CSCI-UA.0473-001 Introduction to Machine Learning

Homework 1

Name: Lejia Hu

Due: September 30, 2020

Goal: The goal of this homework is to practice implementing a logistic regression model and gradient descent as well as to explore some theoretical concepts.

You will need the following packages below to do the homework. Please DO NOT import any other packages.

In [2]:
```
pip install "autograd==1.3"
```

```
Requirement already satisfied: autograd==1.3 in /Users/lejiahu/opt/anaconda3/lib/python3.8/site-packages (1.3)
Requirement already satisfied: future>=0.15.2 in /Users/lejiahu/opt/anaconda3/lib/python3.8/site-packages (from autograd==1.3) (0.18.2)
Requirement already satisfied: numpy>=1.12 in /Users/lejiahu/opt/anaconda3/lib/python3.8/site-packages (from autograd==1.3) (1.18.5)
Note: you may need to restart the kernel to use updated packages.
```

In [3]:
```
from autograd import grad
import autograd.numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal as mvn
```

Problem 1: Gradient Descent (30 pts total)

In this problem you will implement gradient descent as a general purpose optimization algorithm.

Part (a) (5 pts) Implementing gradient descent with a fixed learning rate

Using autograd to compute the derivative, implement gradient descent with a fixed learning rate `lr` for a general scalar function `fun`.

In [4]:
```
"""
Perform a fixed number of iterations of gradient descent on a function using a fixed learning rate.

Input:
    fun : function handle: function that takes in a numpy array of shape (d,) and returns a float
    x0  : initial point: numpy array of shape (d,)
    lr  : fixed learning rate: a positive float
    iterations : number of iterations to perform: int

Return:
    x   : minimizer to fun: numpy array of shape (d,)
"""
def gd_fixed(fun, x0, lr, iterations):
    #compute gradient module using autograd
    gradient = grad(fun)
    #run the gradient descent loop
    for k in range(iterations):
        x0 = x0 - lr*gradient(x0)
    #set the final x0 to x
    x = x0
    return x
```

Part (b) (10 pts) Using a variable learning rate

Sometimes it is necessary to decrease our learning rate as we iterate to help gradient descent converge. Implement gradient descent below where the learning rate at iteration i is given by

$$\mathrm{lr}_i = \frac{\mathrm{lr}}{i+1}.$$

In [5]:
```
"""
Perform a fixed number of iterations of gradient descent on a function using a variable learning rate.

Input:
    fun : function handle: function that takes in a numpy array of shape (d,) and returns a float
    x0  : initial point: numpy array of shape (d,)
    lr  : initial learning rate: a positive float
    iterations : number of iterations to perform: int

Return:
    x   : minimizer to fun: numpy array of shape (d,)
"""
def gd_variable(fun, x0, lr, iterations):
    # TO DO:
    #compute gradient module using autograd
    gradient = grad(fun)
    #run the gradient descent loop
    for i in range(iterations):
        lr_i = lr/(i+1) #change to variable learning rate
        x0 = x0 - lr_i * gradient(x0)
    #set the final x0 to x
    x = x0

    return x
```

Part (c) (10 pts) Choosing the learning rate

Let α denote the fixed learning rate and consider the function $f(x) = \frac{1}{2}x^2$. The gradient descent update rule at iteration $n+1$ is given by

$$x_{n+1} = x_n - \alpha f'(x_n)$$

Is there a critical value α_0 so that if $\alpha \geq \alpha_0$ then gradient descent will not converge for $f(x)$? If so what is it? Give a proof.

Answer goes here: The derivation of $f(x)$ is $f'(x) = x$, and $x = 0$ is the optimum of $f(x)$, the critical value $\alpha_0 = 2$.
When $\alpha_0 = 2$, $x_{n+1} = x_n - 2 * x_n = -x_n$, hence the sequence is alternating and $f(x_n) == f(x_{n+1})$.
When $\alpha > 2$, $f(x_n) \leq f(x_{n+1})$. With the iteration going by, $f(x_n)$ becomes bigger. So it cannot be converaged.

