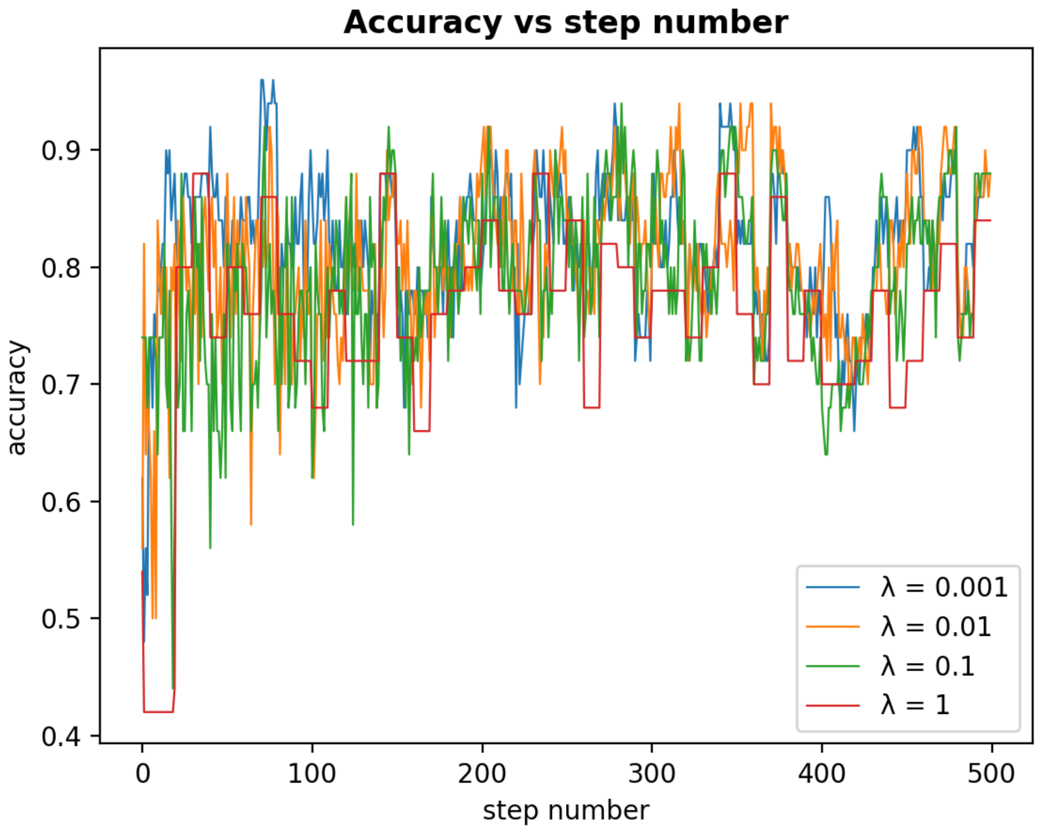
CS498 AML HW2 Name: Yu Che Wang/ Netid: yuchecw2

