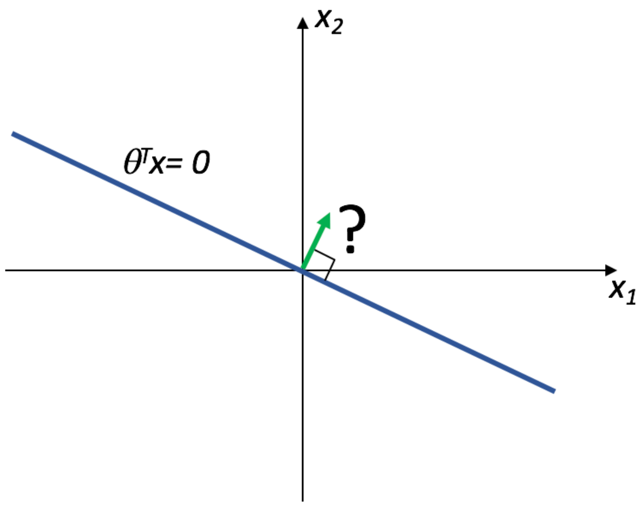
These warmup exercises are divided into two sections:

- 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), 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:

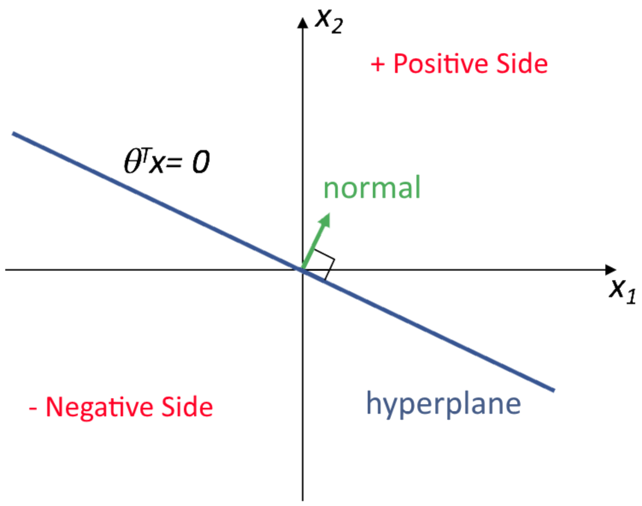
$$\theta^T x = \theta \cdot x = \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_d x_d$$

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 $x \in R^2$ such that $\theta^T x = 0$:

Enter your answer as a Python list of numbers.

Vector on hyperplane: [-2,1]

Ask for Help 100.00%

You have infinitely many submissions remaining.

Solution: `[2, -1]`

Explanation:

There are multiple solutions. Since the plane defined by $\theta^T x = 0$, where $\theta = [1, 2]$, any scalar multiple of $[2, -1]$, including $[4, -2]$ and $[-1, .5]$.

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

Vector normal to hyperplane: [-1,-2]

Ask for Help **100.00%**

You have infinitely many submissions remaining.

Solution: `[1, 2]`

Explanation:

There are multiple solutions. Recall that the dot product between two vectors is 0 if the vectors are perpendicular (or normal) to one another. Thus, the plane defined by $\theta^T x = 0$, where $\theta = [1, 2]$, includes all vectors x that are perpendicular to θ. As a result, a vector that is normal to the hyperplane (all solutions to this equation) is $\theta = [1, 2]$, or any scalar multiples of it, including $[2, 4]$ and $[-1, -2]$.

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

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: theta/norm(theta)

Ask for Help **100.00%**

You have infinitely many submissions remaining.

Multiple Possible Solutions:

