MIT 6.036 Spring 2019: Homework 3

This colab notebook provides code and a framework for problems 1-7 of the homework. You can work out your solutions here, then submit your results back on the homework page when ready.

Setup

First, download the code distribution for this homework that contains test cases and helper functions.

Run the next code block to download and import the code for this lab.

In [1]:
```
!pwd
```

```
/home/art/mydir/ref/Учеба_конспекты_решения/mit_6.036_intro_to_ml
```

In [2]:
```
#!rm -rf code_and_data_for_hw3*
#!rm -rf mnist
#!wget --quiet https://introml_oll.odl.mit.edu/6.036/static/homework/hw03/code_and_data_for_hw3.zip
#!unzip code_and_data_for_hw3.zip
#!mv code_and_data_for_hw3/* .

from code_for_hw3_part1 import *
import code_for_hw3_part2 as hw3
```

```
Importing code_for_hw03
Imported tidy_plot, plot_separator, plot_data, plot_nonlin_sep, cv, rv, y, positive, score
Datasets: super_simple_separable_through_origin(), super_simple_separable(), xor(), xor_more()
Tests for part 2: test_linear_classifier_with_features, mul, make_polynomial_feature_fun,
                  test_with_features
Also loaded: perceptron, one_hot_internal, test_one_hot
Importing code_for_hw03 (part 2, imported as hw3)
Imported tidy_plot, plot_separator, plot_data, plot_nonlin_sep, cv, rv, y, positive, score
         xval_learning_alg, eval_classifier
Tests: test_linear_classifier
Dataset tools: load_auto_data, std_vals, standard, raw, one_hot, auto_data_and_labels
               load_review_data, clean, extract_words, bag_of_words, extract_bow_feature_vectors
               load_mnist_data, load_mnist_single
```

In [3]:
```
help(tidy_plot)
```

```
Help on function tidy_plot in module code_for_hw3_part1:

tidy_plot(xmin, xmax, ymin, ymax, center=False, title=None, xlabel=None, ylabel=None)
```

Feature Transformation

Running Perceptron

In problems 1,2 and 3, you will have to run the Perceptron algorithm several times to obtain linear classifiers. We provide you with an implementation of the algorithm which you can use to obtain your results.

The specifications for the `perceptron` method provided are:

- `data` is a numpy array of dimension d by n
- `labels` is numpy array of dimension 1 by n
- `params` is a dictionary specifying extra parameters to this algorithm; your algorithm runs a number of iterations equal to T
- `hook` is either None or a function that takes the tuple `(th, th0)` as an argument and displays the separator graphically.

It should return a tuple of θ (a d by 1 array) and θ_0 (a 1 by 1 array).

Note that you are free to modify the method. For example, a useful modification for this homework would be to make the method return the number of mistakes made on the input data, while it runs.

In [4]:

```python
# Perceptron algorithm with offset.
# data is dimension d by n
# labels is dimension 1 by n
# T is a positive integer number of steps to run
def perceptron(data, labels, params = {}, hook = None):
    # if T not in params, default to 100
    T = params.get('T', 50)
    (d, n) = data.shape

    theta = np.zeros((d, 1)); theta_0 = np.zeros((1, 1))
    for t in range(T):
        for i in range(n):
            x = data[:,i:i+1]
            y = labels[:,i:i+1]
            if y * positive(x, theta, theta_0) <= 0.0:
                theta = theta + y * x
                theta_0 = theta_0 + y
                if hook: hook((theta, theta_0))
    return theta, theta_0

def averaged_perceptron(data, labels, params = {}, hook = None):
    T = params.get('T', 50)
    (d, n) = data.shape

    theta = np.zeros((d, 1)); theta_0 = np.zeros((1, 1))
    theta_sum = theta.copy()
    theta_0_sum = theta_0.copy()
    for t in range(T):
        for i in range(n):
            x = data[:,i:i+1]
            y = labels[:,i:i+1]
            if y * positive(x, theta, theta_0) <= 0.0:
                theta = theta + y * x
                theta_0 = theta_0 + y
                if hook: hook((theta, theta_0))
            theta_sum = theta_sum + theta
            theta_0_sum = theta_0_sum + theta_0
    theta_avg = theta_sum / (T*n)
    theta_0_avg = theta_0_sum / (T*n)
    if hook: hook((theta_avg, theta_0_avg))
    return theta_avg, theta_0_avg


def eval_classifier(learner, data_train, labels_train, data_test, labels_test):
    th, th0 = learner(data_train, labels_train)
    return score(data_test, labels_test, th, th0)/data_test.shape[1]

def positive(x, th, th0):
```

```python
    return np.sign(th.T@x + th0)

def score(data, labels, th, th0):
    return np.sum(positive(data, th, th0) == labels)

def xval_learning_alg(learner, data, labels, k):
    _, n = data.shape
    idx = list(range(n))
    np.random.seed(0)
    np.random.shuffle(idx)
    data, labels = data[:,idx], labels[:,idx]

    score_sum = 0
    s_data = np.array_split(data, k, axis=1)
    s_labels = np.array_split(labels, k, axis=1)
    for i in range(k):
        data_train = np.concatenate(s_data[:i] + s_data[i+1:], axis=1)
        labels_train = np.concatenate(s_labels[:i] + s_labels[i+1:], axis=1)
        data_test = np.array(s_data[i])
        labels_test = np.array(s_labels[i])
        score_sum += eval_classifier(learner, data_train, labels_train,
                                              data_test, labels_test)
    return score_sum/k
```

