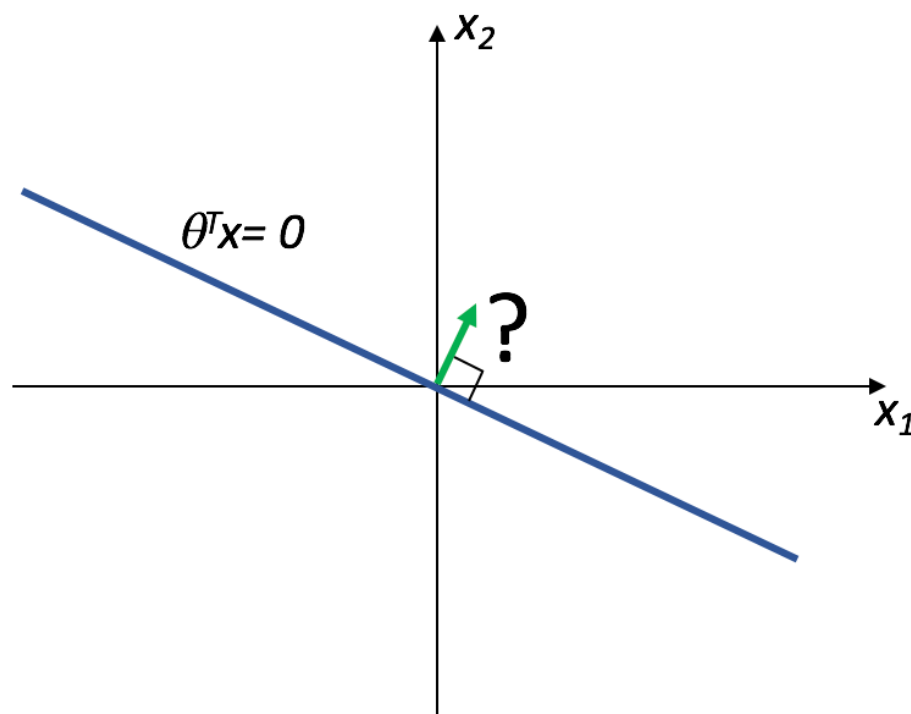


These warmup exercises are divided into two sections:

- Some linear algebra foundational to machine learning
- An introduction to numpy

1) Hyperplanes

We will be using the notion of a **hyperplane** a great deal. A hyperplane is useful for classification, as discussed in the [notes](#).



Some notational conventions:

- x : is a point in d -dimensional space (represented as a column vector of d real numbers), \mathbb{R}^d

- θ : a point in d -dimensional space (represented as a column vector of d real numbers), R^d
- θ_0 : a single real number

We represent x and θ as column vectors, that is, $d \times 1$ arrays. Remember dot products? We write dot products in one of two ways: $\theta^T x$ or $\theta \cdot x$. In both cases:

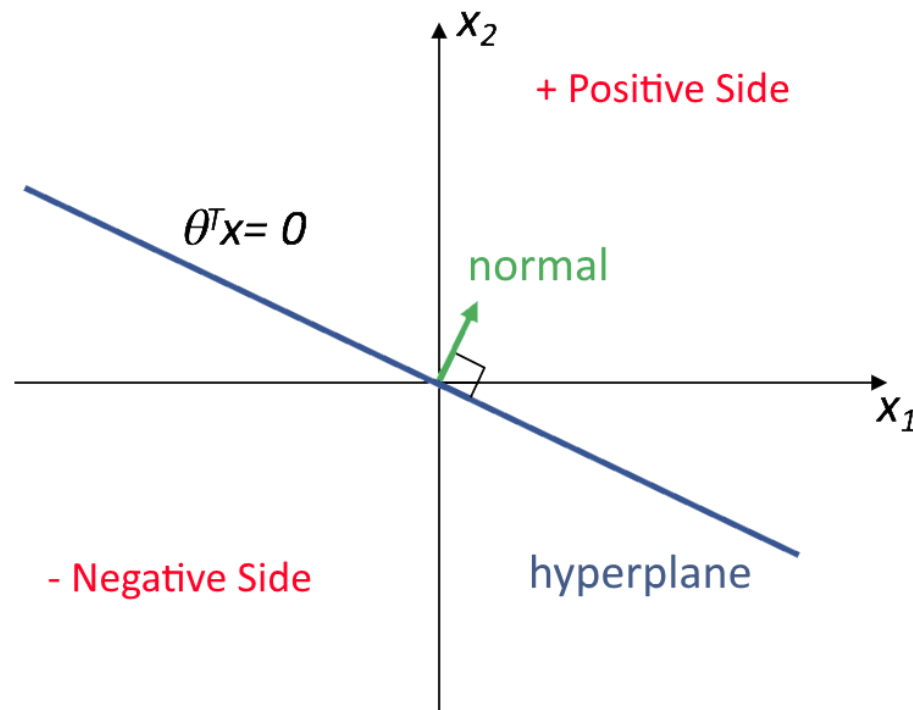
$$\theta^T x = \theta \cdot x = \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_d x_d$$

In a d -dimensional space, a hyperplane is a $d - 1$ dimensional subspace that can be characterized by a normal to the subspace and a scalar offset. For example, any line is a hyperplane in two dimensions, an infinite flat sheet is a hyperplane in three dimensions, but in higher dimensions, hyperplanes are harder to visualize. Fortunately, they are easy to specify.

Hint: When doing the two-dimensional problems below, start by drawing a picture. That should help your intuition. If you get stuck, take a look at this [geometry review for planes and hyperplanes](#).

1.1) Through origin

In d dimensions, any vector $\theta \in R^d$ can define a hyperplane. Specifically, the hyperplane through the origin associated with θ is the set of all vectors $x \in R^d$ such that $\theta^T x = 0$. Note that this hyperplane includes the origin, since $x = 0$ is in the set.



Ex1.1a: In two dimensions, $\theta = [\theta_1, \theta_2]$ can define a hyperplane. Let $\theta = [1, 2]$. Give a vector that lies on the hyperplane given by the set of all $x \in \mathbb{R}^2$ such that $\theta^T x = 0$:

Enter your answer as a Python list of numbers.

Vector on hyperplane:

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Ex1.1b. Using the same hyperplane, determine a vector that is normal to the hyperplane.

Vector normal to hyperplane:

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Ex1.1c. Now, in d dimensions, supply the simplified formula for a **unit** vector normal to the hyperplane in terms of θ where $\theta \in \mathbb{R}^d$.

In this question and the subsequent ones that ask for a formula, enter your answer as a Python expression. Use `theta` for θ , `theta_0` for θ_0 , `x` for any array x , `transpose(x)` for transpose of an array, `norm(x)` for the length (L2-norm) of a vector, and `x@y` to indicate a matrix product of two arrays.

Formula for unit vector normal to hyperplane:

Check Syntax

Submit

View Answer

Ask for Help

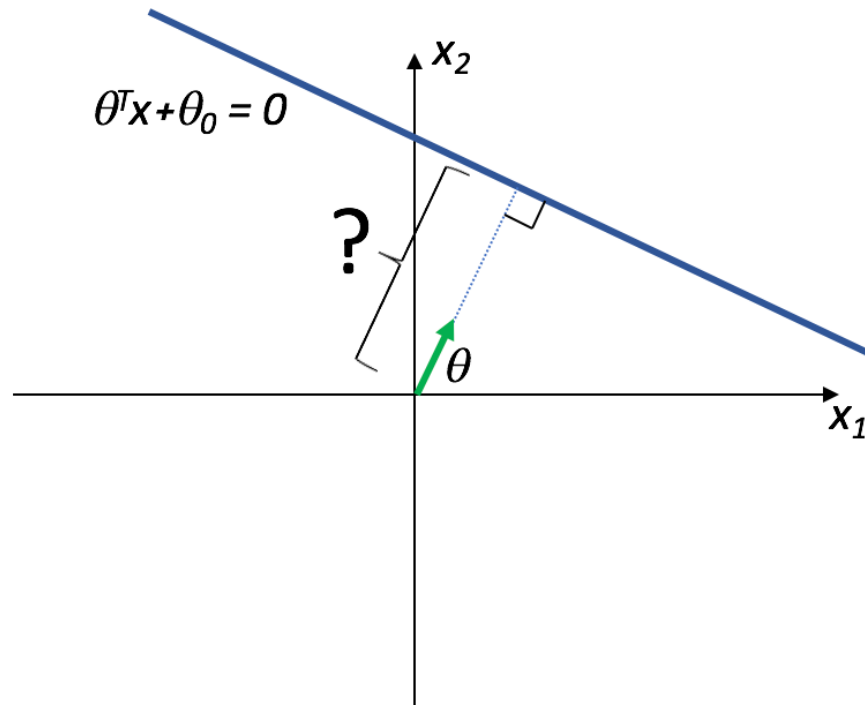
100.00%

You have infinitely many submissions remaining.

1.2) General hyperplane, distance to origin

Now, we'll consider hyperplanes defined by $\theta^T x + \theta_0 = 0$, which do **not** necessarily go through the origin. Distances from points to such general hyperplanes are useful in machine learning models, such as the perceptron ([as described in the notes](#)).

Define the *positive* side of a hyperplane to be the half-space defined by $\{x \mid \theta^T x + \theta_0 > 0\}$, so θ points toward the positive side.



Ex1.2a. In two dimensions, let $\theta = [3, 4]$ and $\theta_0 = 5$. What is the **signed** perpendicular distance from the hyperplane to the origin? The distance should be positive if the origin is on the positive side of the hyperplane, 0 on the hyperplane and negative otherwise. It may be helpful to draw your own picture of the hyperplane (like the one above but with the right intercepts and slopes) with $\theta = [3, 4]$ and $\theta_0 = 5$. **Hint -Draw a picture**

Distance =

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Ex1.2b: Now, in d dimensions, supply the formula for the signed perpendicular distance from a hyperplane specified by θ, θ_0 to the origin. If you

get stuck, take a look at this [walkthrough of point-plane distances](#).

Formula for signed distance to origin:

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

2) Numpy intro

`numpy` is a package for doing a variety of numerical computations in Python. We will use it extensively. It supports writing very compact and efficient code for handling arrays of data. We will start every code file that uses `numpy` with `import numpy as np`, so that we can reference numpy functions with the 'np.' precedent.

You can find general documentation on `numpy` [here](#), and we also have a [6.036-specific numpy tutorial](#).

The fundamental data type in numpy is the multidimensional array, and arrays are usually generated from a nested list of values using the `np.array` command. Every `array` has a `shape` attribute which is a tuple of dimension sizes.

In this class, we will use two-dimensional arrays almost exclusively. That is, **we will use 2D arrays to represent both matrices and vectors!** This is one of several times where we will seem to be unnecessarily fussy about how we construct and manipulate vectors and matrices, but we have our reasons. We have found that using this format results in predictable results from numpy operations.