Solution 1: `theta/norm(theta)`

$$\frac{\theta}{||\theta||}$$

Solution 2: `-theta/norm(theta)`

$$\frac{-\theta}{||\theta||}$$

Solution 3: `theta/((transpose(theta)@theta)**.5)`

$$\frac{\theta}{((\theta)^T\theta)^{.5}}$$

Solution 4: `-theta/((transpose(theta)@theta)**.5)`

$$\frac{-\theta}{((\theta)^T\theta)^{.5}}$$

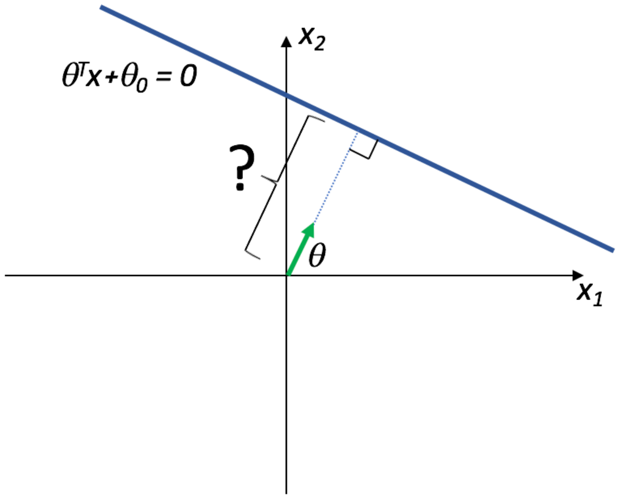
Explanation:

When we say that a hyperplane is 'specified' by θ, the plane is defined by all vectors normal to that vector. Thus, the unit vector normal to the hyperplane is this vector scaled down to unit length. We divide by $||\,\theta\,||$ to accomplish this, giving us $\frac{\theta}{||\,\theta\,||}$. Note that since $-\theta$ is in the colinear to θ (they are the opposite direction but on the colinear), $\frac{-\theta}{||\theta||}$

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 $\{x \mid \theta^T x + \theta_0 \text{ &gt; } 0\}$, 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 = 1

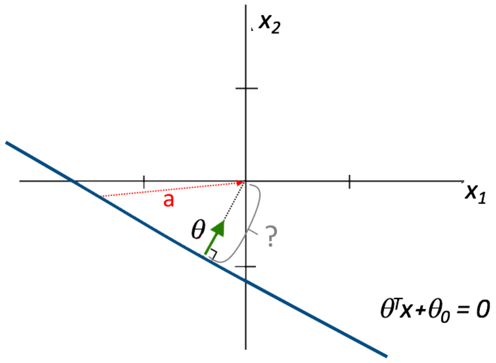
Ask for Help **100.00%**

You have infinitely many submissions remaining.

Solution: `1.0`

Explanation:

One way to measure the distance from the origin to the hyperplane is to measure the length of the projection of some vector a onto θ - that is, to measure the length of the component of a in the direction of θ. We can use formulas for vector projection to help us here.



As displayed in the figure, vector a is defined as a vector from some point x on the line to the origin - it's not important what point. The formula for the projection of vector G onto vector H is $\frac{G \cdot H}{\|H\|}$. The equation is as follows: $\frac{(\langle 0,0\rangle - x)\cdot\theta}{\|\theta\|}$

We can see that the numerator simplifies to $-x\cdot\theta$. Considering the equation for the hyperplane, we see that $\theta^T x + \theta_0 = 0$ - thus, $-\theta^T x = \theta_0$. Substituting this in, we arrive at the following equation:

$$\frac{\theta_0}{\|\theta\|}$$

Plugging in the numbers, $\theta_0 = 5$ and $\|\theta\| = 5$, so the distance from the origin to the hyperplane is 1.

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: theta_0/norm(theta)

Ask for Help **100.00%**

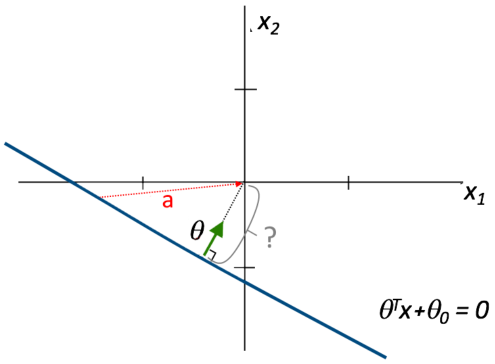
You have infinitely many submissions remaining.

Solution: `theta_0 / norm(theta)`

$$\frac{\theta_0}{\|\theta\|}$$

Explanation:

Another way to measure the distance from the origin to the hyperplane is to measure the length of the projection of vector a onto θ - that is, to measure the length of the component of a in the direction of θ. We can use formulas for vector projection to help us here.



