MIT 6.036 Spring 2019: Homework 4

This homework does not include provided Python code. Instead, we encourage you to write your own code to help you answer some of these problems, and/or test and debug the code components we do ask for. Some of the problems below are simple enough that hand calculation should be possible; your hand solutions can serve as test cases for your code. You may also find that including utilities written in previous labs (like a `sd` or signed distance function) will be helpful, as you build up additional functions and utilities for calculation of margins, different loss functions, gradients, and other functions needed for margin maximization and gradient descent.

In [1]:
```
import numpy as np
```

1)

In [2]:
```
data = np.array([[1, 2, 1, 2, 10, 10.3, 10.5, 10.7],
                 [1, 1, 2, 2,  2,  2,  2, 2]])
labels = np.array([[-1, -1, 1, 1, 1, 1, 1, 1]])
blue_th = np.array([[0, 1]]).T
blue_th0 = -1.5
red_th = np.array([[1, 0]]).T
red_th0 = -2.5
```

In [3]:
```
def gamma(x, y, th, th0):
    return y * (th.T @ x + th0) / np.linalg.norm(th)

def s_sum(x, y, th, th0):
    return sum(gamma(x[:,i:i+1], y[:,i], th, th0) for i in range(x.shape[1]))[0][0]

def s_min(x, y, th, th0):
    return min(gamma(x[:,i:i+1], y[:,i], th, th0) for i in range(x.shape[1]))[0][0]

def s_max(x, y, th, th0):
    return max(gamma(x[:,i:i+1], y[:,i], th, th0) for i in range(x.shape[1]))[0][0]
```

In [34]:
```
[
    s_sum(data, labels, red_th, red_th0),
    s_min(data, labels, red_th, red_th0),
    s_max(data, labels, red_th, red_th0)
]
```

Out[34]:
```
[31.5, -1.5, 8.2]
```

In [35]:
```
[
    s_sum(data, labels, blue_th, blue_th0),
    s_min(data, labels, blue_th, blue_th0),
    s_max(data, labels, blue_th, blue_th0)
]
```

Out[35]:
```
[4.0, 0.5, 0.5]
```

3)

In [2]:
```
np.linalg.norm([[-0.0737901],[2.40847205]]), np.linalg.norm([[-0.23069578],[2.5573550
```

Out[2]:
```
(2.409602165190182, 2.567739315055543)
```

In [3]:
```
np.linalg.norm([[-0.01280916],[-1.42043497]]), np.linalg.norm([[0.45589866],[-4.50220
```

Out[3]:
```
(1.4204927238739404, 4.525230920153827, 4.131878940912039)
```

4)

```
            [1, 2],
            [2, 3],
            [3, 5],
            [1, 4]
   1  2  3  1
   2  3  5  4
```

1+4+9+1 2+6+15+4 2+6+15+4 4+9+25+16

15 27 27 44

15 *44 - 27* 27

-69

6) Implementing gradient descent

In this section we will implement generic versions of gradient descent and apply these to the SVM objective.

Note: If you need a refresher on gradient descent, you may want to reference this week's notes.

6.1) Implementing Gradient Descent

We want to find the x that minimizes the value of the *objective function* $f(x)$, for an arbitrary scalar function f. The function f will be implemented as a Python function of one argument, that will be a numpy column vector. For efficiency, we will work with Python functions that return not just the value of f at $f(x)$ but also return the gradient vector at x, that is, $\nabla_x f(x)$.

We will now implement a generic gradient descent function, `gd`, that has the following input arguments:

- `f` : a function whose input is an `x`, a column vector, and returns a scalar.
- `df` : a function whose input is an `x`, a column vector, and returns a column vector representing the gradient of `f` at `x`.
- `x0` : an initial value of x, `x0`, which is a column vector.
- `step_size_fn` : a function that is given the iteration index (an integer) and returns a step size.
- `max_iter` : the number of iterations to perform

Our function `gd` returns a tuple:

- `x` : the value at the final step
- `fs` : the list of values of `f` found during all the iterations (including `f(x0)`)
- `xs` : the list of values of `x` found during all the iterations (including `x0`)

Hint: This is a short function!