Using 2D arrays for matrices is clear enough, but what about column and row vectors? We will represent a column vector as a $d \times 1$ array and a row vector as a $1 \times d$ array. So for example, we will represent the three-element column vector,

$$x = \begin{bmatrix} 1 \\ 5 \\ 3 \end{bmatrix},$$

as a 3×1 numpy array. This array can be generated with

```
x = np.array([ [1], [5], [3] ]),
```

or by using the transpose of a 1×3 array (a row vector) as in,

```
x = np.transpose(np.array([ [1,5,3] ]),
```

where you should take note of the "double" brackets.

It is often more convenient to use the array attribute `.T` , as in

```
x = np.array([ [1,5,3] ]).T
```

to compute the transpose.

Before you begin, we would like to note that in this assignment we will not accept answers that use "loops". One reason for avoiding loops is efficiency. For many operations, numpy calls a compiled library written in C, and the library is far faster than that interpreted Python (in part due to the low-level nature of C, optimizations like vectorization, and in some cases, parallelization). But the more important reason for avoiding loops is that using higher-level constructs leads to simpler code that is easier to debug. So, we expect that you should be able to transform loop operations into equivalent operations on numpy arrays, and we will practice this in this assignment.

Of course, there will be more complex algorithms that require loops, but when manipulating matrices you should always look for a solution without loops.

Numpy functions and features you should be familiar with for this assignment:

- `np.array`
- `np.transpose` (and the equivalent method `a.T`)
- `np.ndarray.shape`
- `np.dot` (and the equivalent method `a.dot(b)`)

- `np.sign`
- `np.sum` (look at the `axis` and `keepdims` arguments)
- Elementwise operators `+`, `-`, `*`, `/`

Note that in Python, `np.dot(a, b)` is the matrix product `a@b`, not the dot product $a^T b$.

2.1) Array

Provide an expression that sets `A` to be a 2×3 `numpy` array (2 rows by 3 columns), containing any values you wish.

```
1 import numpy as np
2 A = np.array([[1,2,3], [4,5,6]])
```

[Run Code](#)[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

2.2) Transpose

Write a procedure that takes an array and returns the transpose of the array. You can use 'np.transpose' or the '.T', but you may not use a loop.

```
1 import numpy as np
2 def tp(A):
3     return A.T
4
```

[Run Code](#)[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

2.3) Shapes

Let **A** be a 4×2 numpy array, **B** be a 4×3 array, and **C** be a 4×1 array. For each of the following expressions, indicate the shape of the result as a tuple of integers (**recall python tuples use parentheses, not square brackets, which are for lists, and a tuple of a single object x is written as $(x,)$ with a comma**) or "none" (as a Python string with quotes) if it is illegal.

Ex2.3a

`C * C`

(4, 1)

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Ex2.3b

`np.dot(C, C)`

"none"

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Ex2.3c

`np.dot(np.transpose(C), C)`

(1,1)

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Ex2.3d

`np.dot(A, B)`

"none"

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Ex2.3e

`np.dot(A.T, B)`

(2,3)

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Ex2.3f

`D = np.array([1,2,3])`

(3,)

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Ex2.3g

A[:,1]

(4,)

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Ex2.3h

A[:,1:2]

(4,1)

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

2.4) Row vector

Write a procedure that takes a list of numbers and returns a 2D numpy array representing a **row** vector containing those numbers.

```
1 import numpy as np
2 def rv(value_list):
3     return np.array([value_list])
4
```

[Run Code](#)[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

2.5) Column vector

Write a procedure that takes a list of numbers and returns a 2D numpy array representing a column vector containing those numbers. You can use the `rv` procedure.

```
1 import numpy as np
2 def cv(value_list):
3     return np.array([value_list]).T
4
```

[Run Code](#)[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

2.6) length

Write a procedure that takes a column vector and returns the vector's Euclidean length (or equivalently, its magnitude) as a [scalar](#). You may not use `np.linalg.norm`, and you may not use a loop.

```
1 import numpy as np
2 def length(col_v):
3     return ((col_v.T@col_v)**.5)[0,0]
4
```

[Run Code](#)[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

2.7) normalize

Write a procedure that takes a column vector and returns a unit vector in the same direction. You may not use a for loop. Use your `length` procedure from above (you do not need to define it again).

```
1 import numpy as np
2 def normalize(col_v):
3     return col_v/length(col_v)
4
```

[Run Code](#)[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

2.8) indexing

Write a procedure that takes a 2D array and returns the final column as a two dimensional array. You may not use a for loop.


```
1 import numpy as np
2 def index_final_col(A):
3     return A[:, -1:]
```

[Run Code](#)[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

2.9) Representing data

Alice has collected weight and height data of 3 people and has written it down below:

Weight, height

150, 5.8

130, 5.5

120, 5.3

She wants to put this into a numpy array such that each row represents one individual's height and weight in the order listed. Write code to set `data` equal to the appropriate numpy array:

```
1 import numpy as np
2 data = np.array([
3     [150, 5.8],
4     [130, 5.5],
5     [120, 5.3]
6 ])
```

[Run Code](#)[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

Now she wants to compute the sum of each person's height and weight as a column vector by multiplying `data` by another numpy array. She has written the following incorrect code to do so and needs your help to fix it:

2.10) Matrix multiplication

```
1 import numpy as np
2 def transform(data):
3     return data.sum(axis=1, keepdims=True)
4     #return (np.dot(data.T, np.array([1, 1]).T))
5
```

[Run Code](#)[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

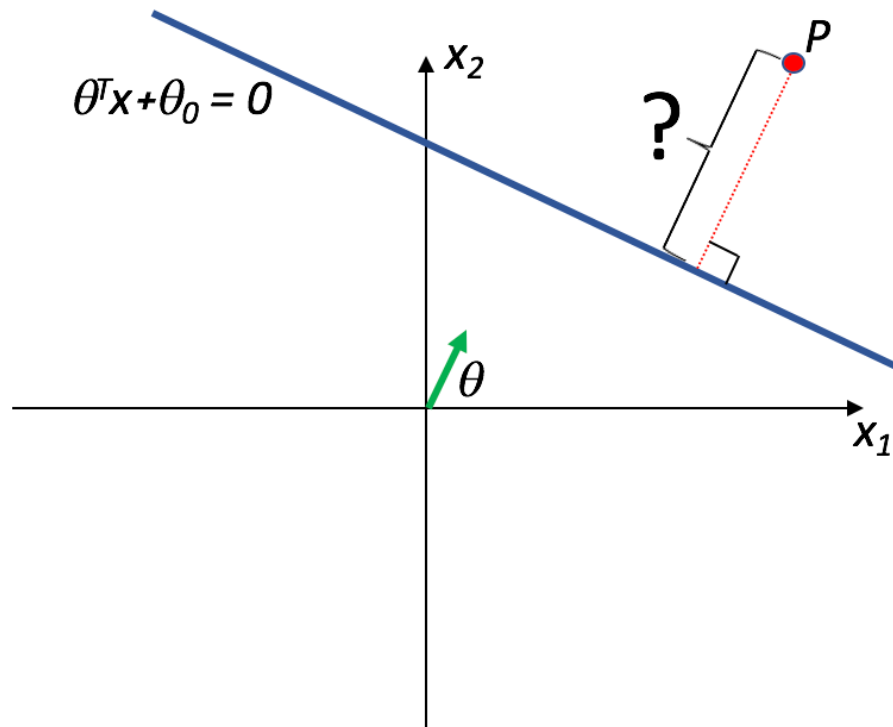
1) Numpy procedures for hyperplanes and separators

Relevant material on linear classifiers in the [notes](#)

Helpful numpy explanations at the [bottom of the page](#).

1.1) General hyperplane, distance to point

Let p be an arbitrary point in \mathbb{R}^d . Give a formula for the **signed** perpendicular distance from the hyperplane specified by θ, θ_0 to this point p .



Enter your answer as a Python expression. Use `theta` for θ , `theta_0` for θ_0 , `p` for the point p , `transpose(x)` for transpose of an array, `norm(x)` for the length (L2-norm) of a vector, and `x@y` to indicate a matrix product of two arrays.

Formula for signed distance:

Check Syntax

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

1.2) Code for signed distance!

Write a Python function using numpy operations (no loops!) that takes column vectors (d by 1) x and th (of the same dimension) and scalar th_0 and returns the signed perpendicular distance (as a 1 by 1 array) from the hyperplane encoded by (th, th_0) to x . Note that you are allowed to use the "length" function defined in previous coding questions (including week 1 exercises).

```
1 import numpy as np
2 def signed_dist(x, th, th0):
3     x = np.array(x)
4     th = np.array(th)
5     return (x.T@th + th0) / length(th)
6
```

[Run Code](#)[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

1.3) Code for side of hyperplane

Write a Python function that takes as input

- a column vector x
- a column vector th that is of the same dimension as x
- a scalar $th0$

and returns

- $+1$ if x is on the positive side of the hyperplane encoded by $(th, th0)$

- 0 if on the hyperplane
- -1 otherwise.

The answer should be a 2D array (a 1 by 1). Look at the [sign](#) function. Note that you are allowed to use any functions defined in week 1's exercises.

```
1 import numpy as np
2 def positive(x, th, th0):
3     return np.sign(signed_dist(x, th, th0))
4
```

[Run Code](#)[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

Now, given a hyperplane and a set of data points, we can think about which points are on which side of the hyperplane. This is something we do in many machine-learning algorithms, as we will explore soon. It is also a chance to begin using numpy on larger chunks of data.

1.4) Expressions operating on data

We define `data` to be a 2 by 5 array (two rows, five columns) of scalars. It represents 5 data points in two dimensions. We also define `labels` to

be a 1 by 5 array (1 row, five columns) of 1 and -1 values.

```
data = np.transpose(np.array([[1, 2], [1, 3], [2, 1], [1, -1], [2, -1]]))
labels = rv([-1, -1, +1, +1, +1])
```

For each subproblem, provide a Python expression that sets `A` to the quantity specified. Note that `A` should always be a 2D numpy array. Only one relatively short expression is needed for each one. No loops!

You can use (our version) of the `length` and `positive` functions; they are already defined, don't paste in your definitions. Those functions if written purely as matrix operations should work with a 2D data array, not just a single column vector as the first argument, with no change.

1. `A` should be a 1 by 5 array of values, either +1, 0 or -1, indicating, for each point in `data`, whether it is on the positive side of the hyperplane defined by `th`, `th0`. **Use `data`, `th`, `th0` as variables in your submission.**


```
1 import numpy as np
2 import numpy as np
3 def point_sign(p, axis):
4     return np.sign(signed_dist(p, th, th0))
5 A = np.apply_over_axes(point_sign, data, axes=1)
6
```

[Run Code](#)[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

2. `A` should be a 1 by 5 array of boolean values, either True or False, indicating for each point in `data` and corresponding label in `labels` whether it is correctly classified by hyperplane $\theta = [1, 1]$, $\theta_0 = -2$. That is, return True when the side of the hyperplane (specified by θ, θ_0) that the point is on agrees with the specified label.

```
1 def point_sign(p, axis):
2     return np.sign(signed_dist(p, th, th0))
3 th = np.array(th)
4 #th0 = -2
5 B = np.apply_over_axes(point_sign, data, axes=1)
6 A = B.T == labels
```

[Run Code](#)[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

1.5) Score

Write a procedure that takes as input

- data: a d by n array of floats (representing n data points in d dimensions)
- labels: a 1 by n array of elements in $(+1, -1)$, representing target labels
- th: a d by 1 array of floats that together with
- th0: a single scalar or 1 by 1 array, represents a hyperplane

and returns the number of points for which the label is equal to the output of the positive function on the point.

