Homework 4

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

Coding assignment: 25 points

Project report: 15 points

Name: Rohitha Ravindra Myla

Link to the github repo: https://github.com/data2060-fall2024/hw-04-rohitharavindra08

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 3.12.5 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.
```

Coding Assignment (25 points)

Introduction

In this assignment, you'll implement Binary Logistic Regression with regularization to perform classification. This classification task is to predict whether or not a given patient has breast cancer based on health data. The regularization method that you will be using is Tikhonov regularization (L2 norm). You will also do cross-validation.

Stencil Code & Data

We have provided the following stencil code within this file:

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

You should not modify any code in the `main`. If you do for debugging or other purposes, please make sure any additions are commented out in the final handin. Do not modify or move the `Check Model` cell! If you do so, you will lose points. The unit tests in that cell make it easy to grade your solution. 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.

UCI Breast Cancer Wisconsin (Diagnostic) Data Set

You will use a preprocessed version of the Breast Cancer Wisconsin (Diagnostic) Data Set from UC Irvine's Machine Learning Repository site. You can read more about the dataset here at https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic). We have split it up into train and validation sets already for you and read them in in `main`.

The Assignment

We provide you with a sigmoid function to use when training your data. In `models`, there are five functions you will implement. They are:

- `RegularizedLogisticRegression`:
  - **`train()`** uses batch stochastic gradient descent to learn the weights. You may find your solution from HW03 to be helpful, but in this assignment, we will train for a finite number of epochs rather than until we reach a particular convergence criteria. The weight update step for this assignment will also be different from HW03.
  - **`predict()`** predicts the labels using the inputs of test data.
  - **`accuracy()`** computes the percentage of the correctly predicted labels over a dataset.
  - **`runTrainTestValSplit()`** trains and evaluates for multiple values of the hyperparameter lambda. This function evaluates models by using train/validation sets, and returns lists of training and validation errors with respect to each value of lambda.
  - **`runKFold()`** evaluates models by implementing k-fold cross validation, and returns a list of errors with respect to each value of lambda. Note that we have defined `_kFoldSplitIndices()` for you, which you may find helpful when implementing this function.

Note: You are not allowed to use any off-the-shelf packages that have already implemented these models, such as scikit-learn. We're asking you to implement them yourself.

Binary Logistic Regression

Similar to homework 3, we are again implementing Logistic Regression for classification. However, note that there are a few key differences. For this assignment, we are performing binary classification, which is a special case of multi-class classification. We are also implementing regularization, so you should think

about how you would need to modify the loss function and gradient provided below to include regularization. For this problem, there are only two classes, which are denoted by $\{0, 1\}$ labels.
Our model will perform the following:

$$h(x) = \frac{1}{1 + e^{-<w,x>}}$$

where w is the model's weights and $h(x)$ is the probability that the data point x has a label of 1. We have implemented this as `sigmoid_function()` for you.

Our loss function will be Binary Log Loss, also called Binary Cross Entropy Loss:

$$L_S(h) = -\frac{1}{m}\sum_{i=1}^{m}(y_i \log h(x_i) + (1 - y_i) \log(1 - h(x_i)))$$

on a sample S of m data points. Therefore, the corresponding gradient of the Binary Log loss with respect to the model's weights is

$$\frac{\partial L_S(h)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}(h(x_i) - y_i)x_{ij}$$

Regularize with Tikhonov Regularization

As mentioned in the introduction part, with Tikhonov regularization, you just need to implement the L2 norm of the weights, which is

$$\lambda||w||_2^2 = \lambda\sum_{i=1}^{d} w_i^2$$

With that added, the gradient used to update the weights has to be adjusted to include

$$\frac{\partial\lambda\sum_{i=1}^{d} w_i^2}{\partial w_j} = 2\lambda w_j$$

Notice that the λ parameter above is used to control the contribution of the regularization term to the overall learning process that you may have to tune a little bit when implementing the code.

Model

In [2]:
```
import numpy as np
import random
import pandas as pd
import matplotlib.pyplot as plt

def sigmoid_function(x):
    return 1.0 / (1.0 + np.exp(-x))
```

```
class RegularizedLogisticRegression(object):
    '''
    Implement regularized logistic regression for binary classification.
    The weight vector w should be learned by minimizing the regularized l
    L(h, (x,y)) = log(1 + exp(-y <w, x>)) + lambda |w|_2^2. In other word
    function that we are trying to minimize is the log loss for binary lo
    plus Tikhonov regularization with a coefficient of lambda.
    '''
    def __init__(self, batch_size = 15):
        self.learningRate = 0.00001 # Feel free to play around with this
        self.num_epochs = 10000 # Feel free to play around with this if y
        self.batch_size = batch_size # Feel free to play around with this
        self.weights = None
        self.lmbda = 0.006 # tune this parameter