In [ ]:
```python
perceptron(data, labels, params = {'T':100}, hook = None)
```

In [41]:
```
def my_perceptron(data, labels, params={}, hook=None):
    # if T not in params, default to 100
    T = params.get('T', 100)
    d = data.shape[0]
    n = data.shape[1]
    theta = np.transpose([[0.0] * d])
    theta_0 = 0.0
    #ax = plot_data(data, labels)
    num = 0
    for test in range(T):
        founderror = False
        for i in range(n):
            xi = data[:,i:i+1]
            yi = labels[0, i]
            if yi * np.sign(theta.T @ xi + theta_0) <= 0:
                num += 1
                founderror = True
                theta += yi * xi
                theta_0 += yi
                if hook:
                    hook(theta, theta_0)
        if not founderror:
            break
    #plot_separator(ax=ax, th=theta, th_0=theta_0)
    return (theta, np.array([[theta_0]]), num)
```

In [62]:
```
data = np.array([
    [200, 800, 200, 800],
    [0.2,  0.2,  0.8,  0.8],
    [1, 1, 1, 1]
])
labels = np.array([[-1, -1, 1, 1]])
th = np.array([[0, 1, -0.5]])

gamma = np.abs(np.min(labels.T * (th @ data) / np.linalg.norm(th) ))
r = np.max(np.linalg.norm(data, axis=0))
r, gamma, (r / gamma)**2
```

Out[62]:
```
(800.0010249993434, 0.2683281572999748, 8888911.666666666)
```

In [43]:
```
data = np.array([
    [200, 800, 200, 800],
    [0.2,  0.2,  0.8,  0.8]
])
labels = np.array([[-1, -1, 1, 1]])
```

In [44]:
```
perceptron(data, labels, params={'T':1000})
```
Out[44]:
```
(array([[  0.],
        [600.]]),
 array([[0.]]),
 2000)
```
In [45]:
```
perceptron(data, labels, params={'T':10000})
```
Out[45]:
```
(array([[   0.],
        [6000.]]),
 array([[0.]]),
 20000)
```
In [46]:
```
perceptron(data, labels, params={'T':100000})
```
Out[46]:
```
(array([[  600.        ],
        [69997.80000031]]),
 array([[0.]]),
 233326)
```
In [47]:
```
perceptron(data, labels, params={'T':1000000})
```
Out[47]:
```
(array([[-2.000000e+02],
        [ 2.000068e+05]]),
 array([[-4.]]),
 666696)
```
In [63]:
```
data = np.array([
    [200, 800, 200, 800],
    [0.2,  0.2,  0.8,  0.8],
    [1, 1, 1, 1]
])
data[0:2] *= 0.001
labels = np.array([[-1, -1, 1, 1]])
th = np.array([[0, 1, -0.0005]])

gamma = np.abs(np.min(labels.T * (th @ data) / np.linalg.norm(th) ))

data, gamma
r = np.max(np.linalg.norm(data, axis=0))
r, gamma, (r / gamma)**2
```
Out[63]:
```
(1.2806250973645645, 0.00029999996250000706, 18222233.88889067)
```

In [64]:
```
data = np.array([
    [200, 800, 200, 800],
    [0.2,  0.2,  0.8,  0.8],
    [1, 1, 1, 1]
])
data[0:1] *= 0.001
labels = np.array([[-1, -1, 1, 1]])
th = np.array([[0, 1, -0.5]])

gamma = np.abs(np.min(labels.T * (th @ data) / np.linalg.norm(th) ))

data, gamma
r = np.max(np.linalg.norm(data, axis=0))
r, gamma, (r / gamma)**2
```

Out[64]:
```
(1.50996688705415, 0.2683281572999748, 31.666666666666664)
```

In [65]:
```
perceptron(data, labels, params={'T':1000000})
```

Out[65]:
```
(array([[-0.2],
        [ 2.8],
        [-1. ]]),
 array([[-1.]]),
 11)
```

2D) Encoding Discrete Values

It is common to encode sets of discrete values, for machine learning, not as a single multi-valued feature, but using a one hot encoding. So, if there are k values in the discrete set, we would transform that single multi-valued feature into k binary-valued features, in which feature i has value $+1$ if the original feature value was i and has value 0 (or -1) otherwise.

