Module 6 - Linear Classification

- 6.1 A Simple Linear Classifier, Incomplete
- 6.2 Support Vector Machines I, Incomplete
- 6.3 Support Vector Machines II, Incomplete
- 6.4 Duality
- 6.5 Multiclass Linear Prediction

总结:

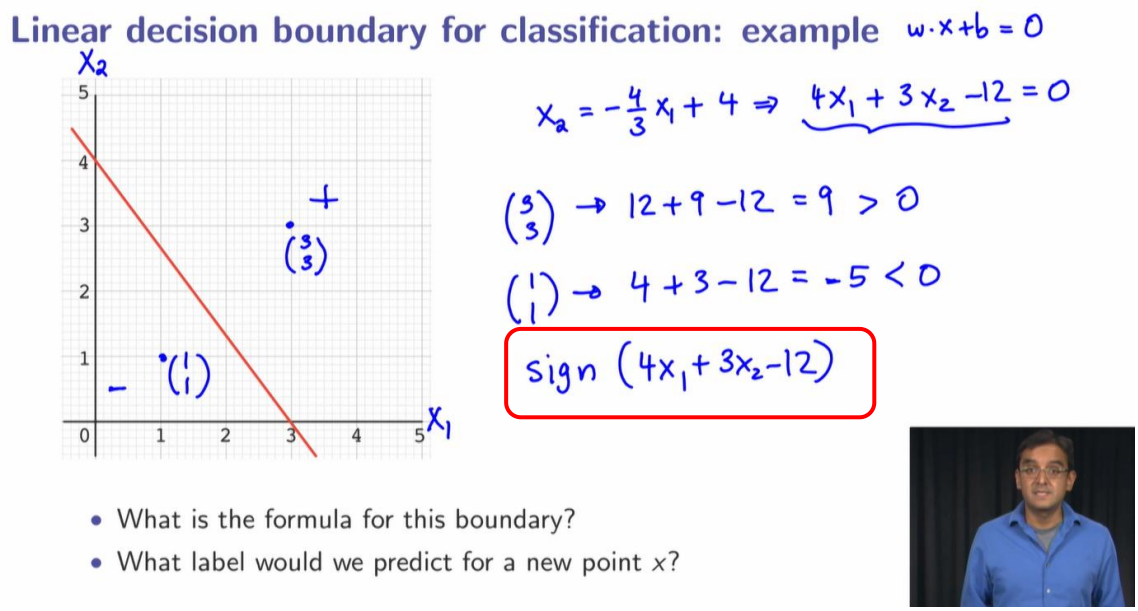
6.1 A Simple Linear Classifier, Incomplete

Topics we'll cover

1. Linear decision boundary for binary classification
2. A loss function for classification
3. The Perceptron algorithm

Recently, we've been talking a lot about regression and conditional probability estimation. The way we've been approaching these problems is to phrase the learning task as an optimization problem, defining a suitable loss function, and then looking at ways to minimize it. What we're going to do now is to circle back round to classification, bringing with us this new optimization-centric mindset.

We'll talk about binary classificationand how it might be solved using a linear separator. We'll define a suitable loss function for classification, and then look at a stochastic gradient descent procedure for it. This will yield the **Perceptron algorithm**.



Here is a linear boundary, in R2. What is the equation of this line? Let's go ahead and figure this out. We have, let's call this feature x one and this feature x two. The equation of this line is something of the form x two equals the slope times x one plus the intercept. So what is the slope over here? It's minus 4/3. So minus 4/3 x one plus the intercept on the y-axis, which is four, that's the equation of the line. Now, we'll be writing it in a slightly different way. So let's multiply through by three and then this becomes four x one plus three x two minus 12 equals zero. And we'll always be writing linear decision boundaries in this way, in the form w dot x plus b equals zero. The form w dot x plus b equal zero. That's the equation of this line and that's gonna be a decision boundary. Points on this side we'll say are plus, points on this side we'll say are minus. Now, how do we make a prediction on a new point? Let's say we get a new point x, maybe this point over here, three three. How do we predict the label at that point? Well, we plug it into our linear function. So we have this linear function over here. We plug in three three. What do we get? We get four times three, 12, plus nine minus 12, that's nine. That's greater than zero. So we classify it as positive. Let's take another point, maybe one one over here. Let's plug it into our linear function, four x one plus three x two minus 12, what do we get? We get four plus three minus 12. What is that? That is negative five. That's less than zero. So we classify that point as negative.

So the classification rule is very simple. When we get a new point, we simply compute our linear function, which is four x one plus three x two minus 12, and then we look at the sign of that. So we take this and we look at its sign. If it's a positive number, we say the class is plus one, if it's a negative number, we say the class is minus one.

总结:

Linear classifiers

Binary classification problem: data $x \in \mathbb{R}^d$ and labels $y \in \{-1, +1\}$

- Linear classifier:
  - Parameters: $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$
  - Decision boundary $w \cdot x + b = 0$
  - On point x, predict label $\text{sign}(w \cdot x + b)$
- If the true label on point x is y: (x, y)
  - Classifier correct if $y(w \cdot x + b) > 0$

Correct $\iff$ y has same sign as $w \cdot x + b$

$\iff$ either $y = +1$ and $w \cdot x + b > 0$

or $y = -1$ and $w \cdot x + b < 0$

$\iff$ $y(w \cdot x + b) > 0$

In a little bit more generality, if we have data now in d dimensional space, so points x and R d, and two possible labels, plus one and minus one, this is a general setting for binary classification. We're gonna be looking at situations where we have a linear classifier. So there's a decision boundary which is linear, it's of this form, w dot x plus b equals zero. And here, w and b are parameters. They're the parameters of the linear function that we have to figure out, typically from training data. So these are our parameters over here. When we get a new point x and we have to predict its label, plus one or minus one, the way we do it is, we compute this linear function and then we just return its sign. We just see whether it's positive or negative. And that's the answer. So when are we correct? Let's say we get a new point x y. When are we correct on that point? We're correct if the label of y, plus one or minus one, has the same sign as w dot x plus b. So y has the same sign as w dot x plus b. To spell it out even more, what that means is that either y is plus one and w dot x plus b is greater than zero, let's say greater than zero, or y is minus one and w dot x plus b is less than zero. If that happens, then we're correct.
And now we have two conditions over there, we can combine them into a single, more concise condition, like this. We can just say that y times w dot x plus b is greater than zero. Either they're both greater than zero or they're both less than zero. So this is a nice, concise way of writing the condition that we are correct on point x y. We'll be using this formulation quite a bit more.

A loss function for classification

What is the **loss** of our linear classifier (given by w, b) on a point (x, y)?

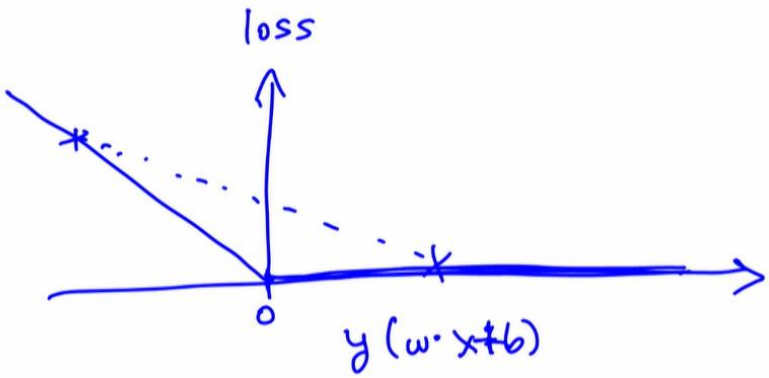
One idea for a loss function:

- If $y(w \cdot x + b) > 0$: correct, no loss
- If $y(w \cdot x + b) < 0$: loss $= -y(w \cdot x + b)$

$y = +1$

$w \cdot x + b = -0.1$

$w \cdot x + b = -6$



总结:

Now let's come up with a loss function for classification. We have some linear classifier given by w b and we want to know what is the loss incurred by this classifier on a point x y? How should we define this loss? Should we use squared loss or logistic loss? What's a suitable loss function? There are actually many ways to do this. There are many different loss functions for classification. They all have their strengths and weaknesses. And we'll just look at one particular plausible loss function right now. We have a point x y. Let's say we are correct on that point. So y times w dot x plus b is greater than zero. If we're correct, we'll just say no loss. We are correct. Nothing to worry about. On the other hand, if we are wrong, if y times w dot x plus b is less than zero, if y and w dot x plus b have different signs, then we are wrong. How much penalty should we incur? Well, one thing we can do is to say that our penalty is the amount by which we are wrong. What does that mean? Let's say that the correct label is plus one. And when we compute w dot x plus b it turns out to be negative .1. So we are wrong because we predict minus. But we're only wrong by a little bit, only by .1. So we shouldn't feel too bad about it. In that case, we'll think about a loss as being just .1. On the other hand, if y equals plus one and w dot x plus b equals minus six, then we're pretty far wrong. We also predict minus. But this time we're really far from that boundary. We're at minus six. So in this situation, we'll say our loss is six. Here is sort of a picture of the loss function. Over here I'll draw the possible values of y times w dot x plus b. And we want this to be positive. So we'd like this to be on this side of zero.
Now let's look at the loss. I'll draw that on the y-axis. If we classify the point correctly, then our loss is zero. Then we lie along this line. No loss at all, we're correct. If we are wrong, then our loss looks like this. Now, one notable feature of this loss function is that it's convex. Why is that?Because if you look at any two points on the curve and you connect them, the graph, the curve lies below that line. So this is a convex loss function. Okay, that's something that should in general make us feel good.

A simple learning algorithm

Fit a linear classifier w, b to the training set using **stochastic gradient descent**.

