Homework 3

Due: **October 1st, 5pm** (late submission until October 4th, 5pm -- no submission possible afterwards)

Written assignment: 10 points

Coding assignment: 25 points

Project report: 15 points

Name: Rohitha Ravindra Myla

Link to the github repo: https://github.com/data2060-fall2024/hw-03-rohitharavindra08/tree/main

Written assignment

Gradient Descent (10 points)

Consider using gradient descent to find the minimum of f, where,

- f is a convex function over the closed interval [-b,b], b > 0
- f' is the derivative of f
- α is some positive number which will represent a learning rate parameter

The steps of gradient descent are as follows:

- Start at $x_0 = 0$
- At each step, set $x_{t+1} = x_t - \alpha f'(x_t)$
- If x_{t+1} falls below -b, set it to -b, and if it goes above b, set it to b.

We say that an optimization algorithm (such as gradient descent) ϵ*-converges* if, at some point, x_t stays within ϵ of the true minimum. Formally, we have ϵ*-convergence* at time t if

$|x_{t'} - x_{min}| \le \epsilon$, where $x_{min} = argmin_{x \in [-b,b]} f(x)$ for all $t' \ge t$.

Question 1

For α = 0.1, b = 1, and ϵ = 0.001, find a convex function f so that running gradient descent does not ϵ-converge. Specifically, make it so that x0 = 0, x_1 = b, x_2 = - b, x_3 = b, x_4 = - b, etc.

Solution:

Parameters: $\alpha = 0.1$, $b = 1$, $\epsilon = 0.001$

Construct a convex function $f(x)$ that oscillates between -1 and 1, let's consider:

[f'(x) = 5x^3 + 15x - 5]

- The derivative of $f(x)$ is: [f'(x) = 5x^3 + 15x - 5]
- Integrating $f'(x)$ gives: [f(x) = $\int(5x^3+15x-5)$, dx = $\frac{5}{4}x^4 + \frac{15}{2}x^2 - 5x$]
- This polynomial function ensures oscillation between -1 and 1.

Question 2

For α = 0.1, $b=1$, and ϵ = 0.001, find a convex function f so that gradient descent does ϵ - *converge*, but only after at least 10,000 steps.

Solution:

Parameters: $\alpha=0.1$, $b=1$, $\epsilon=0.001$

- Let's think of a linear gradient function: [f'(x) = Ax + B, $\text{where } A>0 \text{ (to ensure convexity)}$.]
- Then, the gradient descent update rule becomes: [x_n = x_{n-1} - $\alpha(Ax_{n-1}+B)$ = $(1-\alpha A)x_{n-1}$ - $\alpha B = Cx_{n-1}+D$] Here, $C=1-\alpha A$ and $D=-\alpha B$.

Convergence condition: [x_n - x_{n-1} = C(x_{n-1} - x_{n-2}) = $\ldots = DC^{n-1}$] To converge after $n>10,000$ steps: [|x_n - x_{n-1}| = |DC^{n-1}| < 0.001 = 10^{-3}.] Taking the logarithm: [\log D + (n-1)\log C < -3.]

Assuming $D=1$: [(n-1)\log C < -3 $\Longrightarrow n-1>\frac{-3}{\log C}$]

Finding C: To ensure convergence after $n>10,000$ steps: [10^4 > $\frac{-3}{\log C} \Longrightarrow \log C > \frac{-3}{10^4}$, C > $10^{$\frac{-3}{10^4}$} \approx 0.9993$.]

Finding A and B:

- From $C=1-\alpha A$: [$1 - \alpha A > $0.9993 $\Longrightarrow \alpha A < 0.0007 \Longrightarrow A$ > $\frac{0.0007}{\alpha}$.] For $\alpha=0.1$: [A > \frac{0.0007}{0.1} = 0.007.] Let $A=0.008$.
- From $D=-\alpha B$, and assuming $D=1$: [B = $\frac{D}{-\alpha}$ = $\frac{1}{-0.1} = -10$.]

Final function:

- Gradient: [f'(x) = 0.008x - 10]
- Function: [f(x) = \int (0.008x - 10) , dx = 0.004x^2 - 10x]

Coding Assignment (25 points)

Run the evironmant test below, make sure you get all green checks, if not, you will lose 2 points for each red flag.

```python
from __future__ import print_function
from packaging.version import parse as Version
from platform import python_version

OK = '\x1b[42m[ OK ]\x1b[0m'
FAIL = "\x1b[41m[FAIL]\x1b[0m"

try:
    import importlib
except ImportError:
    print(FAIL, "Python version 3.12.5 is required,"
                " but %s is installed." % sys.version)

def import_version(pkg, min_ver, fail_msg=""):
    mod = None
    try:
        mod = importlib.import_module(pkg)
        if pkg in {'PIL'}:
            ver = mod.VERSION
        else:
            ver = mod.__version__
        if Version(ver) == Version(min_ver):
            print(OK, "%s version %s is installed."
                  % (lib, min_ver))
        else:
            print(FAIL, "%s version %s is required, but %s installed."
                  % (lib, min_ver, ver))
    except ImportError:
        print(FAIL, '%s not installed. %s' % (pkg, fail_msg))
    return mod


# first check the python version
pyversion = Version(python_version())

if pyversion >= Version("3.12.5"):
    print(OK, "Python version is %s" % pyversion)
elif pyversion < Version("3.12.5"):
    print(FAIL, "Python version 3.12.5 is required,"
                " but %s is installed." % pyversion)
else:
    print(FAIL, "Unknown Python version: %s" % pyversion)


print()
```

```
requirements = {'matplotlib': "3.9.1", 'numpy': "2.0.1",'sklearn':
"1.5.1",
                'pandas': "2.2.2"}

# now the dependencies
for lib, required_version in list(requirements.items()):
    import_version(lib, required_version)
```

```
[ OK ] Python version is 3.12.5

[ OK ] matplotlib version 3.9.1 is installed.
[ OK ] numpy version 2.0.1 is installed.
[ OK ] sklearn version 1.5.1 is installed.
[ OK ] pandas version 2.2.2 is installed.
```

Introduction

In this assignment, you will be using a modified version of the UCI Census Income data set to predict the education levels of individuals based on certain attributes collected from the 1994 census database. You can read more about the dataset here: `https://archive.ics.uci.edu/ml/datasets/Census+Income`.

Stencil Code

We have provided the following stencil code within this file:

- `Model` contains the `LogisticRegression` model you will be implementing.
- `Check Model` contains a series of tests to ensure you are coding your model properly.
- `Main` is the entry point of program which will read in the dataset, run the model, and print the results.

You should not modify any code in `Check Model` and `Main`. If you do for debugging or other purposes, please make sure any additions are commented out in the final handin. All the functions you need to fill in reside in this notebook, marked by `TODO`s. You can see a full description of them in the section below.

The Assignment

In `Model`, there are a few functions you will implement. They are:

- `LogisticRegression`:
  - **train()** uses stochastic gradient descent to train the weights of the model.
  - **loss()** calculates the log loss of some dataset divided by the number of examples.

- **predict()** predicts the labels of data points using the trained weights. For each data point, you should apply the softmax function to it and return the label with the highest assigned probability.
- **accuracy()** computes the percentage of the correctly predicted labels over a dataset.

Note: You are not allowed to use any packages that have already implemented these models (e.g. scikit-learn). We have also included some code in `main` for you to test out the different random seeds and calculate the average accuracy of your model across those random seeds.

Logistic Regression

Logistic Regression, despite its name, is used in classification problems. It learns sigmoid functions of the inputs

$$h_{\bf w}(x)_j = \phi_{sig}(\langle {\bf w}_j, {\bf x} \rangle)$$

where $h_{\bf w}(x)_j$ is the probability that sample $\bf x$ is a member of class *j*.

In multi-class classification, we need to apply the `softmax` function to normalize the probabilities of each class. The loss function of a Logistic Regression classifier over *k* classes on a *single* example (*x*,*y*) is the **log-loss**, sometimes called **cross-entropy loss**:

$$\ell(h_{\bf w}, ({\bf x}, y)) = - \sum_{j = 1}^{k} \left\{\begin{array}{lr} \log(h_{\bf w}({\bf x})_j), & y = j\\ 0, & \text{otherwise} \\ \end{array}\right\}$$

Therefore, the ERM hypothesis of **w** on a dataset of *m* samples has weights

$$ {\bf w} = \underset{\bf w}{argmin} (-\frac{1}{m}\sum_{i = 1}^m \sum_{j = 1}^{k} \left\{\begin{array}{lr} \log(h_{\bf w}({\bf x}_i)_j), & y_{i} = j\\ 0, & \text{otherwise} \\ \end{array}\right\}) $$

To learn the ERM hypothesis, we need to perform gradient descent. The partial derivative of the loss function on a single data point

$$ \frac{\partial L_S(h_{\bf w})}{\partial {\bf w}_{st}} = \left\{\begin{array}{lr} h_{\bf w}({\bf x})_s - 1, & y = s\\ h_{\bf w}({\bf x})_s, & \text{otherwise} \\ \end{array}\right\} {\bf x}_t $$

With respect to a single row in the weights matrix, ${\bf w}_s$, the partial derivative of the loss is

$$ \frac{\partial L_S(h_{\bf w})}{\partial {\bf w}_{s}} = \left\{\begin{array}{lr} h_{\bf w}({\bf x})_s - 1, & y = s\\ h_{\bf w}({\bf x})_s, & \text{otherwise} \\ \end{array}\right\} {\bf x} $$

You will need to descend this gradient to update the weights of your Logistic Regression model.

Stochastic Gradient Descent

You will be using Stochastic Gradient Descent (SGD) to train your `LogisticRegression` model. Below, we have provided pseudocode for SGD on a sample *S*:

$\text{initialize parameters } {\bf w}\text{, learning rate } \alpha \text{, and batch size b}$ converge = False while not converge: epoch + 1 shuffle training examples calculate last epoch loss for $i=0,1,\ldots,\lceil n_{examples}/b \rceil - 1$: – iterate over batches: $X_{batch}=X[i \cdot b:(i+1) \cdot b]$ – select the X in the current batch $\quad \quad \quad {\bf y}_{batch} = {\bf y}[i \cdot b: (i+1) \cdot b] \text{ -- select the labels in the current batch}$ $\quad \quad \quad \text{initialize } \nabla L_{{\bf w}} \text{ to be a matrix of zeros}$ $\quad \quad \quad \text{for each pair of training data point } ({\bf x},y)\in (X_{batch}, {\bf y}_{batch}):$ for $j=0,1,\ldots,n_{classes}-1$: – calculate the partial derivative of the loss with respect to – a single row in the weights matrix $\quad \quad \quad \quad \quad \text{if }y = j: \nabla L_{{\bf w}_j} \text{ += } (softmax(\langle {\bf w}_j, {\bf x} \rangle) - 1) \cdot {\bf x} $ $\quad \quad \quad \quad \quad \text{else: }\nabla L_{{\bf w}_j} \text{ += } (softmax(\langle {\bf w}_j, {\bf x} \rangle)) \cdot {\bf x}$ $\quad \quad \quad {\bf w} = {\bf w} - \frac{\alpha \nabla L_{\bf w}}{len(X_{batch})} \text{ -- update the weights}$ calculate this epoch loss $\quad \quad \text{if |Loss}(X,{\bf y})_{this-epoch}-Loss(X,{\bf y})_{last-epoch}| < \text{CONV-THRESHOLD: }$ converge = True – break the loop if loss converged

Hints: Consistent with the notation in the lecture, ${\bf w}$ are initialized as a k x d matrix, where k is the number of classes and d is the number of features (with the bias term). With n as the number of examples, X is a n x d matrix, and ${\bf y}$ is a vector of length n.

Tuning Parameters

Convergence is achieved when the change in loss between iterations is some small value. Usually, this value will be very close to but not equal to zero, so it is up to you to tune this threshold value to best optimize your model's performance. Typically, this number will be some magnitude of 10-x, where you experiment with *x*. Note that when calculating the loss for checking convergence, you should be calculating the loss for the entire dataset, not for a single batch (i.e., at the end of every epoch).

You will also be tuning batch size (and one of the report questions addresses the impact of batch size on model performance). In order to reach the accuracy threshold, you will need to tune both parameters. α would typically be tuned during the training process, but we are fixing α = 0.03 for this assignment. **Please do not change α in your code**.

You can tune the batch size and convergence threshold in `Main`.

Extra: Numpy Shortcuts

While optional, there are many numpy shortcuts and functions that can make your code cleaner. We encourage you to look up numpy documentation and learn new functions.

Some useful shortcuts:

- `A @ B` is a shortcut for `np.matmul(A, B)`
- `X.T` is a shortcut for `np.transpose(X)`
- `X.shape` is a shortcut for `np.shape(X)`

Model

```
import random
import numpy as np

def softmax(x):
    '''
    Apply softmax to an array
    @params:
        x: the original array
    @return:
        an array with softmax applied elementwise.
    '''
    e = np.exp(x - np.max(x))
    return (e + 1e-6) / (np.sum(e) + 1e-6)

class LogisticRegression:
    '''
    Multiclass Logistic Regression that learns weights using
    stochastic gradient descent.
    '''
    def __init__(self, n_features, n_classes, batch_size,
conv_threshold):
        '''
        Initializes a LogisticRegression classifer.
        @attrs:
            n_features: the number of features in the classification
problem
            n_classes: the number of classes in the classification
problem
            weights: The weights of the Logistic Regression model
            alpha: The learning rate used in stochastic gradient
descent
        '''
        self.n_classes = n_classes
        self.n_features = n_features
        self.weights = np.zeros((n_classes, n_features + 1))  # An
extra row added for the bias
        self.alpha = 0.03  # DO NOT TUNE THIS PARAMETER
        self.batch_size = batch_size
        self.conv_threshold = conv_threshold

    def train(self, X, Y):
        '''
        Trains the model using stochastic gradient descent
        @params:
            X: a 2D Numpy array where each row contains an example,
padded by 1 column for the bias
            Y: a 1D Numpy array containing the corresponding labels
for each example
```

```
        @return:
            num_epochs: integer representing the number of epochs
taken to reach convergence
        '''
        # [TODO]
        converge = False
        num_epoch = 0
        num_example = np.shape(X)[0]
        max_batch = int(np.ceil(num_example/self.batch_size))
        Y_temp = np.reshape(Y,(-1,1))
        dataset = np.hstack((Y_temp,X))

        np.random.shuffle(dataset)

        Y_shuffle = dataset[:,0]
        X_shuffle = dataset[:,1:]

        loss_old = 0

        while converge == False:
            num_epoch += 1

            for i in range(max_batch):
                X_batch = dataset[i*self.batch_size:
(i+1)*self.batch_size,1:]
                Y_batch = dataset[i*self.batch_size:
(i+1)*self.batch_size,0]

                grad_loss = np.zeros((self.n_classes, self.n_features
+ 1))

                for j in range(np.shape(X_batch)[0]):
                    x = X_batch[j,:]
                    y = Y_batch[j]

                    for k in range(self.n_classes):
                        if k==y:
                            grad_loss[k,:] += ( softmax( x @
np.transpose(self.weights) )[k] -1) * x

                        else:
                            grad_loss[k,:] += ( softmax( x @
np.transpose(self.weights) )[k] ) * x

                self.weights += -self.alpha/self.batch_size*grad_loss

            loss_new = self.loss(X_shuffle,Y_shuffle)
            if(abs(loss_new-loss_old)<self.conv_threshold):
                converge=True
```

```
            loss_old = loss_new

        return num_epoch

    def loss(self, X, Y):
        '''
        Returns the total log loss on some dataset (X, Y), divided by
the number of examples.
        @params:
            X: 2D Numpy array where each row contains an example,
padded by 1 column for the bias
            Y: 1D Numpy array containing the corresponding labels for
each example
        @return:
            A float number which is the average loss of the model on
the dataset
        '''
        # [TODO]
        total_log_loss = 0 # The total log loss on dataset

        for i in range(X.shape[0]):
            total_log_loss += -np.log( softmax( X[i,:] @ np.transpose(
self.weights ) ) [int(Y[i])] )

        total_log_loss = total_log_loss/X.shape[0]

        return total_log_loss

    def predict(self, X):
        '''
        Compute predictions based on the learned weigths and examples
X
        @params:
            X: a 2D Numpy array where each row contains an example,
padded by 1 column for the bias
        @return:
            A 1D Numpy array with one element for each row in X
containing the predicted class.
        '''
        # [TODO]
        y_predict = np.argmax( softmax( X @ np.transpose( self.weights
) ), axis=1 )

        return y_predict


    def accuracy(self, X, Y):
        '''
        Outputs the accuracy of the trained model on a given testing
dataset X and labels Y.
```

```
        @params:
            X: a 2D Numpy array where each row contains an example,
padded by 1 column for the bias
            Y: a 1D Numpy array containing the corresponding labels
for each example
        @return:
            a float number indicating accuracy (between 0 and 1)
        '''
        # [TODO]
        y_predict = self.predict(X)

        count = 0
        for i in range (Y.shape[0]):
            if y_predict[i] == Y[i]:
                count +=1

        accuracy = count/Y.shape[0]

        return accuracy
```

Check Model

```
import pytest
# Sets random seed for testing purposes
random.seed(0)
np.random.seed(0)

# Creates Test Model with 2 predictors, 2 classes, a Batch Size of 5
and a Threshold of 1e-2
test_model1 = LogisticRegression(2, 2, 5, 1e-2)

# Creates Test Data
x_bias = np.array([[0,4,1], [0,3,1], [5,0,1], [4,1,1], [0,5,1]])
y = np.array([0,0,1,1,0])
x_bias_test = np.array([[0,0,1], [-5,3,1], [9,0,1], [1,0,1], [6,-
7,1]])
y_test = np.array([0,0,1,0,1])

# Creates Test Model with 2 predictors, 1 classes, a Batch Size of 1
and a Threshold of 1e-2
test_model2 = LogisticRegression(2, 3, 1, 1e-2)

# Creates Test Data
x_bias2 = np.array([[0,0,1], [0,3,1], [4,0,1], [6,1,1], [0,1,1],
[0,4,1]])
y2 = np.array([0,1,2,2,0,1])
x_bias_test2 = np.array([[0,0,1], [-5,3,1], [9,0,1], [1,0,1]])
y_test2 = np.array([0,1,2,0])

# Test Model Loss
```

```
assert test_model1.loss(x_bias, y) == pytest.approx(0.693, .001) #
Checks if answer is within .001
assert test_model2.loss(x_bias2, y2) == pytest.approx(1.099, .001) #
Checks if answer is within .001

# Test Train Model and Checks Model Weights
assert test_model1.train(x_bias, y) == 14
assert test_model1.weights == pytest.approx(np.array([[-0.218, 0.231,
0.0174], [ 0.218, -0.231, -0.0174]]), 0.01) # Answer within .01

assert test_model2.train(x_bias, y) == 9
assert test_model2.weights == pytest.approx(np.array([[-0.300,  0.560,
0.093], [ 0.523, -0.257,  0.032], [-0.226, -0.304, -0.123]]), .05)

# Test Model Predict
assert (test_model1.predict(x_bias_test) == np.array([0., 0., 1., 1.,
1.])).all()
assert (test_model2.predict(x_bias_test2) == np.array([0, 0, 1,
1])).all()

# Test Model Accuracy
assert test_model1.accuracy(x_bias_test, y_test) == .8
assert test_model2.accuracy(x_bias_test2, y_test2) == .25
```

Main

```
from sklearn.model_selection import train_test_split

DATA_FILE_NAME = 'normalized_data.csv'
# DATA_FILE_NAME = 'unnormalized_data.csv'
# DATA_FILE_NAME = 'normalized_data_nosens.csv'

CENSUS_FILE_PATH = DATA_FILE_NAME

NUM_CLASSES = 3
BATCH_SIZE = 1  # [TODO]: tune this parameter
CONV_THRESHOLD = 1 # [TODO]: tune this parameter

def import_census(file_path):
    '''
        Helper function to import the census dataset
        @param:
            train_path: path to census train data + labels
            test_path: path to census test data + labels
        @return:
            X_train: training data inputs
            Y_train: training data labels
            X_test: testing data inputs
            Y_test: testing data labels
    '''
```

```
    data = np.genfromtxt(file_path, delimiter=',', skip_header=False)
    X = data[:, :-1]
    Y = data[:, -1].astype(int)
    X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=0.3, random_state=0)
    return X_train, Y_train, X_test, Y_test

def test_logreg():
    X_train, Y_train, X_test, Y_test = import_census(CENSUS_FILE_PATH)
    num_features = X_train.shape[1]

    # Add a bias
    X_train_b = np.append(X_train, np.ones((len(X_train), 1)), axis=1)
    X_test_b = np.append(X_test, np.ones((len(X_test), 1)), axis=1)

    ### Logistic Regression ###
    model = LogisticRegression(num_features, NUM_CLASSES, BATCH_SIZE,
CONV_THRESHOLD)
    num_epochs = model.train(X_train_b, Y_train)
    acc = model.accuracy(X_test_b, Y_test) * 100
    print("Test Accuracy: {:.1f}%".format(acc))
    print("Number of Epochs: " + str(num_epochs))

# Set random seeds. DO NOT CHANGE THIS IN YOUR FINAL SUBMISSION.
random.seed(0)
np.random.seed(0)
test_logreg()
```

```
Test Accuracy: 74.5%
Number of Epochs: 1
```

Check Model (Cont'd)

```
### test your model on the census dataset
X_train, Y_train, X_test, Y_test = import_census(CENSUS_FILE_PATH)
num_features = X_train.shape[1]

# Add a bias
X_train_b = np.append(X_train, np.ones((len(X_train), 1)), axis=1)
X_test_b = np.append(X_test, np.ones((len(X_test), 1)), axis=1)

# Logistic Regression, average accross 10 random states
random.seed(0)
num_states = 10
num_epochs, test_accuracies = [], []

for _ in range(num_states):
    random_state = random.randint(1, 1000)
    random.seed(random_state)
    np.random.seed(random_state)
```

```
    model = LogisticRegression(num_features, n_classes=3,
batch_size=1, conv_threshold=0.1)
    num_epochs.append(model.train(X_train_b, Y_train))
    test_accuracies.append(model.accuracy(X_test_b, Y_test) * 100)

avg_test_accuracy = sum(test_accuracies) / num_states
avg_num_epochs = sum(num_epochs) / num_states
print("Average Test Accuracy: {:.1f}%".format(avg_test_accuracy))
print("Average Number of Epochs: " + str(avg_num_epochs))

assert 1.5 < avg_num_epochs < 2.5
assert 75 < avg_test_accuracy < 80

Average Test Accuracy: 78.2%
Average Number of Epochs: 2.0
```

Report Questions (15 points)

Question 1

Make sure that you have implemented a variable batch size using the constructor given for `LogisticRegression`. Try different batch sizes ([1, 8, 64, 512, 4096] - there are ~5700 points in the dataset), and try different convergence thresholds ([1e-1, 1e-2, 1e-3]) in the cell below. Visualize the accuracy and number of epochs taken to converge.

Answer the following questions:

- What tradeoffs exist between good accuracy and quick convergence?
- Why do you think the batch size led to the results you received?

Fill in the `generate_array()` and `generate_heatmap()` functions so you can visualize how accuracy and number of epochs taken changes as we change batch size and convergence threshold. Fill out BATCH_SIZE_ARR and CONV_THRESHOLD_ARR with the values described above.