Since numpy treats False as 0 and True as 1, you can take the sum of a collection of Boolean values directly.

```
1 def length(col_v):
2     return ((col_v.T@col_v)**.5)[0,0]
3
4
5 def signed_dist(x, th, th0):
6     x = np.array(x)
7     th = np.array(th)
8     return (x.T@th + th0) / length(th)
9
10 def score(data, labels, th, th0):
11     def point_sign(p, axis):
12         return np.sign(signed_dist(p, th, th0))
13     return np.sum(np.apply_over_axes(point_sign, data, axes=1).T == labels)
14
```

[Run Code](#)[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

1.6) Best separator

Now assume that we have some "candidate" classifiers that we want to pick the best one out of. Assume you have \mathbf{ths} , a d by m array of m candidate θ 's (each θ has dimension d by 1), and $\mathbf{th0s}$, a 1 by m array of the corresponding m candidate θ_0 's. Each of the θ, θ_0 pair represents a hyperplane that characterizes a binary classifier.

Write a procedure that takes as input

- `data`: a d by n array of floats (representing n data points in d dimensions)

- `labels`: a 1 by n array of elements in $(+1, -1)$, representing target labels
- `ths`: a d by m array of floats representing m candidate θ 's (each θ has dimension d by 1)
- `th0s`: a 1 by m array of the corresponding m candidate θ_0 's.

and finds the hyperplane with the highest score on the data and labels. In case of a tie, return the first hyperplane with the highest score, in the form of

- a tuple of a d by 1 array and an offset in the form of 1 by 1 array.

The function `score` that you wrote above was for a single hyperplane separator. Think about how to generalize it to multiple hyperplanes and include this modified (if necessary) definition of `score` in the answer box.

Note: Look below the answer box for useful numpy functions!

```
1
2 def best_separator(data, labels, ths, th0s):
3     mysum = np.sum(np.sign(ths.T @ data + th0s.T) == labels, axis=1)
4     i = np.argmax(mysum)
5     return ths[:,i:], th0s[:,i:i+1]
```

Ask for Help

100.00%

You have infinitely many submissions remaining.

Here is the solution we wrote:

```
import numpy as np
# data is dimension d by n
# labels is dimension 1 by n
# ths is dimension d by m
# th0s is dimension 1 by m
# return matrix of integers indicating number of data points correct for
# each separator: dimension m x 1
def score_mat(data, labels, ths, th0s):
    pos = np.sign(np.dot(np.transpose(ths), data) + np.transpose(th0s))
    return np.sum(pos == labels, axis = 1, keepdims = True)
def best_separator(data, labels, ths, th0s):
    best_index = np.argmax(score_mat(data, labels, ths, th0s))
```

```
return cv(th0s[:,best_index]), th0s[:,best_index:best_index+1]
```

Explanation:

First, let's break up the best classifier problem into three subproblems:

1. Extend the score() function to a second dimension, allowing us to generate scores for multiple hyperplanes.
2. Apply this new score() function to the data and the array of hyperplanes and select the best score (across the second dimension).
3. Once we've found the best score (or the index of the best score), use that to return the correspondingly best (θ , θ_0) hyperplane parameters.

Let's tackle each subproblem in order:

To extend the score() function to generate scores for multiple hyperplanes, we can start by using the same expression in 3.3.1 to generate an $m \times n$ array of 1, 0, or -1 values corresponding to how each hyperplane classifies each point:

```
pos = positive(data, ths, th0s.T)
```

Be careful of dimension matching: `np.dot(np.transpose(ths), data)` has dimensions $m \times n$ and `th0s` has dimensions $1 \times m$

Now that we've generated an array of classification values, we can compare them to the label values the same way we did in problem 3.3.2, using `(pos == labels)` or `np.equal(pos, labels)`. Since the second dimension of the two arrays are both n , there's no danger of dimension mismatch (although, if you like, you can create m copies of *labels* and tile them along the first dimension using the `np.tile` function) - this will do an element-wise comparison over the first dimension.

Finally, we want to sum these over the second dimension to create a $m \times 1$ array of scores corresponding to each hyperplane. We can achieve this with `np.sum()` in the following way:

```
score_mat = np.sum((pos == labels), axis=1, keep_dims=True)
```

Two important things to keep note of here: first, we need to sum over *only* the second dimension, so we need to use the axis parameter so that we only reduce the second dimension. Second, np.sum() will remove all dimensions we sum over, so just writing np.sum((pos == labels), axis=1) will return a 1-D array. We still want a 2-D array, so we set the keep_dims flag to True so the axis we sum over is left as a dimension of size 1.

Now that we have a matrix corresponding to each hyperplane's score on classifying *data*, we want to then find the highest score and its corresponding hyperplane. Note that we don't actually care about the value of the highest score, just the index so we can select the corresponding values in ths and th0s. To do this, we can use the `np.argmax()` function as such:

```
best_index = np.argmax(score_mat)
```

Finally, we can select the corresponding θ and θ_0 from ths and th0s, remembering to select along the second dimension and convert the final θ into a column vector as we did in 3.3.1:

```
return cv(ths[:, best_index], th0s[:, best_index])
```

Reference Material: Handy Numpy Functions and Their Usage

In order to avoid using for loops, you will need to use the following numpy functions. (So that you replace for loops with matrix operations)

A. `np.sum` with axis

`np.sum` can take an optional argument `axis`. Axis 0 is row and 1 is column in a 2D numpy array. **The way to understand the “axis” of numpy sum is that it sums(collapses) the given matrix in the direction of the specified axis. So when it collapses the axis 0 (row), it becomes just one row and column-wise sum.** Let's look at examples.

```
>>> np.sum(np.array([[1,1,1],[2,2,2]]), axis=1)
array([3, 6])
>>> np.sum(np.array([[1,1,1],[2,2,2]]), axis=0)
array([3, 3, 3])
```

Note that `axis=1` (column) will "squash" (or collapse) sum `np.array([[1,1,1],[2,2,2]])` in the column direction. On the other hand, `axis=0` (row) will collapse-sum `np.array([[1,1,1],[2,2,2]])` in the row direction.

B. Comparing matrices of different dimensions / advanced `np.sum`

Note that two matrices `A`, `B` below have same number of columns but different row dimensions.

```
>>> A = np.array([[1,1,1],[2,2,2],[3,3,3]])
>>> B = np.array([[1,2,3]])
>>> A==B
array([[ True, False, False],
       [False,  True, False],
       [False, False,  True]])
```

The operation `A==B` copies `B` three times row-wise so that it matches the dimension of `A` and then element-wise compares `A` and `B`.

We can apply `A==B` to `np.sum` like below.

```
>>> A = np.array([[1,1,1],[2,2,2],[3,3,3]])
>>> B = np.array([[1,0,0],[2,2,0],[3,3,3]])
>>> np.sum(A==B, axis=1)
```



```
array([1, 2, 3])
```

C. `np.sign`

`np.sign`, given a numpy array as input, outputs a numpy array of the same dimension such that its element is the sign of each element of the input. Let's look at an example.

```
>>> np.sign(np.array([-3,0,5]))  
array([-1,  0,  1])
```

D. `np.argmax`

`np.argmax`, given a numpy array as input, outputs the index of the maximum element of the input. Let's look at an example.

```
>>> np.argmax(np.array([[1,2,3],[4,5,6]]))  
5
```

Note that the `argmax` index is given assuming the input array is flattened. So in our case, with 6 being the maximum element, 5 was returned instead of something like (1,2).

E. `np.reshape`

For a np array `A`, you can call `A.reshape((dim1_size,dim2_size,...))` in order to change the shape of the array.

```
>>> A = np.array([[1,2,3],[4,5,6]])  
>>> A.reshape((3,2))  
array([[1, 2],  
       [3, 4],  
       [5, 6]])
```

Note, the new shape has to have the same number of elements as the original.

For these exercises, it will be helpful to review the notes on [Linear Classifiers](#) and the [Perceptron](#). You may also find it helpful to write some test code with a local python installation or in a [google colab notebook](#).

1) Classification

Consider a linear classifier through the origin in 4 dimensions, specified by

$$\theta = (1, -1, 2, -3)$$

Which of the following points x are classified as positive, i.e. $h(x; \theta) = +1$?

1. $(1, -1, 2, -3)$
2. $(1, 2, 3, 4)$
3. $(-1, -1, -1, -1)$
4. $(1, 1, 1, 1)$

Enter a Python list with a subset of the numbers 1, 2, 3, 4:

100.00%

You have infinitely many submissions remaining.

Solution: [1, 3]

Explanation:

In a classification problem, a point is considered positive if $\text{sign}(\theta \cdot x)$ is positive and otherwise negative. Note that $\theta \cdot x$ is equal to

$$\sum_i x_i \theta_i$$

Therefore, we have that

$$\theta \cdot x_1 = (1, -1, 2, -3) \cdot (1, -1, 2, -3) = 15$$

So the first point x_1 is classified positively. Similar computations show that x_3 classified positively.

2) Classifier vs Hyperplane

Consider another parameter vector

$$\theta' = (-1, 1, -2, 3)$$

Ex2a

Does θ' represent the same hyperplane as θ does?

100.00%

You have infinitely many submissions remaining.

Solution: yes

Explanation:

Note that the hyperplane defined by a norm vector θ is the set of points x such that $\theta \cdot x = 0$. Now, θ' determines the same hyperplane as θ . This is because a hyperplane H is defined as the set of points x such that $x \cdot \theta$ is equal to 0 (x is perpendicular to θ) for some arbitrary θ . For our specific θ in this problem, note that

$$\theta \cdot x = 0 = -\theta \cdot x$$

Which shows that the set of points perpendicular to $-\theta$ is equivalent to the points perpendicular to θ .

Ex2b

Does θ' represent the same classifier as θ does?

100.00%

You have infinitely many submissions remaining.

Solution: no

Explanation:

Note that hyperplanes specified by norm vectors θ' and θ are the same. However, because θ' and θ point in opposite directions, θ' determines a different classifier than θ since the $\text{sign}(\theta \cdot x)$ is different.

3) Linearly Separable Training

As [discussed in lecture and in the lecture notes](#), note that $\mathcal{E}_n(\theta, \theta_0)$ refers to the training error of the linear classifier specified by θ, θ_0 , and $\mathcal{E}(\theta, \theta_0)$ refers to its test error. What does the fact that the training data are *linearly separable* imply?

Select "yes" or "no" for each of the following statements:

Ex3a

There must exist θ, θ_0 such that $\mathcal{E}(\theta, \theta_0) = 0$

100.00%

You have infinitely many submissions remaining.