Leaderboard accuracy

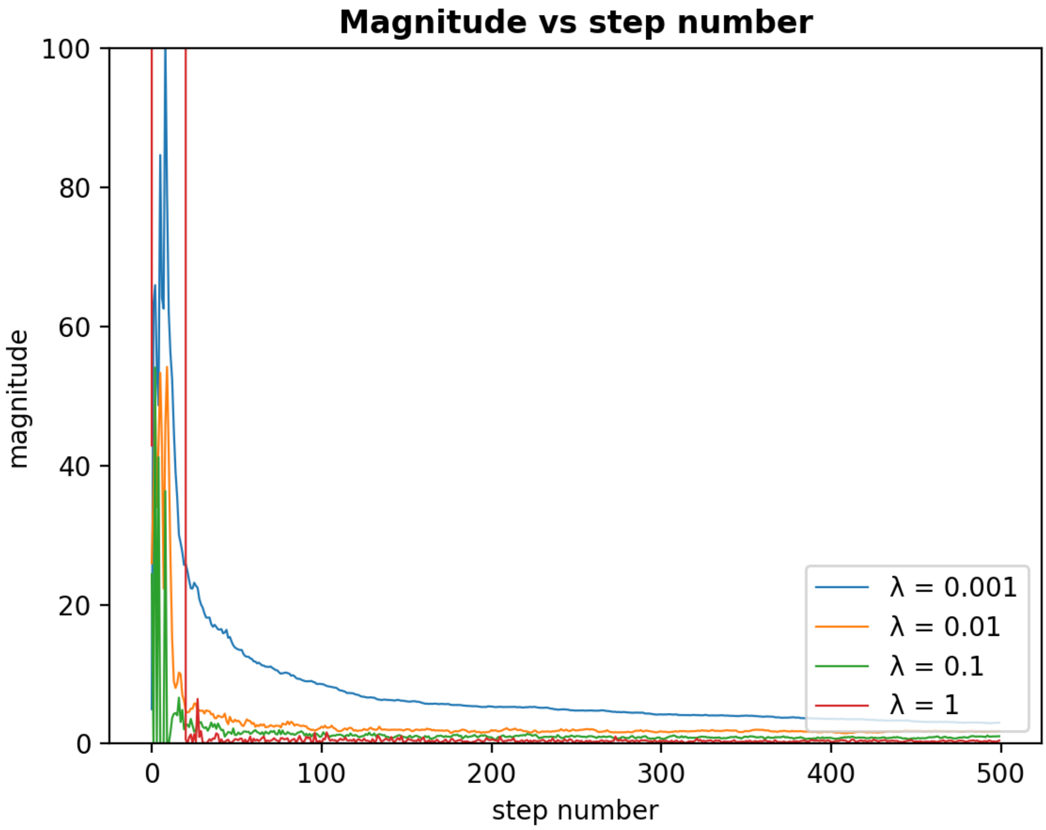
#	△1w	Team Name	Kernel	Team Members	Score	Entries	Last
1	new	**Junli Wu**			0.83497	12	1d
2	new	**Yuyang**			0.83230	10	16h
3	new	**Jeffrey Wang**			0.82637	1	2d
Your Best Entry ↑ Your submission scored 0.82637 Tweet this!							
4	new	**Kunyang**			0.82596	31	1d
5	new	**Hao Wu**			0.81961	13	2d
6	new	**zzachw**			0.81244	26	5h
7	new	**Yue Wan**			0.81142	12	2d
8	new	**Yichong Guo**			0.81122	4	3d
9	new	**awepuzxoicznfawev**			0.81122	26	17h
10	new	**Jinghan Huang**			0.81081	6	3m

Best test dataset accuracy: 0.82637

A plot of the accuracy every 30 steps, for each value of the regularization constant.



A plot of the magnitude of the coefficient vector every 30 steps, for each value of the regularization constant.



Brief explanation:
The classifier has highest performance when λ, the regularization constant, is 0.001 and 0.01 based on the plot of accuracy versus steps. From the plot of the magnitude of the coefficient vector versus steps, the stochastic gradient descent algorithm converges faster when λ is 0.01 than λ is 0.001. Thus, among 0.001 and 0.01, I chose **λ to be 0.01** for my classifier. In addition, I chose the learning rate to be $\frac{m}{n+epoch}$, where m is 1 and n is 0.1. The classifier has high performance in terms of accuracies when m is ranging from 0.1 to 10 and n ranging from 0.01 to 0.1.

Code:

Main function

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mlxtend.preprocessing import standardize
from sklearn.cross_validation import train_test_split
import random
import argparse
from process import *

parser = argparse.ArgumentParser()
parser.add_argument('--num_epochs', default=50, type=int)
parser.add_argument('--num_steps', default=300, type=int)
parser.add_argument('--eval_steps', default=30, type=int, help='compute the accu
parser.add_argument('--batch_size', default=1, type=int)
parser.add_argument('--m', default=1, type=float, help='parameter for learning ra
parser.add_argument('--n', default=0.1, type=float, help='parameter for learning
parser.add_argument('--lamda', default=0.001, type=float, help='regularization pa
parser.add_argument('--num_held_out', default=50, type=int)
parser.add_argument('--test', default=False, type=bool)
parser.add_argument('--path', default='yuchecw2_prediction.csv', type=str)
args = parser.parse_args()

train_data = pd.read_csv("train.data.csv")
label_list = [i for i in train_data['class']]
label = np.empty(len(label_list))
for i in range(label.shape[0]):
    if (label_list[i] == ' >50K'):
        label[i] = 1
    else:
        label[i] = -1

feature = pd.DataFrame.as_matrix(train_data)[:, :-1]
# Preprocess the data
feature = preprocess(feature)
# Standardize columns in train_feature
feature = standardize(feature)
# Train-validation split
train_feature, val_feature, train_label, val_label = train_test_split(
                feature, label, test_size=0.1)

plt.figure()
for lamda in [0.001, 0.01, 0.1, 1]:
    # Initialize a, b
    a = np.random.randn(train_feature.shape[1], 1)  # shape=(14, 1)
    b = 0
    accuracy_list = []
    mag_list = []
    loss_list = []
    for epoch in range(args.num_epochs):
        actual_train_feature, held_out_feature, actual_train_label, held_out_label =
            train_test_split(train_feature, train_label, test_size=args.num_held_out/train_fe
        actual_train_size = actual_train_feature.shape[0]
        for step in range(args.num_steps):
            lr = compute_lr(args.m, args.n, epoch)
            batch_num = random.sample(range(actual_train_size), args.batch_size)
            a = update_a(a, b, actual_train_feature, actual_train_label, batch_num, lr, lamda
            b = update_b(b, a, actual_train_feature, actual_train_label, batch_num, lr, lamda
            if (step%args.eval_steps == 0):
                eval_dict = evaluate(held_out_feature, held_out_label, a, b, lamda)
                accuracy_list.append(eval_dict['accuracy'])
                mag_list.append(eval_dict['mag'])
                loss_list.append(eval_dict['loss'])