Part (d) (10 pts) Feature scaling

Oftentimes it is advantageous to rescale or normalize our features. As a toy example suppose we want to predict a person's weight (in kgs.) based on their height. We would like an algorithm that gives equally good predictions whether height is measured in centimeters or kilometers. For a concrete example, consider both of the following 2-dimensional optimization problems:

$$\mathrm{argmin}_{x\in\mathbb{R}^2}\ x^T\Sigma_i x, \quad i = 1, 2$$

where

$$\Sigma_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \Sigma_2 = \begin{bmatrix} 1 & 0 \\ 0 & 100 \end{bmatrix}.$$

Suppose our starting point is the same $x_0 = (1, 1)^T$ for both optimization problems. For a fixed learning rate, will gradient descent perform better on problem $i = 1$ or $i = 2$? Explain why.

Answer goes here: I think GD performs better on when i=1. The optimization problem is on 2-dimensional surface. With the i = 1, the gradient of two dimension is equal, causing the speed of decline to be even. But with the parameter i =2, the speed of decline in the two dimensions have a large difference, which would have an impact on the algorithm's performance.

Problem 2: Logistic Regression (60 pts total)

In this problem you will implement all of the steps that are taken care of whenever the `fit` function from the sci-kit learn package is called.

Part (a) (5 pts) Logisitc Unit

For binary classification with label $y \in \{0, 1\}$ we can model the posterior probability $p(y = 1|x)$ with the logistic unit

$$h(x; w) = \frac{1}{1 + \exp(-w^T x)}.$$

We will use the convention that $x_0 = 1$. Implement this function below using the skeletal outline as a guide. Suppose we use a discriminant function $f(x)$ which assigns the label $y = 1$ if $p(y = 1|x) \geq 0.5$ and 0 otherwise. In other words, with a discriminant function we do not need the posterior probabilities but rather skip straight to the classification. Implement the following method below to compute this quantity for every point in a dataset.

In [6]:
```
"""
Evaluate the logistic unit h(x;w) on each point x in the dataset X (each row is a different point).  This code
should be vectorized for efficient computation (i.e. no for loops).

Input:
    w: weight vector: numpy array of shape (d,) where d is the dimension
    X: data: numpy array of shape (n, d) where n is the number of data points

Return:
    logits : h(x;w): a numpy array of shape (n,) that has the values
"""
def logistic_unit(X, w):
    # TO DO:
    h = 1 / (1 + np.exp(np.matmul(w, X.T)))
    return h
```

Part (b) (10 pts) Discriminant function

Give a geometric interpretation of our classifier once the parameters w have been learned. What does the angle between the vectors w and x tell us about the predicted class value? For which angles do we predict $y = 1$? Write down the discriminant function $f(x)$ as simply as possible.

Answer goes here:
As the angle between w and x approaches to 0, the dot product gets bigger, and thus $1 + \exp(-w^T x)$ converges to 0 as well, so $p(y = 1)$ converges to 1, which means model has more confidence to predict the class to 1.
The angles between [0, $\frac{\pi}{2}$] and angles between [$\frac{3\pi}{4}$, 0] can make y=1.

$$f(x) = \begin{cases} 1 & p(y = 1|x) \geq 0.5; \\ 0 & p(y = 1|x) < 0.5 \end{cases}$$

Part (c) (10 pts) Deriving the loss function

We have implicitly made the assumption that $y|x \sim \mathrm{Bernoulli}(h(x; w))$. If we have an iid dataset $\{(x_i, y_i)\}_{i=1}^N$, then we can write the data likelihood as

$$p(\vec{y}|X, w) = \prod_{i=1}^{N} p(y_i|x_i, w)$$

We can learn the parameter w by maximizing this probability (i.e. we find the maximum likelihood estimator) or equivalently minimizing $J(w) = -\log p(\vec{y}|X, w)$. Derive the loss function $J(w)$ step-by-step and implement it using the skeletal outline below. In your implementation you may use the convention that $0 \log 0 = 0$.