- Update w, b based on just one data point (x, y) at a time
- If $y(w \cdot x + b) > 0$: zero loss, no update
- If $y(w \cdot x + b) \leq 0$: loss is $-y(w \cdot x + b)$

$$\frac{d}{dw} = -yx$$

$$\frac{d}{db} = -y$$

$$w \leftarrow w + yx$$

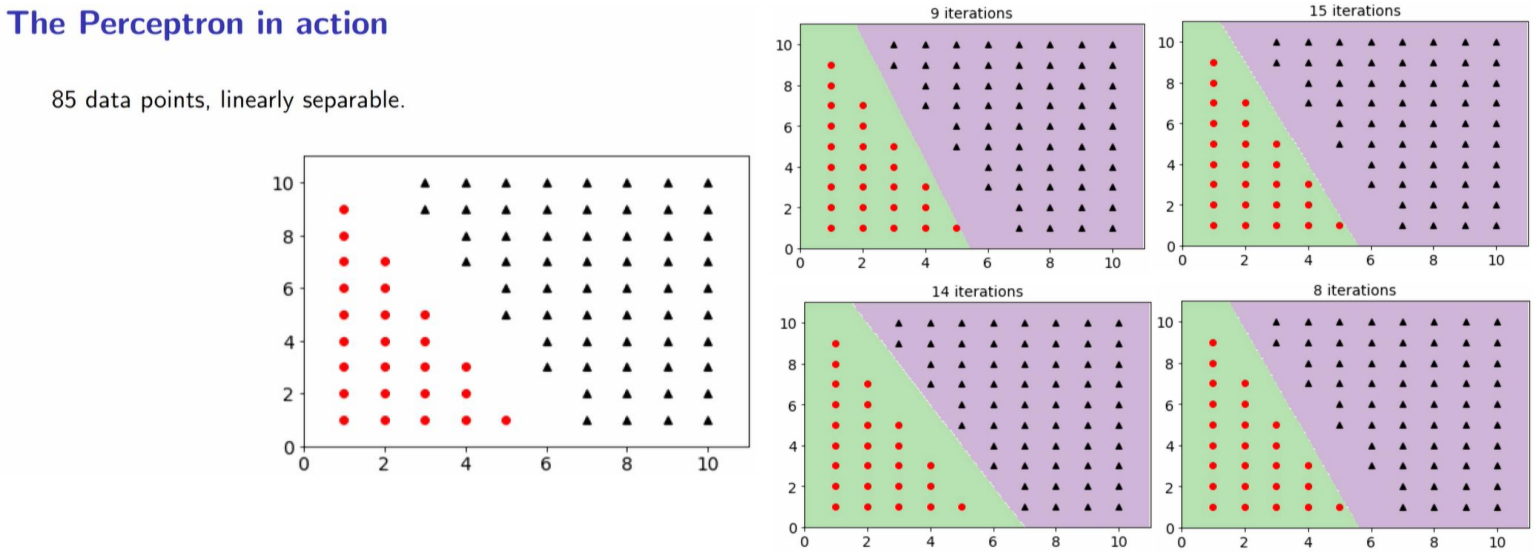
$$b \leftarrow b + y$$

The Perceptron algorithm

- Initialize $w = 0$ and $b = 0$
- Keep cycling through the training data (x, y):
  - If $y(w \cdot x + b) \leq 0$ (i.e. point misclassified):
    - $w = w + yx$
    - $b = b + y$

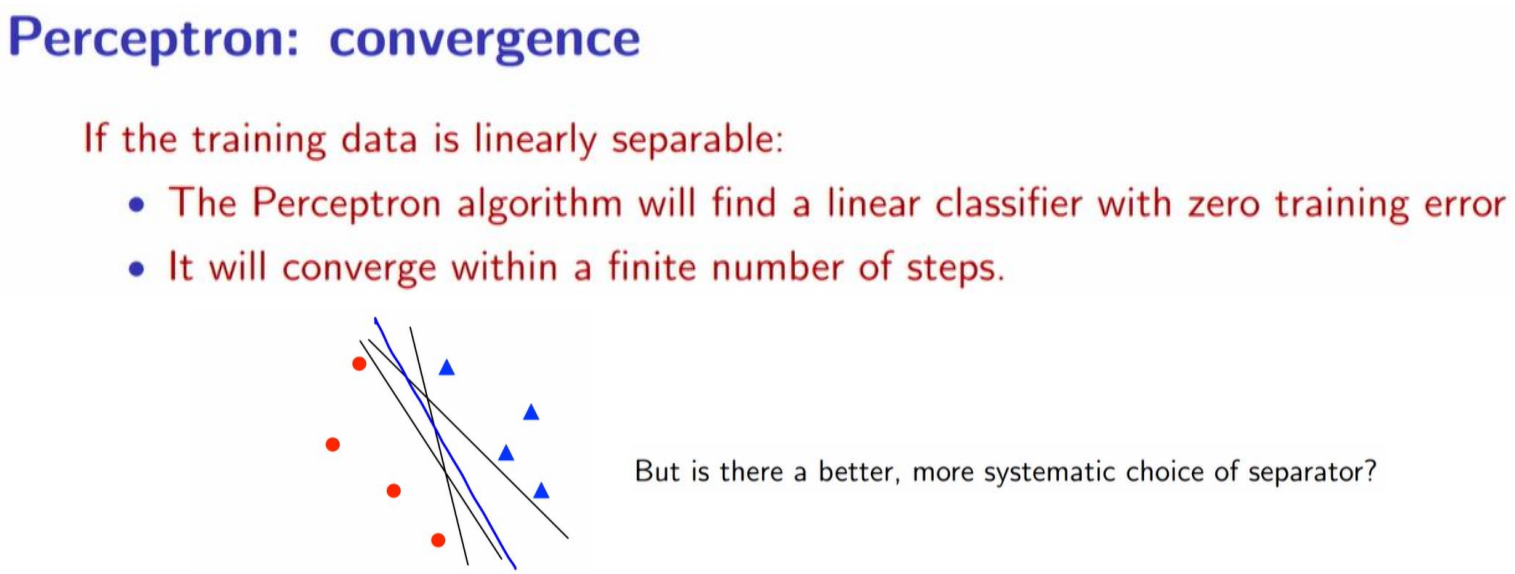
Now let's go about trying to find a linear classifierthat is minimizing this loss. We could do that by gradient descent if we wanted to. But to keep things really simple, let's just use stochastic gradient descent. The way this is gonna work, this is our local search method. So we start by setting w and b to any old values. Let's say we start them at zero. Then we keep updating them. So we update w and b a little bit and then we update it some more and we keep going in this way. And in stochastic gradient descent, each update is just based on a single data point. So we first update on the first data point, then we update on the second data point, and we keep going through the dataset. When we reach the bottom, we cycle back round to the top and then update on the first point, second point, and so on. What are these updates? The update is to take w and b and then move in the negative direction of the derivative. Okay, so let's work that out. We have our current guesses, w and b. New data point arrives, the next on the list. That's some point x and y. What's our update gonna be? Well, let's say we actually get x and y correct. So let's say y times w dot x plus b is greater than zero. In that case, the loss is zero and the derivative is zero, so there's not update. That's easy. What if we get it wrong? What if w times x plus b has a different sign from y? In that case, the loss that we've formulated is minus y times w dot x plus b. What is the derivative of this thing? Let's see, what is the derivative of this with respect to w? Well, it's just minus y x. And what's the derivative of this with respect to b? It is just minus y. So the stochastic gradient update in this case would be to take w and move in the negative direction of the derivative. So we take w and we go plus y x, and we have a step size, eta. We take b and we also move in the negative direction, so step size times y.
That's the stochastic gradient update. Very simple. Well, there's the matter of the step size. We have to decide what to set eta to.Let's just keep things extra simple. Let's just make it equal to one so we can get rid of this. That's gonna be our update. When we do things in this way, we get the **Perceptron algorithm**.
Here it is. Very simple algorithm. We start w and b at zero. Then we just keep cycling through the data. If we're correct on the next data point, then nothing to do. If we are wrong on the data point, then we simply take w and add y x to it and we take b and add y to it. That's it. This is literally four lines of code.

总结:



Let's see it at work. Here are 85 data points, and they're linearly separable. So there really, it's a line that separates the pluses from the minuses, or in this case, the red circles from the black triangles. Let's run the Perceptron algorithm on this dataset and see what linear separator it finds. Now, usually when you run Perceptron or a stochastic gradient descent algorithm of some kind, one thing that is often done is to begin by randomly permuting the data, in case they're in some sort of weird order. Now, in the case of Perceptron, different random permutations can lead to different outcomes. Let's just look at several runs.

This is the first run.It found a linear separator that perfectly classifies the data. It made nine updates. What I mean by that is that, if we go back to the algorithm over here, this is what I'm calling an update. When it gets to a point on which it's wrong, it updates the parameters. In this case, it only did nine updates. Once it did those nine updates, it cycled through the dataset and it was correct on everything it saw. So it stopped at that point. Here's another iteration. This time it made 15 updates. Another one, this one did eight updates. Another one, 14 updates. Another one. This one was eight updates. Here's one question. In this case, it did eight updates. What do you think the final value of b was in this case? Let's go back to the algorithm. b starts out at 0. And each time you make a mistake, you change b by adding y to it. Now, y is either +1 or -1. Each time you do an update, b either goes up by one or goes down by one. That means that if you make eight updates total, the final value of b is an integer which is somewhere between -8 and +8. That's something that we can infer just from the fact that there were eight updates.



So we had these multiple different runs,and every single time, it found a perfect classifier. In fact, this wasn't just a lucky dataset. The Perceptron algorithm is guaranteed to do this. So there's a mathematical theorem that says that if the data is linearly separable, in other words if there is some linear boundary that separates the pluses from the minuses, then the Perceptron algorithm will find such a boundary. It's guaranteed to. And it will converge within a finite number of steps. Not bad. A four-line algorithm that is guaranteed to always return the correct answer.
Now, if we have a dataset like this, for instance.So here we have eight points. What this theorem is telling us is that the final answer is gonna be some separator like the one shown. Something that really does separate the reds from the blues. Now, one thing one might think at this point here, another way now that we have this under control, maybe we can get a little bit greedy and say, "Well, maybe these are not all equally good. "Can we get something that's more central? "Maybe something that's right in the middle?" And that's exactly the question that we'll tackle next time.
For now, that's it. We have seen the Perceptron algorithm and we derived it by formulating a loss function and looking at the stochastic gradient descent algorithm for that loss function. And the result is a four-line algorithm that any of us could code up in five or 10 minutes. It's one of the real gems of machine learning. See you next time.