Solution: no

Explanation:

There doesn't necessarily exist a hyperplane such that $\mathcal{E}(\theta, \theta_0) = 0$ since just because the training data is separable by a specific hyperplane doesn't mean the entire underlying data distribution will be separable by a hyperplane

Ex3b

There must exist θ, θ_0 such that $\mathcal{E}_n(\theta, \theta_0) = 0$

100.00%

You have infinitely many submissions remaining.

Ex3c

A separator with 0 training error exists

100.00%

You have infinitely many submissions remaining.

Ex3d

A separator with 0 testing error exists, for all possible test sets

100.00%

You have infinitely many submissions remaining.

Ex3e

The perceptron algorithm will find θ, θ_0 such that $\mathcal{E}_n(\theta, \theta_0) = 0$

100.00%

You have infinitely many submissions remaining.

Solution: yes

Explanation:

If the data is linearly separable, then perceptron will find a separator.

4) Separable Through Origin?

Provide two points, (x_0, x_1) and (y_0, y_1) in two dimensions that are linearly separable but not linearly separable through the origin. If you get stuck try drawing a picture and review the notes on [offsets](#).

Enter a Python list with two entries of the form `[[x0, x1], label]` where label is 1 or -1. (So each entry represents a point with 2 dimensions and its label)

Submit

View Answer

100.00%

You have infinitely many submissions remaining.

MIT 6.036 Spring 2019: Homework 2

This colab notebook provides code and a framework for problems 7-10 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 (such as `positive`).

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

```
In [3]: !rm -f code_for_hw02.py*
        !wget --no-check-certificate --quiet https://introml_oll.odl.mit.edu/6.036/static/homework/hw02/code_for_hw02.py
        from code_for_hw02 import *
```

Importing code_for_hw02

New procedures added: tidy_plot, plot_separator, plot_data, plot_nonlin_sep, cv,
rv, y, positive, score

Data Sets: super_simple_separable_through_origin(), super_simple_separable(), xor(),
xor_more()

Test data for problem 2.1: data1, labels1, data2, labels2

Test data for problem 2.2: big_data, big_data_labels, gen_big_data(), gen_lin_separable(),
big_higher_dim_separable(), gen_flipped_lin_separable()

Test functions: test_linear_classifier(), test_perceptron(), test_averaged_perceptron(),
test_eval_classifier(), test_eval_learning_alg(), test_xval_learning_alg()

For more information, use 'help', e.g. 'help tidy_plot'

Done with import of code_for_hw02

```
In [4]: help(tidy_plot)
```

Help on function tidy_plot in module code_for_hw02:

```
tidy_plot(xmin, xmax, ymin, ymax, center=False, title=None, xlabel=None, ylabel=None)
    Set up axes for plotting
    xmin, xmax, ymin, ymax = (float) plot extents
    Return matplotlib axes
```

```
In [5]: def test(a):
        return a + 53
```

```
In [6]: def methodB(a):
        return test(a)
```

```
In [7]: def someMethod():
        test = 7
        return methodB(test + 3)
```

```
In [8]: someMethod()
```

```
Out[8]: 63
```

****7) Implement perceptron****

Implement [the perceptron algorithm](#), where

- `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 should run 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. We won't be testing this in the Tutor, but it will help you in debugging on your own machine.

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

We have given you some data sets in the code file for you to test your implementation.

Your function should initialize all parameters to 0, then run through the data, in the order it is given, performing an update to the parameters whenever the current parameters would make a mistake on that data point. Perform T iterations through the data.

```
In [9]: import numpy as np
```

```
In [248... def 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)
    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:
                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]]))
```

```
In [249... test_perceptron(perceptron)
```

```
-----Test Perceptron 0-----
Passed!
```

```
-----Test Perceptron 1-----
Passed!
```

8) Implement averaged perceptron

Regular perceptron can be somewhat sensitive to the most recent examples that it sees. Instead, averaged perceptron produces a more stable output by outputting the average value of \mathbf{th} and $\mathbf{th0}$ across all iterations.

Implement averaged perceptron with the same spec as regular perceptron, and using the pseudocode below as a guide.

```
procedure averaged_perceptron({(x^(i), y^(i)), i=1,...,n}, T)
  th = 0 (d by 1); th0 = 0 (1 by 1)
  ths = 0 (d by 1); th0s = 0 (1 by 1)
  for t = 1,...,T do:
    for i = 1,...,n do:
      if y^(i)(th . x^(i) + th0) <= 0 then
        th = th + y^(i)x^(i)
        th0 = th0 + y^(i)
        ths = ths + th
        th0s = th0s + th0
  return ths/(nT), th0s/(nT)
```

```
In [367... import numpy as np

def averaged_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]
    th = np.transpose([[0.0] * d])
    th0 = 0.0
    ths = np.transpose([[0.0] * d])
    th0s = 0.0
    #ax = plot_data(data, labels)
    for test in range(T):
        for i in range(n):
            xi = data[:,i:i+1]
            yi = labels[0, i]
            if yi * np.sign(th.T @ xi + th0) <= 0:
                founderror = True
                th += yi * xi
                th0 += yi
                if hook:
                    hook(th, th0)
            ths += th
            th0s += th0
    #plot_separator(ax=ax, th=th, th_0=th0)
    return (ths/(n*T), np.array([[th0s/(n*T)]]))
```

```
In [251... test_averaged_perceptron(averaged_perceptron)

-----Test Averaged Perceptron 0-----
Passed!

-----Test Averaged Perceptron 1-----
Passed!
```

9) Implement evaluation strategies

9.1) Evaluating a classifier

To evaluate a classifier, we are interested in how well it performs on data that it wasn't trained on. Construct a testing procedure that uses a training data set, calls a learning algorithm to get a linear separator (a tuple of θ , θ_0), and then reports the percentage correct on a new testing set as a float between 0. and 1..

The learning algorithm is passed as a function that takes a data array and a labels vector. Your evaluator should be able to interchangeably evaluate `perceptron` or `averaged_perceptron` (or future algorithms with the same spec), depending on what is passed through the `learner` parameter.

The `eval_classifier` function should accept the following parameters:

- `learner` - a function, such as `perceptron` or `averaged_perceptron`
- `data_train` - training data
- `labels_train` - training labels
- `data_test` - test data
- `labels_test` - test labels

Assume that you have available the function `score` from HW 1, which takes inputs:

- `data`: a d by n array of floats (representing n data points in d dimensions)
- `labels`: a 1 by n array of elements in $(+1, -1)$, representing target labels
- `th`: a d by 1 array of floats that together with
- `th0`: a single scalar or 1 by 1 array, represents a hyperplane

and returns 1 by 1 matrix with an integer indicating number of data points correct for the separator.


```
In [252... import numpy as np

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

```
In [253... test_eval_classifier(eval_classifier,perceptron)
```

```
-----Test Eval Classifier 0-----
Passed!
```

```
-----Test Eval Classifier 1-----
Passed!
```

9.2) Evaluating a learning algorithm using a data source

Construct a testing procedure that takes a learning algorithm and a data source as input and runs the learning algorithm multiple times, each time evaluating the resulting classifier as above. It should report the overall average classification accuracy.

You can use our implementation of `eval_classifier` as above.

Write the function `eval_learning_alg` that takes:

- `learner` - a function, such as `perceptron` or `averaged_perceptron`
- `data_gen` - a data generator, call it with a desired data set size; returns a tuple (data, labels)
- `n_train` - the size of the learning sets
- `n_test` - the size of the test sets
- `it` - the number of iterations to average over

and returns the average classification accuracy as a float between 0. and 1..

Note: Be sure to generate your training data and then testing data in that order, to ensure that the pseudorandomly generated data matches that in the test code.

```
In [254... import numpy as np

def eval_learning_alg(learner, data_gen, n_train, n_test, it):
    sum_ = 0
    for i in range(it):
        n = n_train + n_test
        x, y = data_gen(n)
        inxs = np.arange(n)
        #np.random.shuffle(inxs)
        data_train = x[:, inxs[:n_train]]
        labels_train = y[:, inxs[:n_train]]
        data_test = x[:, inxs[n_train:]]
        labels_test = y[:, inxs[n_train:]]
        sum_ += eval_classifier(learner, data_train, labels_train, data_test, labels_test)
    return sum_ / it
```

```
In [255... test_eval_learning_alg(eval_learning_alg, perceptron)
```

```
-----Test Eval Learning Algo-----
Passed!
```

9.3) Evaluating a learning algorithm with a fixed dataset

Cross-validation is a strategy for evaluating a learning algorithm, using a single training set of size n . Cross-validation takes in a learning algorithm L , a fixed data set \mathcal{D} , and a parameter k . It will run the learning algorithm k different times, then evaluate the accuracy of the resulting classifier, and ultimately return the average of the accuracies over each of the k "runs" of L . It is structured like this:

```
divide D into k parts, as equally as possible; call them  $D_i$  for  $i = 0 \dots k-1$ 
# be sure the data is shuffled in case someone put all the positive examples first in the data!
for j from 0 to k-1:
     $D_{\text{minus}_j}$  = union of all the datasets  $D_i$ , except for  $D_j$ 
     $h_j = L(D_{\text{minus}_j})$ 
     $\text{score}_j$  = accuracy of  $h_j$  measured on  $D_j$ 
return average( $\text{score}_0, \dots, \text{score}_{k-1}$ )
```

So, each time, it trains on $k-1$ of the pieces of the data set and tests the resulting hypothesis on the piece that was not used for training.

When $k=n$, it is called *leave-one-out cross validation*.

Implement cross validation **assuming that the input data is shuffled already** so that the positives and negatives are distributed randomly. If the size of the data does not evenly divide by k , split the data into $n \% k$ sub-arrays of size $n/k + 1$ and the rest of size n/k . (Hint: You can use [numpy.array_split](#) and [numpy.concatenate](#) with axis arguments to split and rejoin the data as you desire.)

Note: In Python, n/k indicates integer division, e.g. $2//3$ gives 0 and $4//3$ gives 1.

