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 hhh: 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 perferct score for D_train.

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

D_test = G() # return new data set test_score = eval_classifier(h, D_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 hhh, if Dtest\mathcal{D}*{\it test}Dtest came from a different distribution than G\mathcal{G}G, but Dtrain\mathcal{D}*{\it train}Dtrain 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 Dtrain1\mathcal{D}_{\it train*1}Dtrain1 and Dtrain2\mathcal{D}*{\it train_2}Dtrain2 produce the same classifier? In other words, would h1h*1h1 = L(Dtrain1)L(\mathcal{D}*{\it train_1})L(Dtrain1) be the same classifier as h2h*2h2 = L(Dtrain2)L(\mathcal{D}*{\it train_2})L(Dtrain2)? 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 algorith 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\mathcal{G}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]
```