总结:

6.2 Support Vector Machines I

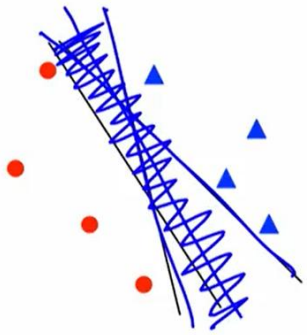
Topics we'll cover

1. The margin of a linear classifier
2. Maximizing the margin
3. A convex optimization problem
4. Support vectors

Today, we'll study a fantasticallycompetent linear classifier. It is called the support vector machine. So we'll begin about talk about the margin of a linear classifier, and then we'll formulate the problem of finding the linear classifier that has the largest margin possible. This will turn out to be a convex optimization problem, which means that it's actually pretty easy to solve. And it'll turn out that the solution has a unique property. It only depends on a view of the training points, which we call support vectors. And this is why the method overall is called a **support vector machine**.

Improving upon the Perceptron

For a linearly separable data set, there are in general many possible separating hyperplanes, and Perceptron is guaranteed to find one of them.



Is there a better, more systematic choice of separator?
The one with the most buffer around it, for instance?

When we were talking about the Perceptron algorithm, we mentioned that there is this convergence theorem which says that if a data set is linearly separable, then the Perceptron algorithm, those four lines of code, are guaranteed to find a linear classifier that perfectly separates the training set. So in this picture, for example, it'll return something that could be one of these lines, or maybe this one, or maybe this one, something like that. But what if we want the linear classifier that is the most central? Maybe something like this. The classifier that has the most buffer around it, for example. How would we even formulate that? What does that mean? So let's take a closer look.

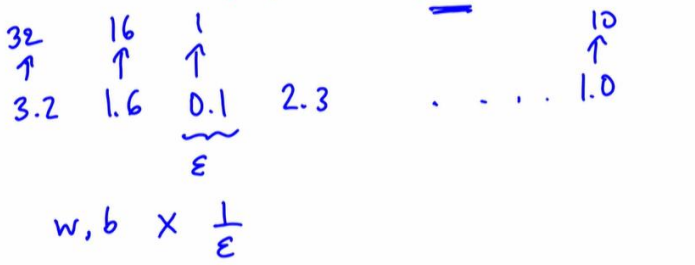
The learning problem

Given: training data $(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)}) \in \mathbb{R}^d \times \{-1, +1\}$.

Find: $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$ such that $y^{(i)}(w \cdot x^{(i)} + b) > 0$ for all i.

By scaling w, b, can equivalently ask for

$$y^{(i)}(w \cdot x^{(i)} + b) \geq 1 \quad \text{for all } i$$



总结:

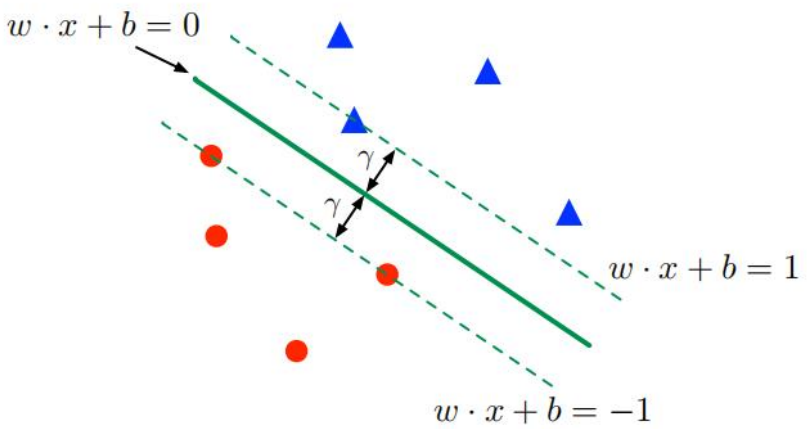
So the setting is that we havetraining data, say n points, x one through xn. These are some d dimensional vectors. And we're looking at binary classification. So each point has its label y, which is either plus one or minus one. Now, we want to find a linear classifier. So we want to find w and b that perfectly classifies this data. Okay, so if y is plus one, w dot x plus b should be positive. If y is minus one, w dot x plus b should be negative. And last time we saw that a concise way to write this is to simply say that y times w dot x plus b should be greater than zero. Either they're both greater than zero or they're both less than zero. Okay, and we want this condition to hold for every one of the n training points.

Now, in order to start talking about margins, what we're going to do is to take this condition and reformulate it just a tiny bit. We'll write it like this. Instead of asking that y times y dot x plus b be greater than zero, we'll ask that it be greater than or equal to one. Okay, why can we do this? It seems like a stronger condition. Certainly if y times w dot x plus bis greater than or equal to one, then it's greater than zero. So it certainly implies that. But it seems like it's something more. It seems like it's requiring more. And it's actually not. If you can solve this, if you can find w and b that solve that, you can also find w and b that solve this. And let me explain why. So let's say you manage to find a w and b such that for all the training points, y times w dot x plus b is greater than zero. So let's look at what these values actually are. Maybe for the first training point is 3.2. Something greater than zero. Maybe for the next training point it's 1.6. Maybe for the next one it's 0.1. Maybe for the next one it's 2.3. And all the way to the nth point maybe it's 1.0. So we have these n different numbers. And they're all greater than zero. Let's look at the smallest of them. Let's say it's this one and let's call it epsilon. Now, if we take w and b and we multiply them by one over epsilon, we just scale them up by a factor of one over epsilon, then w dot x plus b also gets magnified by one over epsilon. In this case, by 10. So the first number will become 32. And the next number will become 16. And the next number will become one. And the last one will become 10. And in this way, all the numbers become greater than or equal to one.
So if you can find w and b such that y times w times x plus b is greater than zero, then by simply scaling up w and b, you can get the second condition as well. You can make all of these values be greater than or equal to one. So this is the condition we'll be working with. This will be our linear separability condition. Okay, that makes sense, but why did we actually do this? How did this help us to rewrite our condition in this way? Well, it will turn out that if we write our separability condition in this way, it's very easy to come up with a nice formula for the margin. And let's see what that is.

Maximizing the margin

Given: training data $(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)}) \in \mathbb{R}^d \times \{-1, +1\}$.
Find: $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$ such that

$$y^{(i)}(w \cdot x^{(i)} + b) \geq 1 \quad \text{for all } i.$$



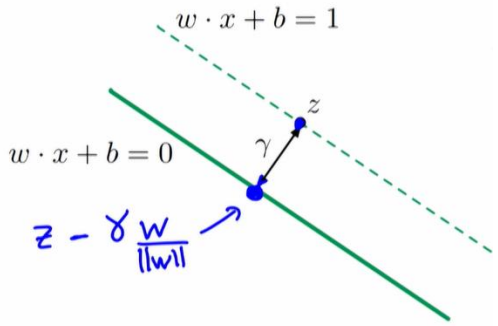
Maximize the **margin** γ.

So now we're gonna have some decision boundary,w dot x plus b equals zero. And that's the thick green line in the middle. What is w dot x plus b equal to one? That is a parallel hyperplane. And it's shown by the dotted line over there. And what's w dot x plus b equals minus one? That's another parallel hyperplane on the other side. So, in this condition over here, in our separability condition, what we're insisting is that all the positive points, in this case, the blue triangles, have to be to the right of the right hyperplane. And all the negative points, in this case the red circles, have to be to the left of the left hyperplane. Okay, so not only do they have to be on the correct side of the boundary, but also there should not be any points in that middle buffer zone. And the margin is the width of the buffer zone for which I'm using the great letter gamma. That's the margin. And that's the thing we want to maximize.

总结:

A formula for the margin

Close-up of a point z on the positive boundary.



$w \cdot z + b = 1$

$w \cdot \left(z - \gamma \frac{w}{\|w\|}\right) + b = 0$

A quick calculation shows that $\gamma = 1/\|w\|$.

In short: to maximize the margin, minimize $\|w\|$.

Now, it turns out that there's a simple formulafor the margin. The margin is simply one over the length of w. Why is that? Well, you can just take my word for it. It's just a simple geometric argument. You don't need to know what it is. In case you're curious, I'll just give you a quick hint. So, let's look at our right hyperplane and look at any point z on it. So the point lies on the hyperplane. So we know w dot z plus b equals one. Now, we also know that if we move gamma units in this direction, we land on the decision boundary. What is this point over here? What is this point gamma units in that direction? Well, it's z minus gamma in the direction of w. Okay. Actually, w might not be a unit vector. So we have to normalize it to make it a unit vector. And if we do that, then we end up with that point. Now, since that point lies on the decision boundary, we know that w dot that point plus b equals zero. And when we solve these two equations, we get our formula for the margin.

Okay. At any rate, we now know what the margin is. It's one over the length of w. Since we want to maximize the margin, what we need to do is to simply minimize the length of w. This is the optimization problem we want to solve.

Maximum-margin linear classifier

- Given $(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)}) \in \mathbb{R}^d \times \{-1, +1\}$

$$\min_{w \in \mathbb{R}^d, b \in \mathbb{R}} \|w\|^2$$

$$\text{s.t.: } y^{(i)}(w \cdot x^{(i)} + b) \geq 1 \quad \text{for all } i = 1, 2, \ldots, n$$

- This is a **convex optimization problem**:
  - Convex objective function
  - Linear constraints

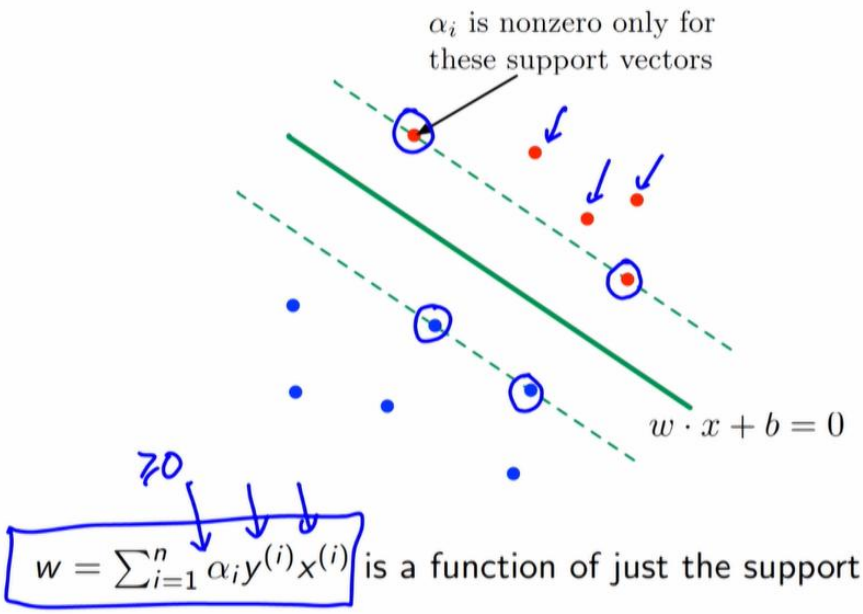
- This means that:
  - the optimal solution can be found efficiently
  - **duality** gives us information about the solution

We have our n data points.We want to perfectly classify all of them. That's this condition over here. And while perfectly classifying those points, we also want to maximize the margin. So we want to minimize the length of w or equivalently minimize the length of w squared. So this is an optimization problem that we want to solve in which the things we are solving for are w and b. So how do we solve this problem? Well, the nice thing is that this turns out to be a convex optimization problem. Now, soon, maybe in a couple of lectures, we'll define exactly what this means. But for the time being, the reason it's a convex optimization problem is the thing we're minimizing is convex. We know the length of w squared is convex. That's one of the examples we saw. And all the constraints are linear functions of w and b. So when you have a convex thing that you are minimizing all the constraints are linear, that's a convex optimization problem. And it means that it's pretty easy to solve. You can just hand this over to some standard package and it'll give you back the answer. Now, there's a second benefit of convex optimization, which is not only can you solve the problem efficiently, but you can use the theory of duality to also tell you a little bit about what the answer looks like. And this also, I'm sure, sounds somewhat mysterious. And it's also something we'll talk a little bit more about later on.

总结：

Support vectors

Support vectors: training points right on the margin, i.e. $y^{(i)}(w \cdot x^{(i)} + b) = 1$.



But let me tell you what duality theory tells us.It tells us that the solution is of a particular form. The solution looks like this. It is the sum over all data points of the data point times its label, which is plus one or minus one, times some coefficient, which is greater than or equal to zero. So for each data point, there's some coefficient alphas of i, and the solution w, the optimal w, is just a linear function of this form. Alpha i times the label plus one minus one times xi. Now, the remarkable thing is that most of these alpha is may well be zero. The only alpha is that are going to be nonzero are for the point, the data points that lie exactly on the margin. In this data set, there's just four of them. These are called the support vectors. For all the other points, the coefficients alpha i will be zero. So in this case, in this example, w will be a function just of the four support vectors, the four data points that I've circled. Okay, so you could have a data set of a million points. But if there's just 100 of them that lie on those margins, then the final answer is a function just of those 100 points.

Small example: Iris data set

Fisher's **iris** data

150 data points from three classes:

- iris setosa
- iris versicolor
- iris virginica

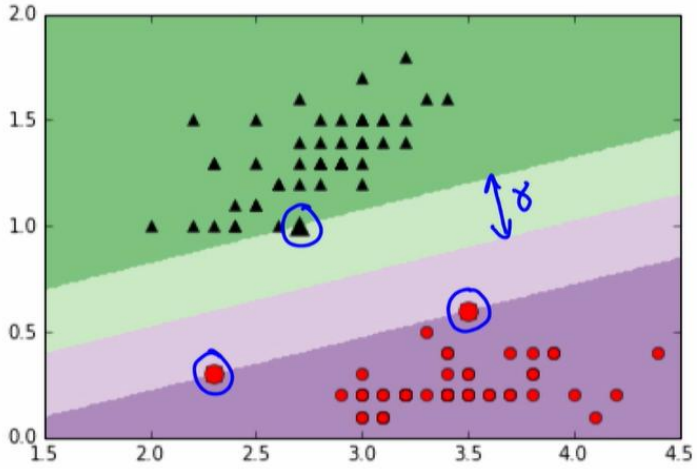
Four measurements: petal width/length, sepal width/length

Let's see a quick example of the svm at work.And this is a very famous data set called the iris data. It consists of measurements of 150 flowers which are all irises of three different types. There's iris setosa, iris versicolor, and iris virginica. There's 50 of each of them. So 150 points total. And for each flower, four measurements were taken. The pedal length and width. And the sepal length and width. And the goal is to take these four measurements and predict which of the three types of iris it comes from.

总结:

Small example: Iris data set

Two features: sepal width, petal width.
Two classes: setosa (red circles), versicolor (black triangles)



So here's a plot of the data. Since we're doing binary classification, I just pulled out two of the three classes. So I pulled out setosa and versicolor. The setosa are shown by red circles. Versicolor are shown by black triangles. These are the two classes. And for visualization purposes, although there are four features, I just pulled out two of them. Okay, so we'll just use two of the features. The widths of the sepals and pedals. So you can see the data set. Is it linearly separable? It certainly is. So if we were to use the Perceptron algorithm, it would indeed find a classifier that was perfect on the training set. But let's see what the svm does. This is what it returns. A very nice linear classifier, the one right in the middle. So this linear classifier depends on just these three training points. It's a function of just those three support vectors. And the margin is this distance over here, this gamma. And this is the largest margin possible on this data set. So a very competent linear classifier indeed.

Okay, well, that's it for today.We've got our first glimpse of the support vector machine. And what we did today was to focus on the case where the data is linearly separable. Now, in practice, this is usually not the case, or at least fairly often not the case. So what we'll do next time is to look at the case of arbitrary data. And it'll turn out that this same methodology extends very easily to that more general setting. So, see you next time.

POLL
Which is not a condition for a SVM solution?

结果

Each training set data point is classified correctly	16%
The margin, γ, is maximized	3%
Each new data point is classified correctly	74%
The training set data points are linearly separable	7%

总结:

6.3 Support Vector Machines II

Topics we'll cover

1. Data that isn't linearly separable
2. Adding slack variables for each point
3. Revised convex optimization problem
4. Setting the slack parameter

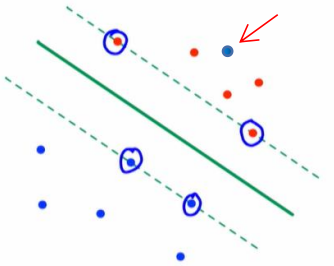
Last time,we introduced the support vector machine. Given a training set, this finds the linear classifier with the widest margin by solving a convex program. It works beautifully, but it's only for data that is linearly separable. That is, data for which there truly is a linear classifier that perfectly separates the plusses from the minuses. In practice, this is often not the case. So what we'll do today is to look at another version of the support vector machine called the **soft-margin SVM**. This can deal with any type of data.

So, we'll see how situation of non-separable datacan be handled quite easily by introducing new slack variables into the convex program. We'll then look at how this works in practice and at some of the issues that arise.

Recall: maximum-margin linear classifier

Given: $(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)}) \in \mathbb{R}^d \times \{-1, +1\}$.
Find: the linear separator w that perfectly classifies the data and has maximum margin.

$$\min_{w \in \mathbb{R}^d, b \in \mathbb{R}} \|w\|^2$$
$$\text{s.t.: } y^{(i)}(w \cdot x^{(i)} + b) \geq 1 \quad \text{for all } i = 1, 2, \ldots, n$$



Solution $w = \sum_{i=1}^{n} \alpha_i y^{(i)} x^{(i)}$ is a function of just the support vectors.

What if data is not separable?

So here is the situation from last time.We have a data set, a training data set of n points, x one through x n, and each point has a label, y one through y n. These are plus one or minus one. It's a binary classification task. The support vector machine finds a linear classifier by solving a convex program. This program over here. So it looks for a linear classifier given by a vector w, and an offset b. And this program, this convex program contains n linear constraints, these constraints over here, one for each data point. What the constraint says is that that data point has to be correctly classified. Now, subject to these constraints, the program additionally wants to find the classifier that has the widest margin. And as we saw, this is the same as minimizing the length of w. Or equivalently, the square length of w. Now, this is a convex program, which means that it can be solved efficiently. And there are packages that will just do this for us. We don't have to worry about the algorithmics behind it. Moreover, the solution, the w that arises in the end, is of a very nice form. It's a linear combination of the data points, of the x i's, and moreover, this linear combination only uses some of the data points. The ones that happen to lie exactly on the margin. These are the support vectors. So it's a very nice picture. This is all very well. But this only works if data is linearly separable, and the question is, what happens in the non-separable case?

For instance, suppose there were a blue point over here (见图中红色箭头). If we had that, then there would be no linear classifier that could perfectly classify the craning data, so we would not be able to satisfy all of these constraints. We'd have to give up on some of them. How do we do that?

总结:

The non-separable case

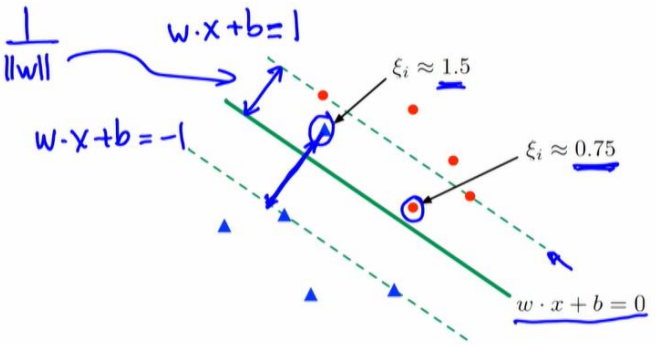
$y^{(i)}(w \cdot x^{(i)} + b) \geq 1$

Idea: allow each data point $x^{(i)}$ some **slack** ξ_i.

$$\min_{w \in \mathbb{R}^d, b \in \mathbb{R}, \xi \in \mathbb{R}^n} \|w\|^2 + C \sum_{i=1}^{n} \xi_i$$

s.t.: $y^{(i)}(w \cdot x^{(i)} + b) \geq 1 - \xi_i$ for all $i = 1, 2, \ldots, n$

$\xi \geq 0$



$y^{(i)}(w \cdot x^{(i)} + b) \geq 1$

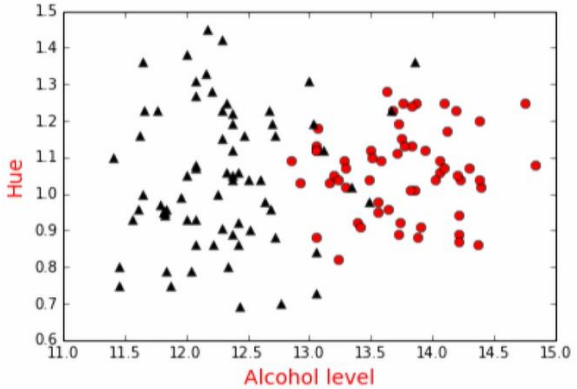
0.25

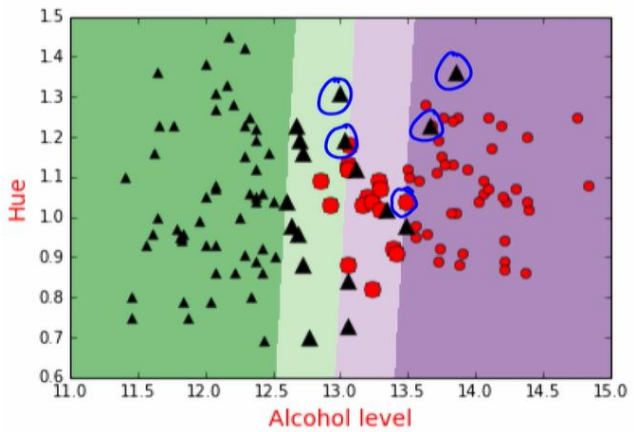
$\sum_{i=1}^{n} \xi_i = 2.25$

It turns out that a very simple way of doing thisis to introduce new **slack variables**. And let me explain what that means.
So previously, for each data point i, we insisted that y i times w dot x i times plus b be greater than or equal to one. That was a hard constraint. Now, we are in general not going to be able to satisfy all n of these constraints. So what we'll do is we'll allow some of these constraints to be violated. And we will introduce a new slack variable, xis of i, this is the Greek letter xi. We'll introduce these variables, one per data point, and then capture the extent of the violation. Okay? So for data point i, we would ideally like y times w dot x plus b to be greater than or equal to one. But we'll allow it to be violated, and xis of i tells us how large of a violation we have on that particular point. Now, the thing we want to minimize is the same thing as before, w squared, because we want to widen the margin. But at the same time, we want to make sure that too many constraints aren't violated. So we also add in the sum of the xi i's. The overall extent to which the constraints are violated. So to understand these slack variables a little bit more, let's look at this picture. So here we have a decision binary, w dot x plus b equals zero. The margin on this side is w dot x plus b equals one. And the parallel hyperplane on the other side is w dot x plus b equals minus one. Okay. The margin of this case is this distance over here. And as we saw before, this distance is just one over the length of w. So that's all the same as last time. But what's different now is that the points aren't all exactly where we would like them to be. Okay? Look at this point for instance. It's on the correct side of the boundary, it's on the red side of the boundary. But we want it to be to the right of this margin. And it's not. It falls a little bit short. For that particular point, we would ideally like w dot x i plus b to be greater than or equal to one. But it's not. It's actually equal to something like .25.
So we make up that difference using the slack variable.The slack variable on that particular point is therefore .75. The difference between the actual value of y times w dot x plus b, and what we would like it to be. Now let's look at this point. This point is actually on the wrong side of the boundary. It's on the red side of the boundary. So it's very far from where we want it to be. We want it to be all the way there. So what is the slack variable on that? Well, there's a slack of one just to get to the boundary, and an additional slack of about point five. So the slack variable on that point will be something like 1.5. Now in this picture, those are the only two data points on which any slack at all is used. The other ones are just fine. They're where we want them to be.
And so the total amount of slack that we've used, which is the sum of these xi i's from i equals one to n, is in this picture, just .75 plus 1.5. So 2.25. So let's see this at work.

Wine data set

Here $C = 1.0$





总结:

So this is the winery data set from a few weeks ago. If you don't remember it, that's fine. All that matters here is that it's a two-dimensional data set. It's got two classes, over here shown as red and black. And it's not linearly separable. Okay, there's no linear separator that will separate the reds and blacks. So let's see what the soft-margin support vector machine does on it. Here's the result. Not too bad, okay? It's found a reasonable boundary. The points that I've shown in bold are the support vectors. They now include any point that's actually on the margin, like that point, as well as any point that's not on the right side of the margin. So that includes all the points that are in the middle as well as these points over here that are far out. So in this picture, there are quite a lot of support vectors. But it seems to be working fairly well. Now, over here, I've said that c equals 1.0. What is that? What is this c over here?

The tradeoff between margin and slack

$$\min_{w\in\mathbb{R}^d, b\in\mathbb{R}, \xi\in\mathbb{R}^n} \quad \|w\|^2 + C\sum_{i=1}^{n}\xi_i$$

$$\text{s.t.:} \quad y^{(i)}(w\cdot x^{(i)} + b) \geq 1 - \xi_i \quad \text{for all } i = 1, 2, \ldots, n$$

$$\xi \geq 0$$

$C = 0$

slack is free!

$w = 0$

$C \to \infty$

slack is infinitely expensive

hard-margin SVM

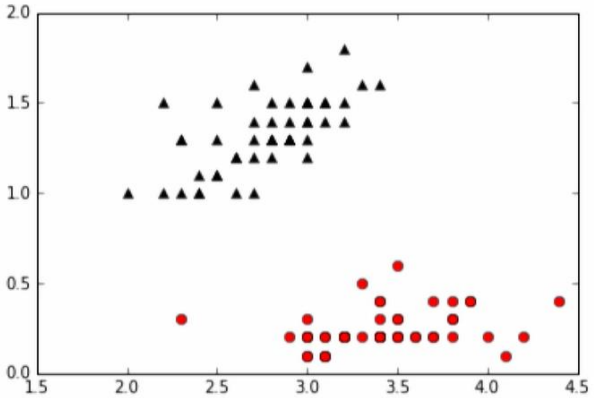
Well, let's go back and look at the convex program. Oh, here is the c that I'm referring to. What is that thing? Well, there are two things we're trying to minimize. We're trying to minimize the length of w, because we want a large margin. And we're also trying to minimize the total amount of slack. The sum of the xi i's. This is because we don't want too many constraints to be violated. So there're two things we're trying to minimize, and there will inevitably be a tradeoff between these two things. So c is a constant that manages that tradeoff. It's something that we have to set appropriately.

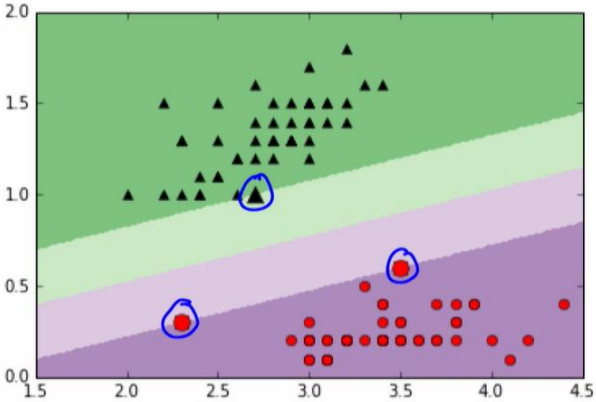
So to understand the effect of c, let's look at the two extremes. Let's look at when c is as small as possible, when c is zero, and the other extreme, when c is really large. When c goes to infinity. And let's see what happens in each case.

- So what happens when c is zero? When c is zero, this second term basically vanishes. It says that slack is free. We can use as much slack as we like, there's no charge for it. We can go ahead and violate as many constraints as we like. We can make those xi i's as large as we like. It's all free. The only thing we want to do is to minimize the length of w. So what are we going to do? We're going to set w to zero.
- Now, let's look at the other extreme. If c equals infinity, that means that slack is infinitely expensive. If we use even a tiny amount of slack, the cost for it is infinity. So we're not going to use any slack at all, which gets us back to the previous hard margin SVM. We're going to have to classify every point perfectly. No slack allowed. Now, of course, we're going to pick some number between these two extremes, and let's see a small example of how this pans out.

Back to Iris

$C = 10$

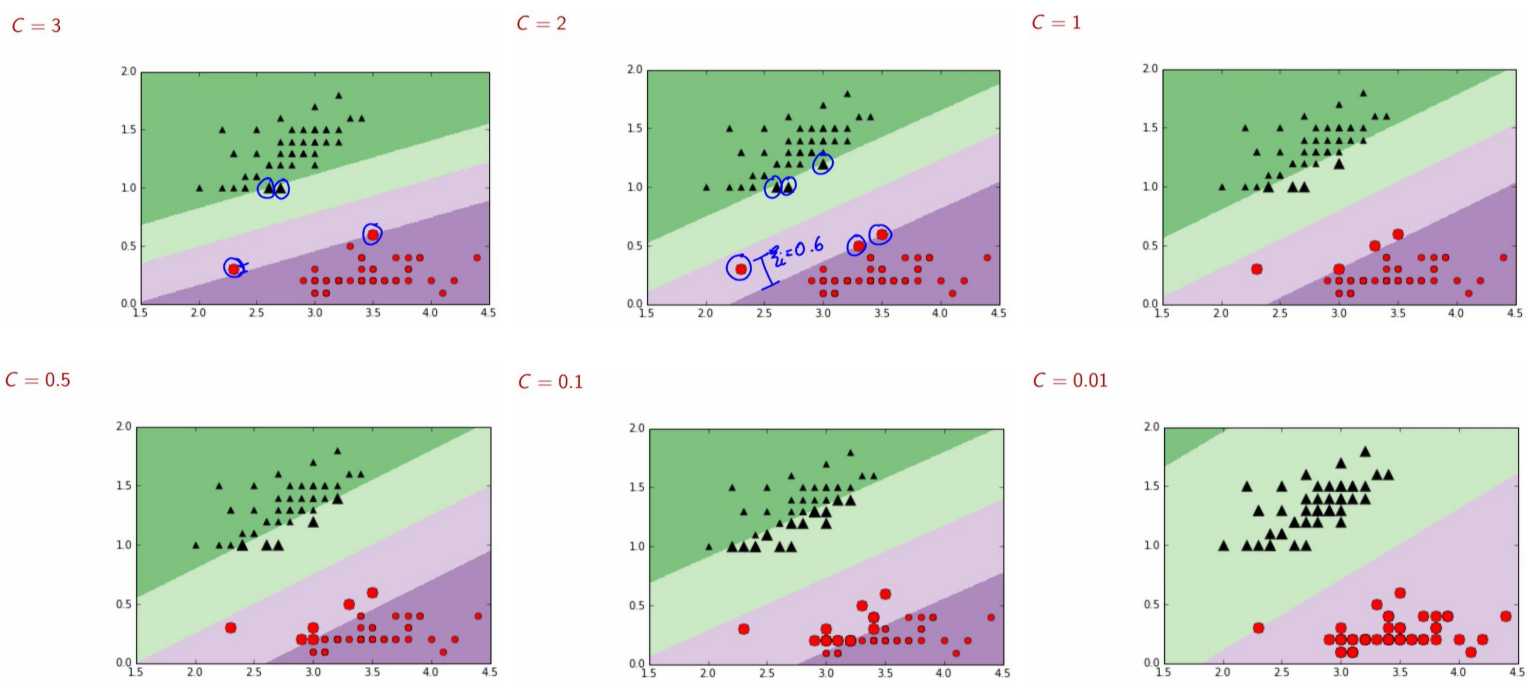




总结:

So this is the Iris data set from last time, and over here I'm just showing two different classes, two of those three types of flower. Okay, so we have the reds and the blacks. This is two-dimensional data.

And as you can see, it's linearly separable. So let's start with a large value of c. If you'll remember, that means that slack is expensive. And so we're essentially backing the case of the hard-margin SVM. So in that case, this is the solution we get. All constraints are satisfied perfectly, and there are three support vectors. These three points. This is the hard-margin SVM solution. Now let's reduce c a little bit.
Let's make slack a little bit cheaper. This is what happens. So we reduce c to three. And now we're starting to use some slack. We're using a little bit of slack on this point. And we're using a little bit of slack on this point. And now there are four support vectors. We have two support vectors that are actually on the margin, and then there are two points on which we're using slack. A total of four support vectors. So what exactly is going on over here? The data is linearly separable. There is no need to violate any constraints. Why is it violating constraints? Why is it doing this? And it's doing this because it's allowed to violate constraints. Slack is not too expensive. And because in doing so, it gets a wider margin. See, this is the margin we had before. And this one is slightly larger. We've got a slightly larger margin.



So let's reduce c a little bit more. Oh, now we're violating some more constraints. This one is violating quite a bit. What do you think the slack variable on that is? Well, it's probably something like, I don't know. The xi on that might be point six roughly. Quite a lot of slack on that one. And now we have six support vectors. We have these six points over here. They're all support vectors. And we have a wider margin. Let's reduce c some more. Even more support vectors, even wider margin. And wider, and wider, and now all the constraints are violated, and we have a very wide margin. So this is what happens as you vary that parameter c.

Sentiment data

Sentences from reviews on Amazon, Yelp, IMDB, each labeled as positive or negative.

- `Needless to say, I wasted my money.`
- `He was very impressed when going from the original battery to the extended battery.`
- `I have to jiggle the plug to get it to line up right to get decent volume.`
- `Will order from them again!`

Data details:

- Bag-of-words representation using a vocabulary of size 4500
- 2500 training sentences, 500 test sentences

总结:

What C to use?

C	training error (%)	test error (%)	# support vectors
0.01	23.72	28.4	2294
0.1	7.88	18.4	1766
1	1.12	16.8	1306
10	0.16	19.4	1105
100	0.08	19.4	1035
1000	0.08	19.4	950

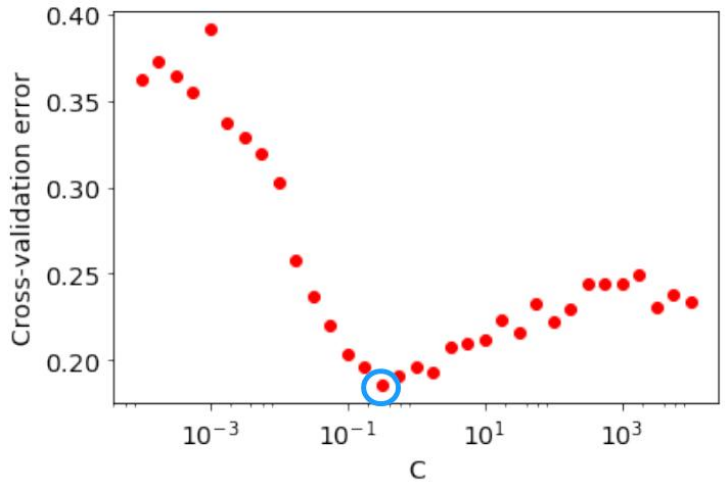
slack more expensive

Now, let's look back at the sentiment data set. So if you'll recall, this is the data set that contains sentences that have been pulled out from reviews of products or movies or restaurants. So each data point is just one sentence, and it's been labeled as positive or negative. Is it a positive review or a negative review? Now when we were talking about logistic regression, we were looking at this data set, and we saw that you can convert each sentence into a vector by using the bag-of-words representation. So each sentence is now a vector, and we have 2,500 training points and about 500 test points. Not a huge amount of data. Okay? In order to use the soft-margin support vector machine on data like this, we have to decide how to set that parameter c.

So let's look at the effect of different settings of c. So here, I'm increasing c by orders of magnitude. And if you remember, as c gets larger, it means slack becomes more and more expensive. So slack, more expensive. When slack becomes more expensive, it means fewer and fewer constraints are violated, and as a result, the training error goes down. That explains that. And because fewer constraints are violated, the number of support vectors is also going down, if you'll notice over here. That's because the support vectors include the points on the margin, but also any point for which the constraint is violated. So as the number of violations is going down, it's reasonable that the number of support vectors is also shrinking. But what happens to the test error? So we had that separate test set of 500 points. The test error goes down a little bit, and then it starts to go up. So the sweet spot seems to be somewhere between point one and ten. Somewhere in there. Which c do we pick? How do we choose a value of c? In reality, we wouldn't have a test set that we could peek at. We'd just have the training set. We'd just have the training error. How do we pick c? As always, we would pick it using **cross-validation**.

Cross-validation

Results of 5-fold cross-validation:



Chose $C = 0.32$. Test error: 15.6%

And let's see how that works out. So what I've done over here is to try a whole bunch of different values of c. And I've shown them over here on a log scale. So I've tried values of c that go from one over a thousand to well above a thousand. And for each value of c, we have to estimate the resulting error using only the training set. The way we do that is with cross-validation, and in this case I've used 5-fold cross-validation. So there's a training set of 2,500 points. We divide it into five batches, okay? So first, I treat the first batch of 500 points as a test set, train on the remaining 2,000 points, and look at the error on those 500 points. Then treat the next batch of 500 points as the test set, remain- train on the remaining 2,000 points, obtain the error on those 500 points, and so on. And in this way, I get five error numbers, and I take their average. And that's how I get an error estimate for every particular value of c. So for each value of c, I do 5-fold cross-validation and I get some error estimate. As you can see, these estimates dip quite a bit at first, and then they start going up. So which is the best value of c to use? Well, based on this cross-validation, we would pick this one right there. Okay? And that turns out to be c equals .32, and the resulting test error is 15.6%. Not bad at all. Okay, well that's it for today.

总结:

Today we've seen the soft-margin SVM, which is the form of the SVM that's typically used in practice. It's an extremely competent linear classifier. The amazing thing is that you can actually use it to go even further, to obtain boundaries that are quadratic or polynomial or beyond, and in the upcoming week, we'll see how to do this. See you later.

POLL

When choosing a value of C for the soft-margin SVM, what effect does a larger value have on the margin of the classifier?

结果

A larger value of C results in a smaller margin	74%
A larger value of C results in a larger margin	24%
A larger value of C results does not affect the margin	2%

总结:

6.4 Duality

Topics we'll cover

1. Dual form of the Perceptron
2. Dual form of the support vector machine

We have seen some beautiful and powerfullinear classification methods, the Perceptron Algorithm and two forms of the support vector machine. It turns out that you can actually do much more with them, go beyond just linear boundaries to boundaries that are quadratic or polynomial or even more general. The first step in doing this is to look at these algorithms in a slightly different way, to look at their **dual form**.
So this might sound a little bit mysterious and we'll try to shed as much light on this as possible. What we'll see is that the Perceptron and the two forms of the support vector machine can be expressed in a slightly different way that is also very intuitive.

Dual form of the Perceptron solution

Given a training set of points $\{(x^{(i)}, y^{(i)}) : i = 1 \dots n\}$:

Perceptron algorithm

- Initialize $w = 0$ and $b = 0$
- While some training point (x, y) is misclassified:
  - $w = w + yx$
  - $b = b + y$

α_i = # updates on pt i

0, 1, 2, ...

The final answer is of the form:

$$w = \sum_i \alpha_i y^{(i)} x^{(i)},$$

where α_i = # of times an update occurred on point i.

Can equivalently represent w by $\alpha = (\alpha_1, \dots, \alpha_n)$.

Let's start with the Perceptron. So here's the algorithm. The setup is that we have training points, Xi, Yi. The Xs are vectors in, say, some D dimensional space. The Ys are the labels, plus one or minus one. Now we're gonna learn a linear classifier given by W and B, and the way we're gonna do this is just by cycling through the data set, and each time we come to a point on which we are wrong, each time we come to some point X, Y that are misclassified by the current W and B, we do a simple update. We just add YX to W and we add Y to B. That's it, just four lines of code total. Now what this means is that the final answer W is of a very simple form. Here it is. The final answer is just the sum over all data points of Yi, Xi, times the number of times we updated of Yi, Xi, times the number of times we updated on that data point. So it's interesting because it means that we can represent the final answer as the vector W or alternatively, equivalently, we can represent the final answer just by providing these coefficients alpha one to alpha N. **So alpha I(i) is just the number of times we updated on point I(i)**. So we never updated on that point, which might be the case for many of the data points, then it would take on a value of zero. If we updated just once on it, it would take on a value of one. If we updated it twice, it would take on a value of two and so on.
So alphas of I is a non-negative integer, and if we just take this vector alpha, which gives us these update counts for each of the end data points, we can recover W using this formula here. So there are two ways to represent the answer, either as the vector W, which as dimension D, or as this vector of counts, alpha, that has dimension N. We will call this the **dual form** of the solution. **The original, primal form of the solution is W and the dual form is alpha**. Okay, now we can take this one step further by rewriting the entire algorithm just in terms of the alphas. Let's see what we get.

总结:

Dual form of the Perceptron algorithm

Perceptron algorithm: primal form

- Initialize $w = 0$ and $b = 0$
- While some training point $(x^{(i)}, y^{(i)})$ is misclassified:
  - $w = w + y^{(i)}x^{(i)}$
  - $b = b + y^{(i)}$

Perceptron algorithm: dual form

- Initialize $\alpha = 0$ and $b = 0$
- While some training point $(x^{(i)}, y^{(i)})$ is misclassified:
  - $\alpha_i = \alpha_i + 1$
  - $b = b + y^{(i)}$

Answer: $w = \sum_i \alpha_i y^{(i)} x^{(i)}$

So on top we have the original form of the Perceptron algorithm, which is the primal form, the same thing we saw before. Let's look at the dual form. So here, we initialize the alphas to zeros. So alpha is a vector with end entries, we initialize it to zero because we haven't updated on any of the points yet. Then we start cycling through the points, and each time we come to some point Xi, Yi that we've misclassified, it means we're gonna update on that point, so we simply increment alphas of I. We're doing one more update on that point. That's it, and we make the same change to B as before. Now at any given point, the corresponding W is just given by this formula, but we have now a different form of the algorithm that works entirely with the alphas and in some ways it's simpler than the original Perceptron algorithm. At each step, we aren't adding a vector to W, we are just incrementing a single count. So these are two ways of looking at exactly the same algorithm, and the same solution can be represented in these two ways, as W or as alpha.

Hard-margin SVM

- Given $(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)}) \in \mathbb{R}^d \times \{-1, +1\}$

$$\text{(PRIMAL)} \quad \min_{w \in \mathbb{R}^d, b \in \mathbb{R}} \|w\|^2$$
$$\text{s.t.: } y^{(i)}(w \cdot x^{(i)} + b) \geq 1 \quad \text{for all } i = 1, 2, \ldots, n$$

- This is a **convex optimization problem**:
  - Convex objective function
  - Linear constraints
- As such, it has a **dual maximization problem**.
- The **primal** and **dual** problems have the same optimum value.

Now let's move on to the support vector machine. If you remember, we had two forms of SVM. We have the hard-margin SVM, which enforces all constraints strongly and works only if the data is linearly separable. Then we have the version that's more commonly used in practice, the soft-margin SVM. So let's start with the hard-margin case. We have training data and we solve for a linear classifier given by W and B via a convex program. So this program has a linear constraint for each data point. We insist that that data point be correctly classified, and subject to all those constraints, we also wanna maximize the margin, which we've seen is the same as minimizing the length of W. So this the specification of an optimization problem. It doesn't tell us how to solve the problem, but it turns out that this is a nice problem. It's a convex optimization problem, and that means automatically that it can be solved efficiently. We don't really have to worry too much about how that's done. We're just gonna hand it over to a package and it will give us the answer. It doesn't matter what exactly the algorithm is, we don't have to worry about it. So where does the dual come in? Well, it turns out that every optimization problem has a dual optimization problem. So this particular problem, for example, which is a minimization problem, so the primal problem is a minimization, it has a dual problem that is a maximization problem. These two optimization problems have exactly the same optimal value.

总结：

The dual program

$$\text{(PRIMAL)} \quad \min_{w\in\mathbb{R}^d, b\in\mathbb{R}} \|w\|^2$$
$$\text{s.t.: } y^{(i)}(w\cdot x^{(i)}+b)\geq 1 \quad \text{for all } i=1,2,\ldots,n$$

$$\text{(DUAL)} \quad \max_{\alpha\in\mathbb{R}^n} \sum_{i=1}^n \alpha_i - \frac{1}{2}\sum_{i,j=1}^n \alpha_i\alpha_j y^{(i)}y^{(j)}(x^{(i)}\cdot x^{(j)})$$
$$\text{s.t.: } \sum_{i=1}^n \alpha_i y^{(i)} = 0$$
$$\alpha \geq 0$$

Complementary slackness: At optimality, $w=\sum_{i=1}^n \alpha_i y^{(i)}x^{(i)}$ and

$$\alpha_i > 0 \;\Rightarrow\; y^{(i)}(w\cdot x^{(i)}+b)=1$$

Points $x^{(i)}$ with $\alpha_i > 0$ are **support vectors**.

So what is this dual problem? Here it is. This is the original primal problem that we just solved and this is the dual. What exactly is this doing? Well, it now has a different set of variables. It's not solving for W and B, it's solving for alphas, alpha one through alpha N, and these are the same alphas that we saw in the case of the Perceptron. So this is solving the problem in alpha space. Once again, once we solve the problem, the solution we want, the W, will turn out to have the same form as it did with the Perceptron. The sum of data points where the coefficient of each point is alpha I. So there two things to mention over here.

- The first thing is that the primal problem we said was a nice problem, a convex minimization problem. The dual problem is also nice. It's a concave maximization problem. It's also one of these problems that can be solved efficiently.
- It might be a little hard to figure out exactly what it's doing, but there's this beautiful theory called **Complementary Slackness** that tells us the relationship between the primal solution and the dual solution, and what it tells us in this case is that in terms of the alphas, the only points on which alpha is non-zero are the points that lie exactly on the margin. This is where the notion of support vector comes from. Duality gives it to us. Duality is what tells us that the solution W depends only on the support vectors, the points that lie exactly on the margin.

Dual of soft-margin SVM

$$\text{(PRIMAL)} \quad \min_{w\in\mathbb{R}^d, b\in\mathbb{R}, \xi\in\mathbb{R}^n} \|w\|^2 + C\sum_{i=1}^n \xi_i$$
$$\text{s.t.: } y^{(i)}(w\cdot x^{(i)}+b)\geq 1-\xi_i \quad \text{for all } i=1,2,\ldots,n$$
$$\xi \geq 0$$

$$\text{(DUAL)} \quad \max_{\alpha\in\mathbb{R}^n} \sum_{i=1}^n \alpha_i - \frac{1}{2}\sum_{i,j=1}^n \alpha_i\alpha_j y^{(i)}y^{(j)}(x^{(i)}\cdot x^{(j)})$$
$$\text{s.t.: } \sum_{i=1}^n \alpha_i y^{(i)} = 0$$
$$0 \leq \alpha_i \leq C$$

At optimality, $w=\sum_i \alpha_i y^{(i)}x^{(i)}$, with

$$0<\alpha_i<C \;\Rightarrow\; y^{(i)}(w\cdot x^{(i)}+b)=1$$
$$\alpha_i = C \;\Rightarrow\; y^{(i)}(w\cdot x^{(i)}+b)=1-\xi_i$$

Okay, so let's move on to the soft-margin SVM. So this is rather like the hard-margin problem, except that we also have these slack variables. We are no longer in a situation where the data is necessarily linearly separable. So instead of having hard consraints, that every point has to be perfectly classified, we allow some of these constraints to be violated and these slack variables, xi(ξ) of I(i), capture the extent of the violation. So this is the primal problem, and we saw it last time, and once again we can write down a dual problem.

The dual, in fact, looks a whole lot like the dual for the hard-margin SVM. The solution, again, gives us these coefficients alpha, where the W we want is just given by a linear combination of the data points with these coefficients alpha. Now when we use Complementary Slackness, we find that in this case, there are actually two kinds of support vectors. There are two kinds of points on which alpha is non-zero. There are the points that lie exactly on the margin, and then there are points on which we've used slack. So there are two kinds of support vectors, and again it is this beautiful theory of duality that gives us this surprising insight.

总结:

So that's it for today. We have got just a very small glimpse of duality. Duality is central to optimization, and as you go further and further into machine learning, it's a concept that you're gonna bump into again and again. For our purposes, we've seen just enough of it to give us what we need to take Perceptron and support vector machine to the next level to use them not just for linear boundaries, but for polynomial and beyond. See you next time.

POLL

For the hard-margin SVM, a data point, (x_i,y_i), with a coefficient of $\alpha_i = 0$ resides where?

结果

On the correct side of the margin, but not on the margin	60%
Exactly on the margin	34%
On the wrong side of the margin	6%

总结:

6.5 Multiclass Linear Prediction

Topics we'll cover

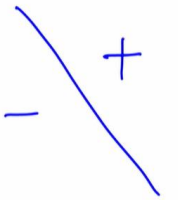
1. Multiclass logistic regression
2. Multiclass Perceptron
3. Multiclass support vector machines

We have recently seen some beautiful methods for linear classification. Logistic regression, the proceptron algorithm, and support vector machines. But we've only talked about the binary case, where the possible labels are plus one or minus one. What happens when there are more clauses? How can we accommodate them? So we'll see that there's a simple, unifying formalism that lets us extend all three of these methods to multiclass. And we'll look at them one by one.

Multiclass classification

Of the classification methods we have studied so far, which seem inherently binary?

- Nearest neighbor?
- Generative models?
- Linear classifiers?



So in this class we've studied all sorts of different methods for classification, and some of them are inherently multiclass. Nearest neighbor for instance. It automatically adapts to however many classes there are. You don't need to do anything special at all. Generative models are also inherently multiclass. You just fit a Gaussian or some other distribution to each individual class, and that's it. Nothing else needs to be done. Linear classifiers on the other hand, seem a little bit tricky. After all, when you have a line there are only two sides to it.

The main idea

Remember Gaussian generative models...