Write a function `one_hot` that takes as input x, a single feature value (between 1 and k), and k, the total possible number of values this feature can take on, and transform it to a numpy column vector of k binary features using a one-hot encoding (remember vectors have zero-based indexing).

In [71]:
```
data = np.array([[2, 3,  4,  5]])
labels = np.array([[1, 1, -1, -1]])
th, th0 = perceptron(data, labels)

th, th0, labels * (th @ data + th0), th @ np.array([[1, 6]]) + th0
```

Out[71]:
```
(array([[-2.]]), array([[7.]]), array([[3., 1., 1., 3.]]), array([[ 5., -5.]]))
```

In [84]:
```
a = np.zeros((1, 10))
a[:, 1] = 1
a
```

Out[84]:
```
array([[0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

In [8]:
```
def one_hot(x, k):
    a = np.zeros((k, 1))
    a[x-1,0] = 1.0
    return a
```

In [9]:
```
test_one_hot(one_hot)
```

```
Passed!
```

In [16]:
```
data =   [[2, 3,  4,  5]]
data[0]
```

Out[16]:
```
[2, 3, 4, 5]
```

In [47]:
```
data =   np.array([[2, 3,  4,  5]])
labels = np.array([[1, 1, -1, -1]])
data = np.concatenate(
    [one_hot(e, 6) for e in data[0]],
    axis=1
)
data, labels
```

Out[47]:
```
(array([[0., 0., 0., 0.],
        [1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.],
        [0., 0., 0., 0.]]),
 array([[ 1,  1, -1, -1]]))
```

In [49]:
```
th, th0 = perceptron(data, labels)
th.T, th0
```

Out[49]:
```
(array([[ 0.,  2.,  1., -2., -1.,  0.]]), array([[0.]]))
```

In [71]:
```
samsung = one_hot(1, 6)
nokia = one_hot(6, 6)
samsung, nokia, th.T
```

Out[71]:
```
(array([[1.],
        [0.],
        [0.],
        [0.],
        [0.],
        [0.]]),
 array([[0.],
        [0.],
        [0.],
        [0.],
        [0.],
        [1.]]),
 array([[ 0.,  2.,  1., -2., -1.,  0.]]))
```

In [70]:
```
[ th.T @ xi + th0 for xi in (samsung, nokia)]
```

Out[70]:
```
[array([[0.]]), array([[0.]])]
```

In [72]:
```
[ (th.T @ xi + th0) / np.linalg.norm(th) for xi in (samsung, nokia)]
```

Out[72]:
```
[array([[0.]]), array([[0.]])]
```

In [75]:
```
data =   [[1, 2, 3, 4, 5, 6]]
labels = np.array([[1, 1, -1, -1, 1, 1]])

data = np.concatenate(
    [one_hot(e, 6) for e in data[0]],
    axis=1
)
data, labels
```

Out[75]:
```
(array([[1., 0., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0., 0.],
        [0., 0., 1., 0., 0., 0.],
        [0., 0., 0., 1., 0., 0.],
        [0., 0., 0., 0., 1., 0.],
        [0., 0., 0., 0., 0., 1.]]),
 array([[ 1,  1, -1, -1,  1,  1]]))
```

In [76]:
```
th, th0 = perceptron(data, labels)
th.T, th0
```

Out[76]:
```
(array([[ 1.,  1., -2., -2.,  1.,  1.]]), array([[0.]]))
```

3) Polynomial Features

One systematic way of generating non-linear transformations of your input features is to consider the polynomials of increasing order. Given a feature vector $x = [x_1, x_2, ..., x_d]^T$, we can map it into a new feature vector that contains all the factors in a polynomial of order d. For example, for $x = [x_1, x_2]^T$ and order 2, we get $$\phi(x) = [1, x_1, x_2, x_1x_2, x_1^2, x_2^2]^T$$ and for order 3, we get $$\phi(x) = [1, x_1, x_2, x_1x_2, x_1^2, x_2^2, x_1^2x_2, x_1x_2^2, x_1^3, x_2^3]^T.$$

In the code that has been loaded, we have defined `make_polynomial_feature_fun` that, given the order, returns a feature transformation function (analogous to ϕ in the description). You should use it in doing this problem.

In [77]:

```
## For example, make_polynomial_feature_fun could be used as follows:
import numpy as np