Answer goes here:
Let $p(y_i|x_i, w) = p(y_i = 1|x_i, w)^{y_i}(1 - p(y_i = 1|x_i, w))^{1-y_i}$,

$$\begin{aligned} \max_w \prod_{i=1}^{N} p(y_i|x_i, w) &\Leftrightarrow \min_w \left(-\sum_{i=1}^{N} \log p(y_i|x_i, w) \right) \\ &= \min_w \left(-\sum_{i=1}^{N} y_i \log p(y_i = 1|x_i, w) + (1 - y_i)\log(1 - p(y_i = 1|x_i, w)) \right) \\ &= \min_w \left(-\sum_{i=1}^{N} y_i \log h(x_i; w) + (1 - y_i)\log(1 - h(x_i; w)) \right) \end{aligned}$$

therefore,

$$J(w) = -\sum_{i=1}^{N} y_i \log h(x_i; w) + (1 - y_i)\log(1 - h(x_i; w)$$

then, we derive this function $J(w)$ as below,

$$\begin{aligned} \frac{\partial J(w)}{\partial w} &= -\sum_{i=1}^{N} y_i * (\log h(x_i; w))' + (1 - y_i) * (\log(1 - h(x_i; w)))' \\ &= -\sum_{i=1}^{N} y_i * \frac{h(x_i; w)'}{h(x_i; w)} + (1 - y_i) * \frac{-h(x_i; w)'}{1 - h(x_i; w)} \end{aligned}$$

Because the derivation of sigmoid function to w is

$$\begin{aligned} \frac{\partial h(x_i; w)}{\partial w} &= \frac{\exp(-w^T x_i)}{(1 + \exp(-w^T x_i))^2} * x_i \\ &= h(x_i; w)(1 - h(x_i; w)) * x_i \end{aligned}$$

so, we can get

$$\begin{aligned} \frac{\partial J(w)}{\partial w} &= -\sum_{i=1}^{N} y_i * \frac{h(x_i; w)'}{h(x_i; w)} + (1 - y_i) * \frac{-h(x_i; w)'}{1 - h(x_i; w)} \\ &= -\sum_{i=1}^{N} y_i * \frac{h(x_i; w)(1 - h(x_i; w)) * x_i}{h(x_i; w)} + (1 - y_i) * \frac{-h(x_i; w)(1 - h(x_i; w)) * x_i}{1 - h(x_i; w)} \\ &= -\sum_{i=1}^{N} x_i y_i (1 - h(x_i; w)) - x_i(1 - y_i)h(x_i; w) \\ &= -\sum_{i=1}^{N} x_i(y_i - h(x_i; w)) \\ &= \sum_{i=1}^{N} x_i(h(x_i; w) - y_i) \end{aligned}$$

In [7]:
```
"""
Compute the loss function using the formula you derived.  This code does not need to be vectorized and you
may use the logistic_unit function.

Input:
    w : weight vector: numpy array of shape (d+1,) (remember that x_0 = 1)
    X : dataset features: numpy array of shape (n, d)
    y : dataset targets: numpy array of shape (n,) contains only 0's or 1's

Return:
    J : loss of w given X,y: float
"""
def loss(w, X, y):
    # TO DO:
    h = logistic_unit(X, w)
    J = -(np.sum(y * np.log(h)) + np.sum((1 - y) * np.log(1 - h)))
    return J
```

Making data (Nothing to do here)

The following method generates random data to test your algorithm on. **DO NOT CHANGE THE METHOD!**

In [8]:
```
# The following method generates a random dataset.  DO NOT ALTER THIS METHOD.
def make_data(n_samples = 500):
    # Generate data features.
    X1 = mvn.rvs(mean = np.array([1.1, 0]), cov = 0.2*np.eye(2), size = n_samples//2)
    X2 = mvn.rvs(mean = np.array([-1.1, 0]), cov = 0.2*np.eye(2), size = n_samples - n_samples//2)
    # Append data labels and combine.
    X1 = np.hstack( (X1, np.ones((X1.shape[0], 1))))
    X2 = np.hstack( (X2, np.zeros((X2.shape[0], 1))))
    X = np.vstack([X1, X2])
    # Randomly permute data.
    np.random.shuffle(X)
    return X
```

Part (d) (5 pts) Splitting the data for training and testing

Now we'll actually learn a logistic regression model for some synthetic data.