Hint 2: If you do `temp_x = x` where `x` is a vector (numpy array), then `temp_x` is just another name for the same vector as `x` and changing an entry in one will change an entry in the other. You should either use `x.copy()` or remember to change entries back after modification.

Some utilities you may find useful are included below.

In [4]:
```
def rv(value_list):
    return np.array([value_list])

def cv(value_list):
    return np.transpose(rv(value_list))

def f1(x):
    return float((2 * x + 3)**2)

def df1(x):
    return 2 * 2 * (2 * x + 3)

def f2(v):
    x = float(v[0]); y = float(v[1])
    return (x - 2.) * (x - 3.) * (x + 3.) * (x + 1.) + (x + y -1)**2

def df2(v):
    x = float(v[0]); y = float(v[1])
    return cv([(-3. + x) * (-2. + x) * (1. + x) + \
               (-3. + x) * (-2. + x) * (3. + x) + \
               (-3. + x) * (1. + x) * (3. + x) + \
               (-2. + x) * (1. + x) * (3. + x) + \
               2 * (-1. + x + y),
               2 * (-1. + x + y)])
```

The main function to implement is `gd`, defined below.

In [8]:
```
def gd(f, df, x0, step_size_fn, max_iter):
    fs = [f(x0)]
    xs = [x0]
    for t in range(max_iter):
        xs += [xs[-1] - step_size_fn(t) * df(xs[-1])]
        fs += [f(xs[-1])]
    return xs[-1], fs, xs
```

To evaluate results, we also use a simple `package_ans` function, which checks the final `x`, as well as the first and last values in `fs`, `xs`.

In [9]:
```
def package_ans(gd_vals):
    x, fs, xs = gd_vals
    return [x.tolist(), [fs[0], fs[-1]], [xs[0].tolist(), xs[-1].tolist()]]
```

The test cases are provided below, but you should feel free (and are encouraged!) to write more of your own.

In [10]:
```
# Test case 1
ans=package_ans(gd(f1, df1, cv([0.]), lambda i: 0.1, 1000))
print(ans)

# Test case 2
ans=package_ans(gd(f2, df2, cv([0., 0.]), lambda i: 0.01, 1000))
print(ans)
```

```
[[[-1.5]], [9.0, 0.0], [[[0.0]], [[-1.5]]]]
[[[-2.2058239041648853], [3.205823890926977]], [19.0, -20.967239611348752], [[[0.0],
[0.0]], [[-2.2058239041648853], [3.205823890926977]]]]
```

6.2) Numerical Gradient

Getting the analytic gradient correct for complicated functions is tricky. A very handy method of verifying the analytic gradient or even substituting for it is to estimate the gradient at a point by means of *finite differences*.

Assume that we are given a function $f(x)$ that takes a column vector as its argument and returns a scalar value. In gradient descent, we will want to estimate the gradient of f at a particular $x_0.$

The i^{th} component of $\nabla_x f(x_0)$ can be estimated as $$\frac{f(x_0+\delta^{i}) - f(x_0-\delta^{i})}{2\delta}$$ where δ^{i} is a column vector whose i^{th} coordinate is δ, a small constant such as 0.001, and whose other components are zero. Note that adding or subtracting δ^{i} is the same as incrementing or decrementing the i^{th} component of x_0 by δ, leaving the other components of x_0 unchanged. Using these results, we can estimate the i^{th} component of the gradient.