```
In [291... a = np.array([[1, 2], [3, 4]])
          b = np.array([[5, 6]])
```

```
In [292... np.concatenate((a, b), axis=0), np.concatenate((a, b.T), axis=1), np.concatenate((a, b), axis=None)
```

```
Out[292]: (array([[1, 2],
                  [3, 4],
                  [5, 6]]),
          array([[1, 2, 5],
                  [3, 4, 6]]),
          array([1, 2, 3, 4, 5, 6]))
```

```
In [347... import numpy as np

def xval_learning_alg(learner, indata, inlabels, k):
    ds = np.split(indata, k, axis=1)
    ls = np.split(inlabels, k, axis=1)
    scores = []
    for j in range(k):
        data = np.concatenate(ds[:j] + ds[j+1:], axis=1)
        labels = np.concatenate(ls[:j] + ls[j+1:], axis=1)
        th, th0 = learner(data, labels)
        correctnum = score(ds[j], ls[j], th, th0)
        n = len(ls[j][0])
        print(correctnum, n)
        scores += [correctnum * 1. / n]
    return np.average(scores)
```

```
In [348... test_xval_learning_alg(xval_learning_alg,perceptron)
```

```
11 20
12 20
10 20
13 20
15 20
-----Test Cross-eval Learning Algo-----
Passed!
```

10) Testing

In this section, we compare the effectiveness of perceptron and averaged perceptron on some data that are not necessarily linearly separable.

Use your `eval_learning_alg` and the `gen_flipped_lin_separable` generator in the code file to evaluate the accuracy of `perceptron` vs. a `veraged_perceptron`. `gen_flipped_lin_separable` can be called with an integer to return a data set and labels. Note that this generates linearly separable data and then "flips" the labels with some specified probability (the argument `pflip`); so most of the results will not be linearly separable. You can also specify `pflip` in the call to the generator. You should use the default values of `th` and `th_0` to retain consistency with the Tutor.

Run enough trials so that you can confidently predict the accuracy of these algorithms on new data from that same generator; assume training/test sets on the order of 20 points. The Tutor will check that your answer is within 0.025 of the answer we got using the same generator.

```
In [358... print(eval_learning_alg(perceptron, gen_flipped_lin_separable(pflip=.1), 20, 20, 1000))
0.75905000000000006
```

```
In [359... print(eval_learning_alg(averaged_perceptron, gen_flipped_lin_separable(pflip=.1), 20, 20, 1000))
0.80484999999999997
```

```
In [360... print(eval_learning_alg(perceptron, gen_flipped_lin_separable(pflip=.25), 20, 20, 1000))
0.58735000000000001
```

```
In [361... print(eval_learning_alg(averaged_perceptron, gen_flipped_lin_separable(pflip=.25), 20, 20, 1000))
0.63750000000000007
```

```
In [362... def eval_learning_alg_wrong(learner, data_gen, n_train, n_test, it):
    sum_ = 0
    for i in range(it):
        n = n_train + n_test
        x, y = data_gen(n)
        inxs = np.arange(n)
        #np.random.shuffle(inxs)
        data_train = x[:, inxs[:n_train]]
        labels_train = y[:, inxs[:n_train]]
        data_test = x[:, inxs[n_train:]]
        labels_test = y[:, inxs[n_train:]]
        sum_ += eval_classifier(learner, data_train, labels_train, data_test, labels_test)
    return sum_ / it
```

```
In [363... print(eval_learning_alg_wrong(perceptron, gen_flipped_lin_separable(pflip=.1), 20, 20, 1000))
0.8195
```

```
In [364... print(eval_learning_alg_wrong(averaged_perceptron, gen_flipped_lin_separable(pflip=.1), 20, 20, 1000))
0.8665999999999997
```

```
In [365... print(eval_learning_alg_wrong(perceptron, gen_flipped_lin_separable(pflip=.25), 20, 20, 1000))
0.6714499999999998
```

```
In [366... print(eval_learning_alg_wrong(averaged_perceptron, gen_flipped_lin_separable(pflip=.25), 20, 20, 1000))
0.7329499999999997
```