First split the dataset into a training, validation, and test set. Use a 40/40/20 split (roughly 40/40/20 is fine). You do not need to use a random splitting, although in practice it is usually a good idea.

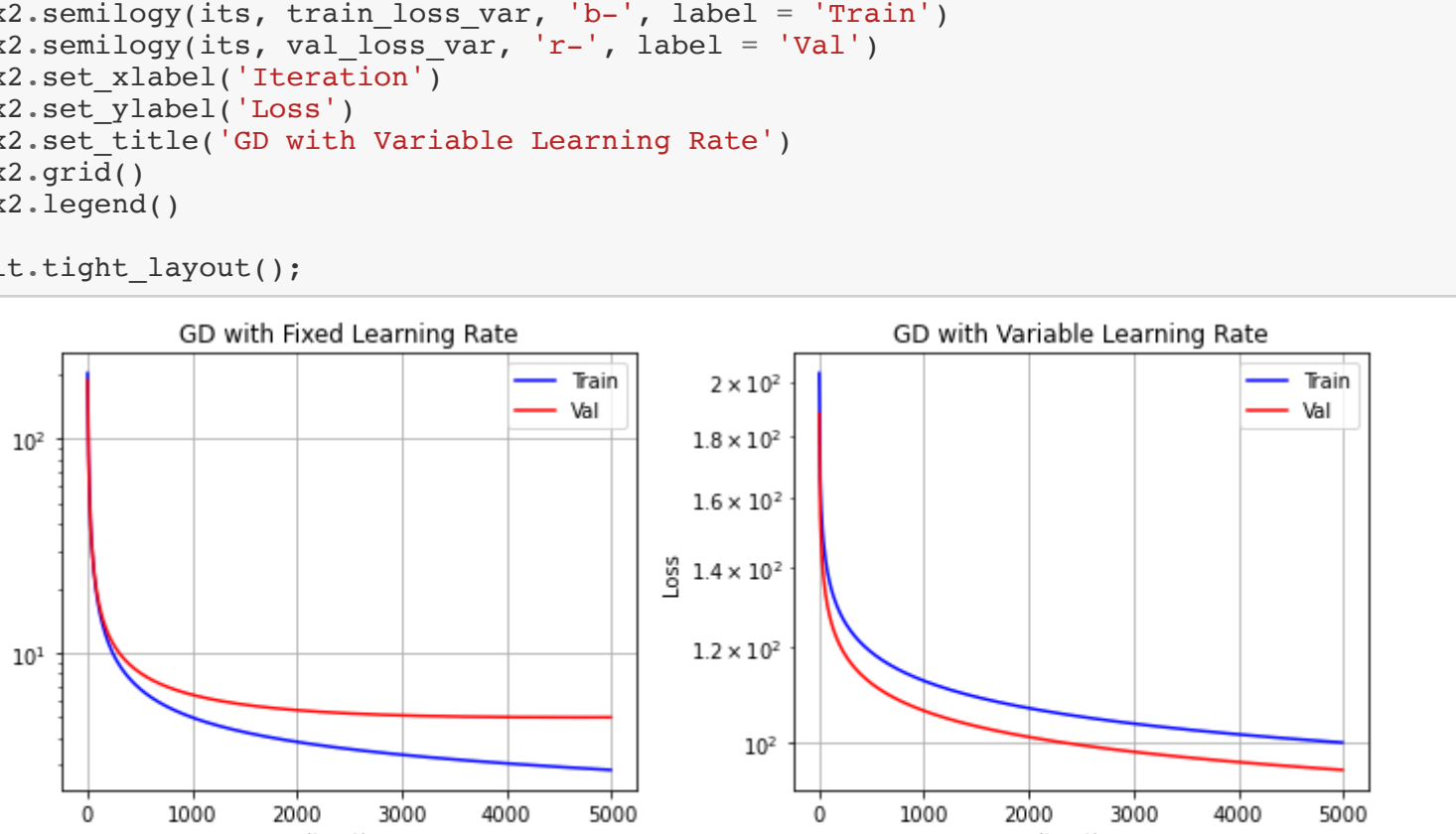
In [9]:
```
# We first generate some fake data.
np.random.seed(2) # Don't change the random seed.
data = make_data()
X = data[:,:-1] # Features
y = data[:,-1]  # Labels

# We'll also augment the data so that x_0 = 1 for the intercept term.
X = np.append(np.ones((len(X), 1)), X, axis = 1)


# TO DO:
## Your part starts here.
size = X.shape[0]
X_train= X[: int(0.4 * size), :]
y_train = y[: int(0.4 * size)]
X_valid = X[int(0.4 * size): int(0.8 * size), :]
y_valid = y[int(0.4 * size): int(0.8 * size)]
X_test = X[int(0.8 * size):, :]
y_test = y[int(0.8 * size):]
## Ends here.
```