# Data
data = np.zeros((5,1))

# Generate transformation of order 2
transformation = make_polynomial_feature_fun(2)

# Use transformation on data
print(transformation(data))
```

```
[[1.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
```

In [79]:
```python
#  Enter a list of 6 integers indicating the number of polynomial features of degrees [1, 10, 20, 30, 40, 50] for a 2-dimension
# (x1, x2)
# 1: 2
# 10:
# 1, 1
# 2 11 2
# 3 21 12 3
# 4 31 22 13 4
# 5 41 32 23 14 5
# 6 51 42 33 24 15 6
# ..
# 10 91 82 73 64 55 46 37 28 19 10
#     1 + 2 + 3 + 4 + ..
# 20: 2 + .. + 21 = 23 * 20 / 2 = 230
# 30: 2 + .. + 31 = 33 * 30 / 2 = 330 + 165 = 495
# 40: 43 * 40 / 2 = 860
# 50: 53 * 50 / 2 = 5300 / 4 = 1325
# 2, 65, 230, 495, 860, 1325 ; +1
[1 + (n+3) * n // 2 for n in (10, 20, 30, 40, 50)]
```

Out[79]:
```
[66, 231, 496, 861, 1326]
```

Note that iterative animations, which update a plot within a loop, don't work the same way in colab, as with a local python console installation. One workaround for colab to be able to show such plot iterations is to show all the plots, and this can be done for the test code using this patched function:

In [162…
```
def test_linear_classifier_with_features(dataFun, learner, feature_fun,
                             draw = True, refresh = True, pause = True):
    raw_data, labels = dataFun()
    data = feature_fun(raw_data) if feature_fun else raw_data
    if draw:
        def hook(params):
            ax = plot_data(raw_data, labels)   # create plot axis on each iteration
            (th, th0) = params
            predictor = lambda x1,x2: int(positive(feature_fun(cv([x1, x2])), th, th0))
            plot_nonlin_sep(
                predictor,
                ax = ax)
            plot_data(raw_data, labels, ax)
            plt.show()                         # force plot to push to the colab notebook and be displayed
            print('th', th.T, 'th0', th0)
            if pause: input('press enter here to continue:')
    else:
        hook = None
    th, th0 = learner(data, labels, hook = hook)
    if hook: hook((th, th0))
    print("Final score", int(score(data, labels, th, th0)))
    print("Params", np.transpose(th), th0)

def test_with_features(dataFun, order = 2, draw=True, pause=True, learner=perceptron):
    test_linear_classifier_with_features(
        dataFun,                        # data
        learner,                     # learner
        make_polynomial_feature_fun(order), # feature maker
        draw=draw,
        pause=pause)
```

Here's a test you can run to see plots:

In [163…
```
def perceptron_with_params(T=100):
    myparams ={'T': T}
    def f(data, labels, params = {}, hook = None):
        return perceptron(data, labels, myparams, hook)
    return f
```

In [144…
```
print(super_simple_separable_through_origin()[0].shape)
test_with_features(super_simple_separable_through_origin, order=2, draw=False, pause=False)
```

```
(2, 4)
Final score 4
Params [[  2.   4.  17. -46.  59. 107.]] [[2.]]
```

In [143…
```
print(super_simple_separable()[0].shape)
test_with_features(super_simple_separable, order=2, draw=False, pause=False, learner=perceptron_with_params(T=1000))
```

```
(2, 4)
Final score 4
Params [[ -11.  -26.   11. -190.  140.  235.]] [[-11.]]
```

In [115…
```
print(xor()[0].shape)
test_with_features( xor , order=2, draw=False, pause=False, learner=perceptron_with_params(T=1000))
```

```
(2, 4)
Final score 4
Params [[ 1. -1. -1. -5. 11. -5.]] [[1.]]
```

In [165…
```
print(xor_more())
test_with_features( xor_more , order=3, draw=False, pause=False, learner=perceptron_with_params(T=10000))
```

