Homework 3

Due: **October 8th, 5pm** (late submission until October 11th, 5pm -- no submission possible afterwards)

Written assignment: 10 points

Coding assignment: 25 points

Project report: 15 points

Name: [Yawen]

Link to the github repo: [https://github.com/IsabellaTan/Brown-DATA2060-HW3.git]

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

$$|x_{t'} - x_{\min}| \leq \epsilon, \quad \text{where } x_{\min} = \underset{x \in [-b,b]}{argmin} f(x) \text{ for all } t' \geq t.$$

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 x_0 = 0, x_1 = b, x_2 = - b, x_3 = b, x_4 = - b, etc.

Solution:

Define function f(x) = 15 x^2 - 10x over the close interval [-1, 1] where b = 1

Firstly we check if f(x) is convex.

Since f(x) = 15 x^2 - 10x, then f'(x) = 30x - 10 which monotonically nondecrease, and f''(x) = 30 which is nonnegative, so that f(x) is convex function.

Secondly we check if gradient descent doesn't ϵ-converge.

Since f'(x) = 30x - 10 and $x_{t+1} = x_t - \alpha f'(x_t)$ and α = 0.1, then $x_{t+1} = x_t - 0.1(30x_t - 10)$ = $-2x_t$ + 1

Let x_0 = 0

x_1 = $-2x_0$ + 1 = 1

x_2 = $-2x_1$ + 1 = -1

x_3 = $-2x_2$ + 1 = 3 = 1 because when goes above b, set it to b

x_4 = $-2x_3$ + 1 = -1

x_5 = $-2x_4$ + 1 = 3 = 1

.....

Since x_i is always equal to 1 or -1, then f gradient descent doesn't ϵ-converge.

Thus, convex function f(x) = 15 x^2 - 10x so that running gradient descent does not ϵ-converge.

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:

Define function f(x) = $0.001(x-1)^2$ over the close interval [-1, 1] where b = 1

Firstly we check if f(x) is convex.

Since f(x) = $0.001(x-1)^2$, then f'(x) = 0.002(x-1) which monotonically nondecrease, and f''(x) = 0.002 which is nonnegative, so that f(x) is convex function.

Secondly we check if gradient descent does ϵ-converge for ϵ = 0.001

Since f(x) = $0.001(x-1)^2$, then x_{min} = 1, so that ϵ-convergence at time t is $|x_{t'} - 1| \leq 0.001$ for all $t' \geq t$.

Let $d_t = |x_t - 1|$ be the distance between x_t the minimum (x_{min} = 1)

Since f'(x) = 0.001x and $x_{t+1} = x_t - \alpha f'(x_t)$ and α = 0.1, then $x_{t+1} = x_t - 0.0002(x_t - 1)$, so that we can get:

-> $x_{t+1} - x_t$ = -0.0002(x_t - 1)

-> x_{t+1} – 1 = 0.9998(x_t –1)

Initial point: $x_0 = 0, d_0 = |\ x_0 - 1\ | = |\ 0 - 1\ | = 1$

Step 1: $x_1 - 1 = 0.9998(x_0 - 1) = 0.9998(-1) = -0.9998$, then $d_1 = |\ x_1 - 1\ | = |\ -0.9998\ | = 0.9998$

Step 2: $x_2 - 1 = 0.9998(x_1 - 1) = 0.9998(-0.9998) \approx -0.9996$, then $d_2 = 0.9996$

Step 3: $x_3 - 1 = 0.9998(x_2 - 1) = 0.9998(-0.9996) = 0.9998 * 0.9998 * (-0.9998) \approx -0.9994$, then $d_3 = 0.9994$

.....

We can find $d_t = 0.9998^t$ which is the distance between x_t the minimum.

When $d_t \leq 0.001$, $0.9998^t \leq 0.001$. Then when $0.9998^t = 0.001$, t ≈ 34500 > 10000 steps

Thus, convex function f(x) = $0.001(x-1)^2$ so that gradient descent does ϵ-converge after 34500 steps (>10000 steps).

Coding Assignment (25 points)

Run the evironment test below, make sure you get all green checks. If not, you will lose 2 points for each red or missing sign.