```
In [ ]:
```

This homework will aid in your understanding of the Perceptron Algorithm by having you:

- practice applying the perceptron algorithm to toy data
- implement the perceptron algorithm and one of its variants
- apply your perceptron implementation on larger, more interesting data sets

In addition to the Numpy functions and features mentioned in hw 1, here are some you should be familiar with for this assignment:

- `np.argmax`
- `np.array_split`, look at the `axis` argument
- `np.concatenate`, look at the `axis` argument
- `np.zeros`
- `numpy.ndarray.shape`
- numpy [logic functions](#) e.g. `np.array([[1, 2], [3, 4]]) == 3`

For this homework, it will be helpful to review the [notes](#) on perceptron.

For problems 7-10, [here](#) is a code file that will be useful for debugging on your computer; alternatively, you may find it helpful to use [this preformatted google colab notebook](#).

Note: for all of the problems in this assignment, you are allowed to use `for` loops.

1) Perceptron Mistakes

Let's apply [the perceptron algorithm \(through the origin\)](#) to a small training set containing three points:

i **Data Points** $x^{(i)}$ **Labels** $y^{(i)}$

i **Data Points** $x^{(i)}$ **Labels** $y^{(i)}$

1 $[1, -1]$ 1

2 $[0, 1]$ -1

3 $[-1.5, -1]$ 1

Given that the algorithm starts with $\theta^{(0)} = [0, 0]$, **the first point that the algorithm sees is always a mistake**. The algorithm starts with **some** data point (to be specified in the question), and then cycles through the data until it makes no further mistakes.

1.1) Take 1

1.1a) How many mistakes does the algorithm make until convergence if the algorithm starts with data point $x^{(1)}$?

Number of mistakes is:

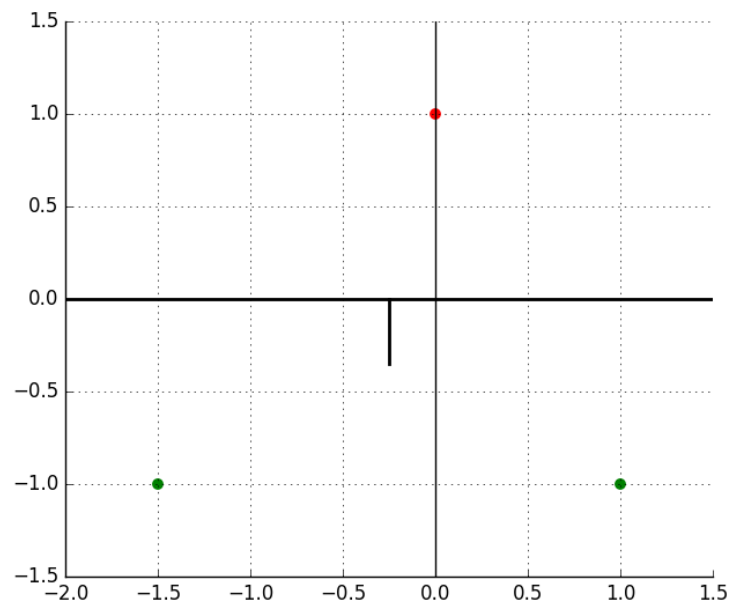
Submit

View Answer

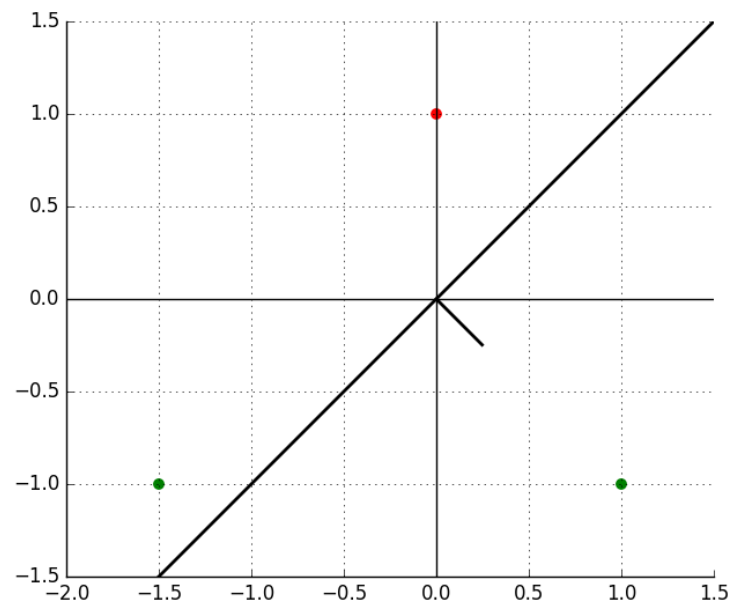
Ask for Help

100.00%

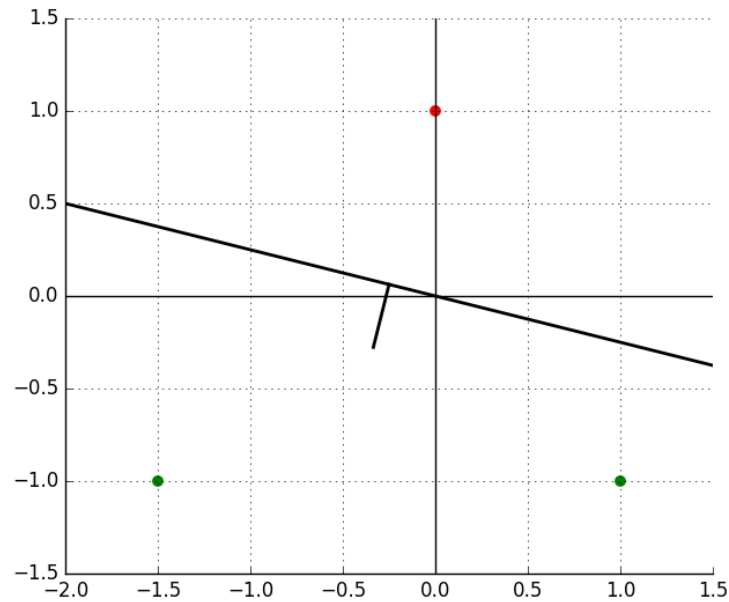
You have infinitely many submissions remaining.



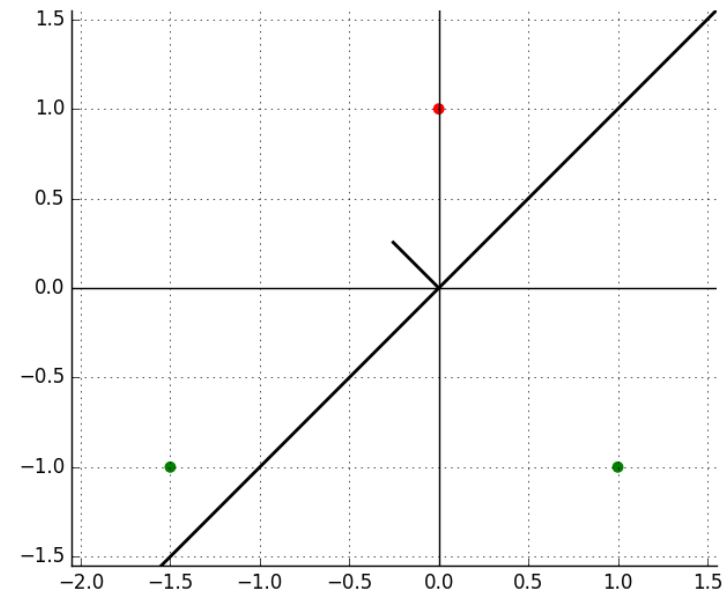
(1)



(2)



(3)



(4)

1.1b) Which of the above plot(s) correspond to the progression of the hyperplane as the algorithm cycles? Ignore the initial 0 weights, and include an entry only when θ changes.

Please provide the plot number(s) in the order of progression as a Python list.

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

1.1c) How many mistakes does the algorithm make if it starts with data point $x^{(2)}$ (and then does $x^{(3)}$ and $x^{(1)}$)?

Number of mistakes is:

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

1.1d) Again, if it starts with data point $x^{(2)}$ (and then does $x^{(3)}$ and $x^{(1)}$), which plot(s) correspond to the progression of the hyperplane as the algorithm cycles? Ignore the initial 0 weights.

Please provide the plot number(s) in the order of progression as a Python list.

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

1.2) Take 2

Now assume that $x^{(3)} = [-10, -1]$, with label 1.

1.2a) How many mistakes does the algorithm make until convergence if cycling starts with data point $x^{(1)}$?

Number of mistakes is

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

1.2b) How many mistakes if it starts with data point $x^{(2)}$?

Number of mistakes is

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

2) Initialization

2.1) If we were to initialize the perceptron algorithm with $\theta = [1000, -1000]$, how would it affect the number of mistakes made in order to separate the data set from question 1?

- ☐ It would have small or no effect on the number of mistakes.
- ☐ It would significantly decrease the number of mistakes.
- ☒ It would significantly increase the number of mistakes.

Ask for Help

100.00%

You have infinitely many submissions remaining.

Solution: It would significantly increase the number of mistakes.

Explanation:

When θ is initialized with larger values, corrections to θ have a smaller impact. Therefore, in general, it takes more corrections until θ can separate the data points.

2.2) Provide a value of $\theta^{(0)}$ for which running the perceptron algorithm (through origin) on the data set from question 1 returns a different result than using $\theta^{(0)} = [0, 0]$. The data set is repeated below:

i	Data Points $x^{(i)}$	Labels $y^{(i)}$
1	$[1, -1]$	1
2	$[0, 1]$	-1
3	$[-1.5, -1]$	1

Enter the 2 coordinates of θ as a Python list or the string 'none' if none exists.'

[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

3) Dual View

The following table shows a data set and the number of times each point is misclassified during a run of the perceptron algorithm (with offset). θ is initialized to the zero vector and θ_0 is initialized to 0.

i	$x^{(i)}$	$y^{(i)}$	times misclassified
1	$[-3, 2]$	1	2
2	$[-1, 1]$	-1	4
3	$[-1, -1]$	-1	2
4	$[2, 2]$	-1	1
5	$[1, -1]$	-1	0

3.1) What is the post training θ ?

Provide it as a python list of the form $[a, b]$.

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

3.2) What is the post training θ_0 ?

Provide it as a number.

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

4) Decision Boundaries

4.1) AND

Consider the AND function defined over three binary variables: $f(x_1, x_2, x_3) = (x_1 \wedge x_2 \wedge x_3)$.

If you are unfamiliar with the AND function, it simply returns 1 if all three variables are true (value of 1) and 0 otherwise.

We aim to find a θ such that, for any $x = [x_1, x_2, x_3]$, where $x_i \in \{0, 1\}$:

$$\theta \cdot x + \theta_0 > 0 \text{ when } f(x_1, x_2, x_3) = 1, \text{ and}$$

$$\theta \cdot x + \theta_o \leq 0 \text{ when } f(x_1, x_2, x_3) = 0.$$

4.1a) For each of the combination of values of (x_1, x_2, x_3) , that is,

$[(0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)]$ enter the values of $f(x_1, x_2, x_3)$.

Please enter the values of $f(x_1, x_2, x_3)$ as a Python list.

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

4.1b) Assuming $\theta_0 = 0$ (no offset), enter θ as a Python list of length 3 or enter 'none' as a Python string (with quotes) if none exists.

Enter a Python list of 3 numbers or the string 'none'

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

4.1c) Assuming θ_0 is non-zero (offset), enter a θ and θ_0 as a Python list of length 4 (θ_0 last) or enter 'none' as a Python string (with quotes) if none exists.

Enter a Python list with 4 numbers or the string 'none'

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

4.2) Families

You are given the following labeled data points:

- Positive examples: $[-1, 1]$ and $[1, -1]$,
- Negative examples: $[1, 1]$ and $[2, 2]$.

For each of the following parameterized families of classifiers, find the parameters of a family member that can correctly classify the above data, or think about why no such family member exists.

Is there a classifier of the following forms that can correctly classify the above data?

Recall from lecture that we consider a point lying exactly on the separator to be classified as negative.

4.2a) Inside or outside of an origin-centered circle with radius r

Enter a value for r or the string 'none' if none exists.

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

4.2b) Inside or outside of a circle centered on x_0 with radius r

Enter a list with 3 entries for coordinates of x_0 and r ($[x_0_1, x_0_2, r]$) or the string 'none' if none exists.'

[2,2,2]

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

4.2c) On one side of a line through the origin with normal θ (recall normal vector points into positive half-space).

Enter a list with 2 entries for coordinates of θ or the string 'none' if none exists.'

'none'

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

4.2d) On one side of a line with normal θ and offset θ_0 (recall normal vector points into positive half-space).

Enter a list with 3 entries for coordinates of θ and θ_0 ($[\theta_1, \theta_2, \theta_0]$) or the string 'none' if none exists.'

[-1,-1, 0.5]

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

4.2e)

Which of the above are families of linear classifiers?

☐ 4.2a

☐ 4.2b

☒ 4.2c

☒ 4.2d

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

5) Separation

Indicate if the following datasets are:

- not linearly separable
- linearly separable without an offset
- linearly separable only with a non-zero offset

HINT: You shouldn't have to work through the perceptron algorithm to figure this out.

5.1) Dataset 1:

i **Data Points** $x^{(i)}$ **Labels** $y^{(i)}$

1 $[1, -1]$ -1

2 $[1, 1]$ 1

i **Data Points** $x^{(i)}$ **Labels** $y^{(i)}$

3 $[2, -1]$ 1

4 $[2, 1]$ -1

This dataset is: Not linearly separable ▼

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

5.2) Dataset 2:

i **Data Points** $x^{(i)}$ **Labels** $y^{(i)}$

1 $[1, -1]$ 1

2 $[1, 1]$ 1

3 $[2, -1]$ -1

4 $[2, 1]$ -1

This dataset is: Linearly separable only with a non-zero offset ▼

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

5.3) Dataset 3:


i Data Points $x^{(i)}$ Labels $y^{(i)}$

1 $[1, -1, 1]$ 1

2 $[1, 1, 1]$ 1

3 $[2, -1, 1]$ -1

4 $[2, 1, 1]$ -1

This dataset is: Linearly separable without an offset 

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

5.4) Compare datasets 2 and 3, and see if you can generalize what you learned from those two examples. Which of the following transformations allow one to transform *any* dataset that is only linearly separable *with* an offset to a dataset that is linearly separable *without* an offset, such that the transformation doesn't depend on the values of the datapoints? Select all which are valid:

This dataset is:

- ☐ Remove the final dimension of each of the datapoints
- ☐ Add an extra dimension to all of the datapoints using any (nonzero) numbers
- ☒ Add an extra dimension to all of the datapoints using the same (nonzero) number for each point
- ☒ Add an extra dimension to all of the datapoints using a 1 for each point
- ☐ Add an extra dimension to all of the data points, using 0 for each point
- ☐ A transformation with these properties is impossible

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

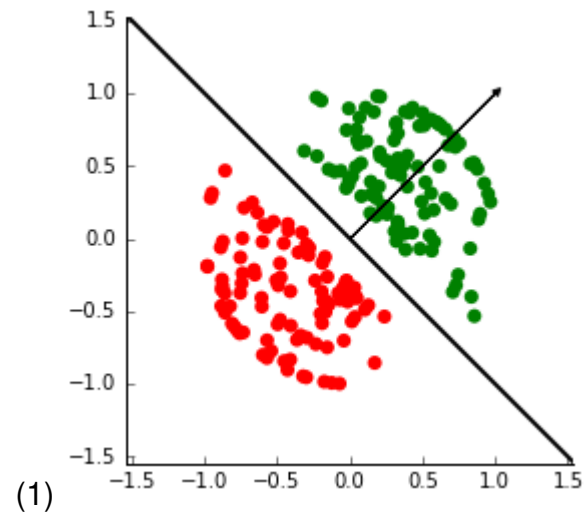
6) Mistakes and generalization

Please refer to [the Perceptron Convergence Theorem](#).

6.1) Plots

Consider the following plots. For each one estimate plausible values of R (an upper bound on the magnitude of the training vectors) and γ (the margin of the separator for the dataset). Consider values of R in the range $[1, 10]$ and values of γ in the range $[0.01, 2]$.

6.1a)



Enter a Python list with 2 floats, a value of R and a value of gamma:

Submit

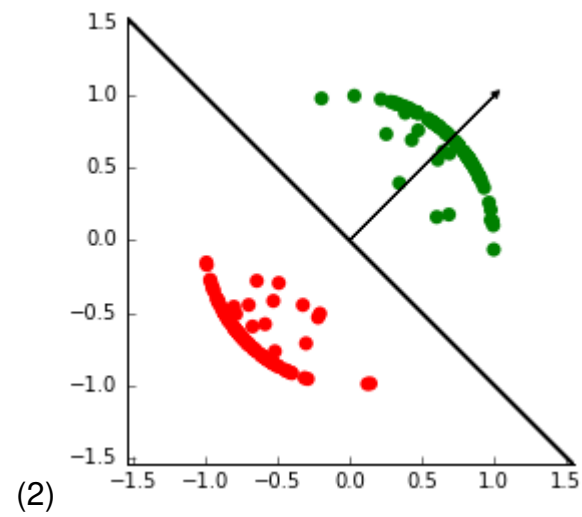
View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

6.1b)



Enter a Python list with 2 floats, a value of R and a value of gamma:

Submit

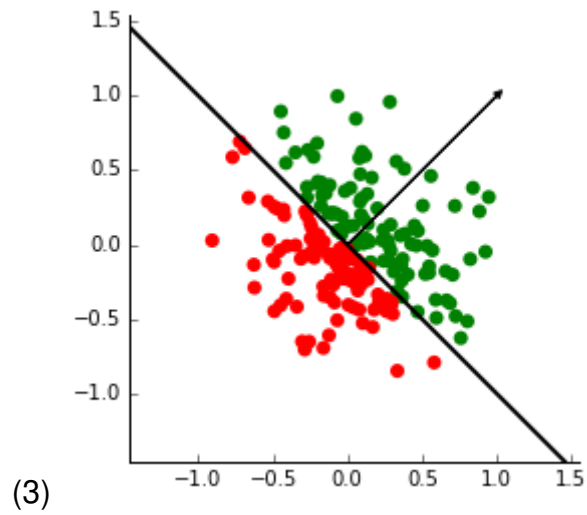
View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

6.1c)



Enter a Python list with 2 floats, a value of R and a value of gamma:

Submit

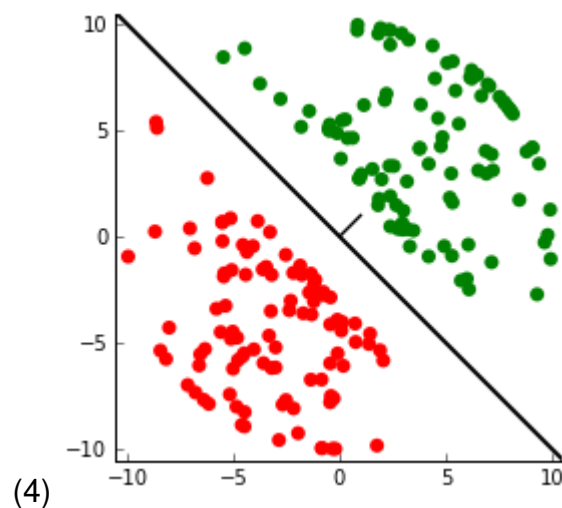
View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

6.1d)



Enter a Python list with 2 floats, a value of R and a value of gamma:

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

6.2) Mistake Bound

6.2a) What is an upper bound on mistakes when $R = 1, n = 1000$ for each of the following values of γ ?

(.00001, .0001, .001, .01, .1, .2, .5)

Recall that given a linear separator that passes through the origin, the perceptron algorithm makes at most $(\frac{R}{\gamma})^2$ mistakes.

Check yourself: Think about how you can compute this bound when the separator does not pass through the origin.

Enter a Python list with 7 floats.

Submit

View Answer

Ask for Help

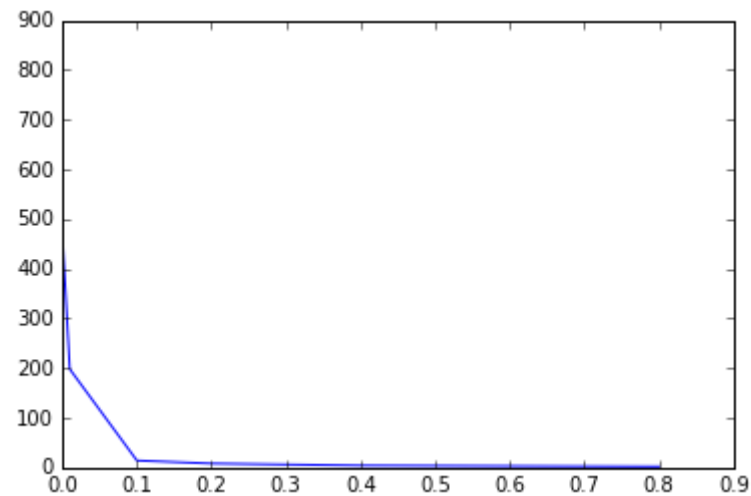
100.00%

You have infinitely many submissions remaining.

As a sanity check, we have provided a plot for $R = 1$, $d = 4$, $n = 1000$ of the actual numbers of mistakes made by the perceptron on one particular run, as a function of γ .

Check yourself: Make sure you understand this plot and why it agrees with our theoretical bounds.

The actual numbers of mistakes (on y axis) are: [862, 414, 446, 198, 14, 8, 4].



7) Implement perceptron

For this problem and the remaining problems below, [here](#) is a code file that will be useful for debugging on your computer; alternatively, you may

find it helpful to use [this preformatted google colab notebook](#).

7) Implement [the perceptron algorithm](#), where

- `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 should run 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. We won't be testing this in the Tutor, but it will help you in debugging on your own machine.

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

We have given you some data sets in the code file for you to test your implementation.

Your function should initialize all parameters to 0, then run through the data, in the order it is given, performing an update to the parameters whenever the current parameters would make a mistake on that data point. Perform T iterations through the data. After every parameter update, if `hook` is defined, call it on the current `(th, th0)` (as a single parameter in a python tuple).

When debugging on your own, you can use the procedure `test_linear_classifier` for testing. By default, it pauses after every parameter update to show the separator. For data sets not in 2D, or just to get the answer, set `draw = False`. **See the top of problem 7 for the the code distribution.**