```
(array([[1, 2, 1, 2, 2, 4, 1, 3],
       [1, 2, 2, 1, 3, 1, 3, 3]]), array([[ 1,  1, -1, -1,  1,  1, -1, -1]]))
2202
Final score 8
Params [[ -78.   28.  -39.   72.  248.  -19.   76. -522.  476. -153.]] [[-78.]]
```

We know that a better way to do this exists (eg using colab plot animations) - if you are willing to contribute some nice code which lets our plotting functions do this, please do share!

Experiments

4) Evaluating algorithmic and feature choices for AUTO data

We now want to build a classifier for the auto data, focusing on the numeric data. In the code file for this part of the assignment, we have supplied you with the `load_auto_data` function, that can be used to read the relevant .tsv file. It will return a list of dictionaries, one for each data item.

We then have to specify what feature function to use for each column in the data. The file `hw3_part2_main.py` has an example for constructing the data and label arrays using `raw` feature function for all the columns. Look at the definition of `features` in `hw3_part2_main.py`, this indicates a feature name to use and then a feature function, there are three defined in the `code_for_hw3_part2.py` file (`raw`, `standard` and `one_hot`). `raw` just uses the original value, `standard` subtracts out the mean value and divides by the standard deviation and `one_hot` does the encoding described in the notes.

The function `auto_data_and_labels` will process the dictionaries and return `data, labels` where `data` are arrays of dimension $(d, 392)$, with d the total number of features specified, and `labels` is of dimension $(1, 392)$. The data in the file is sorted by class, but it will be shuffled when you read it in.

In [166…

```
# Returns a list of dictionaries.  Keys are the column names, including mpg.
auto_data_all = hw3.load_auto_data('auto-mpg.tsv')

# The choice of feature processing for each feature, mpg is always raw and
# does not need to be specified.  Other choices are hw3.standard and hw3.one_hot.
# 'name' is not numeric and would need a different encoding.
features = [('cylinders', hw3.raw),
            ('displacement', hw3.raw),
            ('horsepower', hw3.raw),
            ('weight', hw3.raw),
            ('acceleration', hw3.raw),
            ## Drop model_year by default
            ## ('model_year', hw3.raw),
            ('origin', hw3.raw)]

# Construct the standard data and label arrays
auto_data, auto_labels = hw3.auto_data_and_labels(auto_data_all, features)
print('auto data and labels shape', auto_data.shape, auto_labels.shape)
```

```
avg and std {}
entries in one_hot field {}
auto data and labels shape (6, 392) (1, 392)
```

In [ ]:

In [172…

```
hw3.xval_learning_alg(
    lambda data, labels: perceptron(data, labels, {"T": 1}),
    auto_data,
    auto_labels,
    10)
```

```
97
97
95
90
98
94
99
97
92
94
```

Out[172]:

```
0.6526282051282052
```

In [173…
```
hw3.xval_learning_alg(
    lambda data, labels: averaged_perceptron(data, labels, {"T": 1}),
    auto_data,
    auto_labels,
    10)
```

Out[173]: 0.8441025641025641

In [174…
```
features2 = [('cylinders', hw3.one_hot),
             ('displacement', hw3.standard),
             ('horsepower', hw3.standard),
             ('weight', hw3.standard),
             ('acceleration', hw3.standard),
             ## Drop model_year by default
             ## ('model_year', hw3.raw),
             ('origin', hw3.one_hot)]

# Construct the standard data and label arrays
auto_data2, auto_labels2 = hw3.auto_data_and_labels(auto_data_all, features2)
print('auto data and labels shape', auto_data2.shape, auto_labels2.shape)
```

```
avg and std {'displacement': (388.3482142857143, 302.0458123396403), 'horsepower': (509.3545918367347, 333.6521151716361), 'weight': (2977.5841836734694, 848.3184465698365), 'acceleration': (15.541326530612228, 2.7553429127509963)}
entries in one_hot field {'cylinders': [3.0, 4.0, 5.0, 6.0, 8.0], 'origin': [1.0, 2.0, 3.0]}
auto data and labels shape (12, 392) (1, 392)
```

In [188…
```
hw3.xval_learning_alg(
    lambda data, labels: perceptron(data, labels, {"T": 1}),
    auto_data2,
    auto_labels2,
    10),  hw3.xval_learning_alg(
    lambda data, labels: averaged_perceptron(data, labels, {"T": 1}),
    auto_data2,
    auto_labels2,
    10)
```

Out[188]: (0.7908333333333333, 0.9004487179487182)

In [182…
```
hw3.xval_learning_alg(
    lambda data, labels: perceptron(data, labels, {"T": 10}),
    auto_data,
    auto_labels,
    10),  hw3.xval_learning_alg(
    lambda data, labels: averaged_perceptron(data, labels, {"T": 10}),
    auto_data,
    auto_labels,
    10)
```

Out[182]: (0.7423076923076924, 0.8366025641025641)

In [177…
```
hw3.xval_learning_alg(
    lambda data, labels: perceptron(data, labels, {"T": 10}),
    auto_data2,
    auto_labels2,
    10),  hw3.xval_learning_alg(
    lambda data, labels: averaged_perceptron(data, labels, {"T": 10}),
    auto_data2,
    auto_labels2,
    10)
```