For example, if $x_0 = (1,1,\dots,1)^T$ and $\delta = 0.01$, we may approximate the first component of $\nabla_x f(x_0)$ as $$\frac{f((1,1,1,\dots)^T+(0.01,0,0,\dots)^T) - f((1,1,1,\dots)^T-(0.01,0,0,\dots)^T)}{2\cdot 0.01}.$$ (We add the transpose so that these are column vectors.) **This process should be done for each dimension independently, and together the results of each computation are compiled to give the estimated gradient, which is d dimensional.**

Implement this as a function `num_grad` that takes as arguments the objective function `f` and a value of `delta`, and returns a new **function** that takes an `x` (a column vector of parameters) and returns a gradient column vector.

Note: As in the previous part, make sure you do not modify your input vector.

In [11]:
```
def num_grad(f, delta=0.001):
    def df(x):
        d, n = x.shape
        ds = np.identity(d) * delta
        gr = np.zeros((d, n))
        for i in range(d):
            gr[i] = ( f(x + ds[:,i:i+1]) - f(x - ds[:,i:i+1]) ) / (2 * delta)
        return gr
    return df
```

The test cases are shown below; these use the functions defined in the previous exercise.

In [83]:
```
x = cv([0.])
#ans=(num_grad(f1)(x).tolist(), x.tolist())
#print(ans)

x = cv([0.1])
#ans=(num_grad(f1)(x).tolist(), x.tolist())
#print(ans)

x = cv([0., 0.])
ans=(num_grad(f2)(x).tolist(), x.tolist())
expected = [[6.99999899999959], [-2.000000000000668]],
print(ans ,expected)
print(ans[0] == expected)


x = cv([0.1, -0.1])
ans=(num_grad(f2)(x).tolist(), x.tolist())
print(ans)
```

```
ds [[0.001 0.   ]
 [0.    0.001]]
ds[:,i:i+1] [[0.001]
 [0.   ]]
i gr 0 [[6.999999]
 [0.      ]]
ds[:,i:i+1] [[0.   ]
 [0.001]]
i gr 1 [[ 6.999999]
 [-2.      ]]
([[6.99999899999959], [-2.000000000000668]], [[0.0], [0.0]]) ([[6.99999899999959], [-
2.000000000000668]],)
False
ds [[0.001 0.   ]
 [0.    0.001]]
ds[:,i:i+1] [[0.001]
 [0.   ]]
i gr 0 [[4.7739994]
 [0.       ]]
ds[:,i:i+1] [[0.   ]
 [0.001]]
i gr 1 [[ 4.7739994]
 [-2.       ]]
([[4.7739994000011166], [-2.000000000000668]], [[0.1], [-0.1]])
```

In [ ]:

A faster (one function evaluation per entry), though sometimes less accurate, estimate is to use: $$\frac{f(x_0+\delta^{i}) - f(x_0)}{\delta}$$ for the i^{th} component of $\nabla_x f(x_0).$

6.3) Using the Numerical Gradient

Recall that our generic gradient descent function takes both a function `f` that returns the value of our function at a given point, and `df`, a function that returns a gradient at a given point. Write a function `minimize` that takes only a function `f` and uses this function and numerical gradient descent to return the local minimum. We have provided you with our implementations of `num_grad` and `gd`, so you should not redefine them in the code box below. You may use the default of `delta=0.001` for `num_grad`.

Hint: Your definition of `minimize` should call `num_grad` exactly once, to return a function that is called many times. You should return the same outputs as `gd`.

In [12]:
```
def minimize(f, x0, step_size_fn, max_iter):
    fs = [f(x0)]
    xs = [x0]
    df = num_grad(f, delta=0.001)
    for t in range(max_iter):
        xs += [xs[-1] - step_size_fn(t) * df(xs[-1])]
        fs += [f(xs[-1])]
    return xs[-1], fs, xs
```

The test cases are below.

In [13]:
```
ans = package_ans(minimize(f1, cv([0.]), lambda i: 0.1, 1000))
print(ans)

ans = package_ans(minimize(f2, cv([0., 0.]), lambda i: 0.01, 1000))
print(ans)
```

```
[[[-1.5]], [9.0, 0.0], [[[0.0]], [[-1.5]]]]
[[[-2.2058237062057517], [3.205823692967833]], [19.0, -20.967239611347775], [[[0.0],
[0.0]], [[-2.2058237062057517], [3.205823692967833]]]]
```

7) Applying gradient descent to SVM objective

Note: In this section, you will code many individual functions, each of which depends on previous ones. We **strongly recommend** that you test each of the components on your own to debug.

7.1) Calculating the SVM objective

Implement the single-argument hinge function, which computes L_h, and use that to implement hinge loss for a data point and separator. Using the latter function, implement the SVM objective. Note that these functions should work for matrix/vector arguments, so that we can compute the objective for a whole dataset with one call.

```
x is d x n, y is 1 x n, th is d x 1, th0 is 1 x 1, lam is a scalar
```

Hint: Look at `np.where` for implementing `hinge`.

In [6]:
```
a = np.arange(10)
np.where(a < 5, a, 0)
```

Out[6]:
```
array([0, 1, 2, 3, 4, 0, 0, 0, 0, 0])
```

In [14]:
```
def hinge(v):
    return max(0, 1 - v)

# x is dxn, y is 1xn, th is dx1, th0 is 1x1
def hinge_loss(x, y, th, th0):
    return np.where(
        y * (th.T @ x + th0) < 1,
        1 - y * (th.T @ x + th0),
        0)

# x is dxn, y is 1xn, th is dx1, th0 is 1x1, lam is a scalar
def svm_obj(x, y, th, th0, lam):
    n = y.shape[1]
    return np.sum(hinge_loss(x, y, th, th0) + lam * np.linalg.norm(th)**2) / n
```

In [15]:
```
# add your tests here

assert hinge(1) == 0, 'Actual: "{}"'.format(hinge(1))
assert hinge(-1) == 2
assert hinge(1.5) == 0
assert hinge(0) == 1
assert hinge(0.5) == 0.5
```

In the test cases for this problem, we'll use the following `super_simple_separable` test dataset and test separator for some of the tests. A couple of the test cases are also shown below.

In [16]:
```
def super_simple_separable():
    X = np.array([[2, 3, 9, 12],
                  [5, 2, 6, 5]])
    y = np.array([[1, -1, 1, -1]])
    return X, y

sep_e_separator = np.array([[-0.40338351], [1.1849563]]), np.array([[-2.26910091]])

# Test case 1
x_1, y_1 = super_simple_separable()
th1, th1_0 = sep_e_separator
ans = svm_obj(x_1, y_1, th1, th1_0, .1)
print(ans)

# Test case 2
ans = svm_obj(x_1, y_1, th1, th1_0, 0.0)
print(ans)
```

```
0.15668396890496103
0.0
```

7.2) Calculating the SVM gradient

Define a function `svm_obj_grad` that returns the gradient of the SVM objective function with respect to θ and θ_0 in a single column vector. The last component of the gradient vector should be the partial derivative with respect to θ_0. Look at `np.vstack` as a simple way of stacking two matrices/vectors vertically. We have broken it down into pieces that mimic steps in the chain rule; this leads to code that is a bit inefficient but easier to write and debug. We can worry about efficiency later.

In [ ]:
```
```

In [17]:
```python
#eta = 1
# Returns the gradient of hinge(v) with respect to v.
def d_hinge(v):
    return np.where(v >= 1, 0, -1)

# Returns the gradient of hinge_loss(x, y, th, th0) with respect to th
def d_hinge_loss_th(x, y, th, th0):
    d, n = x.shape
    g = np.where(y * (th.T @ x + th0) < 1, -1, 0)
    return x * y * g

# Returns the gradient of hinge_loss(x, y, th, th0) with respect to th0
def d_hinge_loss_th0(x, y, th, th0):
    return y * np.where(y * (th.T @ x + th0) < 1, -1, 0)

# Returns the gradient of svm_obj(x, y, th, th0) with respect to th
def d_svm_obj_th(x, y, th, th0, lam):
    d, n = x.shape
    # x - d x n
    # y - 1 x n
    dhlth = d_hinge_loss_th(x, y, th, th0)
    assert dhlth.shape == (d, n)
    s = np.sum(dhlth, axis=1, keepdims=True)
    return s / n  + 2 * lam * th

# Returns the gradient of svm_obj(x, y, th, th0) with respect to th0
def d_svm_obj_th0(x, y, th, th0, lam):
    d, n = x.shape
    dth0s = d_hinge_loss_th0(x, y, th, th0)
    assert dth0s.shape == (1, n)
    return np.sum(dth0s, axis=1, keepdims=True) / n

# Returns the full gradient as a single vector
def svm_obj_grad(X, y, th, th0, lam):
    return np.vstack((
        d_svm_obj_th(X, y, th, th0, lam),
        d_svm_obj_th0(X, y, th, th0, lam)
    ))
```

Some test cases that may be of use are shown below.

In [18]:
```python
X1 = np.array([[1, 2, 3, 9, 10]])
y1 = np.array([[1, 1, 1, -1, -1]])
th1, th10 = np.array([[-0.31202807]]), np.array([[1.834     ]])
X2 = np.array([[2, 3, 9, 12],
               [5, 2, 6, 5]])
y2 = np.array([[1, -1, 1, -1]])
th2, th20=np.array([[ -3.,  15.]]).T, np.array([[ 2.]])
```

In [19]:
```python
X2 * X2
```

Out[19]:
```
array([[  4,   9,  81, 144],
       [ 25,   4,  36,  25]])
```

In [20]:
```python
(
d_hinge(np.array([[ 71.]])).tolist(),
d_hinge(np.array([[ -23.]])).tolist(),
d_hinge(np.array([[ 71, -23.]])).tolist(),
)
```

Out[20]:
```
([[0]], [[-1]], [[0, -1]])
```

In [21]:
```python
def mytest(actual, expected):
    assert actual == expected, f"Actual: '{actual}' <> Expected: '{expected}'"
```

In [22]:
```python
mytest(d_hinge_loss_th(X2[:,0:1], y2[:,0:1], th2, th20).tolist(), [[0], [0]])

#d_hinge_loss_th(X2, y2, th2, th20).tolist()
#d_hinge_loss_th0(X2[:,0:1], y2[:,0:1], th2, th20).tolist()
#d_hinge_loss_th0(X2, y2, th2, th20).tolist()

mytest(
    d_hinge_loss_th(X2, y2, th2, th20).tolist(),
    [[0, 3, 0, 12], [0, 2, 0, 5]]
)

mytest(
    d_hinge_loss_th0(X2, y2, th2, th20).tolist() ,
[[0.0, 1.0, 0.0, 1.0]]
)
```

In [23]:
```python
mytest(
    d_svm_obj_th(X2[:,0:1], y2[:,0:1], th2, th20, 0.01).tolist(),
    [[-0.06], [0.3]]
)

#d_svm_obj_th(X2[:,0:1], y2[:,0:1], th2, th20, 0.01).tolist()
#d_svm_obj_th(X2, y2, th2, th20, 0.01).tolist()

#d_svm_obj_th0(X2[:,0:1], y2[:,0:1], th2, th20, 0.01).tolist()
# d_svm_obj_th0(X2, y2, th2, th20, 0.01).tolist()
```

In [24]:
```python
mytest(
    d_svm_obj_th(X2, y2, th2, th20, 0.01).tolist(),
    [[3.69], [2.05]]
)
```

In [25]:
```python
mytest(
    d_svm_obj_th0(X2, y2, th2, th20, 0.01).tolist(),
    [[0.5]]
)
```

In [26]:
```python
svm_obj_grad(X2, y2, th2, th20, 0.01).tolist()
svm_obj_grad(X2[:,0:1], y2[:,0:1], th2, th20, 0.01).tolist()
```

Out[26]:
```
[[-0.06], [0.3], [0.0]]
```

7.3) Batch SVM minimize

Putting it all together, use the functions you built earlier to write a gradient descent minimizer for the SVM objective. You do not need to paste in your previous definitions; you can just call the ones defined by the staff. You will need to call `gd`, which is already defined for you as well; your function `batch_svm_min` should return the values that `gd` does.