```

11     for i in range(n):
12         xi = data[:,i:i+1]
13         yi = labels[0, i]
14         if yi * np.sign(theta.T @ xi + theta_0) <= 0:
15             founderror = True
16             theta += yi * xi
17             theta_0 += yi
18             if hook:
19                 hook(theta, theta_0)
20         if not founderror:
21             break
22     #plot_separator(ax=ax, th=theta, th_0=theta_0)
23     return (theta, np.array([[theta_0]]))
24

```

[Run Code](#)
[Submit](#)
[View Answer](#)
[Ask for Help](#)
100.00%

You have infinitely many submissions remaining.

8) Implement averaged perceptron

Regular perceptron can be somewhat sensitive to the most recent examples that it sees. Instead, averaged perceptron produces a more stable output by outputting the average value of `th` and `th0` across all iterations.

Implement averaged perceptron with the same spec as regular perceptron, and using the pseudocode below as a guide.

```

procedure averaged_perceptron({(x_(i), y_(i)), i=1,...,n}, T)
    th = 0 (d by 1); th0 = 0 (1 by 1)
    ths = 0 (d by 1); th0s = 0 (1 by 1)
    for t = 1,...,T do:

```

```

for i = 1,...,n do:
    if y_(i)(th . x_(i) + th0) <= 0 then
        th = th + y_(i)x_(i)
        th0 = th0 + y_(i)
    ths = ths + th
    th0s = th0s + th0
return ths/(nT), th0s/(nT)

```

```

13     for test in range(T):
14         for i in range(n):
15             xi = data[:,i:i+1]
16             yi = labels[0, i]
17             if yi * np.sign(th.T @ xi + th0) <= 0:
18                 founderror = True
19                 th += yi * xi
20                 th0 += yi
21                 if hook:
22                     hook(th, th0)
23             ths += th
24             th0s += th0
25     #plot_separator(ax=ax, th=th, th_0=th0)
26     return (ths/(n*T), np.array([[th0s/(n*T)]]))

```

[Run Code](#)
[Submit](#)
[View Answer](#)
[Ask for Help](#)
100.00%

You have infinitely many submissions remaining.

9) Implement evaluation strategies

9.1) Evaluating a classifier

To evaluate a classifier, we are interested in how well it performs on data that it wasn't trained on. Construct a testing procedure that uses a training data set, calls a learning algorithm to get a linear separator (a tuple of θ, θ_0), and then reports the percentage correct on a new testing set as a float between 0. and 1..

The learning algorithm is passed as a function that takes a data array and a labels vector. Your evaluator should be able to interchangeably evaluate `perceptron` or `averaged_perceptron` (or future algorithms with the same spec), depending on what is passed through the `learner` parameter.

Assume that you have available the function `score` from HW 1, which takes inputs:

- `data`: a d by n array of floats (representing n data points in d dimensions)
- `labels`: a 1 by n array of elements in $(+1, -1)$, representing target labels
- `th`: a d by 1 array of floats that together with
- `th0`: a single scalar or 1 by 1 array, represents a hyperplane

and returns a scalar number of data points that the separator correctly classified.

The `eval_classifier` function should accept the following parameters:

- `learner` - a function, such as `perceptron` or `averaged_perceptron`
- `data_train` - training data
- `labels_train` - training labels
- `data_test` - test data
- `labels_test` - test labels

and returns the percentage correct on a new testing set as a float between 0. and 1..


```
1 import numpy as np
2
3 def eval_classifier(learner, data_train, labels_train, data_test, labels_test):
4     th, th0 = learner(data_train, labels_train)
5     n = data_test.shape[1]
6     correctnum = score(data_test, labels_test, th, th0)
7     return correctnum / n
8
```

[Run Code](#)[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

9.2) Evaluating a learning algorithm using a data source

Construct a testing procedure that takes a learning algorithm and a data source as input and runs the learning algorithm multiple times, each time evaluating the resulting classifier as above. It should report the overall average classification accuracy.

You can use our implementation of `eval_classifier` as above.

Write the function `eval_learning_alg` that takes:

- `learner` - a function, such as `perceptron` or `averaged_perceptron`
- `data_gen` - a data generator, call it with a desired data set size; returns a tuple (data, labels)

- `n_train` - the size of the learning sets
- `n_test` - the size of the test sets
- `it` - the number of iterations to average over

and returns the average classification accuracy as a float between 0. and 1..

Note: Be sure to generate your training data and then testing data in that order, to ensure that the pseudorandomly generated data matches that in the test code.

```
4     sum_ = 0
5     for i in range(it):
6         n = n_train + n_test
7         x, y = data_gen(n)
8         inxs = np.arange(n)
9         #np.random.shuffle(inxs)
10        data_train = x[:,inxs[:n_train]]
11        labels_train = y[:,inxs[:n_train]]
12        data_test = x[:,inxs[n_train:]]
13        labels_test = y[:,inxs[n_train:]]
14        sum_ += eval_classifier(learner, data_train, labels_train, data_test, labels_test)
15    return sum_ / it
16
17
```

[Run Code](#)[Submit](#)[View Answer](#)[Ask for Help](#)**100.00%**

You have infinitely many submissions remaining.

9.3) Evaluating a learning algorithm with a fixed dataset

Cross-validation is a strategy for evaluating a learning algorithm, using a single training set of size n . Cross-validation takes in a learning

algorithm L , a fixed data set \mathcal{D} , and a parameter k . It will run the learning algorithm k different times, then evaluate the accuracy of the resulting classifier, and ultimately return the average of the accuracies over each of the k "runs" of L . It is structured like this:

```
divide D into k parts, as equally as possible; call them D_i for i == 0 .. k-1
# be sure the data is shuffled in case someone put all the positive examples first in the data!
for j from 0 to k-1:
    D_minus_j = union of all the datasets D_i, except for D_j
    h_j = L(D_minus_j)
    score_j = accuracy of h_j measured on D_j
return average(score_0, ..., score(k-1))
```

So, each time, it trains on $k-1$ of the pieces of the data set and tests the resulting hypothesis on the piece that was not used for training.

When $k = n$, it is called *leave-one-out cross validation*.

Implement cross validation **assuming that the input data is shuffled already** so that the positives and negatives are distributed randomly. If the size of the data does not evenly divide by k , split the data into $n \% k$ sub-arrays of size $n//k + 1$ and the rest of size $n//k$. (Hint: You can use [numpy.array_split](#) and [numpy.concatenate](#) with axis arguments to split and rejoin the data as you desire.)

Note: In Python, $n//k$ indicates integer division, e.g. $2//3$ gives 0 and $4//3$ gives 1.

