss for monitoring training progress
    unreg_te_loss: float = round(get_unregularized_loss_func(y_te_pred, y_te),4)
    reg_te_loss: float = round(get_regularized_loss_func(y_te_pred, y_te, best_params['weights'], best_params['regularization_strength']),4)
    print(f"\nComputing Loss on Test Data Set\nUnRegulaized Test Loss: {unreg_te_loss}%, Regulaized Test Loss: {reg_te_loss}%")
```

```
(4000, 2304) (4000,) (2500, 2304) (2500,)
80 / 20 Split :  3200
(3200, 2304) (3200,) (2500, 2304) (2500,) (800, 2304) (800,)

Best Hyperparameters: {'batch_size': 160, 'learning_rate': 0.003, 'num_epochs': 64, 'regularization_strength': 1e-07, 'weights': array([ 0.88437179,  0.77671802,  0.41219991, ..., -0.28188384,
       -0.22051565, -0.25063656]), 'bias': 0.1},
Best Validation Loss: 114.3292

Computing Loss on Test Data Set
UnRegulaized Test Loss: 112.385%, Regulaized Test Loss: 112.385%
```

hw2 (4) Linear Regression with MAE minimization via Gradient Descent

$$f_{MAE}(w,b) = \frac{1}{n} \sum_{i=1}^{n} \left| (x^{(i)})^T w + b - y^{(i)} \right|$$

$$t = \left[(x^{(i)})^T w + b - y^{(i)} \right]$$

(A) For prediction that overestimates

$$\hat{y}^{(i)} < \left[(x^{(i)})^T w + b \right]$$

(B) For predictions that underestimates

$$\hat{y}^{(i)} > \left[(x^{(i)})^T w + b \right]$$

Consider overestimation case

when $t > 0$ $\frac{\partial |t|}{\partial t} = 1$

when $t < 0$ $\frac{\partial |t|}{\partial t} = \frac{\partial(-t)}{\partial t} = -1$

Considering the summation, yields

$$\frac{\partial f_{MAE}(w,b)}{\partial b} = \frac{1}{n} \sum_{i=1}^{n^+} \frac{\partial |t|}{\partial t}$$

$= \frac{1}{n}$ [no. of cases $t > 0$ - no. of cases $t < 0$]

$= \frac{n^+}{n}$ .... (1)

Consider underestimation case

when $t > 0$ $\frac{\partial |t|}{\partial t} = 1$

when $t < 0$ $\frac{\partial |t|}{\partial t} = \frac{\partial(-t)}{\partial t} = -1$

Similarly we get $\left(-\frac{n^-}{n}\right)$

$$\therefore \nabla_b f_{MAE}(w,b) = \frac{n^+}{n} - \frac{n^-}{n}$$

$$= \frac{1}{n}(n^+ - n^-)$$

DATE / /

hw (4b) Expression for $\nabla_w f_{MAE}(w, b)$

$$t = \left((x^{(i)})^T w + b - y^{(i)} \right)$$

when $t > 0$ $\quad \frac{\partial |t|}{\partial w} = x^{(i)}$

when $t < 0$ $\quad \frac{\partial |t|}{\partial w} = -\left(x^{(i)}\right)$

Case where predictions overestimates and underestimates we get

$$\frac{\partial f_{MAE}(w, b)}{\partial w} = \frac{1}{n} \left[\text{no. of cases } t > 0 \cdot x^{(i)} - \text{no. of cases } t < 0 \; x^{(i)} \right]$$

$$\nabla_w f_{MAE}(w, b) = \frac{1}{n} \sum_{i=1}^{n} \left[(n^+ \, x^{(i)}) - (n^- \cdot x^{(i)}) \right]$$

```python
# problem 4_c

import numpy as np

def get_update_b_with_gradient_descent(X: np.ndarray, y: np.ndarray, w: np.ndarray, b: float, lr: float) -> float:

    # Compute the predicted values
    y_hat_pred: np.ndarray = np.dot(X, w) + b

    # we compute the gradient b for using MAE equation
    grad_b: np.ndarray = (1/X.shape[0]) * np.sum(np.sign(y_hat_pred-y))

    # update parameter b
    b_new: float = b - lr * grad_b

    return b_new
```

```python
print(b)
print(get_update_b_with_gradient_descent(X=X_te, y=y_te, w=w, b=b, lr=initial_learning_rate))
```

```
0.1
0.1024096
```

hw$_2$ (5) $p(y|x) = N(\mu = x^T w, \sigma^2)$

$$= \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - x^T w)^2}{2\sigma^2}\right)$$

To prove:

$$w = \left(\sum_{i=1}^{n} x^{(i)} (x^{(i)})^T\right)^{-1} \left(\sum_{i=1}^{n} x^{(i)} y^{(i)}\right)$$

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (x^{(i)T} w - y^{(i)})^2$$

DATE / /

$$P(D \mid w, \sigma^2) = \prod_{i=1}^{n} P(y^{(i)} \mid x^{(i)}, w, \sigma^2)$$

Taking log of likelihood

$$\log P(D \mid w, \sigma^2) = \sum_{i=1}^{n} \log P(y^{(i)} \mid x^{(i)}, w, \sigma^2)$$

Now, $\log P(y^{(i)} \mid x^{(i)}, w, \sigma^2) = -\frac{1}{2} \log (2\pi\sigma^2) - \frac{(y^{(i)} - (x^{(i)})^T w)^2}{2\sigma^2}$

$$= \frac{1}{2\sigma^2} \sum_{i=1}^{n} (y^{(i)} - (x^{(i)})^T w)^2$$

Taking ∇ wrt w we get after setting to 0

$$\frac{1}{\sigma^2} \sum_{i=1}^{n} (x^{(i)} (x^{(i)})^T w - x^{(i)} y^{(i)}) = 0$$

$$\sum_{i=1}^{n} x^{(i)} x^{(i)T} w = \sum_{i=1}^{n} x^{(i)} y^{(i)}$$

$$\therefore w = \left(\sum_{i=1}^{n} x^{(i)} x^{(i)T}\right)^{-1} \left(\sum_{i=1}^{n} x^{(i)} y^{(i)}\right)$$

Now for σ^2

$$\frac{\partial}{\partial \sigma^2} P(D \mid w, \sigma^2) = -\frac{n}{2\sigma^2} + \frac{1}{2\sigma^4} \sum_{i=1}^{n} (y^{(i)} - (x^{(i)})^T w)^2$$

Setting $\frac{\partial}{\partial \sigma^2} P(D \mid w, \sigma^2) = 0$

$$\therefore -\frac{n}{2\sigma^2} + \frac{1}{2\sigma^4} \sum_{i=1}^{n} \left(y^{(i)} - (x^{(i)})^T w\right)^2 = 0$$

multiply both sides by $2\sigma^4$

$$-n\sigma^2 + \sum_{i=1}^{n} \left(y^{(i)} - x^{(i)T} w\right)^2 = 0$$

$$n\sigma^2 = \sum_{i=1}^{n} \left(y^{(i)} - (x^{(i)})^T w\right)^2$$

$$\therefore \sigma^2 = \frac{1}{n} \sum_{i=1}^{n} \left(x^{(i)T} w - y^{(i)}\right)^2$$