```
545
540
563
529
546
540
531
495
532
547
```

Out[177]: (0.8061538461538461, 0.8979487179487181)

In [183…
```
hw3.xval_learning_alg(
    lambda data, labels: perceptron(data, labels, {"T": 50}),
    auto_data,
    auto_labels,
    10),  hw3.xval_learning_alg(
    lambda data, labels: averaged_perceptron(data, labels, {"T": 50}),
    auto_data,
    auto_labels,
    10)
```

Out[183]: (0.6909615384615384, 0.8366025641025641)

In [187…
```
hw3.xval_learning_alg(
    lambda data, labels: perceptron(data, labels, {"T": 50}),
    auto_data2,
    auto_labels2,
    10),  hw3.xval_learning_alg(
    lambda data, labels: averaged_perceptron(data, labels, {"T": 50}),
    auto_data2,
    auto_labels2,
    10)
```

Out[187]: (0.8060256410256409, 0.9005128205128207)

In [189…
```
th, th0 = averaged_perceptron(auto_data2, auto_labels2, params={'T':50})
th, th0
```

Out[189]:
```
(array([[-1.98173469],
        [ 0.34622449],
        [ 0.51530612],
        [-0.95596939],
        [ 2.80469388],
        [-1.46206452],
        [ 0.27203955],
        [-6.55860703],
        [ 0.83288456],
        [-0.10352041],
        [ 1.1647449 ],
        [-0.33270408]]),
 array([[0.72852041]]))
```

5) Evaluating algorithmic and feature choices for review data

We have supplied you with the `load_review_data` function, that can be used to read a .tsv file and return the labels and texts. We have also supplied you with the `bag_of_words` function, which takes the raw data and returns a dictionary of unigram words. The resulting dictionary is an input to `extract_bow_feature_vectors` which computes a feature matrix of ones and zeros that can be used as the input for the classification algorithms. The file `hw3_part2_main.py` has code for constructing the data and label arrays. Using these arrays and our implementation of the learning algorithms, you will be able to compute θ and θ_0. You will need to add your (or the one written by staff) implementation of perceptron and averaged perceptron.

In [5]:
```
# Returns lists of dictionaries.  Keys are the column names, 'sentiment' and 'text'.
# The train data has 10,000 examples
review_data = hw3.load_review_data('reviews.tsv')

# Lists texts of reviews and list of labels (1 or -1)
review_texts, review_label_list = zip(*((sample['text'], sample['sentiment']) for sample in review_data))

# The dictionary of all the words for "bag of words"
dictionary = hw3.bag_of_words(review_texts)

# The standard data arrays for the bag of words
review_bow_data = hw3.extract_bow_feature_vectors(review_texts, dictionary)
review_labels = hw3.rv(review_label_list)
print('review_bow_data and labels shape', review_bow_data.shape, review_labels.shape)
```

```
review_bow_data and labels shape (19945, 10000) (1, 10000)
```

In [6]:
```python
import time
```

In [7]:
```python
for T in (1, 10, 50):
    print('T', T)
    start = time.time()
    print('P', xval_learning_alg(
        lambda data, labels: perceptron(data, labels, {"T": T}),
        review_bow_data, review_labels, 10))
    print(time.time() - start)
    start = time.time()
    print('AP', xval_learning_alg(
        lambda data, labels: averaged_perceptron(data, labels, {"T": T}),
        review_bow_data, review_labels, 10))
    print(time.time() - start)
```

```
T 1
P 0.7672000000000001
7.181673765182495
AP 0.8120999999999998
7.737758159637451
T 10
P 0.7871
31.119680643081665
AP 0.8237
38.83753275871277
T 50
P 0.8036
130.36860251426697
AP 0.8157
179.88108468055725
```

In [14]:
```python
th, th0 = averaged_perceptron( review_bow_data, review_labels, {"T": 10})
th, th0
```

Out[14]:
```
(array([[ 0.15984],
        [-2.74048],
        [-1.23668],
        ...,
        [ 0.     ],
        [-1.2001 ],
        [ 0.     ]]),
 array([[-1.72795]]))
```

In [16]:
```python
mysorted = sorted((e, i) for i,e in enumerate(th))
rdict = hw3.reverse_dict(dictionary)
[rdict[i] for (e, i) in mysorted[:10]], [rdict[i] for (e, i) in mysorted[-10:]]
```

Out[16]:
```
(['worst',
  'awful',
  'unfortunately',
  'horrible',
  'stuck',
  'changed',
  'disappointment',
  'bland',
  'poor',
  'formula'],
 ['great',
  'individually',
  'bright',
  'yummy',
  'skeptical',
  'perfect',
  'easily',
  'satisfied',
  'delicious',
  'excellent'])
```

6) Evaluating features for MNIST data

This problem explores how well the perceptron algorithm works to classify images of handwritten digits, from the well-known ("MNIST") dataset, buiding on your thoughts from lab about extracting features from images. This exercise will highlight how important feature extraction is, before linear classification is done, using algorithms such as the perceptron.

Dataset setup

Often, it may be easier to work with a vector whose spatial orientation is preserved. In previous parts, we have represented features as one long feature vector. For images, however, we often represent a m by n image as a `(m,n)` array, rather than a `(mn,1)` array (as the previous parts have done).

In the code file, we have supplied you with the `load_mnist_data` function, which will read from the provided image files and populate a dictionary, with image and label vectors for each numerical digit from 0 to 9. These images are already shaped as `(m,n)` arrays.

In [55]:
```
mnist_data_all = hw3.load_mnist_data(range(10))