Part (e) (20 pts) Training the model

Now use your gradient descent function to learn the parameters w using `5000` iterations. You may choose the learning rate and initial parameters w_0 for this problem. Compare both the fixed learning rate and variable learning rate gradient descents side by side (i.e. 2 subplots) by computing the loss after every `m=10` iterations. There are 4 things to do for this problem.

1. Set the learning rate and initial points.
2. Implement the loss function $J(w)$.
3. Update the parameters using the two gradient descent methods.
4. Compute the loss after every 10 iterations.

In [47]:
```
iterations = 5000          # Lots of iterations to see the training error go down.
m = 10                     # Get the loss every m iterations.

# TO DO: Set the learning rate.
lr1 = 0.001                # You choose this value, try something small.
lr2 = 0.001


## Store the loss values in these arrays.
# Fixed learning rate.
train_loss_fixed = np.zeros(iterations//m)
val_loss_fixed = np.zeros(iterations//m)

# Variable learning rate.
train_loss_var = np.zeros(iterations//m)
val_loss_var = np.zeros(iterations//m)


# TO DO: Set the initial values, either randomly or some fixed value.
w1 = np.random.rand(3)
w2 = w1

# TO DO: Write the loss function as a function of the parameter only.
J = lambda w: loss(w, X_train, y_train)
grad_J = grad(J)


for i in range(iterations):
    # TO DO: Get the updated parameters from gradient descent with a fixed learning rate.
    w1 = w1 - lr1 * grad_J(w1)
    # TO DO: Get the updated parameters from gradient descent with a variable learning rate.
    # Hint: You may either modify the gd_variable function above or call gd_fixed with a different learning rate
.
    lr_2 = lr2/(i+1)
    w2 = w2 - lr_2 * grad_J(w2)

    # Only compute the loss every m iterations.
    if np.mod(i, m) == 0:
        # TO DO: Compute the training and validation loss of the parameters found with the fixed learning rate.
        train_loss_fixed[i//m] = loss(w1, X_train, y_train)
        val_loss_fixed[i//m] = loss(w1, X_valid, y_valid)
        # TO DO: Compute the training and validation loss of the parameters found with a variable learning rate.
        train_loss_var[i//m] = loss(w2, X_train, y_train)
        val_loss_var[i//m] = loss(w2, X_valid, y_valid)


## Plotting starts here.  Nothing to implement.
its = np.arange(1, iterations + 1, m)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize = (10, 4))

ax1.semilogy(its, train_loss_fixed, 'b-', label = 'Train')
ax1.semilogy(its, val_loss_fixed, 'r-', label = 'Val')
ax1.set_xlabel('Iteration')
ax1.set_ylabel('Loss')
ax1.set_title('GD with Fixed Learning Rate')
ax1.grid()
ax1.legend()

ax2.semilogy(its, train_loss_var, 'b-', label = 'Train')
ax2.semilogy(its, val_loss_var, 'r-', label = 'Val')
ax2.set_xlabel('Iteration')
ax2.set_ylabel('Loss')
ax2.set_title('GD with Variable Learning Rate')
ax2.grid()
ax2.legend()

plt.tight_layout();
```



Part (f) (10 pts) Evaluating the model on the test set

Finally, we have trained our models and are ready to evaluate them on the test set. For binary classification one way to check our classifier is to make a *confusion matrix* of our predictions.

$$C = \begin{bmatrix} \text{Predict 0, Actual 0} & \text{Predict 0, Actual 1} \\ \text{Predict 1, Actual 0} & \text{Predict 1, Actual 1} \end{bmatrix}$$

The diagonal elements are the number of samples that are correctly classified.