- Initialize all the separator parameters to zero,
- use the step size function provided below, and
- specify 10 iterations.

In [59]:
```
def batch_svm_min(data, labels, lam):
    def svm_min_step_size_fn(i):
       return 2/(i+1)**0.5
    d, n = data.shape
    x0 = np.vstack((
        np.repeat(0, d).reshape((d,1)),
        np.repeat(0, 1)
    ))
    print(x0[0:d])
    return gd(
        lambda x: svm_obj(data, labels, x[0:d], x[d:], lam),
        lambda x: svm_obj_grad(data, labels, x[0:d], x[d:], lam),
        x0,
        svm_min_step_size_fn,
        10
    )
```

Test cases are shown below, where an additional separable test data set has been specified.

In [58]:
```
def separable_medium():
    X = np.array([[2, -1, 1, 1],
                  [-2, 2, 2, -1]])
    y = np.array([[1, -1, 1, -1]])
    return X, y
sep_m_separator = np.array([[ 2.69231855], [ 0.67624906]]), np.array([[-3.02402521]])

x_1, y_1 = super_simple_separable()
ans = package_ans(batch_svm_min(x_1, y_1, 0.0001))
x_1,y_1=super_simple_separable()
ans=package_ans(batch_svm_min(x_1, y_1, 0.0001))

mytest(ans,
[
    [[-2.430041469694636], [3.7742410656579057], [-0.40377305279098563]],
    [1.0, 0.37283613860066195],
    [
        [[0.0], [0.0], [0.0]],
        [[-2.430041469694636], [3.7742410656579057], [-0.40377305279098563]]
    ]

])
x_1, y_1 = separable_medium()
ans = package_ans(batch_svm_min(x_1, y_1, 0.0001))
```

```
[[0]
 [0]]
[[0]
 [0]]
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Input In [58], in <cell line: 13>()
     10 x_1,y_1=super_simple_separable()
     11 ans=package_ans(batch_svm_min(x_1, y_1, 0.0001))
---> 13 mytest(ans,
     14 [
     15     [[-2.430041469694636], [3.7742410656579057], [-0.40377305279098563]],
     16     [1.0, 0.37283613860066195],
     17     [
     18         [[0.0], [0.0], [0.0]],
     19         [[-2.430041469694636], [3.7742410656579057], [-0.40377305279098563]]
     20     ]
     21 
     22 ])
     23 x_1, y_1 = separable_medium()
     24 ans = package_ans(batch_svm_min(x_1, y_1, 0.0001))

Input In [21], in mytest(actual, expected)
      1 def mytest(actual, expected):
----> 2     assert actual == expected, f"Actual: '{actual}' <> Expected: '{expecte
d}'"

AssertionError: Actual: '[[[-1.4810507930100065], [4.406219189763341], [-0.4037730527
9098563]], [1.0, 2.457217409802802], [[[0], [0], [0]], [[-1.4810507930100065], [4.406
219189763341], [-0.40377305279098563]]]]' <> Expected: '[[[-2.430041469694636], [3.77
42410656579057], [-0.40377305279098563]], [1.0, 0.37283613860066195], [[[0.0], [0.0],
[0.0]], [[-2.430041469694636], [3.7742410656579057], [-0.40377305279098563]]]]'
```

7.4) Numerical SVM objective (Optional)

Recall from the previous question that we were able to closely approximate gradients with numerical estimates. We may apply the same technique to optimize the SVM objective.

Using your definition of `minimize` and `num_grad` from the previous problem, implement a function that optimizes the SVM objective through numeric approximations.

How well does this function perform, compared to the analytical result? Consider both accuracy and runtime.

In [ ]:
```
# your code here
```