print('mnist_data_all loaded. shape of single images is', mnist_data_all[0]["images"][0].shape)

# HINT: change the [0] and [1] if you want to access different images
def get_data_lables(leftd, rightd):
    d0 = mnist_data_all[leftd]["images"]
    d1 = mnist_data_all[rightd]["images"]
    y0 = np.repeat(-1, len(d0)).reshape(1,-1)
    y1 = np.repeat(1, len(d1)).reshape(1,-1)
    # data goes into the feature computation functions
    data = np.vstack((d0, d1))
    # labels can directly go into the perceptron algorithm
    labels = np.vstack((y0.T, y1.T)).T
    return data, labels
```

```
mnist_data_all loaded. shape of single images is (28, 28)
```

In [15]:
```
np.array([np.average([[1, 2, 3], [4, 5, 6], [7, 8, 9]], axis=0)]).T
```

Out[15]:
```
array([[4.],
       [5.],
       [6.]])
```

In [104…
```
arr = np.arange(24).reshape(2,3,4)
print(arr.ndim)
print(arr)
out_ = np.array([
    np.apply_along_axis(
        lambda a: np.average(a),
        axis=0,
        arr=item
    )
    for item
    in arr
])
out_
```

```
3
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Out[104]:
```
array([[ 4.,  5.,  6.,  7.],
       [16., 17., 18., 19.]])
```

In [174…

```python
# change these implementations to support whole datasets

def raw_mnist_features(x):
    """
    @param x (n_samples,m,n) array with values in (0,1)
    @return (m*n,n_samples) reshaped array where each entry is preserved
    """
    n_samples, m, n = x.shape
    return x.reshape((n_samples, n*m, 1))


def row_average_features(x):
    """
    This should either use or modify your code from the tutor questions.

    @param x (n_samples,m,n) array with values in (0,1)
    @return (m,n_samples) array where each entry is the average of a row
    """
    return np.apply_along_axis(
        lambda a: [np.average(a)],
        axis=1,
        arr=x
    )

def col_average_features(x):
    """
    This should either use or modify your code from the tutor questions.

    @param x (n_samples,m,n) array with values in (0,1)
    @return (n,n_samples) array where each entry is the average of a column
    """
    return np.apply_along_axis(
        lambda a: [np.average(a)],
        axis=0,
        arr=x
    ).T

def top_bottom_features(x):
    """
    This should either use or modify your code from the tutor questions.

    @param x (n_samples,m,n) array with values in (0,1)
    @return (2,n_samples) array where the first entry of each column is the average of the
    top half of the image = rows 0 to floor(m/2) [exclusive]
    and the second entry is the average of the bottom half of the image
    = rows floor(m/2) [inclusive] to m
    """
    n, m = x.shape
```

```
    return cv([np.average(x[0:n//2,:]), np.average(a=x[n//2:,:])])
```

In [175…

```
#Your Code Here
ans=row_average_features(np.array([[1,2,3],[3,9,2]])).tolist()
ans
```

Out[175]:

```
[[2.0], [4.666666666666667]]
```

In [176…

```
ans=col_average_features(np.array([[1,2,3],[3,9,2],[2,1,9]])).tolist()
ans
```

Out[176]:

```
[[2.0], [4.0], [4.666666666666667]]
```

In [177…

```
top_bottom_features(np.arange(12).reshape((3,4))), np.arange(12).reshape((3,4))
```

Out[177]:

```
(array([[1.5],
        [7.5]]),
 array([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]]))
```

In [57]:

```
import time
# use this function to evaluate accuracy
#print( data.shape, raw_mnist_features(data).shape, raw_mnist_features(data).T[0].shape )

for l, r in ((0, 1), (2, 4), (6, 8), (9, 0)):
    start = time.time()

    data, labels = get_data_lables(l, r)
    raw_data = raw_mnist_features(data).T[0]
    acc = hw3.get_classification_accuracy(raw_data, labels)
    print('Done', time.time() - start)
    print('left', l)
    print('right', r)
    print(acc)
```

```
Done 0.46145176887512207
left 0
right 1
0.975
Done 0.47538232803344727
left 2
right 4
0.8641666666666665
Done 0.4565155506134033
left 6
right 8
0.9479166666666667
Done 0.500361442565918
left 9
right 0
0.6470833333333333
```