    def train(self, X, Y):
        '''
        Train the model, using batch stochastic gradient descent
        '''
        num_samples, num_features = X.shape
        self.weights = np.zeros((1, num_features))  # Ensuring weights is

        for epoch in range(self.num_epochs):
            indices = np.random.permutation(num_samples)
            X_shuffled = X[indices]
            Y_shuffled = Y[indices]

            for i in range(0, num_samples, self.batch_size):
                X_batch = X_shuffled[i:i + self.batch_size]
                Y_batch = Y_shuffled[i:i + self.batch_size]

                # Compute predictions
                linear_model = np.dot(X_batch, self.weights.T).flatten()
                predictions = sigmoid_function(linear_model)

                # Compute the gradient of the loss
                errors = predictions - Y_batch
                gradient = np.dot(X_batch.T, errors) / self.batch_size

                # Compute regularization term
                regularization = 2 * self.lmbda * self.weights

                # Update weights
                self.weights -= self.learningRate * (gradient.T + regular



    def predict(self, X):
        '''
        Compute predictions based on the learned parameters and examples
        @params:
            X: a 2D Numpy array where each row contains an example, padde
        @return:
            A 1D Numpy array with one element for each row in X containin
        '''
        #[TODO]
        linear_model = np.dot(X, self.weights.T.flatten())
        predictions = sigmoid_function(linear_model)
        return (predictions >= 0.5).astype(int)
```

```
    def accuracy(self,X, Y):
        '''
        Output the accuracy of the trained model on a given testing datas
        @params:
            X: a 2D Numpy array where each row contains an example, padde
            Y: a 1D Numpy array containing the corresponding labels for e
        @return:
            a float number indicating accuracy (between 0 and 1)
        '''
        #[TODO]
        predictions = self.predict(X)
        accuracy = np.mean(predictions == Y)
        return accuracy

    def runTrainTestValSplit(self, lambda_list, X_train, Y_train, X_val,
        '''
        Given the training and validation data, fit the model with traini
        respect to each lambda. Record the training error and validation
        to (1 - accuracy).
        @params:
            lambda_list: a list of lambdas
            X_train: a 2D Numpy array for trainig where each row contains
            padded by 1 column for the bias
            Y_train: a 1D Numpy array for training containing the corresp
            X_val: a 2D Numpy array for validation where each row contain
            padded by 1 column for the bias
            Y_val: a 1D Numpy array for validation containing the corresp
        @returns:
            train_errors: a list of training errors with respect to the l
            val_errors: a list of validation errors with respect to the l
        '''
        train_errors = []
        val_errors = []
        #[TODO] train model and calculate train and validation errors her
        train_errors = []
        val_errors = []

        for lmbda in lambda_list:
            self.lmbda = lmbda
            self.train(X_train, Y_train)

            train_accuracy = self.accuracy(X_train, Y_train)
            val_accuracy = self.accuracy(X_val, Y_val)

            train_errors.append(1 - train_accuracy)
            val_errors.append(1 - val_accuracy)

        return train_errors, val_errors

    def _kFoldSplitIndices(self, dataset, k):
        '''
        Helper function for k-fold cross validation. Evenly split the ind
        dataset into k groups.
        For example, indices = [0, 1, 2, 3] with k = 2 may have an output
        indices_split = [[1, 3], [2, 0]].

        Please don't change this.
        @params:
```

```
            dataset: a Numpy array where each row contains an example
            k: an integer, which is the number of folds
        @return:
            indices_split: a list containing k groups of indices
        '''
        num_data = dataset.shape[0]
        fold_size = int(num_data / k)
        indices = np.arange(num_data)
        np.random.shuffle(indices)
        indices_split = np.split(indices[:fold_size*k], k)
        return indices_split

    def runKFold(self, lambda_list, X, Y, k = 3):
        '''
        Run k-fold cross validation on X and Y with respect to each lambd
        errors.

        Each run of k-fold involves k iterations. For an arbitrary iterat
        used as testing data while the rest k-1 folds are combined as one
        averaged as the cross validation error.
        @params:
            lambda_list: a list of lambdas
            X: a 2D Numpy array where each row contains an example, padde
            Y: a 1D Numpy array containing the corresponding labels for e
            k: an integer, which is the number of folds, k is 3 by defaul
        @return:
            k_fold_errors: a list of k-fold errors with respect to the la
        '''
        k_fold_errors = []
        indices_split = self._kFoldSplitIndices(X, k)

        for lmbda in lambda_list:
            self.lmbda = lmbda
            fold_errors = []

            for i in range(k):
                test_indices = indices_split[i]
                train_indices = np.hstack([indices_split[j] for j in rang

                X_train, Y_train = X[train_indices], Y[train_indices]
                X_test, Y_test = X[test_indices], Y[test_indices]

                self.train(X_train, Y_train)
                fold_error = 1 - self.accuracy(X_test, Y_test)
                fold_errors.append(fold_error)

            k_fold_errors.append(np.mean(fold_errors))

        return k_fold_errors

    def plotError(self, lambda_list, train_errors, val_errors, k_fold_err
        '''
        Produce a plot of the cost function on the training and validatio
        cost function of k-fold with respect to the regularization parame
        to determine a valid lambda.
        @params:
            lambda_list: a list of lambdas
            train_errors: a list of training errors with respect to the l
            val_errors: a list of validation errors with respect to the l
            k_fold_errors: a list of k-fold errors with respect to the la
```

```
        @return:
            None
        '''
        plt.figure()
        plt.semilogx(lambda_list, train_errors, label = 'training error')
        plt.semilogx(lambda_list, val_errors, label = 'validation error')
        plt.semilogx(lambda_list, k_fold_errors, label = 'k-fold error')
        plt.xlabel('lambda')
        plt.ylabel('error')
        plt.legend()
        plt.show()
```

Check Model

In [3]:
```
import pytest
# Sets random seed for testing purposes
np.random.seed(0)
random.seed(0)

# Creates Test Models
test_model1 = RegularizedLogisticRegression(3)
test_model2 = RegularizedLogisticRegression(3)

# 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])

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([0,1,1,1,0,1])
x_bias_test2 = np.array([[0,0,1], [-5,-3,1], [9,0,1], [1,0,1]])
y_test2 = np.array([0,0,1,0])


# Test Train Model and Checks Model Weights
test_model1.train(x_bias, y)
weights1 = test_model1.weights
assert isinstance(weights1, np.ndarray)
assert weights1.ndim==2 and weights1.shape == (1,3)
assert weights1 == pytest.approx(np.array([[0.12661045, -0.14658517, -0.0

test_model2.train(x_bias2, y2)
weights2 = test_model2.weights
print(weights2)
assert isinstance(weights2, np.ndarray)
assert weights2.ndim==2 and weights2.shape == (1,3)
assert weights2 == pytest.approx(np.array([[0.11113, 0.08361, 0.01943]]),

# Test Model Predict
predict1 = test_model1.predict(x_bias_test)
assert isinstance(predict1, np.ndarray)
assert predict1.ndim==1 and predict1.shape==(5,)
assert (predict1 == np.array([0, 0, 1, 1, 1])).all()

predict2 = test_model2.predict(x_bias_test2)
assert isinstance(predict2, np.ndarray)
assert predict2.ndim==1 and predict2.shape==(4,)
```

```
assert (test_model2.predict(x_bias_test2) == np.array([1, 0, 1, 1])).all(

# Test Model Accuracy
accuracy1 = test_model1.accuracy(x_bias_test, y_test)
assert isinstance(accuracy1, float)
assert accuracy1 == .8

accuracy2 = test_model2.accuracy(x_bias_test2, y_test2)
assert isinstance(accuracy2, float)
assert accuracy2 == .5

from datetime import date
#[TODO] Print your name and the date, using today function from date
print("Name: Rohitha Ravindra Myla")
print("Date: ",date.today())
```

```
[[0.13280188 0.10035246 0.02293966]]
```

```
---------------------------------------------------------------------------
_
AssertionError                            Traceback (most recent call las
t)
Cell In[3], line 34
     32 assert isinstance(weights2, np.ndarray)
     33 assert weights2.ndim==2 and weights2.shape == (1,3)
---> 34 assert weights2 == pytest.approx(np.array([[0.11113, 0.08361, 0.01
943]]), 0.05)
     36 # Test Model Predict
     37 predict1 = test_model1.predict(x_bias_test)

AssertionError:
```

Main

In [4]:

```
def extract():
    X_train = pd.read_csv('X_train.csv',header=None)
    Y_train = pd.read_csv('y_train.csv',header=None)
    X_val = pd.read_csv('X_val.csv',header=None)
    Y_val = pd.read_csv('y_val.csv',header=None)

    Y_train = np.array([i[0] for i in Y_train.values])
    Y_val = np.array([i[0] for i in Y_val.values])

    X_train = np.append(X_train, np.ones((len(X_train), 1)), axis=1)
    X_val = np.append(X_val, np.ones((len(X_val), 1)), axis=1)

    return X_train, X_val, Y_train, Y_val

def main():
    X_train, X_val, Y_train, Y_val = extract()
    X_train_val = np.concatenate((X_train, X_val))
    Y_train_val = np.concatenate((Y_train, Y_val))

    RR = RegularizedLogisticRegression()
    RR.train(X_train, Y_train)
    print('Train Accuracy: ' + str(RR.accuracy(X_train, Y_train)))
    print('Validation Accuracy: ' + str(RR.accuracy(X_val, Y_val)))

    #[TODO] Once implemented, uncomment the following lines of code and:
    # 1. implement runTrainTestValSplit to get the training and validatio
```

```
    #    split to the original dataset
    # 2. implement runKFold to generate errors of each lambda, where k =
    # 3. call plotError to plot those errors with respect to lambdas
    '''
    lambda_list = [1000, 100, 10, 1, 0.1, 0.01, 0.001]
    train_errors, val_errors = RR.runTrainTestValSplit(lambda_list, X_tra
    k_fold_errors = RR.runKFold(lambda_list, X_train_val, Y_train_val, 3)
    print(lambda_list)
    print(train_errors, val_errors, k_fold_errors)
    RR.plotError(lambda_list, train_errors, val_errors, k_fold_errors)
    '''

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

```
Train Accuracy: 0.9648351648351648
Validation Accuracy: 0.9736842105263158
```

Project Report (15 points)

Question 1

Briefly explain how you used batch stochastic gradient descent with regularization to learn the weights. Think about how the regularization is incorporated into the loss function and how that affects the gradient when updating weights.

Solution:

In this case, I used batch stochastic gradient descent to learn the weights, where instead of using the whole dataset at once, the model is updated based on smaller, random batches of data. This makes the training faster and adds some randomness that can help improve the learning process. With each batch, the model calculates the gradient, which tells us the direction to adjust the weights in order to reduce the error. This process repeats many times until the model starts to converge on a set of weights that work well for the data.

The regularization part comes in to prevent the model from overfitting, which is when it gets too focused on the training data and doesn't perform well on new data. Specifically, I used L2 regularization, which adds a penalty based on the size of the weights—basically, it discourages the model from relying too heavily on any one feature by making large weights more costly. So when updating the weights, I included this penalty term, which effectively pulls the weights back towards zero a little bit after each update. This helps keep the model simpler and more general, striking a balance between fitting the data well and not over-complicating things.