Use both trained models (fixed and variable learning rates) to classify samples in the test set according to whether $h(x_i; w) \geq 0.5$ or not and print the confusion matrices. Also print the accuracy rate which is just the percentage of correctly classified examples for both models.

In [44]:
```
C1 = np.zeros((2, 2)) # Confusion matrix for fixed learning rate.
C2 = np.zeros((2, 2)) # Confusion matrix for variable learning rate.
N = len(X_test)       # Number of test samples.


## TO DO STARTS HERE:
pred_y1 = (logistic_unit(X_test ,w1) >= 0.5).astype(np.int16)
pred_y2 = (logistic_unit(X_test ,w2) >= 0.5).astype(np.int16)
C1[0][0] = np.logical_and(pred_y1 == y_test, y_test == 0).sum()
C1[1][1] = np.logical_and(pred_y1 == y_test, y_test == 1).sum()
C1[0][1] = np.logical_and(pred_y1 != y_test, y_test == 1).sum()
C1[1][0] = np.logical_and(pred_y1 != y_test, y_test == 0).sum()

C2[0][0] = np.logical_and(pred_y2 == y_test, y_test == 0).sum()
C2[1][1] = np.logical_and(pred_y2 == y_test, y_test == 1).sum()
C2[0][1] = np.logical_and(pred_y2 != y_test, y_test == 1).sum()
C2[1][0] = np.logical_and(pred_y2 != y_test, y_test == 0).sum()
# Compute the accuracy
acc1 = (C1[0][0] + C1[1][1]) / np.sum(C1)
acc2 = (C2[0][0] + C2[1][1]) / np.sum(C2)

# TO DO ENDS HERE.
print("C_1 = ")
print(C1)
print("C_2 = ")
print(C2)
print('Fixed Learning Rate Accuracy    = {:0.1f}%'.format(100 * acc1))
print('Variable Learning Rate Accuracy = {:0.1f}%'.format(100 * acc2))
```

```
C_1 =
[[51.  3.]
 [ 1. 45.]]
C_2 =
[[52. 12.]
 [ 0. 36.]]
Fixed Learning Rate Accuracy    = 96.0%
Variable Learning Rate Accuracy = 88.0%
```

Problem 3: Loss functions (10 pts total)

We have already seen two examples of loss functions: the squared loss and the logistic loss. In this problem we will look at two more important examples.

Part (a) (5 pts) L^p loss

Similar to the squared loss, we can minimize the L^p loss defined as

$$L(w) = \frac{1}{n}\sum_{i=1}^{n} |y_i - w^T x_i|^p$$

for $p > 1$. Whenever, $p = 1$ the problem is no longer differentiable so we cannot take the gradient. Derive the gradient of this loss function.

Answer goes here:

$$\frac{\partial L(w)}{\partial w} = \frac{1}{n}\sum_{i=1}^{N} g(x_i, y_i, w); \quad g(x_i, y_i, w) = \begin{cases} p(y_i - w^T x_i)^{p-1}(-x_i) & y_i \geq w^T x_i; \\ p(w^T x_i - y_i)^{p-1}(x_i) & y_i < w^T x_i \end{cases}$$

Part (b) (5 pts) Regularized loss

Sometimes it's a good idea to add an additional penalty to the weights. An example, which appears in ridge regression, is

$$L(w) = \frac{1}{n}\sum_{i=1}^{n}(y_i - w^T x_i)^2 + \sum_{j=1}^{d} w_j^2$$

Derive the gradient of this loss function. How does the minimizer w of the regularized loss compare to the minimzer of the squared loss? (A heuristic explanation is fine).

Answer goes here:

$$\frac{\partial L(w)}{\partial w} = \frac{2}{n}\sum_{i=1}^{N}(w^T x_i - y_i)x_i + 2\sum_{j=1}^{d} w_j$$

When adding the item $\sum_{j=1}^{d} w_j^2$, the parameter w wouldn't be bigger because from the geomtric perspective w^2 forms a circle with center at origin and radius = 1, so the parameter w would not exceed 1.

In [ ]: