November 14, 2017

CS 361: Probability & Statistics

Classification

Minimization

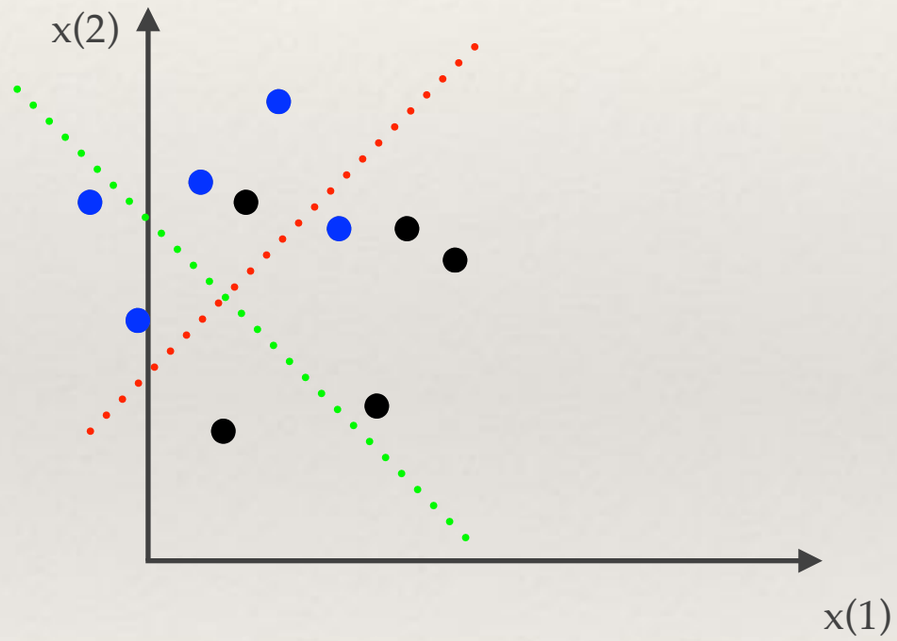
What we are doing is searching for good values of **a** and *b*. Our function g is a function of **a** and *b*. It tells us how good **a** and *b* are as choices

You can think of the search for a good **a** and *b* as: in a large national park we are trying to find the latitude/longitude of the spot in the park with the lowest elevation, we have a guide with us (*g*) who is pretty tricky, the guide will only tell us our current elevation. We could ignore the geography and just take random steps and ask at each step for the new elevation and only take steps that decrease our elevation. Or we could try to always be walking down hill

In this metaphor, our **a** and *b* are where we are in the space. The training data and the loss function interact to say how the space is shaped—what its topography is

Loss functions

We want our algorithm to give us a decision surface that separates the data if possible



If it's not possible we still prefer the red to the green decision surface above

What we did last time

For the Perceptron algorithm we specified a prediction function and a loss function

In order to solve the optimization problem of finding a set of parameters that minimized the loss on the training data, we reduced the problem to an iterative search

We used the gradient of the loss function to figure out how to update our best guess for the parameters at each iteration

This gave us a straightforward set of update equations that we could use to train the Perceptron with our training data

Predicting on new data

After training, we have an **a** and b which we can use to predict the class labels of new data

For this we just use our prediction function with our actual learned values of **a** and b

$$\text{sign}(\mathbf{a} \circ \mathbf{x} + b)$$

The support vector machine

SVM

The support vector machine or SVM is a classifier that is similar to the one we have just constructed. Our prediction function is still

$$\text{sign}(\mathbf{a} \circ \mathbf{x} + b)$$

So our goal is to find an **a** and b that produces a decision boundary that separates the data well.

However, we will make be making some modifications to the loss function to account for the fact that having a loss function that just looks at the training data might not be wise

The loss function

We will choose an **a** and b by choosing values that minimize a cost function. Our cost function will have the form

training error cost + penalty term

First let's focus just on how we can quantify the training error

The training error

For data item i, we will write the prediction that our function makes as

$$\gamma_i = \mathbf{a}^T \mathbf{x}_i + b$$

For training data, we have the true labels of the data, so we compare our prediction with the true label. If we write a function that compares our prediction with the true label

$$C(\gamma_i, y_i)$$

Then our training error cost will be of the form

$$\frac{1}{N} \sum_{i=1}^{N} C(\gamma_i, y_i)$$

Specifying C

What properties should C have?

We have a correct prediction if our γ_i is negative when the true label is -1 or positive when the true label is +1. So we are happy if γ_i and y_i have the same sign

So C should be large when γ_i and y_i have different signs. Furthermore, if the signs are different and γ_i has a large magnitude C should be even larger

The reason is that **ax**+b gets larger in magnitude as **x** gets further from the decision boundary

Specifying C

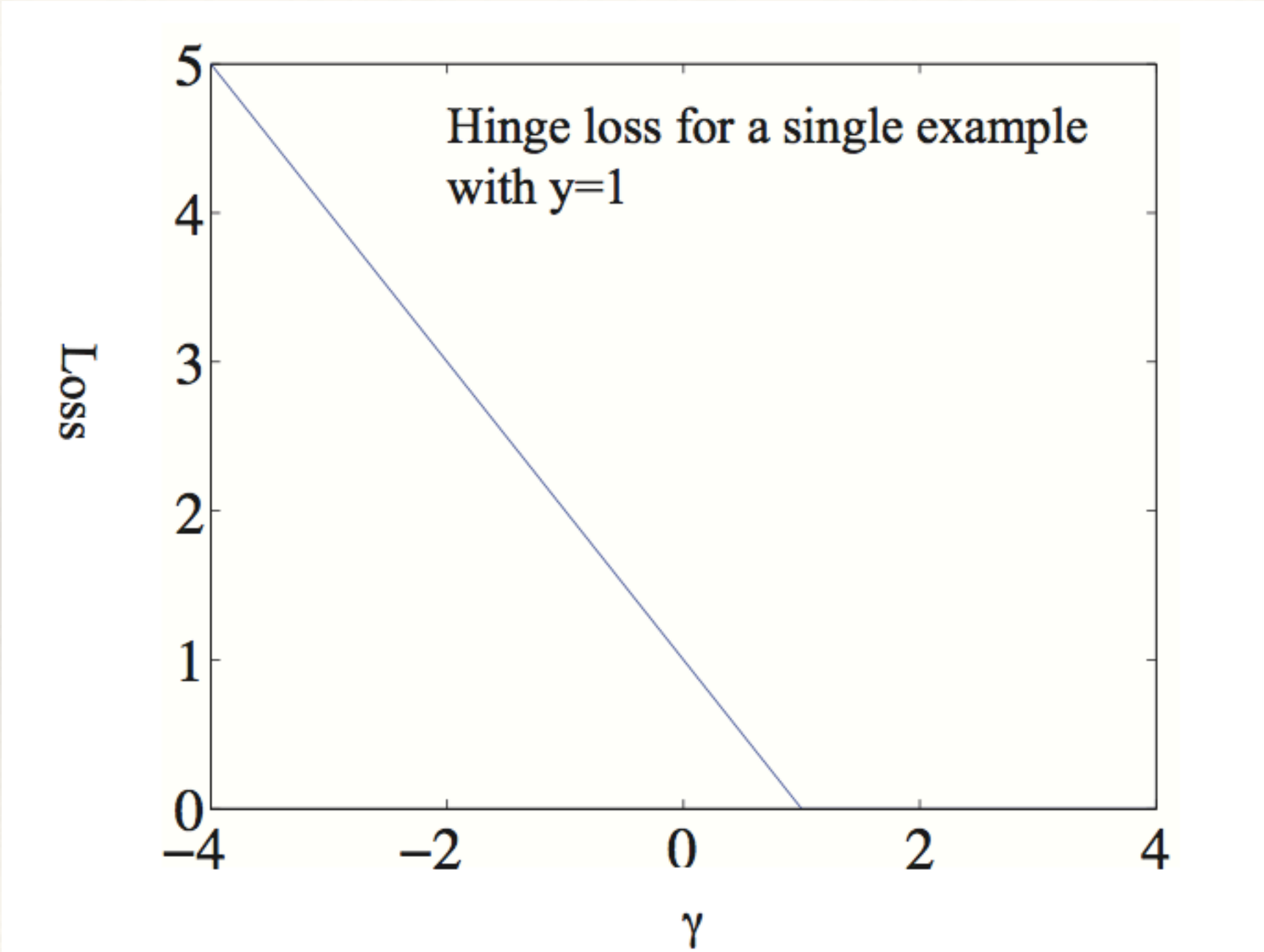
If γ_i and y_i have the same sign but γ_i has a small magnitude, then that means **x_i** is close to the decision boundary, so while **x_i** was correctly predicted, points close to it in the test set may not be. So in order to give ourselves some wiggle room, we will have

$$C(\gamma_i, y_i) = \max(0, 1 - y_i\gamma_i)$$

We see that if the predicted and the actual value have different sign—a wrong prediction—then C > 1. If the prediction and the actual value have the same sign and the magnitude of the prediction is large, i.e. $y_i\gamma_i > 1$ the cost is 0. And if the prediction is correct but very close to the boundary, i.e. $0 \leq y_i\gamma_i \leq 1$ the cost is between 0 and 1

The kind of loss function is called the **hinge loss**

Hinge loss



Loss functions

Our loss function for the Perceptron only depended on the training data

It is not uncommon in practice to include other terms in a loss function that have nothing to do with the training data

Sometimes we might want to express other preferences via the loss function

The penalty term

For test data that we cannot yet observe, imagine we classify an example wrongly. If $||a||$ is small, then this example is at least close to the decision boundary and so nearby examples might be classified correctly. So all else being equal, we prefer a small $||a||$

So if we try to minimize

$$S(\mathbf{a}, b; \lambda) = \left[(1/N) \sum_{i=1}^{N} \max(0, 1 - y_i (\mathbf{a} \cdot \mathbf{x}_i + b)) \right] + \frac{\lambda}{2} \mathbf{a}^T \mathbf{a}$$

We are taking into account both our desire for small $||a||$ and our desire to do well on the training set. The value of λ controls the degree to which we emphasize these two aspects of the loss function. It is sometimes referred to as a regularization parameter

Finding good parameters

$$S(\mathbf{a}, b; \lambda) = \left[(1/N) \sum_{i=1}^{N} \max(0, 1 - y_i (\mathbf{a} \cdot \mathbf{x}_i + b)) \right] + \frac{\lambda}{2} \mathbf{a}^T \mathbf{a}$$

In practice what we will do, is assume we have chosen a λ and then find an $\mathbf{a}$ and b to minimize S. And then separately we will try to estimate a good value of λ

We will use stochastic gradient descent yet again which means we need to compute the gradient of the loss function with respect to a single training example

Gradient

$$S(\mathbf{a}, b; \lambda) = \left[(1/N) \sum_{i=1}^{N} \max(0, 1 - y_i (\mathbf{a} \cdot \mathbf{x}_i + b)) \right] + \frac{\lambda}{2} \mathbf{a}^T \mathbf{a}$$

Our loss function is different this time, so our update rules will be as well, for our step direction, we choose a random k from among our training examples and have

$$p_k = \begin{cases} \begin{bmatrix} \lambda \mathbf{a} \\ 0 \end{bmatrix} & \text{if } y_k (\mathbf{a} \cdot \mathbf{x}_k + b) \geq 1 \\ \begin{bmatrix} \lambda \mathbf{a} - y_k \mathbf{x} \\ -y_k \end{bmatrix} & \text{otherwise} \end{cases}$$

Giving update rules of

$$\mathbf{a}^{(n+1)} = \mathbf{a}^{(n)} - \eta \begin{cases} \lambda \mathbf{a} & \text{if } y_k (\mathbf{a} \cdot \mathbf{x}_k + b) \geq 1 \\ \lambda \mathbf{a} - y_k \mathbf{x} & \text{otherwise} \end{cases}$$

and

$$b^{(n+1)} = b^{(n)} - \eta \begin{cases} 0 & \text{if } y_k (\mathbf{a} \cdot \mathbf{x}_k + b) \geq 1 \\ -y_k & \text{otherwise} \end{cases}$$

The regularization parameter

Our training for the SVM so far was only with respect to the parameters **a** and b. Our prediction function is still just

$$\text{sign}(\mathbf{a} \circ \mathbf{x} + b)$$

This doesn't depend on λ. Nevertheless, our update equations for training did use this parameter, so we can't really find **a** and b without some value for this parameter

For a parameter like this that isn't part of the prediction function, we will just try different values of it and see which one works best rather than derive an optimal value.

We do this by holding out some portion of our training data as a validation set. We run the training procedure on the training data for a few different values of the parameter and evaluate which setting of the parameter gives the best accuracy on the held out data

Classifier evaluation

The normal flow for training and evaluating a classifier is to take the data we have available, keep most of it for training and some for evaluation

You run the training algorithm on the classifier with the training data and then evaluate it against the validation set on which it was not trained.

If there is something like the regularization parameter which the SVM has, the process can be repeated above for multiple different values of λ

We will keep the classifier (characterized by $\mathbf{a}$ and b) which performs best on the validation data and we will report its accuracy: percentage of items in the validation set whose labels were correctly predicted

Classifier evaluation

After we've trained a classifier we have a nice machine that can look at new data and hopefully accurately apply class labels. There are many other design considerations besides accuracy that we may want to take into account

How long does a classifier take to train?

How long does it take to make a prediction?

Is the classifier interpretable? Does looking at the resulting classifier tell us anything interesting about our data?

Among others

Multi-class classification

We have considered the Perceptron and SVM classifiers which are binary classifiers

However, we can use binary classifiers to handle multiple classes in a couple of different ways

In the **all vs all** approach, we would train a classifier for each pair of potential class labels. For instance if we were trying to train a classifier that analyzed radiology images and made a diagnosis of healthy, benign tumor, malignant tumor, in the all vs all approach we would train a classifier for healthy vs benign, healthy vs malignant, and benign vs malignant. We then run all of the classifiers and see which class label wins the most 1vs1 competitions

Multi-class classification

Another approach for using binary classifiers to handle multiple classes is the **one-vs-all** approach

In this case we train a classifier for each label separately. For example we would train a healthy vs non-healthy, benign vs non-benign, and malignant vs non-malignant classifier. To classify a new example, we show it to each of the classifiers and record the score. We output the class label which had the highest score

All vs all requires training O(N^2) classifiers, whereas one vs all requires O(N). In practice both methods work fairly well

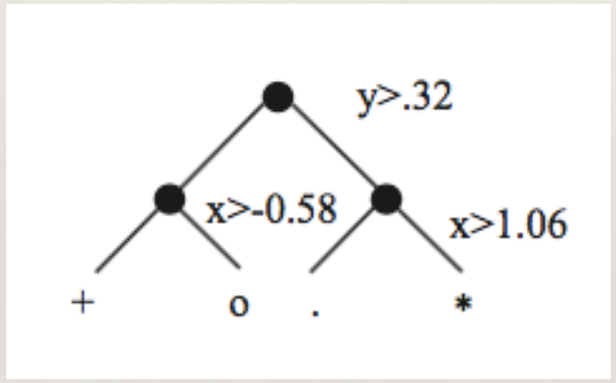
Class confusion matrices

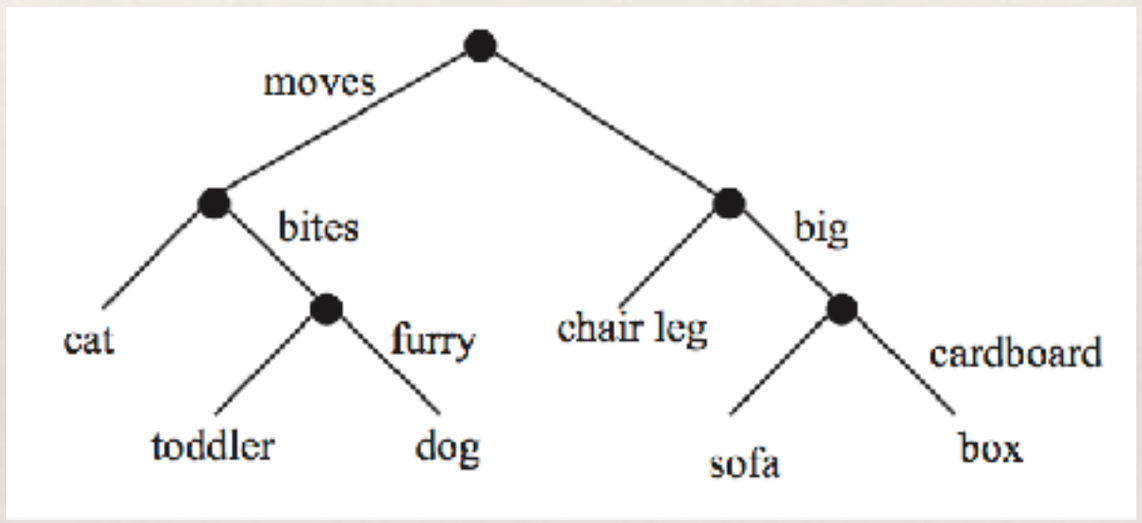
A tool for evaluating multi-class (including binary) classifiers

	Predict 0	Predict 1	Predict 2	Predict 3	Predict 4	Class error
True 0	151	7	2	3	1	7.9%
True 1	32	5	9	9	0	91%
True 2	10	9	7	9	1	81%
True 3	6	13	9	5	2	86%
True 4	2	3	2	6	0	100%

Decision trees

Examples


y>.32
x>-0.58
x>1.06
+
o
.
*


moves
bites
big
cat
furry
chair leg
cardboard
toddler
dog
sofa
box

Decision trees

A decision tree is a type of classifier that operates as follows

Each internal node corresponds to a single dimension of the data and a binary **split** for that dimension, i.e. a set of values for the dimension which points to the left child and the remainder of values which point to the right child

To classify a new data item, you start at the root node. Being an internal (non-leaf) node, the root will refer to one of the dimensions of the data item. Depending on the split, the data item will next visit the left or right child

Continue to traverse the tree according to dimensions and splits until reaching a leaf. Each leaf in the tree gives a class label

How to make a decision tree from training data

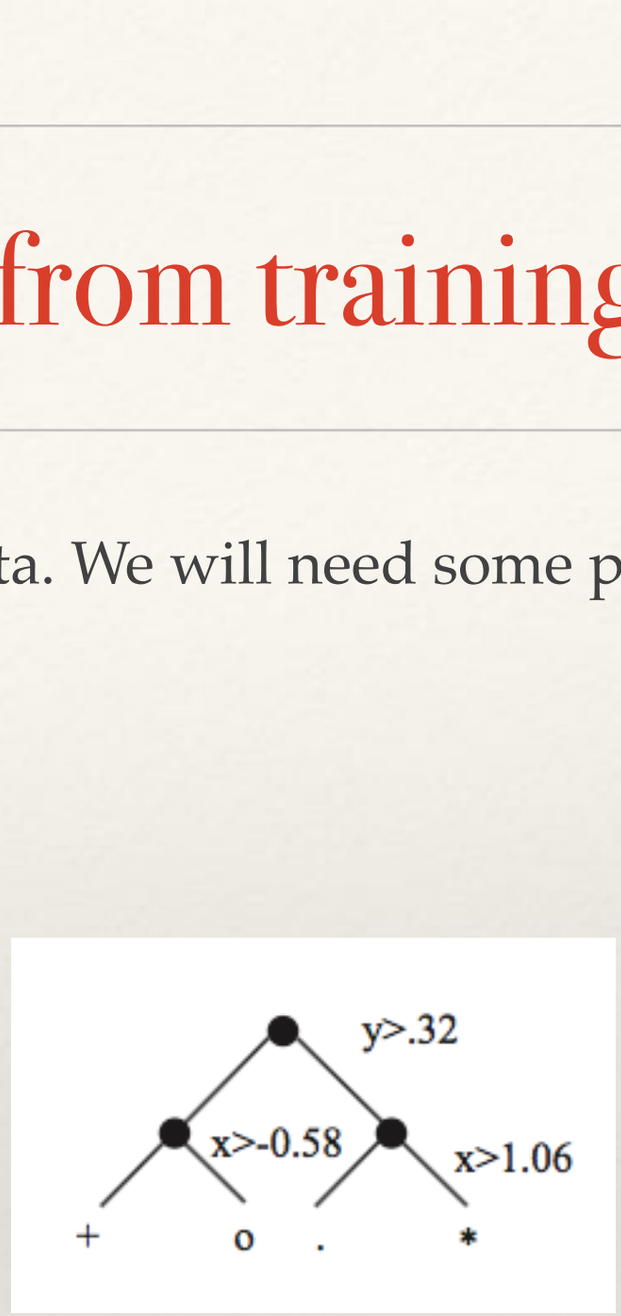
We will recursively build our tree using the training data. We will need some principles in order to

Decide which dimension to split on

Decide how to split the dimension

Decide how deep to make the tree
or equivalently when to end
recursion

Decide which labels to put on the leaves



What kind of splits

We want to split the data in some informative way

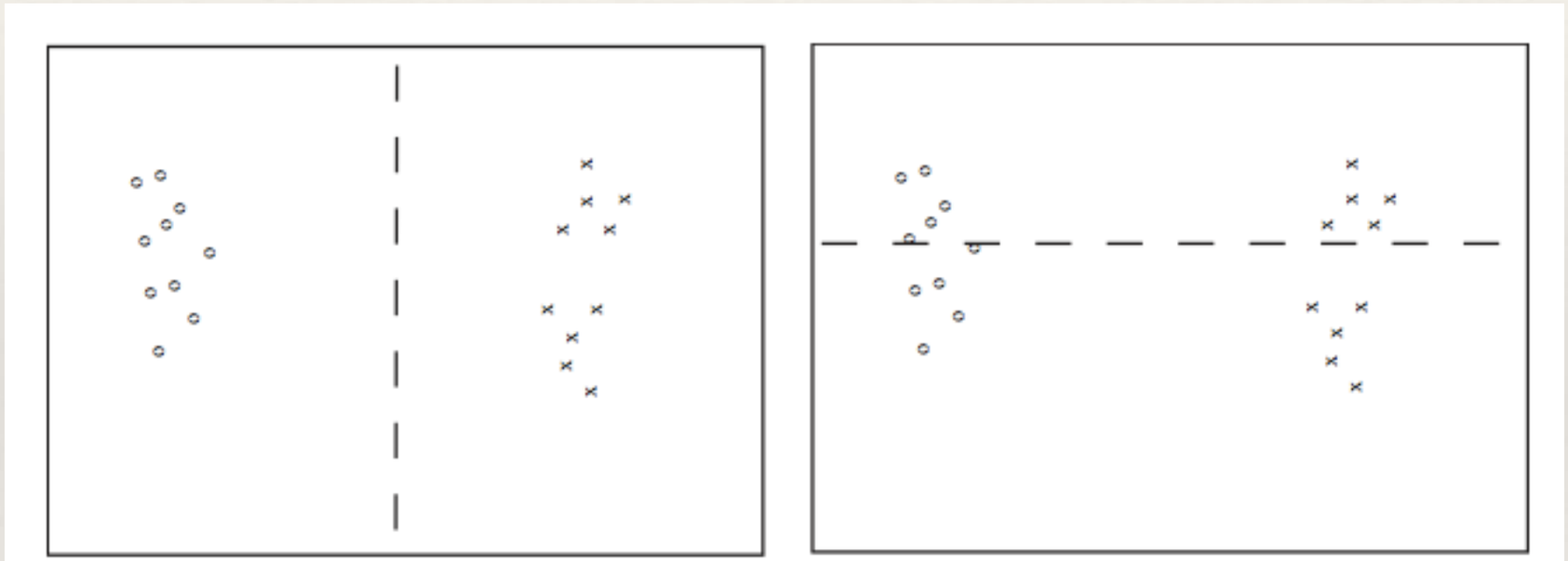
Suppose this was our entire training data set

X1	X2	Class
0	0	-1
0	8.5	-1
0.5	3.25	-1
1.2	-1.5	-1
2.5	5	-1
3.75	1	1
4	1.5	1
4.5	4.5	1
4	3.25	1
5.25	5.5	1

Why is X1>3 a great split? Why is X2 > 3 not as good?

What kind of split

In general we would like it if whatever the "class balance" at the node is (say it starts out 50-50 class 1 and class -1) is more imbalanced in the children



Informative split

Less informative split

Entropy

It turns out that a good way to measure this "class imbalance" feature of a dataset comes to us from communication theory or information theory

The **entropy** of a probability distribution is the smallest number of bits on average you would need to identify an item sampled from a distribution. It can be thought of as measuring the degree of uncertainty of the distribution.

We won't get too into the details but this quantity is large when the probability distribution is close to uniform and small when one class is very likely while the others are unlikely. The entropy is denoted with an *H* and is given by the following formula

$$-\sum_i p(i) \log_2 p(i)$$

Where the *p(i)* are the probability of class *i* which in this case is just the relative frequency of the class

Example

Calculate the entropy for this dataset

$$-\sum_i p(i)\log_2 p(i)$$

X1	X2	Class
0	0	-1
0	8.5	-1
0.5	3.25	-1
1.2	-1.5	-1
2.5	5	-1
3.75	1	1
4	1.5	1
4.5	4.5	1
4	3.25	1
5.25	5.5	1

We get

$$-5/10\log(5/10) - 5/10\log(5/10) = 1$$

Split evaluation

Let H(P) be the entropy of all the training data. In order to evaluate a potential split at the root node of the tree, we want to compute the difference between this entropy and the average entropy remaining in the left and right children nodes, which we will call H(P_l) and H(P_r) respectively.

The average entropy after the split between the left and right subtrees will be

$$\frac{N(\mathcal{P}_l)}{N(\mathcal{P})}H(\mathcal{P}_l) + \frac{N(\mathcal{P}_r)}{N(\mathcal{P})}H(\mathcal{P}_r)$$

Where N(P_l) is, for instance, the number of items that wind up in the left subtree

Information gain

This difference between the entropy or average number of bits needed before and after a split is called the **information gain** of the split. We would like to choose the split with the largest information gain

$$I(\mathcal{P}_l, \mathcal{P}_r; \mathcal{P}) = H(\mathcal{P}) - \left(\frac{N(\mathcal{P}_l)}{N(\mathcal{P})} H(\mathcal{P}_l) + \frac{N(\mathcal{P}_r)}{N(\mathcal{P})} H(\mathcal{P}_r) \right)$$

Example

For the training set below, evaluate the information gain of the split x2>1

X1	X2	Class
0	0	-1
0	8.5	-1
0.5	3.25	-1
1.2	-1.5	-1
2.5	5	-1
3.75	1	1
4	1.5	1
4.5	4.5	1
4	3.25	1
5.25	5.5	1

From our prior example we know that $H(P) = 1$

Let's say that data items {2, 3, 5, 7, 8, 9, 10} go to the right subtree and items {1, 4, 6} go to the left

Example

Let's say that data items {2, 3, 5, 7, 8, 9, 10} go to the right and items {1, 4, 6} go to the left

X1	X2	Class
0	0	-1
0	8.5	-1
0.5	3.25	-1
1.2	-1.5	-1
2.5	5	-1
3.75	1	1
4	1.5	1
4.5	4.5	1
4	3.25	1
5.25	5.5	1

$$-\sum_i p(i) \log_2 p(i)$$

What is H(P_r)?

$$-3/7 \log(3/7) - 4/7 \log(4/7) = 0.985$$

Example

Let's say that data items {2, 3, 5, 7, 8, 9, 10} go to the right and items {1, 4, 6} go to the left

X1	X2	Class
0	0	-1
0	8.5	-1
0.5	3.25	-1
1.2	-1.5	-1
2.5	5	-1
3.75	1	1
4	1.5	1
4.5	4.5	1
4	3.25	1
5.25	5.5	1

$$-\sum_i p(i) \log_2 p(i)$$

What is H(P_l)?

$$-2/3 \log(2/3) - 1/3 \log(1/3) = 0.918$$

Example

$$I(\mathcal{P}_l, \mathcal{P}_r; \mathcal{P}) = H(\mathcal{P}) - \left(\frac{N(\mathcal{P}_l)}{N(\mathcal{P})} H(\mathcal{P}_l) + \frac{N(\mathcal{P}_r)}{N(\mathcal{P})} H(\mathcal{P}_r) \right)$$

So we had H(P) = 1, H(P_l) = 0.918, H(P_r) = 0.985, N(P_l) = 3, N(P_r) = 7, N(P)=10

Giving an information gain of 0.0351