```

3 def xval_learning_alg(learner, indata, inlabels, k):
4     ds = np.split(indata, k, axis=1)
5     ls = np.split(inlabels, k, axis=1)
6     scores = []
7     for j in range(k):
8         data = np.concatenate(ds[:j] + ds[j+1:], axis=1)
9         labels = np.concatenate(ls[:j] + ls[j+1:], axis=1)
10        th, th0 = learner(data, labels)
11        correctnum = score(ds[j], ls[j], th, th0)
12        n = len(ls[j][0])
13        print(correctnum, n)
14        scores += [correctnum * 1. / n]
15    return np.average(scores)
16

```

[Run Code](#)
[Submit](#)
[View Answer](#)
[Ask for Help](#)

100.00%

You have infinitely many submissions remaining.

10) Testing

In this section, we compare the effectiveness of perceptron and averaged perceptron on some data that are not necessarily linearly separable.

Use your `eval_learning_alg` and the `gen_flipped_lin_separable` function in the code file to evaluate the accuracy of `perceptron` vs. `averaged_perceptron`. `gen_flipped_lin_separable` is a wrapper function that returns a generator - `flip_generator`, which can be called with an integer to return a data set and labels. Note that this generates linearly separable data and then "flips" the labels with some specified probability (the argument `pflip`); so most of the results will not be linearly separable. You can also **specifiy** `pflip` in the call to the generator wrapper function. At the bottom of the code distribution is an example.

Run enough trials so that you can confidently predict the accuracy of these algorithms on new data from that same generator; assume

training/test sets on the order of 20 points. The Tutor will check that your answer is within 0.025 of the answer we got using the same generator.

Accuracy for perceptron (with flip probability 0.1):

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Accuracy for averaged perceptron (with flip probability 0.1):

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Accuracy for perceptron (with flip probability 0.25):

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Accuracy for averaged perceptron (with flip probability 0.25):

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Modify your `eval_learning_alg` so that it tests hypothesis on the training data instead of generating a new test data set. Run enough trials that you can confidently predict this "training accuracy" for the two learning algorithms. Note the differences from your results above.

Accuracy for perceptron (with flip probability 0.1) on training data:

0.8195

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Accuracy for averaged perceptron (with flip probability 0.1) on training data:

0.8665999999999997

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Accuracy for perceptron (with flip probability 0.25) on training data:

0.6714499999999998

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

Accuracy for averaged perceptron (with flip probability 0.25) on training data:

0.7329499999999997

Submit

View Answer

Ask for Help

100.00%

You have infinitely many submissions remaining.

```
In [2]: import numpy as np
```

Week 1

```
In [3]: def rv(value_list):
        return np.array([value_list])

def cv(value_list):
    return np.array([value_list]).T

def length(col_v):
    return ((col_v.T@col_v)**.5)[0,0]

def normalize(col_v):
    return col_v/length(col_v)

def signed_dist(x, th, th0):
    x = np.array(x)
    th = np.array(th)
    return (x.T@th + th0) / length(th)

def positive(x, th, th0):
    return np.sign(signed_dist(x, th, th0))
```

```
In [4]: data = np.transpose(np.array([[1, 2], [1, 3], [2, 1], [1, -1], [2, -1]]))
labels = rv([-1, -1, +1, +1, +1])
data, labels
```

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

```
In [5]: def point_sign(p, axis):
        return np.sign(signed_dist(p, th, th0))
        th = np.array([[1], [1]])
        th0 = -2
        B = np.apply_over_axes(point_sign, data, axes=1)
        A = B.T == labels
        B,A
```

```
Out[5]: (array([[ 1.],
               [ 1.],
               [ 1.],
               [-1.],
               [-1.]]),
        array([[False, False,  True, False, False]]))
```

```
In [6]: # 1.5

def length(col_v):
    return ((col_v.T@col_v)**.5)[0,0]

def signed_dist(x, th, th0):
    x = np.array(x)
    th = np.array(th)
    return (x.T@th + th0) / length(th)

def score(data, labels, th, th0):
    def point_sign(p, axis):
        return np.sign(signed_dist(p, th, th0))
    return np.sum(np.apply_over_axes(point_sign, data, axes=1).T == labels)
```



```
In [42]: def length(col_v):
        return ((col_v.T@col_v)**.5)[0,0]

def signed_dist(x, th, th0):
    x = np.array(x)
    th = np.array(th)
    return (x.T@th + th0) / length(th)

def score(data, labels, th, th0):
    def point_sign(p, axis):
        return np.sign(signed_dist(p, th, th0))
    return np.sum(np.apply_over_axes(point_sign, data, axes=1).T == labels)

def best_separator(data, labels, ths, th0s):
    mysum = np.sum(np.sign(ths.T @ data + th0s.T) == labels, axis=1)
    i = np.argmax(mysum)
    return ths[:,i:], th0s[:,i:i+1]
```

```
In [43]: data = np.array([
        [1, 1, 2, 1, 2],
        [2, 3, 1, -1, -1]
    ])
labels = np.array([[ -1, -1, 1, 1, 1]])
ths = np.array([
    [0.98645534, -0.02061321, -0.30421124, -0.62960452, 0.61617711, 0.17344772, -0.21804797, 0.26093651, 0.47179
    [0.87953335, 0.39605039, -0.1105264, 0.71212565, -0.39195678, 0.00999743, -0.88220145, -0.73546501, -0.77697
    ])
th0s = np.array([
    [0.65043158, 0.61626967, 0.84632592, -0.43047804, -0.91768579, -0.3214327, 0.0682113, -0.20678004, -0.3396
    ])

mysum = np.sum(np.sign(ths.T @ data + th0s.T) == labels, axis=1)
i = np.argmax(mysum)
i, mysum, ths.T[i]
best_separator(data, labels, ths, th0s)
```

```
Out[43]: (array([[ 0.32548657],
        [-0.83807759]]),
        array([[0.74308104]]))
```

Week 2

```
In [46]: th = np.array([[1, -1, 2, -3]])
```

```
X = np.array([
    [1, -1, 2, -3],
    [1, 2, 3, 4],
    [-1, -1, -1, -1],
    [1, 1, 1, 1]
])
```

```
X@th.T
```

```
Out[46]: array([[15],
               [-7],
               [ 1],
               [-1]])
```

Lab

1)

A) Percy Eptron suggests reusing the training data to assess h : `eval_classifier(h, D_train)`

Explain why Percy's strategy might not be so good.

Because the training data was used to find the hypothesis, the h so it will always return perfect score for D_{train} .

B) Now write down a better approach for evaluating h , which may use h , G , and D_{train} , and computes a score for h . The syntax is not important, but do write something down. What does this score measure and what is the range of possible outputs?

$D_{\text{test}} = G()$ # return new data set $\text{test_score} = \text{eval_classifier}(h, D_{\text{test}})$ # this measures how h good on this unseen set of data that was pulled independently from D_{train} from the dataset. So it's elements from $n+1$ to $n+n'$, where n is length of D_{train} and n' is length of D_{test} .

C) Explain why your method might be more desirable than Percy's. What problem does it fix?

Because this points h wasn't trained at. It fixes the problem of always predicting 0 error rate for D_{train} .

D) How would your method from B score the classifier h , if D_{test} came from a different distribution than G , but D_{train} was unchanged?

Then the func from B would give irrelevant, meaningless score because h was trained on a different set.

2)

A)

Would running the learning algorithm LLL on two different training datasets D_{train1} and D_{train2} produce the same classifier? In other words, would $h_1 = L(D_{\text{train1}})$ be the same classifier as $h_2 = L(D_{\text{train2}})$? What if those training datasets were pulled from the same distribution?

No, it is unlikely that it will produce the same classifier because L might consider those points in different order or even randomly choose h hypothesis. Even if those where the same distribution L will very likely give different h s.

you'll need to assess the learning algorithm's performance in the context of a particular data source.

That's interesting. No free lunch theorem (NFL) states that there are no prediction strategy that scores better than any other strategy if they are taken for all possible problems.

What is the difference between a classifier and a learning algorithm?

Classifier is a hypothesis function that is used to predict labels (classes). A learning algorithm is a function that produces a model or generalization of a problem.

B)

What are GGG and nnn in the code above?

G is a generator, i.e. an iterator over the dataset that produces a matrix with n points and n labels. n is the number of samples to pull from G .

Explain why Linnea's strategy might not be so good.

- Performance problem. And possible out of data. It pulls the same number of points to eval the algorithm as it did to train it. It is unlikely it needs the same number of points to get the score.
- It doesn't provide hyperparameter to L if it requires it as in our lectures the Perceptron requires T hyperparameter.
- It might pull a set for test that is close to the train set, i.e. they are ordered. Not representful.

C)

Is Linnea's strategy good now? Explain why or why not.

- Better as it is now might pull substantial amount of points to detect bad hypothesis.
- Again performance and out of data problems. It needs 11 times n data points
- Again bad representation if points ordered.

D)

```
def better_eval_learning_alg(L, G, n):
    pulled_set = G(n*2) # Or randomly from all data in source without considering n.
    train, test = randomly divide pulled_set into two sets with 0.8-0.2 ratio or other.
    h = L(train.X, train.Y)
    score = eval_classifier(h, test)
    return score
```

E) Explain why your method might be more desirable than Linnea's.

- Now avoiding the ordered points issue.
- Avoiding the performance issue.
- Avoiding out of data if n times 10 leads to it.
- Evaluating only once.

3)

A) In the last section, you thought about how to evaluate a learning algorithm. Now that you are given only 100 labeled data points in total, how would you evaluate a learning algorithm? Specifically, how would you implement `better_eval_learning_alg` from 2C) without G but instead with your 100 labeled data?

Randomly divide those 100 points into two sets for training and for testing. Perhaps 0.9 ratio.

Homework

```
In [1]: # 3) Dual View

theta = [
    -3*2+4+2-2*1,
    2*2-4+2-2]
theta_0 = 2-4-2-1
theta, theta_0
```

```
Out[1]: ([-2, 0], -5)
```

```
In [1]: import matplotlib.pyplot as plt
```

```
In [ ]: a = [.00001, .0001, .001, .01, .1, .2, .5]
```

For these exercises, it will be helpful to review the notes on [Linear Classifiers](#)

1) Feature representation

For the following feature, pick what might be the best encoding for linear classification. The assumption is that there are other features in the data set.

The point of this question is to think about alternatives; there are many options, many not mentioned here.

Car make, e.g. Chevy, Ford, Toyota, VW, for predicting gas mileage (lo, hi).

4 unary features (one-hot): 1000, 0100, 0010, 0001 ▾

Submit

View Answer

100.00%

You have infinitely many submissions remaining.

2) Feature mapping

Consider the following, one-dimensional, data set. It is not linearly separable in its original form.

1. $x^{(1)} = -1, y^{(1)} = +1$

2. $x^{(2)} = 0, y^{(2)} = -1$

3. $x^{(3)} = 1, y^{(3)} = +1$

Ex2.a.: Which of these feature transformations leads to a separable problem?

1. $\phi(x) = 0.5 * x$

2. $\phi(x) = |x|$

3. $\phi(x) = x^3$

4. $\phi(x) = x^4$

5. $\phi(x) = x^{2k}$ for any positive integer k

Enter a Python list with a subset of the numbers 1, 2, 3, 4, 5.

Submit

View Answer

100.00%

You have infinitely many submissions remaining.

Ex2.b.: Your friend Kernelius uses feature transformation $\phi(x) = (x, x^2)$ on the data above. In the new space, the linear classifier with $\theta = (0, 1)$ and $\theta_0 = -0.25$ achieves perfect accuracy. What points from the original space \mathbb{R} map to this linear classifier in \mathbb{R}^2 ? (It may be helpful to find the equation of the separator.)

Enter a Python list with all values of x which constitute this separator.

Submit

View Answer

100.00%

You have infinitely many submissions remaining.