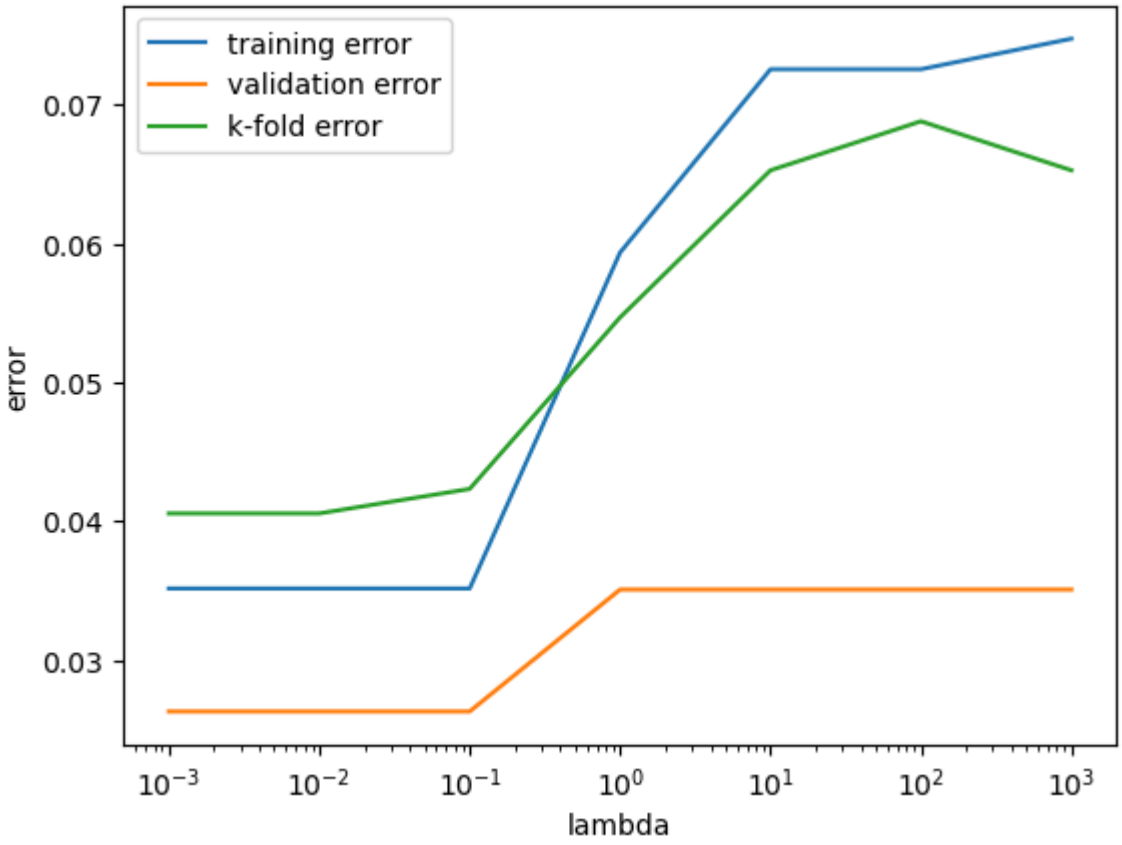
Question 2

Use `plotError()`, which we have implemented for you, to produce a model selection curve. Include your plot here. Then, conclude what the best value of

lambda is and explain why.
Note: It takes about 10-15 minutes to generate a graph.

Solution

In [5]:
```
lambda_list = [1000, 100, 10, 1, 0.1, 0.01, 0.001]

X_train, X_val, Y_train, Y_val = extract()

X_train_val = np.vstack((X_train, X_val))
Y_train_val = np.hstack((Y_train, Y_val))

RR = RegularizedLogisticRegression()

train_errors, val_errors = RR.runTrainTestValSplit(lambda_list, X_train,

k_fold_errors = RR.runKFold(lambda_list, X_train_val, Y_train_val, k=3)

RR.plotError(lambda_list, train_errors, val_errors, k_fold_errors)
```



The graph gives a clear picture of how different lambda values affect the model's performance. On the X-axis, we see lambda values, where smaller values mean less regularization and larger values indicate more. The Y-axis shows the error rate, and lower values here reflect better model accuracy. The blue line traces the training error, the orange line shows validation error, and the green line represents the k-fold error.

At smaller lambda values, particularly between 10^{-3} and 10^{-2}, the training error is quite low, meaning the model fits the training data well. The validation and k-fold errors are also low, which suggests that the model is not overfitting and is

generalizing effectively. As lambda increases (from 10^{-1} to 1), the training error rises as the regularization becomes stronger, but the validation error stays stable, showing that the model is still performing well on unseen data. The k-fold error starts to increase slightly, signaling some inconsistency across different data splits. When lambda gets much larger (10^2 and above), the training error jumps up, indicating that the model is underfitting due to being overly simplified. Interestingly, the validation error remains flat, implying that further regularization does not provide any additional benefits.

To sum up, the most effective lambda value seems to be around 10^{-3} to 10^{-2}. This range offers the best trade-off, where the model avoids both overfitting and underfitting, and performs well on new data. Increasing lambda beyond this range only hurts training accuracy without improving how well the model generalizes.

Question 3

In this project, you used validation data to select a model. Suppose that each patient might've had multiple samples (e.g., multiple lab tests or x-rays) collected and entered into the dataset. Would you need to account for this when splitting your train-validation-test data? If yes, how? If no, why not? (3-5 sentences)

Solution:

When splitting the train-validation-test data, it’s important to account for cases where a patient has multiple samples. If the same patient’s data appears in both the training and validation/test sets, the model might overfit to that specific patient’s data. This can lead to inflated performance estimates, as the model could learn from similar data in training and then easily predict for that same patient in testing. To avoid this issue, you should group all samples from a patient together and assign them entirely to either the training set or the validation/test set. This ensures that the model is evaluated on truly unseen data, preventing any data leakage and giving a more accurate measure of how well the model generalizes to new patients.