In [1]:
```
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 3.12.11 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("3.12.11"):
    print(OK, "Python version is %s" % pyversion)
elif pyversion < Version("3.12.11"):
    print(FAIL, "Python version 3.12.11 is required,"
                " but %s is installed." % pyversion)
else:
    print(FAIL, "Unknown Python version: %s" % pyversion)


print()
requirements = {'matplotlib': "3.10.5", 'numpy': "2.3.2",'sklearn': "1.7.1",
                'pandas': "2.3.2", 'pytest': "8.4.1", 'torch':"2.7.1"}

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

```
[ OK ] Python version is 3.12.11

[ OK ] matplotlib version 3.10.5 is installed.
[ OK ] numpy version 2.3.2 is installed.
[ OK ] sklearn version 1.7.1 is installed.
[ OK ] pandas version 2.3.2 is installed.
[ OK ] pytest version 8.4.1 is installed.
[ OK ] torch version 2.7.1 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

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

where $h_{\mathbf{w}}(x)_j$ is the probability that sample $\mathbf{x}$ 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:**

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

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

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

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

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

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

$$\frac{\partial l_S(h_{\mathbf{w}})}{\partial \mathbf{w}_s} = \left\{ \begin{matrix} h_{\mathbf{w}}(\mathbf{x})_s - 1, & y = s \\ h_{\mathbf{w}}(\mathbf{x})_s, & \text{otherwise} \end{matrix} \right\} \mathbf{x}$$

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*:

initialize parameters $\mathbf{w}$, learning rate α, 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
      $\mathbf{y}_{batch} = \mathbf{y}[i \cdot b : (i+1) \cdot b]$ -- select the labels in the current batch
      initialize $\nabla L_{\mathbf{w}}$ to be a matrix of zeros
      for each pair of training data point $(\mathbf{x}, y) \in (X_{batch}, \mathbf{y}_{batch})$:
        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
          if $y = j$: $\nabla L_{\mathbf{w}_j} \mathrel{+}= (softmax(\langle \mathbf{w}_j, \mathbf{x} \rangle) - 1) \cdot \mathbf{x}$
          else: $\nabla L_{\mathbf{w}_j} \mathrel{+}= (softmax(\langle \mathbf{w}_j, \mathbf{x} \rangle)) \cdot \mathbf{x}$
      $\mathbf{w} = \mathbf{w} - \frac{\alpha \nabla L_{\mathbf{w}}}{len(X_{batch})}$ -- update the weights
    calculate this epoch loss
    if $|\text{Loss}(X, \mathbf{y})_{this-epoch} - Loss(X, \mathbf{y})_{last-epoch}| <$ CONV-THRESHOLD:
      converge $=$ True -- break the loop if loss converged

Hints: Consistent with the notation in the lecture, $\mathbf{w}$ 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 $\mathbf{y}$ 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

In [2]:
```
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]
        num_samples = X.shape[0] # get the number of samples
        num_epochs = 0 # let the number of epochs be 0 initially
        prev_loss = self.loss(X, Y)  # calculate the initial loss
        np.random.seed(42) # set seed for reproducibility
        while True:
            num_epochs += 1 # number of epochs increases 1 for each loop
```

```
            indices = np.arange(num_samples) # get all the indices of samples
            np.random.shuffle(indices) # shuffle the indices
            # shuffle the samples according to the shuffled indices
            X = X[indices]
            Y = Y[indices]

            # run over each batch
            for i in range(0, num_samples, self.batch_size):
                # get the current batch
                X_batch = X[i:i+self.batch_size]
                Y_batch = Y[i:i+self.batch_size]
                # intial the gradient to be zero matrix
                grad = np.zeros_like(self.weights)

                # run over each sample in the batch
                for x, y in zip(X_batch, Y_batch):
                    # calculate ⟨w,x⟩
                    scores = np.dot(self.weights, x)
                    # calculate the softmax probabilities
                    probs = softmax(scores)

                    # run over each class to calculate the gradient
                    for j in range(self.n_classes):
                        if j == y:
                            grad[j] += (probs[j] - 1) * x
                        else:
                            grad[j] += probs[j] * x

                # calculate the average gradient for the batch
                grad /= len(X_batch)
                # update weights
                self.weights -= self.alpha * grad

            # calculate the current loss after each epoch
            curr_loss = self.loss(X, Y)
            # check for convergence
            if abs(prev_loss - curr_loss) < self.conv_threshold: # if covergence, then break the loop
                break
            # if not convergence, then set the previous loss to current loss for next epoch
            prev_loss = curr_loss
```

```
        return num_epochs



    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_loss = 0 # set the total loss to be 0 initially
        n = X.shape[0] # number of samples

        # run over each sample to calculate the loss
        for i in range(n):
            x = X[i] # x is the i-th sample
            y = Y[i] # y is the label of the i-th sample
            # calculate ⟨w,x⟩
            scores = np.dot(self.weights, x)
            # calculate the softmax probabilities for each class
            probs = softmax(scores)
            # accumulate the loss
            total_loss += -np.log(probs[y] + 1e-6)
        # return the average loss
        return total_loss / n


    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]
        predictions = [] # set a list to store the predictions
```

```
        # run over each sample to make predictions
        for x in X:
            # calculate ⟨w,x⟩
            scores = np.dot(self.weights, x)
            # calculate the softmax probabilities for each class
            probs = softmax(scores)
            # get the class with the highest probability
            pred_label = np.argmax(probs)
            # append the predicted label to the predictions list
            predictions.append(pred_label)
        # return the predictions as a numpy array
        return np.array(predictions)


    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]
        preds = self.predict(X)
        # calculate the number of correct predictions
        correct = np.sum(preds == Y)
        # return the accuracy
        return correct / len(Y)
```

Check Model

In [ ]:

```
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

In [ ]:

```
from sklearn.model_selection import train_test_split

DATA_FILE_NAME = 'normalized_data.csv'
# Question 2
# DATA_FILE_NAME = 'unnormalized_data.csv'
# Question 3
# DATA_FILE_NAME = 'normalized_data_nosens.csv'

CENSUS_FILE_PATH = DATA_FILE_NAME

NUM_CLASSES = 3
BATCH_SIZE = 10  # [TODO]: tune this parameter
CONV_THRESHOLD = 0.00001 # [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: 94.8%
Number of Epochs: 230
```

Check Model (Cont'd)

In [29]:

```
### 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: 76.2%
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:

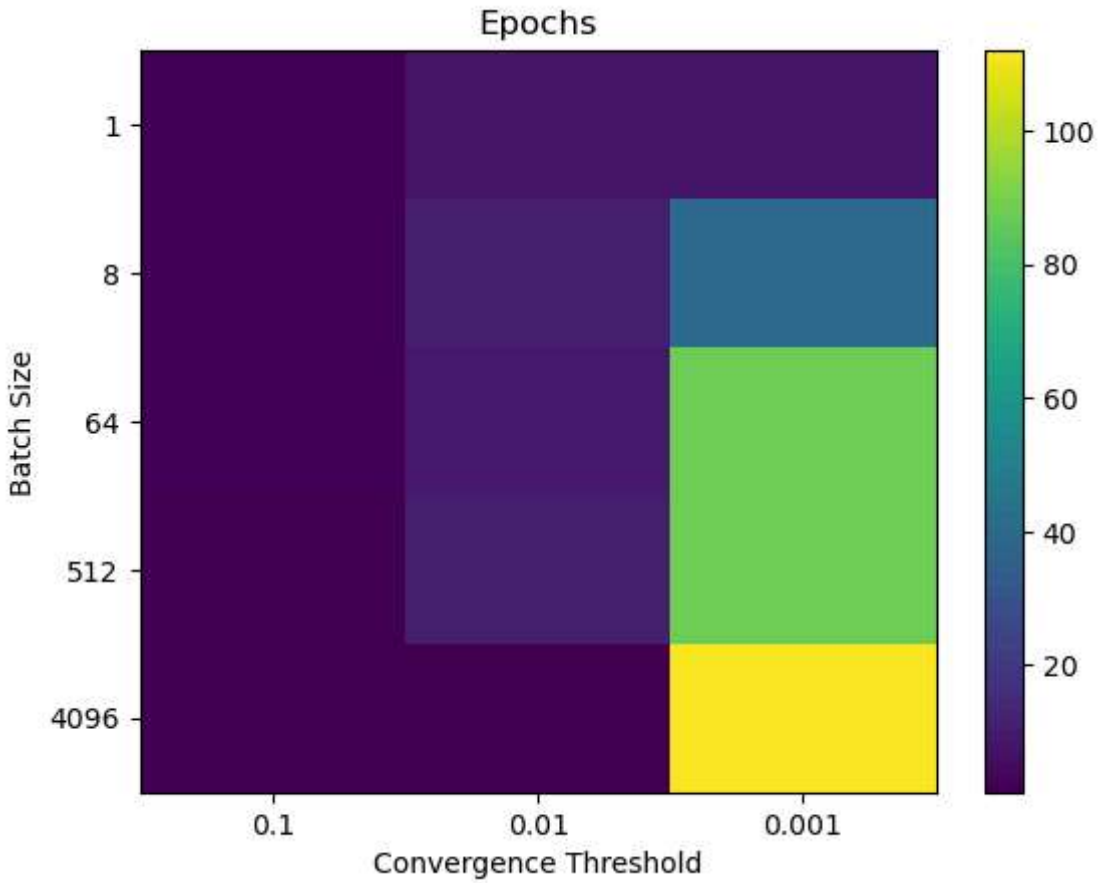
https://matplotlib.org/stable/gallery/images_contours_and_fields/image_annotated_heatmap.html

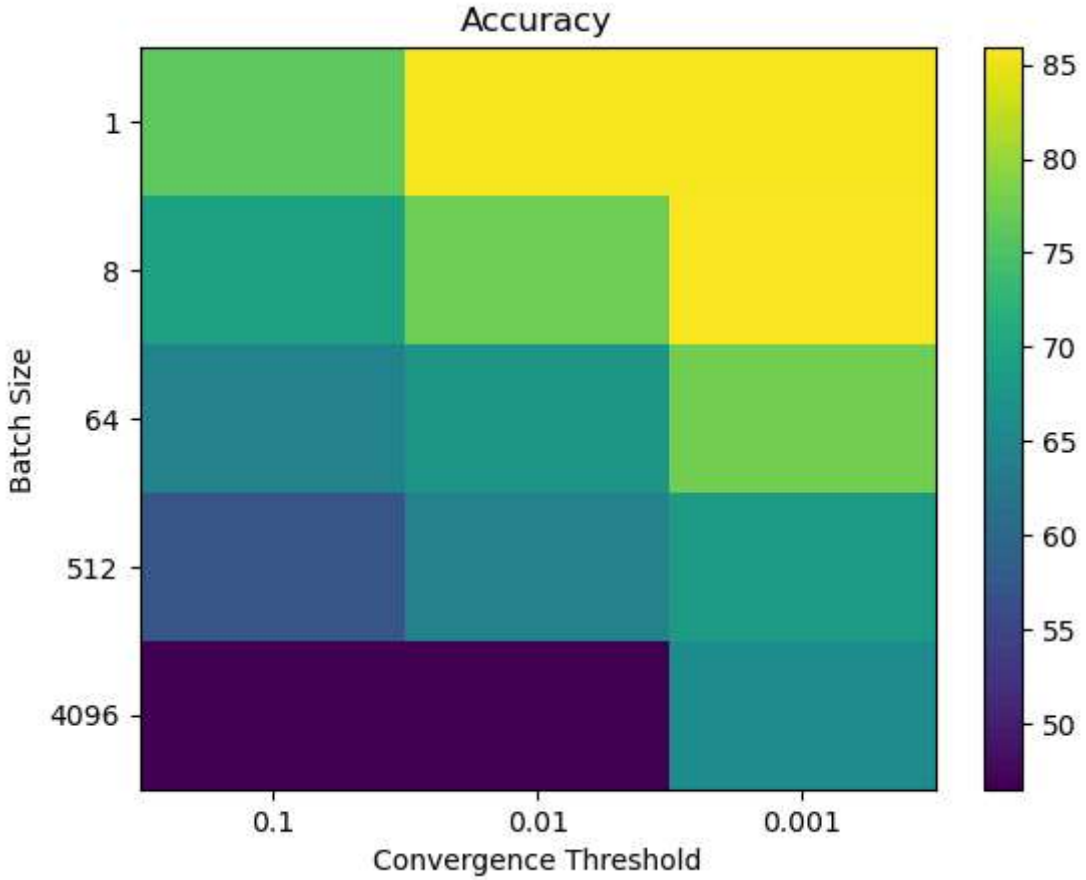
Hint: Runs with large batch sizes and low convergence thresholds might take several minutes to half an hour to complete. We recommend that you develop the code below with a small subset of the parameters (e.g., batch size of [1,2,4] and conv_threshold of [1e-1, 1e-2]). Once your code works and your figures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

In [23]:
```
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]
        # run over each batch size
    for i, batch_size in enumerate(BATCH_SIZE_ARR):
        # run over each convergence threshold
```

```
        for j, conv_threshold in enumerate(CONV_THRESHOLD_ARR):
            # create a logistic regression model for each combination of batch size and conv threshold
            # we defined NUM_CLASSES = 3 previously
            model = LogisticRegression(num_features, NUM_CLASSES, batch_size, conv_threshold)
            # get the number of epoch
            num_epochs = model.train(X_train_b, Y_train)
            # calcuate the accuracy
            acc = model.accuracy(X_test_b, Y_test) * 100
            # save the number of epochs and accuracy to the corresponding arrays
            epoch_arr[i, j] = num_epochs
            acc_arr[i, j] = round(acc, 2)  # round to 2 decimal places

    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]
    plt.figure()
    plt.imshow(arr, cmap='viridis', aspect='auto')
    plt.colorbar()  # display the color bar on the right
    # set x ticks and labels
    plt.xticks(ticks=range(len(CONV_THRESHOLD_ARR)), labels=CONV_THRESHOLD_ARR)
    # set y ticks and labels
    plt.yticks(ticks=range(len(BATCH_SIZE_ARR)), labels=BATCH_SIZE_ARR)
    # set x and y labels
    plt.xlabel('Convergence Threshold')
    plt.ylabel('Batch Size')
    # set title
    plt.title(name)
    plt.show()
```

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





Solution:

- What tradeoffs exist between good accuracy and quick convergence?

Small batch size and small convergence threshold lead to good accuracy; large batch size and large convergence threshold lead to bad accuracy. However, small convergence threshold lead to large epochs (slow convergence) and lead to good accuracy; large convergence threshold lead to small epochs (quick convergence) and lead to bad accuracy. Then we need to balance batch size and convergence threshold in order to balance good accuracy and quick convergence.

- Why do you think the batch size led to the results you received?

Small batch size leads to good accuracy because each update uses only a few samples, which introduces some noise in the gradient. This “shaking” helps the model escape bad solutions and find better ones, improving test accuracy. But large batch size leads to bad accuracy because each update uses many samples, making the gradient very stable. This can make the model get stuck in suboptimal solutions and overfit the training data, lowering test accuracy.

Small batch sizes mean that only a small number of examples are used to update parameters at a time, which can lead to gradient fluctuations and require more rounds of training to converge. Consequently, the number of epochs may be higher. Large batch sizes, which use a large number of examples at a time, stabilize gradients and sometimes allow for rapid convergence. However, they can also lead to suboptimal solutions due to a lack of exploration, resulting in erratic epoch times. Gradient fluctuations and the convergence threshold jointly influence the number of training rounds, so the relationship between batch size and epochs is sometimes directly proportional and sometimes inversely proportional.

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.

Solution:
I set BATCH_SIZE = 10 and CONV_THRESHOLD = 0.00001.

When I run the normalized_data.csv, Test Accuracy: 94.8%, Number of Epochs: 230, Average Test Accuracy: 76.2% and Average Number of Epochs: 2.0

However when the dataset is changed to normalized_data.csv, Test Accuracy: 34.1%, Number of Epochs: 11, Average Test Accuracy: 33.6% and Average Number of Epochs: 9.0 which raise AssertError (because assert 1.5 < avg_num_epochs < 2.5; assert 75 < avg_test_accuracy < 80)

Thus, data normalization scales all features to a similar range, making gradient descent more stable. Without normalization, the scales of features vary widely, and the gradients are dominated by large features, resulting in slow model convergence and low accuracy. While normalization improves training efficiency and predictive performance, models often perform poorly on unnormalized data.

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?

Solution:

I set BATCH_SIZE = 10 and CONV_THRESHOLD = 0.00001.

When I run the normalized_data.csv, Test Accuracy: 94.8%, Number of Epochs: 230, Average Test Accuracy: 76.2% and Average Number of Epochs: 2.0

However,When I run the normalized_data_nosens.csv, Test Accuracy: 94.0%, Number of Epochs: 239, Average Test Accuracy: 76.9% and Average Number of Epochs: 2.0.

After removing race and sex, the model's average test accuracy barely decreased (from 76.2% to 76.9%), and the number of training epochs remained largely unchanged. This suggests that race and sex contribute very little to predicting education level, and the model achieves nearly the same accuracy without these sensitive features. Therefore, model accuracy alone cannot directly prove that race or sex are uncorrelated with education level, but they are not the primary determinants of model predictions.