In [ ]:

```
(0.975, 0.8641666666666665, 0.9479166666666667, 0.6470833333333333)
```

In [178…

```
data, labels = get_data_lables(0, 1)
print(data.shape)
rfd = np.concatenate(
    list(
        row_average_features(xi).T
        for xi
        in data
    )
).T
cfd = np.concatenate(
    list(
        col_average_features(xi).T
        for xi
        in data
    )
).T
tbfd = np.concatenate(
    list(
        top_bottom_features(xi).T
        for xi
        in data
    )
).T

print(rfd.shape)
print(cfd.shape)
print(tbfd.shape)

np.concatenate(
    (rfd, cfd, tbfd),
    axis=0
).shape
```

```
(160, 28, 28)
(28, 160)
(28, 160)
(2, 160)
```

Out[178]:

```
(58, 160)
```

In [183…
```
for l, r in ((0, 1), (2, 4), (6, 8), (9, 0)):
    data, labels = get_data_lables(l, r)
    rfd = np.concatenate(
        list(
            row_average_features(xi).T
            for xi
            in data
        )
    ).T
    cfd = np.concatenate(
        list(
            col_average_features(xi).T
            for xi
            in data
        )
    ).T
    tbfd = np.concatenate(
        list(
            top_bottom_features(xi).T
            for xi
            in data
        )
    ).T
    res = []
    for fdata in (rfd, cfd, tbfd):
        acc = hw3.get_classification_accuracy(fdata, labels)
        res += [acc]
    print(repr(res))
```

```
[0.48125, 0.6375, 0.48125]
[0.7754166666666668, 0.49749999999999994, 0.49749999999999994]
[0.92125, 0.52125, 0.5650000000000001]
[0.49749999999999994, 0.5041666666666667, 0.49749999999999994]
```

In [ ]:
```
0.48125,
0.6375,
0.48125
```

> 6.2F) (Optional) What does it mean if a binary classification accuracy is below 0.5, if your dataset is balanced (same number from each class)? Are these datasets balanced?

Means it is worse than randomly picking up labels.

In [188…

```python
# 6.2G) (Optional) Feel free to classify other images from each other. Which combinations perform the best, and which perform t

res = []
for l in range(10):
    for r in range(10):
        if l == r:
            continue
        start = time.time()
        data, labels = get_data_lables(l, r)
        raw_data = raw_mnist_features(data).T[0]
        acc = hw3.get_classification_accuracy(raw_data, labels)
        res += [(acc, l, r)]
sres = sorted(res)
print(sres[:10])
print(sres[-10:])
```

```
[(0.48250000000000004, 5, 8), (0.4841666666666667, 4, 9), (0.5079166666666667, 4, 6), (0.525, 1, 8), (0.5429166666666666, 9, 8), (0.5487500000000001, 8, 5), (0.575, 5, 3), (0.5758333333333334, 3, 5), (0.5912499999999999, 7, 3), (0.5954166666666667, 0, 5)]
[(0.9737500000000001, 9, 2), (0.975, 0, 1), (0.975, 1, 0), (0.975, 9, 1), (0.9808333333333333, 7, 5), (0.98125, 1, 4), (0.98125, 5, 7), (0.9875, 4, 1), (0.9875, 4, 3), (0.99375, 8, 0)]
```

In [194…

```python
print('Best 10:')
print('\n'.join('{} vs {}: {}'.format(l, r, a) for a,l,r in reversed(sres[-10:])))
print('Worst 10:')
print('\n'.join('{} vs {}: {}'.format(l, r, a) for a,l,r in sres[:10]))
```

```
Best 10:
8 vs 0: 0.99375
4 vs 3: 0.9875
4 vs 1: 0.9875
5 vs 7: 0.98125
1 vs 4: 0.98125
7 vs 5: 0.9808333333333333
9 vs 1: 0.975
1 vs 0: 0.975
0 vs 1: 0.975
9 vs 2: 0.9737500000000001
Worst 10:
5 vs 8: 0.48250000000000004
4 vs 9: 0.4841666666666667
4 vs 6: 0.5079166666666667
1 vs 8: 0.525
9 vs 8: 0.5429166666666666
8 vs 5: 0.5487500000000001
5 vs 3: 0.575
3 vs 5: 0.5758333333333334
7 vs 3: 0.5912499999999999
0 vs 5: 0.5954166666666667
```

In [ ]: