April 4, 2018

CS 361: Probability & Statistics

Classification

Learning to classify

Classification

A classifier is a procedure that accepts a set of features and produces a class label for them

For example, if you wanted to determine whether a given Twitter user was a bot or not, you could watch the account, record some activity, call this activity the features, and decide whether it is or not according to some procedure

Another example: you might look at a long sequence of DNA and want to know whether it contains a gene or not according to some rule

A final example: credit card companies must decide if each transaction it processes is fraudulent or not

Classification

All of the previous examples were **binary classifiers** in that they applied one of two class labels to each input

We can also have multi-class classifiers

Doctors' diagnoses can be thought of as complex multi-class classifiers: you visit your doctor, they observe some features of your presentation, implement some procedure, and spit out a diagnosis

Likewise, the grading policy of a class is a procedure for looking at the features of a student's performance and spitting out a label corresponding to their letter grade

Classifiers

Of course, we will focus on classifiers that are easy for us to formalize mathematically and are easy for us to program a computer to implement

So it's useful for us to think of a classifier as being a function that can map from d dimensional **feature vectors** x_i to **class labels** y_i

We will construct our classifiers by learning. That is, we will have a set of **training data**. Which are **feature vectors** x_i with the correct **class labels** y_i which we will use to come up with a function f which we hope performs well on unseen, run-time data

The BEST classifier

We want to use the training data to find the classifier that does the “best” on unseen data

So we have two big problems:

We have to define what a good classifier is

And we have to perform well on data we can’t and won’t see

Performance

We can summarize the performance of a classifier by using the error rate or the accuracy of the classifier, compared to some baseline

If we have two classes, what is the worst error rate (or accuracy) we can get?

50% would be the worst we could get. Anything lower, we could just predict the opposite of what the classifier says and have a classifier which does better than 50%. Another way to think about it is that if we just randomly predicted class labels, we would get an accuracy of around 50%

If we have C classes, then just by predicting random class labels we would get an accuracy of approximately 1 - 1/C. This baseline is always available to compare our accuracy with

Loss functions

In order to determine how good a classifier is, we need some way to represent the cost of making mistakes. A function which assigns costs to mistakes is called a **loss function**.

A binary classifier can only make two kinds of mistakes. Without loss of generality, we think of the two possible labels in a binary classification problem as being "positive" and "negative"

Such a classifier can make the mistake of a **false negative** in which it applies the label "negative" to something which should have been positive or it can make the mistake of a **false positive** in which it applies the "positive" label to an input which should have been labeled negative

Loss functions

Many classifiers will attempt to minimize the loss on training data

If we wanted to build a classifier to diagnose a disease that can be deadly if untreated but which has a cheap and easy treatment, then the cost of a false negative should be much higher than the cost of a false positive

Likewise a disease with a treatment that is unpleasant relative to its effects untreated might be best served by classifier that penalizes false positives more than false negatives

We could have both kinds of mistakes be equally costly, as well. The so-called **0-1 loss** function assigns loss 1 to every mistake and 0 to every correct answer

The training and test data

What we really want is for the classifier to have a small loss on the unseen data

We generally assume that our training data resembles the unseen data. We want our training data to look a lot like the data that the classifier might encounter out in the wild. So you might say "if we come up with a classifier that can do well on the training data we are happy"

But consider this classifier. On input x, if x was in the training set it goes and returns the label associated with x, otherwise it outputs a random label. This has zero training error but probably won't do well on any unseen data

Classifiers usually perform worse on unseen data than training data because of an effect called **overfitting**—the classifier fits the training data better than the unseen data

The training and test data

In order to have a sense of the amount of overfitting, classifiers should be evaluated on data that was not used in training

One way to do this is to set aside some portion of the training data to use as a **validation set** or **test data**. We would then use the remainder of the training data to train the algorithm and then evaluate it on the basis of the validation set which it hasn't seen

Cross-validation

If we set aside 10% of the training data for testing purposes—to simulate unseen data, there's a problem

Labeled training data is often expensive to get. And the more data we use for training, the better our classifier's accuracy will be.

We wouldn't need to build a classifier if coming across true labels wasn't prohibitively expensive. In some cases training data can be extremely hard to get. Do we want to waste 10% of it on evaluation or do we want to use virtually all of it to train the classifier better?

One thing we can do is split the data into training and test data multiple times and average the error. In the extreme case, we can train the data on all but one training item and evaluate on the held out item, and do this for every item in turn. This is known as leave-one-out cross validation

The perceptron

Overview

Some steps that we will have to take any time we are building a classifier:

1) We will make some assumptions about how features and class labels are encoded

2) We will specify the form of our function for computing class labels from features and identify if and how it is parameterized

3) We will specify a loss function deciding how to score correct and incorrect predictions.

4) We will derive a strategy for finding the function which minimizes loss

The data

We will assume that we have *N* example data points that belong to two classes, indicated by 1 and -1. We can write our dataset as

$$\{(x_1, y_1), \ldots, (x_N, y_N)\}$$

Where the x_i are the feature vectors (of *d* dimensions) and the y_i are the class labels

The prediction function

The prediction function must predict a class label when presented with an input of features

One possible design choice—the one we use in perceptron—will take a linear combination of the features to produce our predicted class label. In particular, we will predict the class label 1 for a data point x if

$$a_1 x^{(1)} + a_2 x^{(2)} + \ldots + a_d x^{(d)} + b \geq 0$$

and label -1 otherwise. Another way of writing the above is

$$\mathbf{a} \circ \mathbf{x} + b \geq 0$$

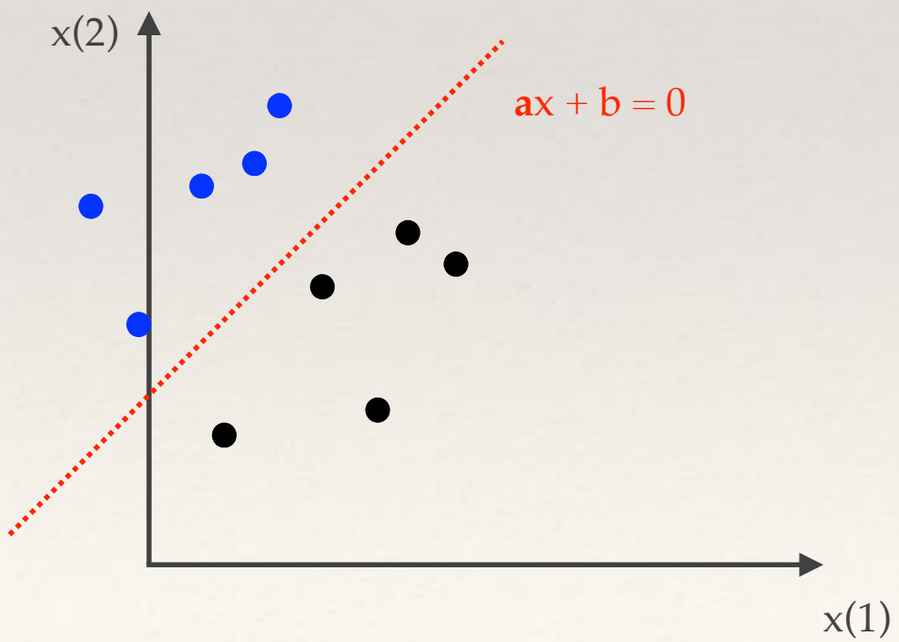
Or that we are reporting the output of

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

The prediction function

The **a** and *b* are what we are learning here. We want to find a good set of coefficients for the linear combination on the last slide

Choosing an **a** and *b*, we can consider the points where **a**x+b=0. This set of points will form a hyperplane which we call a linear decision surface or decision boundary. All points on one side of the boundary will be classified as +1 and on the other will be classified as -1



Example

If our data were one dimensional our prediction function on a data point x_i would be

$$\text{sign}(ax_i + b)$$

Which means our decision boundary is at

$$x_i = -b/a$$

For 2 dimensional data we have the following prediction

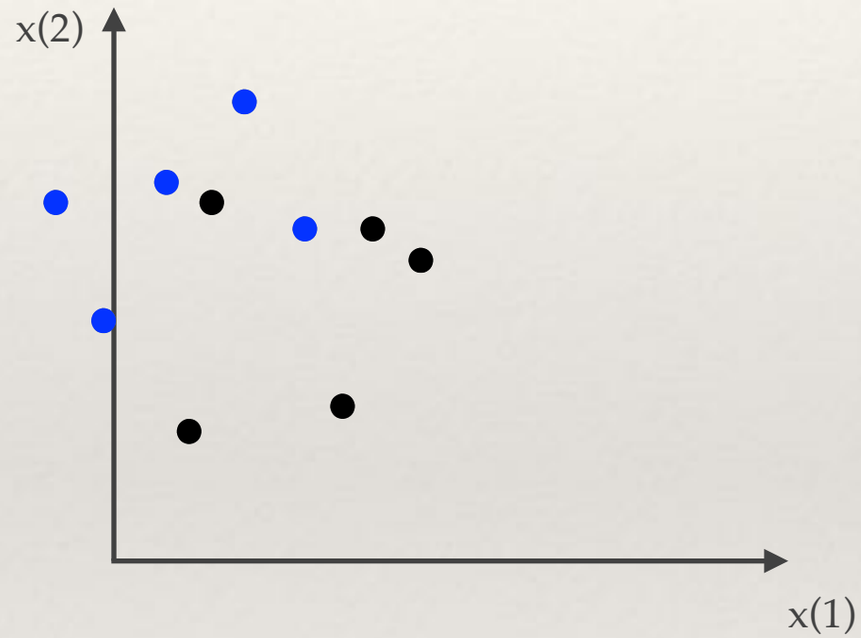
$$\text{sign}(\mathbf{a}^T \mathbf{x_i} + b)$$

The decision boundary is given by the line

$$x_i^{(2)} = -a_1/a_2 x_i^{(1)} - b/a_2$$

The prediction function

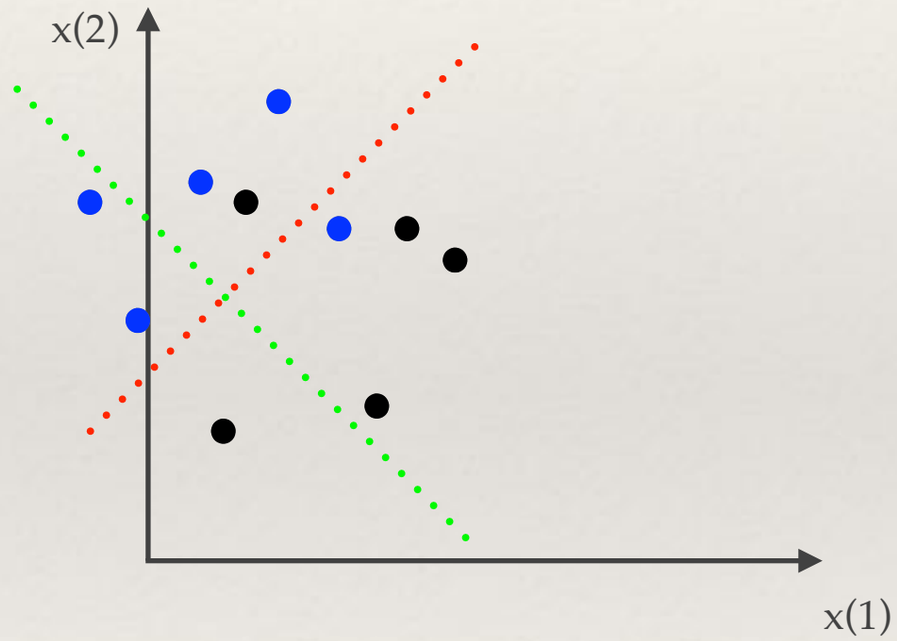
It's not hard to come up with examples that this classifier has trouble with



What we find in practice, however, is that this family of classifiers is strong. In particular, if we can add more features to the data, experience shows that the error rate improves

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

The loss function

We will choose an **a** and b by choosing values that minimize a cost function. For the perceptron our cost will be a function only of how poorly we do on the training data

The training error

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

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

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

A good choice for C, then, is

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

We see, then, that if we have a correct prediction, then $\gamma_i y_i$ will be positive, so we will have C=0. On the other hand, if the signs of γ_i and y_i differ, then our cost will be $\gamma_i y_i$. A cost which will more heavily penalize an incorrect prediction of large magnitude, i.e. an incorrect prediction far from the decision boundary

Our goal, then, will be to find an **a** and *b* to minimize

$$S(\mathbf{a}, b) = \frac{1}{N}\sum_{i=1}^{N} \max(0, -y_i(\mathbf{a}\mathbf{x}_i + b))$$

Gradient descent

Minimization overview

What we are doing is searching for good values of **a** and *b*. Our function S 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 (*S*) 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

Minimizing loss

If we write our parameters as a single vector

$$\mathbf{u} = [\mathbf{a}, b]$$

In general, we are trying to minimize a function S(**u**)

Sometimes this problem is solved by finding the gradient of *S* and finding a value of **u** that makes the gradient equal 0. But usually this doesn't work in practice

In practice, we will **search** for the value of **u** that minimizes *S*. We will do this by beginning with a start point for **u** and then iteratively calculating a direction to step in, **p**, such that **u**+**p** gives a smaller value for *S* than **u** did

Which step

If our current guess for **u** is

$$\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ \cdots \\ u_d \end{pmatrix}$$

We know evaluating

$$\nabla S = \begin{pmatrix} \frac{\partial S}{\partial u_1} \\ \frac{\partial S}{\partial u_2} \\ \cdots \\ \frac{\partial S}{\partial u_d} \end{pmatrix}$$

at **u** gives the direction of largest increase in S at **u**. So if we we were to take a small step in the direction $-\nabla S(\mathbf{u})$ we should get a **u** with a smaller value of S

Which step

So our rule for updating our best guess of **u** at the i-th step, will be

$$\mathbf{u}^{(i+1)} = \mathbf{u}^{(i)} + \eta p^{(i)}$$

where η is our step size, and specifically

$$\mathbf{u}^{(i+1)} = \mathbf{u}^{(i)} - \eta \nabla S(\mathbf{u}^{(i)})$$

But remember our loss function was an average over the whole dataset, so

$$-\nabla S(\mathbf{u}) = -\frac{1}{N}\sum_{i=1}^{N} \nabla S_i(u)$$

Reminder

$\mathbf{u} = [\mathbf{a}, b]$

p is our step direction

Stochastic gradient descent

$$-\nabla S(\mathbf{u}) = -\frac{1}{N}\sum_{i=1}^{N}\nabla S_i(u)$$

But we may not want to have to iterate through our entire dataset for each descent step to calculate which way to descent.

Instead, we estimate the gradient of S by picking a random k in the range $1, \ldots, N$ and choose our step on the basis of just one of the training examples

$$\mathbf{p}_k = -\nabla S_k(\mathbf{u})$$

This is a random quantity. If we took the expected value, we would get

$$E[\mathbf{p}_k] = -\nabla S(\mathbf{u})$$

Stochastic gradient descent

In other words, we are approximating our loss function when we do stochastic gradient descent. Rather than stepping in the direction of the negative gradient of

$$\frac{1}{N}\sum_{i=1}^{N} max(0, -y_i(\mathbf{a}\mathbf{x}_i + b))$$

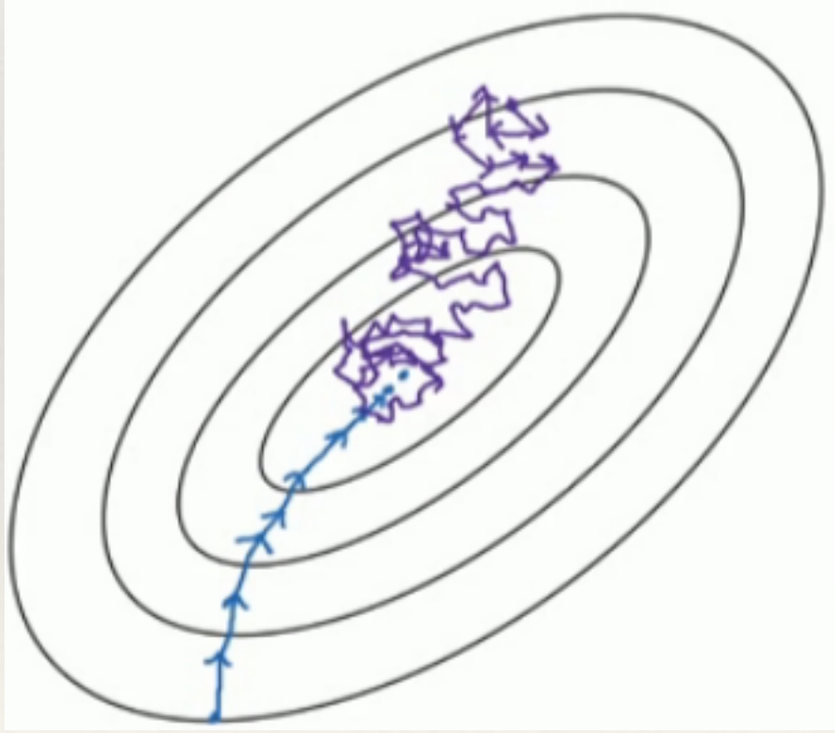
We choose a random k in 1, …, N and step in the direction of the negative gradient of

$$max(0, -y_k(\mathbf{a}\mathbf{x}_k + b))$$

Stochastic gradient descent

$$E[\mathbf{p}_k] = -\nabla S(\mathbf{u})$$

Finding a minimum this way is called **stochastic gradient descent** because rather than stepping along the gradient we are stepping in the random direction dictated by a random training example which is the gradient in expectation



Source: marsggbo@cnblogs.com

The gradient

Moving from the general back to our loss function

$$S(\mathbf{a}, b) = \frac{1}{N}\sum_{i=1}^{N} \max(0, -y_i(\mathbf{a}\mathbf{x}_i + b))$$

To get our update rule for having picked data item *k* randomly, we need the gradient for

$$S_k(\mathbf{u}) = \max(0, -y_k(\mathbf{a}\mathbf{x}_k + b))$$

Taking derivatives we get

$$\frac{\partial S_k}{\partial a_j} = \begin{cases} 0 & \gamma_k y_k > 0 \\ -y_k x_k^{(j)} & \text{otherwise} \end{cases} \quad \text{and} \quad \frac{\partial S_k}{\partial b} = \begin{cases} 0 & \gamma_k y_k > 0 \\ -y_k & \text{otherwise} \end{cases}$$

Update rule

Which produces the following algorithm for finding **a** and b for the perceptron

Choose some random values for **a** and b

Update **a** and b at the i-th step by

Drawing a random point from the training set (x_k, y_k)

Predict the label with the prior step's **a**, if it is correct do nothing, else update with

$$\mathbf{a}^{(i+1)} = \mathbf{a}^{(i+1)} + \eta y_k \mathbf{x} \qquad b^{(i+1)} = b^{(i)} + \eta y_k$$

The Naive Bayes Classifier

A probabilistic classifier

If we suppose there is a probabilistic relationship between our class labels and feature vectors, and we have determined what this relationship is, then the following is a valid classifier

For a feature vector **x**, evaluate $p(y \mid x)$ and output the class y that maximizes this function

Some classifiers attempt to learn or model $p(y \mid x)$ directly using max likelihood estimation (homework problem 9.17 actually did this for a classifier known as logistic regression). With the naive Bayes classifier we will take a slightly different approach

The naive Bayes model

Instead of modeling p(y | **x**) directly, we will use Bayes' theorem to observe that

$$p(y|\mathbf{x}) = \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})}$$

And come up with a model for p(**x** | y) and p(y)

The naive part about naive Bayes is that it makes an assumption that isn't usually true but we nevertheless might be able to get away with. We will assume that the features of **x** are conditionally independent given the class label, i.e. that

$$p(\mathbf{x}|y) = \prod_{j=1}^{d} p(x^{(j)}|y)$$

Naive Bayes prediction

Making the assumption that

$$p(\mathbf{x}|y) = \prod_{j=1}^{d} p(x^{(j)}|y)$$

And using Bayes theorem means that

$$\begin{aligned}
\arg\max_y p(y|\mathbf{x}) &= \arg\max_y \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})} \\
&= \arg\max_y p(\mathbf{x}|y)p(y) \\
&= \arg\max_y \prod_{j=1}^{d} p(x^{(j)}|y)p(y)
\end{aligned}$$

Training and modeling

$$\arg\max_y p(y|\mathbf{x}) = \arg\max_y \prod_{j=1}^{d} p(x^{(j)}|y)p(y)$$

In order to make a prediction, then, we are going to need a model of p(y) and p(x^j | y) for every dimension *j* and for each value of *y*

If feature (dimension) *j* is numerical, perhaps we model p(x^j | y) as a normal random variable. If it is a count perhaps we model p(x^j | y) as a Poisson random variable, etc.

The prior probability p(y) is probably best modeled as a Bernoulli random variable, in the binary classification case, where max likelihood would tell us that p(y=1) = fraction of training examples where y=1

Example

Suppose we had the following training data

1	x1	x2	y
2	3.4	10	1
3	1.1	9	1
4	0	10	-1
5	-2.1	13	-1

And we choose to suppose that p(x1 | y) is normally distributed, p(x2 | y) is Poisson, and p(y) is Bernoulli

Then p(x1 | y=1) has an MLE mean of 2.25 and standard deviation of 1.323
p(x1 | y=-1) has an MLE mean of -1.05 and standard deviation of 1.1025
p(x2 | y=1) has an MLE lambda of 9.5
p(x2 | y=-1) has an MLE lambda of 11.5
and p(y=1) = 0.5 by taking the MLE of a Bernoulli random variable

If we saw some new point (-1, 12), we would predict the class label by evaluating

$$\arg\max_y \prod_{j=1}^{d} p(x^{(j)}|y)p(y)$$

One caveat

If we have a large or even modest number of dimensions, we can run into numerical issues evaluating

$$\arg\max_y \prod_{j=1}^{d} p(x^{(j)}|y)p(y)$$

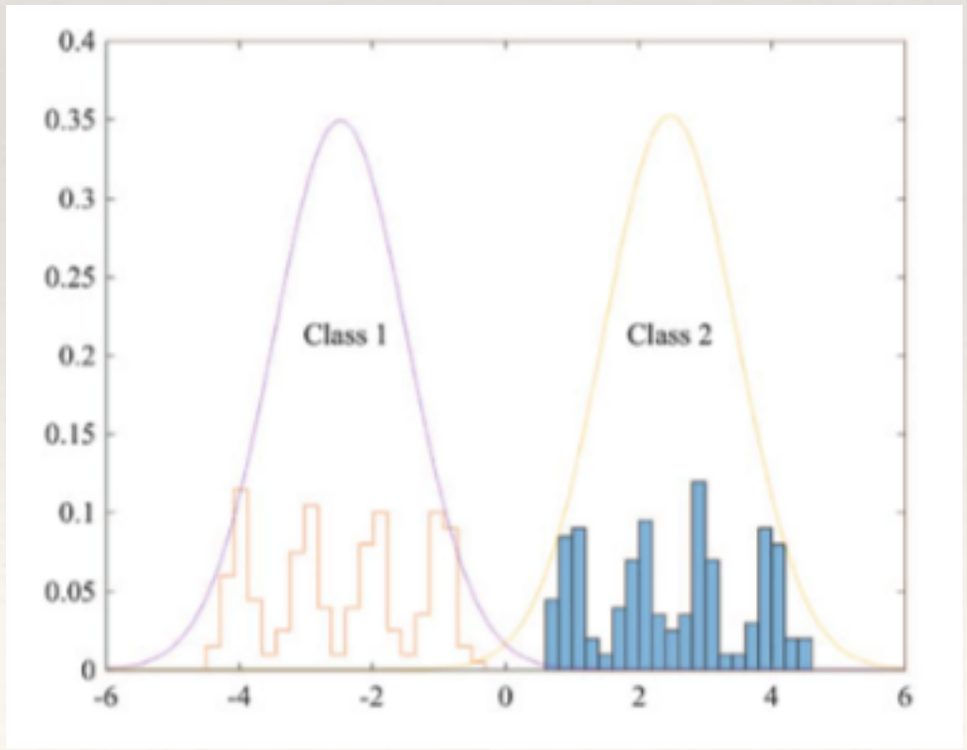
Each p(x^j | y) is some small number between 0 and 1, if we multiply many of these probabilities together, we are very likely to get a numerical underflow. To stop this from happening we often use logarithms since

$$\arg\max_y \prod_{j=1}^{d} p(x^{(j)}|y)p(y) = \arg\max_y \sum_{j=1}^{d} \log(p(x^{(j)}|y)) + \log(p(y))$$

Takeaway

By making a simplistic assumption we are able to get a probabilistic classifier that reduces to basic MLE calculations for training.

We can do this because even if we poorly model $p(y \mid x)$ or even $p(x \mid y)$, we can be okay as long as for any **x** the score for the right class is higher than the score for the wrong class and this can be the case even with poor assumptions



The x-axis is the value of some feature x and the y-axis is the proportion of the training data that has a value around x, i.e. it's a class conditional histogram

Here we assume that $p(x \mid y)$ is normal for some feature x, when the distribution clearly isn't actually normal, but we would still get the right predictions