As displayed in the figure, vector a is defined as a vector from some point x on the line to the origin- it's not important what point. The formula for the projection of vector G onto vector H is $\frac{G \cdot H}{\|H\|}$. The equation is as follows:

$$\frac{(\langle 0,0\rangle - x)\cdot\theta}{\|\theta\|}$$

We can see that the numerator simplifies to $-x \cdot \theta$. Considering the equation for the hyperplane, we see that $\theta^T x + \theta_0 = 0$ - thus, $-\theta^T x = \theta_0$. Substituting this in, we arrive at the following equation:

$$\frac{\theta_0}{\|\theta\|}$$

Plugging in the numbers, $\theta_0 = 5$ and $\|\theta\| = 5$, so the distance to the origin is 1.

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.

```python
import numpy as np
A = 0

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

A[:,1]
```

Run Code Submit View Answer Ask for Help

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.

```python
import numpy as np
def tp(A):
    return A.T

```

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,)

Ask for Help **100.00%**

You have infinitely many submissions remaining.

Solution: `(3,)`

Explanation:

Using one set of square brackets creates a 1-dimensional array, whose dimension is 3, in correspondence with the number of elements, yielding (3,).

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)

Ask for Help **100.00%**

You have infinitely many submissions remaining.

Solution: `(4, 1)`

Explanation:

The colon signifies to create a splice using every row of A and the 1:2 is a splice for the second column. Since both indices are splices, numpy preserves the dimensions, resulting in a 1D array of size (4,1).

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.

```python
import numpy as np
def rv(value_list):
    return np.array([value_list])

```

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.

```python
import numpy as np
def cv(value_list):
    return rv(value_list).T

```

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.

```
import numpy as np
def length(col_v):
    np.sqrt(np.sum(np.square(col_v.T)))

```

Ask for Help **0.00%**

You have infinitely many submissions remaining.

Here is the solution we wrote:

```
import numpy as np
# Takes a d by 1 matrix;  returns a scalar
# of their lengths
def length(col_v):
    return np.sum(col_v * col_v)**0.5
```

Explanation:

First, recall the formula for the Euclidean norm of a row or column vector is $\sqrt{\sum_k c_k^2}$, where the c_k 's are the entries of the vector. Thus, we can think of coding the Euclidean norm as a three step process:

1. Square each item of the original vector (this can be done with element-wise multiplication, `col_v*col_v`, or the built-in square function, `np.square(col_v)`)
2. Sum up all the elements in the new vector (this can be done with `np.sum()`)
3. Take the square root of the previous sum.

Putting these three steps together gives us `np.sum(col_v*col_v)^{0.5}` (or `np.sum(np.square(col_v))^{0.5}`)

Another way to solve this problem is to use an alternate formulation of the Euclidean norm - namely, that the norm of a vector is the square root of the inner product of the vector with itself. In matrix form, this formula can be written as $||\, v \,|| = \sqrt{v^T \cdot v}$. We can write this out in code as `np.asscalar(np.dot(np.transpose(col_v), col_v)^{0.5})`. We need to use `np.scalar()` here because `np.dot` will return an array, and this function must return a scalar.

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).

```python
import numpy as np
def normalize(col_v):
    return col_v/length(col_v)

```

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.

```python
import numpy as np
def index_final_col(A):
    return A[:,A.shape[-1]-1:A.shape[-1]]

```

Ask for Help **100.00%**

You have infinitely many submissions remaining.

Here is the solution we wrote:

```python
import numpy as np
# Takes a 2D matrix;  returns last column as 2D matrix
def index_final_col(A):
   return A[:,-1:]
```

Explanation:

Since we want only the last column, we can index `A[:,-1]`. However, recall from Problem 2.9, that we if we splice an np.array, then it will preserve the dimension. So we provide a splice that will only index for the last column: `A[:,-1:]`

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:

```
import numpy as np
data = 0 #your code here
data = np.array([[150,5.8],[130,5.5],[120,5.3]])
```

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

```python
import numpy as np
def transform(data):
    return np.sum(data, keepdims = True, axis = 1)

```

Ask for Help **100.00%**

You have infinitely many submissions remaining.

Here is the solution we wrote:

```python
import numpy as np
def transform(data):
    return (np.dot(data, np.array([[1], [1]])))
```

Explanation:

Here d is a 3x2 matrix. We can multiply d by a 2x1 column vector of 1s to sum each row of d. This then gives us a 3x1 column vector representing sums of each person's height and weight.