    if (args.test):
        test_data = pd.read_csv("test.data.csv")
        test_feature = pd.DataFrame.as_matrix(test_data)
        test_feature = preprocess(test_feature)
        test_feature = standardize(test_feature)
        prediction = predict(test_feature, a, b)
        prediction[prediction['Label'] == '>'] = ">50K"
        prediction[prediction['Label'] == '<'] = "<=50K"
        prediction.to_csv(args.path)

    step_list = range(len(accuracy_list))

    for step in step_list:
        step *= args.num_steps


    label = 'λ = {}'.format(lamda)
    plt.plot(step_list, mag_list, label=label, linewidth=0.8)
    plt.ylim(0, 100)

plt.xlabel('step number')
plt.ylabel('magnitude')
plt.title('Magnitude vs step number', fontsize=12, fontweight='bold')
plt.legend(loc='lower right')
plt.show()
```

Helper functions

```python

# Input: feature matrix of shape (43958, 14)
def preprocess(train_feature):⋯

# Input
# a: numpy array of size (14, 1)
# feature: numpy array of size (14, 1)
# b: double
# Output
# gamma: double
def compute_gamma(a, feature, b):
    gamma = np.matmul(np.transpose(a), feature).item() + b
    return gamma

# Input
# gamma: numpy array of size (14, 1)
def cost_function(gamma, label):
    cost = max(0, 1-label*gamma)
    return cost

# Input
# batch_num: a list consists of indices of training feature, sup
# train_feature: numpy array of shape (43958, 14)
# train_label: numpy array of shape (43958, )
def hinge_loss(batch_num, train_feature, train_label, a, b):⋯

# Input
# lamda: regularization parameter
# a: numpy array of shape (14, 1)
def regularization_loss(lamda, a):⋯

# This is the loss we're going to minimize using stochastic grad
def total_loss(batch_num, train_feature, train_label, a, b, lamd
    return hinge_loss(batch_num, train_feature, train_label, a, 

# Output
# u: numpy array of shape (15, 1)
def obtain_u(a, b):
    b_arr = np.expand_dims(np.array([b]), axis=1)
    u = np.concatenate((a, b_arr), axis=0)
    return u

def compute_lr(m, n, epoch):
    return m/(n+epoch)
```

```python
# Input
# current_a: numpy array of shape (14, 1)
# current_b: double
def update_a(current_a, current_b, train_feature, train_label, batch_num, lr, lamda):
    grad = np.zeros(current_a.shape)
    for i in batch_num:
        if (cost_function(compute_gamma(current_a, train_feature[i,:], current_b), train_label[i]) == 0):
            grad += lamda*current_a
        else:
            grad = grad + (lamda*current_a - np.expand_dims(train_label[i]*train_feature[i, :], axis=1))
    grad /= len(batch_num)
    return current_a - lr*grad

def update_b(current_b, current_a, train_feature, train_label, batch_num, lr, lamda):
    grad = 0
    for i in batch_num:
        if (cost_function(compute_gamma(current_a, train_feature[i, :], current_b), train_label[i]) == 0):
            continue
        else:
            grad += (-train_label[i])
    grad /= len(batch_num)
    return current_b - lr*grad

def evaluate(held_out_feature, held_out_label, a, b, lamda):
    n_correct = 0
    for i in range(held_out_feature.shape[0]):
        gamma_i = compute_gamma(a, held_out_feature[i,:], b)
        if gamma_i*held_out_label[i] > 0:
            n_correct += 1
    accuracy = n_correct / held_out_feature.shape[0]
    mag = math.sqrt(np.matmul(a.T, a).item() + b**2)
    loss = total_loss(range(held_out_feature.shape[0]), held_out_feature, held_out_label, a, b, lamda)
    return {'accuracy': accuracy, 'mag': mag, 'loss': loss}

def predict(test_feature, a, b):
    test_label = np.empty((test_feature.shape[0]), dtype=str)
    for i in range(test_feature.shape[0]):
        gamma = compute_gamma(a, np.expand_dims(test_feature[i,:], axis=1), b)
        if (gamma > 0):
            test_label[i] = ">50K"
        else:
            test_label[i] = "<=50K"
    return pd.DataFrame(data={'Label': test_label}, dtype=str)
```