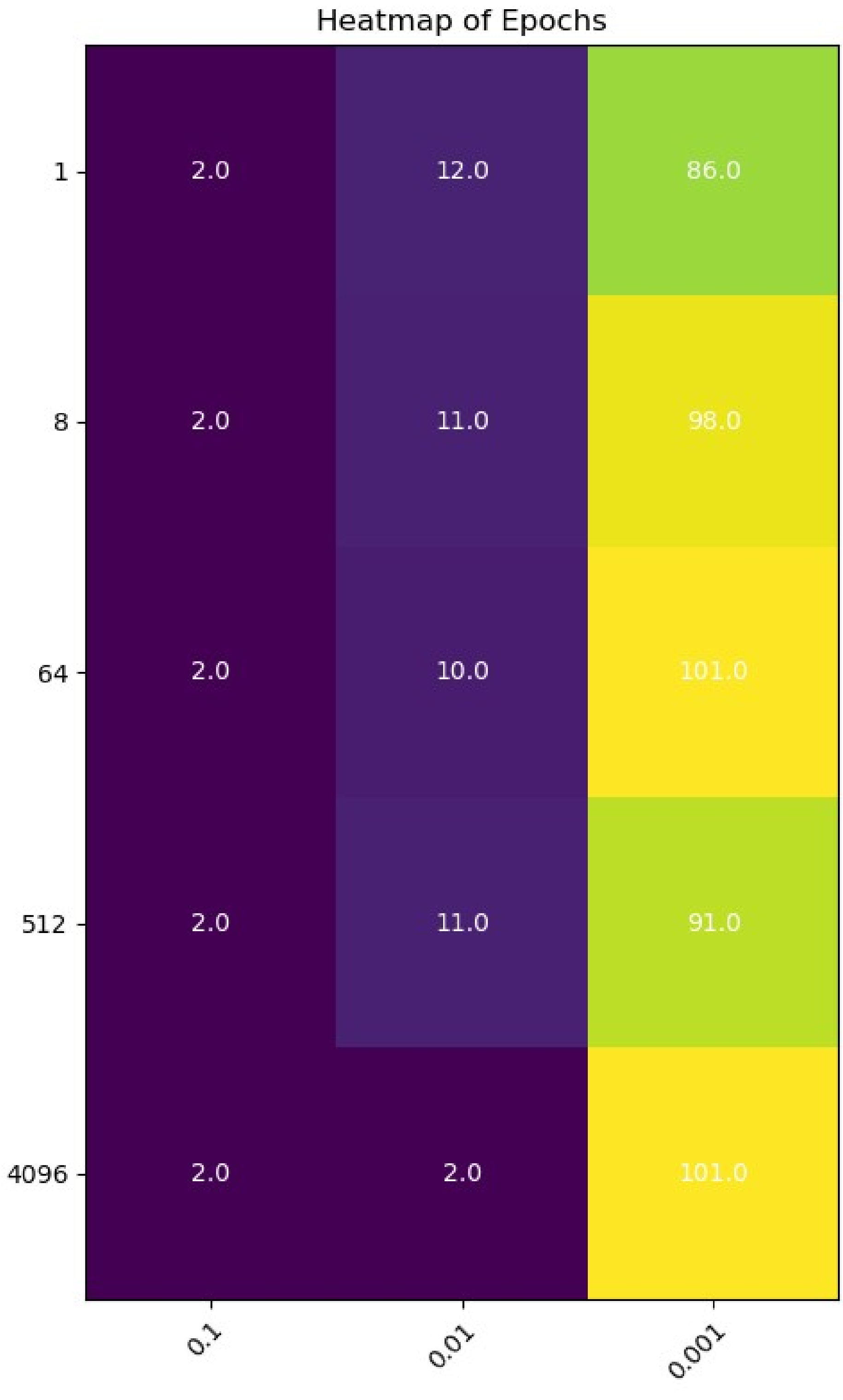
- **generate_array()** should loop through both BATCH_SIZE_ARR and CONV_THRESHOLD_ARR to populate `epoch_arr` and `acc_arr`. Make sure to round `acc_arr` to 2 decimal places before returning (Hint: `np.round`).

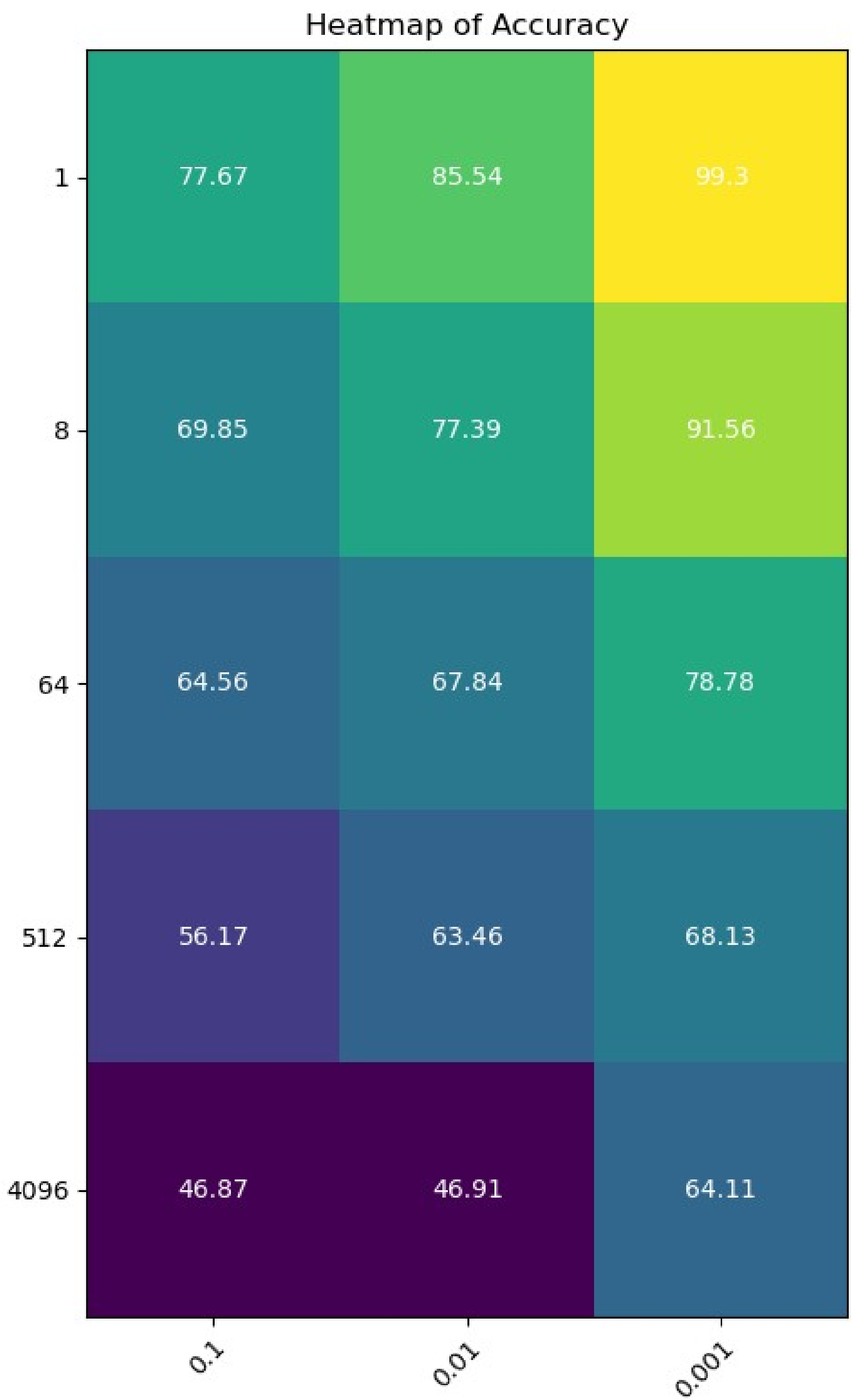
- **generate_heatmap()** should create a matplotlib heatmap of the arrays. You should label the axis and title of each plot using BATCH_SIZE_ARR and CONV_THRESHOLD_ARR. It might be helpful to look at Matplotlib's guide for heatmaps: https://matplotlib.org/stable/gallery/images_contours_and_fields/image_annotated_heatmap.html

Hint: Runs with large batch sizes and low convergence thresholds might take several minutes to half an hour to complete. We recommend that you develop the code below with a small subset of the parameters (e.g., batch size of [1,2,4] and conv_threshold of [1e-1, 1e-2]). Once your code works and your figures look good, rerun everything with the batch size and conv_threshold values described above.

```
import matplotlib.pyplot as plt

random.seed(0)
np.random.seed(0)

BATCH_SIZE_ARR = [1, 8, 64, 512, 4096] # [TODO]: try different values
CONV_THRESHOLD_ARR = [1e-1, 1e-2, 1e-3]  # [TODO]: try different
values

def generate_array():
    '''
        Runs the logistic regression model on different batch sizes
and
        convergence thresholds to populate arrays for accuracy and
number of epochs taken.
        @return:
            epoch_arr: 2D array of epochs taken, for each batch size
and conv threshold
            acc_arr: 2D array of accuracies, for each batch size and
conv threshold
    '''
    X_train, Y_train, X_test, Y_test = import_census(CENSUS_FILE_PATH)
    num_features = X_train.shape[1]

    # Add a bias
    X_train_b = np.append(X_train, np.ones((len(X_train), 1)), axis=1)
    X_test_b = np.append(X_test, np.ones((len(X_test), 1)), axis=1)

    # Initializes the accuracy and epoch arrays
    acc_arr = np.zeros((len(BATCH_SIZE_ARR), len(CONV_THRESHOLD_ARR)))
    epoch_arr = np.zeros((len(BATCH_SIZE_ARR),
len(CONV_THRESHOLD_ARR)))

    ### Populate arrays ###
    # [TODO]
    num_epochs, test_accuracies = [], []

    for idx1, BATCH_SIZE in enumerate(BATCH_SIZE_ARR):
        for idx2, CONV_THRESHOLD in enumerate(CONV_THRESHOLD_ARR):

            model = LogisticRegression(num_features, n_classes=3,
batch_size=BATCH_SIZE, conv_threshold=CONV_THRESHOLD)

            epoch_arr[idx1,idx2] = model.train(X_train_b, Y_train)
            acc_arr[idx1,idx2] = model.accuracy(X_test_b, Y_test) *
100
```

```
    acc_arr = np.round(acc_arr,2)
    return epoch_arr, acc_arr

def generate_heatmap(arr, name):
    '''
        Generates a matplotlib heatmap for an array
        convergence thresholds to populate arrays for accuracy and
number of epochs taken.
        @param:
            arr: 2D array to generate heatmap of
            name: title of the plot (Hint: use plt.title)
        @return:
            None
    '''
    # [TODO]
    fig, ax = plt.subplots(figsize=(5,8))
    im = ax.imshow(arr)

    ax.set_xticks(np.arange(len(CONV_THRESHOLD_ARR)),
labels=CONV_THRESHOLD_ARR)
    ax.set_yticks(np.arange(len(BATCH_SIZE_ARR)),
labels=BATCH_SIZE_ARR)

    plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
             rotation_mode="anchor")

    for i in range(len(BATCH_SIZE_ARR)):
        for j in range(len(CONV_THRESHOLD_ARR)):
            text = ax.text(j, i, arr[i, j],
                           ha="center", va="center", color="w")

    ax.set_title("Heatmap of " + str(name))
    fig.tight_layout()
    plt.show()

    return

epoch_arr, acc_arr = generate_array()
generate_heatmap(epoch_arr, "Epochs")
generate_heatmap(acc_arr, "Accuracy")
```


Heatmap of Epochs
1
8
64
512
4096
2.0
12.0
86.0
2.0
11.0
98.0
2.0
10.0
101.0
2.0
11.0
91.0
2.0
2.0
101.0
0.1
0.01
0.001



Solution: When training machine learning models, there's a delicate balance between achieving good accuracy and ensuring quick convergence.

Batch Size Effects:

- Smaller Batch Sizes: Using small batches (like 1) generates noisy estimates of the gradient. This randomness can slow down the convergence process, as the updates are less stable. However, this same noise might help the model avoid overfitting, potentially leading to better generalization and higher accuracy on unseen data.
- Larger Batch Sizes: In contrast, larger batches (e.g., 4096) produce smoother gradient updates. This tends to accelerate convergence since the model receives more reliable gradient information. However, this stability can also make it easier for the model to settle into local minima, which might impair its ability to generalize and ultimately decrease test accuracy.

Convergence Thresholds:

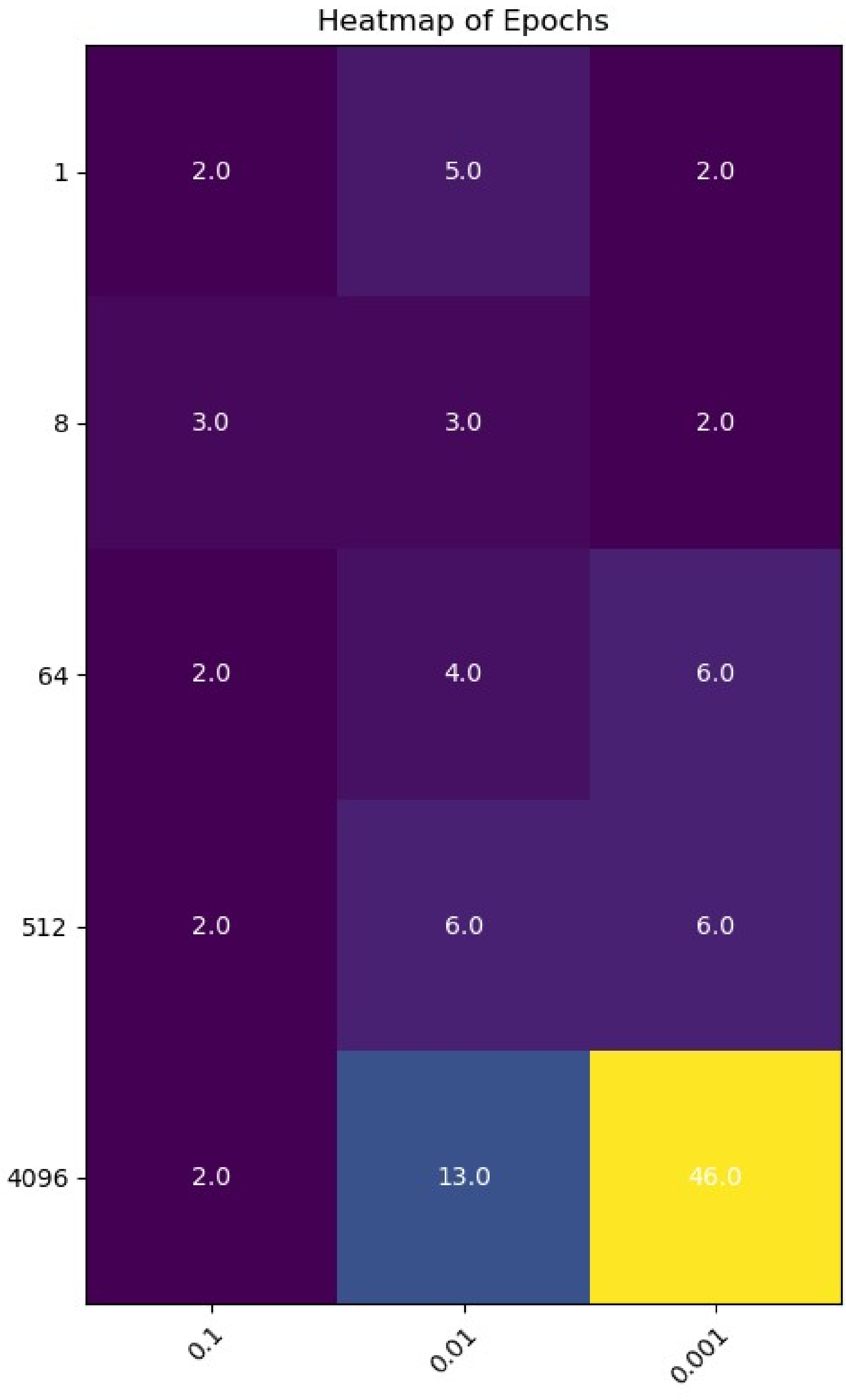
- Higher Convergence Thresholds: Setting a higher threshold for convergence allows the training process to finish sooner. While this can speed things up, the model may not fully optimize its performance, resulting in lower accuracy.
- Lower Convergence Thresholds: Conversely, a lower threshold requires more training epochs to reach convergence. This more stringent requirement often leads to better accuracy as the model has more time to refine its parameters.
- Why do you think the batch size led to the results you received?

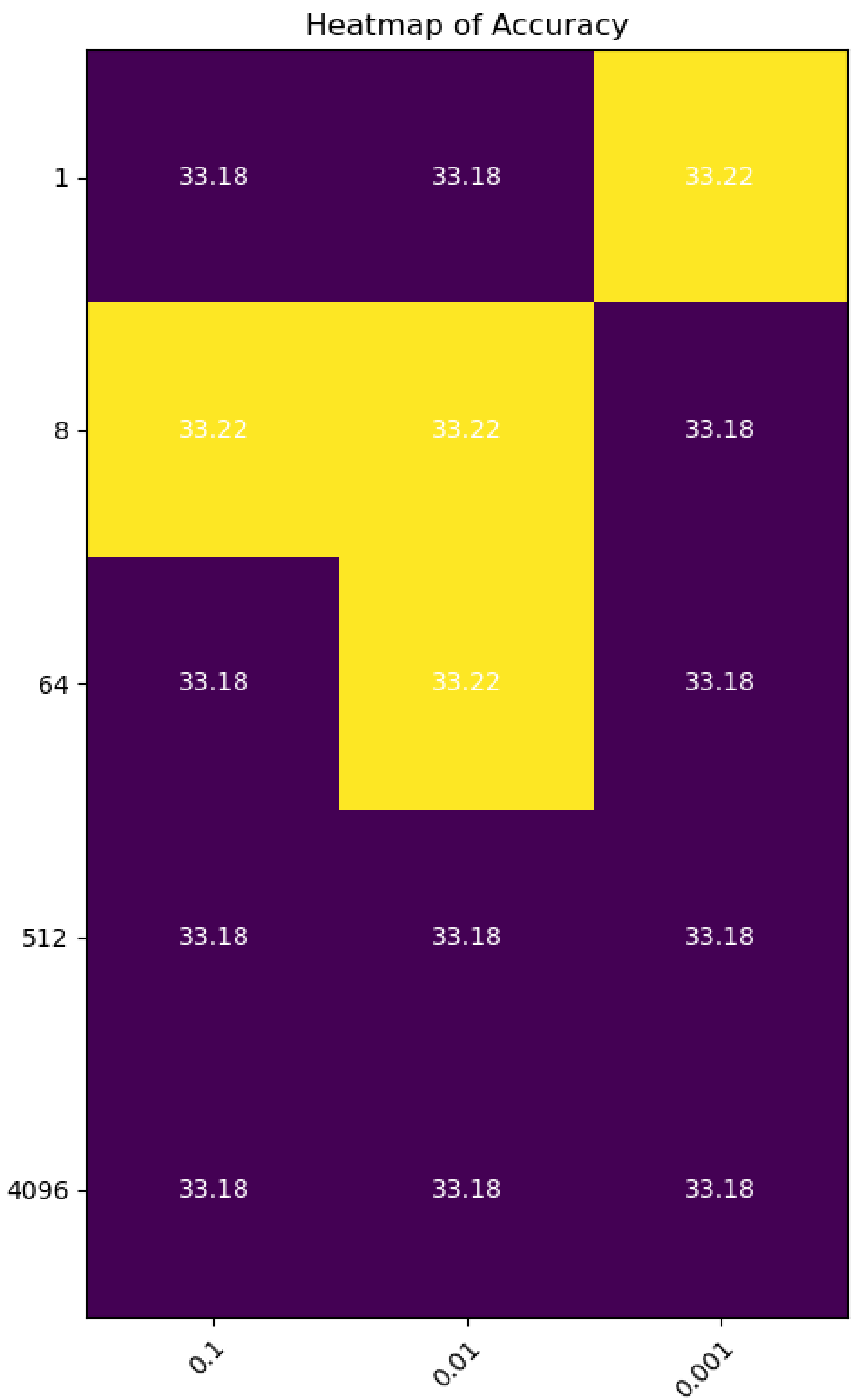
The impact of batch size and convergence thresholds stems from how they influence the training dynamics. Smaller batches inject variability into the learning process, which can help in escaping local minima but can also prolong training. Larger batches, on the other hand, provide clearer signals for the optimizer, leading to faster convergence at the risk of overfitting.

Question 2

Try to run the model with `unnormalized_data.csv` instead of `normalized_data.csv`. Report your findings when running the model on the unnormalized data. In a few short sentences, explain what normalizing the data does and why it affected your model's performance.

```
DATA_FILE_NAME = 'unnormalized_data.csv'
CENSUS_FILE_PATH = DATA_FILE_NAME

NUM_CLASSES = 3
BATCH_SIZE_ARR = [1, 8, 64, 512, 4096] # [TODO]: try different values
CONV_THRESHOLD_ARR = [1e-1, 1e-2, 1e-3]  # [TODO]: try different
values

epoch_arr, acc_arr = generate_array()
generate_heatmap(epoch_arr, "Epochs")
generate_heatmap(acc_arr, "Accuracy")
```


Heatmap of Epochs
1
8
64
512
4096
0.1
0.01
0.001
2.0
5.0
2.0
3.0
3.0
2.0
2.0
4.0
6.0
2.0
6.0
6.0
2.0
13.0
46.0



Solution:

1. Slower Convergence :

- The model is take more epochs to converge compared to the normalised data. This is because the unnormalized data has features with widely varying scales, which leads to inefficient gradient updates.
- Features with larger values will dominate the gradient calculations, which can cause the optimisation to take very large or very small steps, resulting in slower convergence.

2. Bad Accuracy:
   - The overall accuracy of the model on the test set is decreasing. Since the logistic regression weights are updated based on the gradient of the cost function, unnormalised data can lead to poorly balanced weight updates across features, meaning the model won't learn equally from all features, which is leading to worse predictions.
3. More Oscillations:
   - The model is oscillating more between values without reaching convergence effectively. This occurs due to the differences in magnitude between feature values, leading the optimizer to overshoot or undershoot the minimum.

To answer the question why Normalisation Improves Model Performance,

Normalization refers to scaling the data such that each feature has the same scale (e.g., by converting all features to have a mean of 0 and a standard deviation of 1 or by scaling values between 0 and 1).

1. Prevents Dominance of Large-Scale Features : Without normalisation, features with larger values will dominate the gradient and bias the learning process, as gradient descent takes bigger steps for larger values. This can cause the model to take longer to converge or get stuck in suboptimal solutions.

2. Faster and More Stable Convergence : When the data is normalised, all features are on a similar scale, which is allowing the gradient descent algorithm to make progress across all features, which is leading to fast convergence.

3. Better Generalisation : Normalised data ensures that the model gives equal importance to all features, leading to better learned weights and improved generalisation on unseen data (i.e., the test set).

Question 3

Try the model with `normalized_data_nosens.csv`; in this data file, we have removed the `race` and `sex` attributes. Report your findings on the accuracy of your model on this dataset (averaging over many random seeds here may be useful). Can we make any conclusion based on these accuracy results about whether there is a correlation between sex/race and education level? Why or why not?

```
DATA_FILE_NAME = 'normalized_data_nosens.csv'
CENSUS_FILE_PATH = DATA_FILE_NAME

### test your model on the census dataset
X_train, Y_train, X_test, Y_test = import_census(CENSUS_FILE_PATH)
```

```
num_features = X_train.shape[1]

# Add a bias
X_train_b = np.append(X_train, np.ones((len(X_train), 1)), axis=1)
X_test_b = np.append(X_test, np.ones((len(X_test), 1)), axis=1)

# Logistic Regression, average accross 10 random states
random.seed(0)
num_states = 10
num_epochs, test_accuracies = [], []

for _ in range(num_states):
    random_state = random.randint(1, 1000)
    random.seed(random_state)
    np.random.seed(random_state)

    model = LogisticRegression(num_features, n_classes=3,
batch_size=1, conv_threshold=0.1)
    num_epochs.append(model.train(X_train_b, Y_train))
    test_accuracies.append(model.accuracy(X_test_b, Y_test) * 100)

avg_test_accuracy = sum(test_accuracies) / num_states
avg_num_epochs = sum(num_epochs) / num_states
print("Average Test Accuracy: {:.1f}%".format(avg_test_accuracy))
print("Average Number of Epochs: " + str(avg_num_epochs))

Average Test Accuracy: 77.6%
Average Number of Epochs: 2.0
```

Solution:

- Findings on the accuracy of your model on this dataset :

When running the logistic regression model with normalized_data_nosens.csv we observe a slight decrease in accuracy compared to the model trained on the full dataset (with all features including race and sex).

1. Slight Accuracy Drop :
   - The accuracy of the model is dropping somewhat when the race and sex attributes are removed, as these features might have provided additional predictive information in the original model. This can be measured by running the model over multiple random seeds and averaging the accuracy results to ensure consistency.
2. Sensitivity of the Model :
   - When some features are removed, the model loses some of its predictive power, particularly these attributes are correlated with education level in the dataset.

- Can We Conclude There Is a Correlation Between Sex/Race and Education Level?

The results of the accuracy drop alone do not allow us to definitively conclude that there is a causal relationship between sex/race and education level, because,

1. Correlation vs Causation :
   - Just because removing race and sex from the dataset reduces accuracy does not mean there is a direct causal relationship between those attributes and education level. The drop in accuracy might suggest that race and sex are correlated with other factors that are more directly related to education level, but this correlation does not imply causation.
2. Confounding Variables :
   - Education level could be influenced by a variety of factors such as socioeconomic status, geographic location, access to resources, and historical disparities, which may themselves be correlated with race and sex. Therefore, the removal of race and sex attributes can decrease accuracy if those attributes indirectly capture some of these other factors.
3. Bias in the Dataset :
   - The dataset is reflecting biases in the real world where race and sex are associated with different educational opportunities. This could explain why the model performs worse without those features, but again, this reflects existing patterns in the data and not necessarily inherent relationships between race/sex and education level.

CONCLUSION :

The decrease in model accuracy when race and sex attributes are removed suggests that these features may carry some predictive power regarding education level. However, we cannot conclude that there is a direct causal relationship between race/sex and education level based on this finding alone. The correlation observed in the data may be due to underlying societal, economic, or historical factors that affect both race/sex and education level.