$x \in \mathbb{R}^d$, $y \in \{1, 2, \ldots, k\}$

Class 1: $w_1 \cdot x + b_1$

Class 2: $w_2 \cdot x + b_2$

⋮

Class k: $w_k \cdot x + b_k$

So how can we possibly accommodate more classes? So to get the main idea, let's think back to Gaussian generative models. What we were doing there, was to fit a Gaussian to each class. So maybe a Gaussian for class one, another Gaussian to class two, another Gaussian to class three, another one for class four and so on. We observe then, that the log of the Gaussian density is a quadratic function. And so effectively, each class has got its own quadratic function. When a new point comes along, each class evaluates its quadratic function, and the one with the highest value is the winner. That's the label we predict.

So we can do something similar to this for linear classification. We can let each class have its own linear function. Okay, so let's see what that means. So let's say we have data in d dimensional space. So our data points x line rd. And let's say that the linguals now have k possible value. Say one, two, all the way to class number k. We will let each class have its own linear function. Okay, so class one has some linear function. Lets call it w one dot x plus b one. And here w one is a vector in d dimensional space and b one is just an offset, just a number. Class two has its own linear function, say w two dot x plus b two. All the way to class k. Say wk dot x plus bk. And now when a new point comes along, when a new point x arrives, every class evaluates its linear function, and the one with the largest value is the winner. And that's the label we predict.

总结:

From binary to multiclass logistic regression

Binary logistic regression: for $\mathcal{X} = \mathbb{R}^d$, classifier given by $w \in \mathbb{R}^d$ and $b \in \mathbb{R}$:

$$\Pr(y = 1|x) = \frac{e^{w \cdot x + b}}{1 + e^{w \cdot x + b}}$$

Labels $\mathcal{Y} = \{1, 2, \ldots, k\}$: specify a classifier by $w_1, \ldots, w_k \in \mathbb{R}^d$ and $b_1, \ldots, b_k \in \mathbb{R}$:

$$\Pr(y = j|x) \propto e^{w_j \cdot x + b_j}$$

- What is the fully normalized form of the probability?

$$\Pr(y=j \mid x) = \frac{e^{w_j \cdot x + b_j}}{e^{w_1 \cdot x + b_1} + \ldots + e^{w_k \cdot x + b_k}}$$

- Given a point x, which label to predict?

$$\operatorname{argmax}_j \; w_j \cdot x + b_j$$

So let's see how this works out for logistic regression. So if a binary logistic regression, which is the case we studied before, if you remember, we just had one linear function, w dot x plus b. And for any given point x the probability that y equals one was given by this formula over here. Now let's move to the case where we have k possible labels, one through k. We'll now have a linear function for each label. So for label one we have w one b one. For label two, w two b two. For label k wk, bk. And for any given point x, the probability that the label is j, the probability that y equals j is going to be proportional to e raised to the jth linear function. Wj dot x plus bj. So the higher this linear function, the larger the value of this function, the more likely it is that the label is j. Okay so we said proportional to. What is the exact formula for the probability? We just need these to add up to one. So we just need to normalize. So the probability that y is j given x is exactly that term up there. E to the wj dot x plus bj. And now we normalize it so we divide it by the term for class one, plus the term for class two, all the way to the term for class k. Okay, so that's the probability that y equals j given x. Now what label would we predict for x? Well, the one for which this function is the highest. The one for which wj dot x plus bj is largest. In other words the argmax of this. And that's it.

Multiclass logistic regression

- **Label space**: $\mathcal{Y} = \{1, 2, \ldots, k\}$
- **Parametrized classifier**: $w_1, \ldots, w_k \in \mathbb{R}^d$, $b_1, \ldots, b_k \in \mathbb{R}$:

$$\Pr(y = j|x) = \frac{e^{w_j \cdot x + b_j}}{e^{w_1 \cdot x + b_1} + \cdots + e^{w_k \cdot x + b_k}}$$

- **Prediction**: given a point x, predict label $\arg\max_j (w_j \cdot x + b_j)$.
- **Learning**: Given: $(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})$.
  Find: $w_1, \ldots, w_k \in \mathbb{R}^d$ and $b_1, \ldots, b_k$ that maximize the likelihood

$$\prod_{i=1}^{n} \Pr(y^{(i)}|x^{(i)})$$

Taking negative log gives a convex minimization problem.

So here's a summary over here, of our prediction rule and our model. How about learning? So let's say we have a training set, x one, y one, all the way through xn, yn. How do we learn these k linear functions? Well, we'll do exactly what we did in the binary case. We'll just pick the parameters that maximize the likelihood of the data. That is, that maximize the probability of the first data point times the probability of the second data point all the way to the probability of the nth data point. So that's this function over here. And as usual, products are a little bit tricky to deal with, so we can just take the logarithm to make it into a sum. And if we wanna make it into a minimization problem, we can just stick a minus sign in front. And when we do that, we end up with a convex minimization problem. In other words, it's a problem that's quite eaxi to solve, and in practice it will just hand the data over to a package which will then return the solution to us. So multiclass logistic regression is not a problem at all.

总结:

Multiclass Perceptron

Setting: $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \{1, 2, \ldots, k\}$

Model: $w_1, \ldots, w_k \in \mathbb{R}^d$ and $b_1, \ldots, b_k \in \mathbb{R}$

Prediction: On instance x, predict label $\arg\max_j (w_j \cdot x + b_j)$

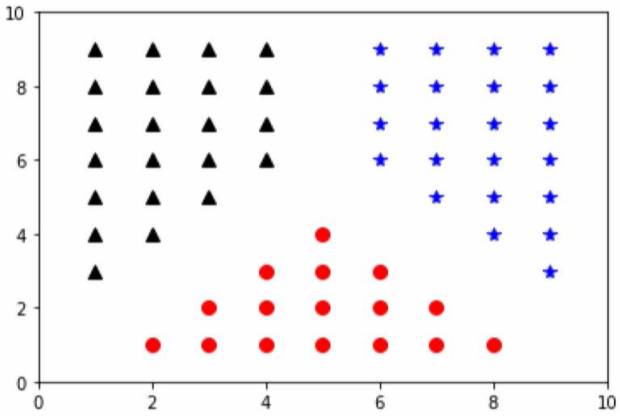
Learning. Given training set $(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})$:

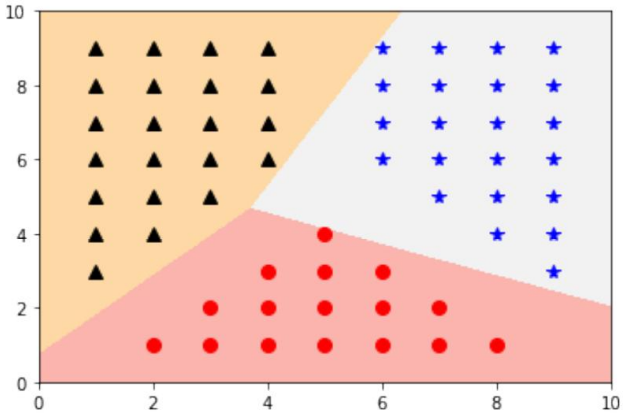
- Initialize $w_1 = \cdots = w_k = 0$ and $b_1 = \cdots = b_k = 0$
- Repeat while some training point (x, y) is misclassified:

$$\begin{aligned} \text{for correct label } y: \quad & w_y = w_y + x \\ & b_y = b_y + 1 \\ \text{for predicted label } \widehat{y}: \quad & w_{\widehat{y}} = w_{\widehat{y}} - x \\ & b_{\widehat{y}} = b_{\widehat{y}} - 1 \end{aligned}$$

Now let's move on to the perceptron algorithm. That was just four lines of code. Can we somehow adapt the perceptron to deal with multiple classes? And it turns out that it's actually really simple to do so. Here's the modified algorithm: also very short. So we have a training set, x one, y one all the way through xn, yn. And as usual, we begin by setting all the parameters to zero. By setting all the linear functions to zero. In this case we have k and m. Then we start going through the data set, and each time we come to a point that's misclassified we do a little update. So we go through the data, and let's say that we arrive at a particular point xy. The first thing we have to do is to figure out what label we would predict on x given our current parameters. And that label is simply the label j that maximizes wj dot x plus bj, where the ws and bs are our current parameter values. So let's say that the label we predict is y hat. Now if that's the correct label, if y hat equals y, then we don't have to do anything. But if we predicted the wrong label, if y hat is not equal to y, then we have to update the parameters. But which parameters do we update? We have k different linear functions. Well we're only going to update two out of the k functions. We're gonna update the function for the correct label, and the function for the label that we actually predicted. And both of these updates are very simple. So for the correct label, we simply add x to the w vector, and increment b. For the label we predicted, we just subtract x from the w vector, and decrement b. And that's it. The multiclass proceptron algorithm. So let's see it at work on a simple data set.

Multiclass Perceptron: example





So here is some data in two dimensions, and withthree classes. We have the black triangles, the blue stars, and the red circles. What does multiclass perceptron do on this? Well this is the bine boundary that it finds. It perfectly classifies the data. Now as you can see, the boundary that it finds is made up of three linear pieces. There's this piece over here, this piece over here, and this piece over here. And basically it keeps cycling through the data, until it finds this perfect classifier at which point it immediately holds. It does not for instance, try to optimize the boundary further for instance by trying to center the boundary or by trying to maximize the margin in some way. So that makes us wonder, what about the support vector machine? Can we make that multiclass as well? And indeed we can, and it's quite simple.

总结:

Multiclass SVM

Model: $w_1, \ldots, w_k \in \mathbb{R}^d$ and $b_1, \ldots, b_k \in \mathbb{R}$

Prediction: On instance x, predict label $\arg\max_j (w_j \cdot x + b_j)$

Learning. Given training set $(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})$:

$$\min_{w_1,\ldots,w_k \in \mathbb{R}^d, b_1,\ldots,b_k \in \mathbb{R}, \xi \in \mathbb{R}^n} \sum_{j=1}^{k} \|w_j\|^2 + C \sum_{i=1}^{n} \xi_i$$

$$w_{y^{(i)}} \cdot x^{(i)} + b_{y^{(i)}} - w_y \cdot x^{(i)} - b_y \geq 1 - \xi_i \quad \text{for all } i \text{ and all } y \neq y^{(i)}$$

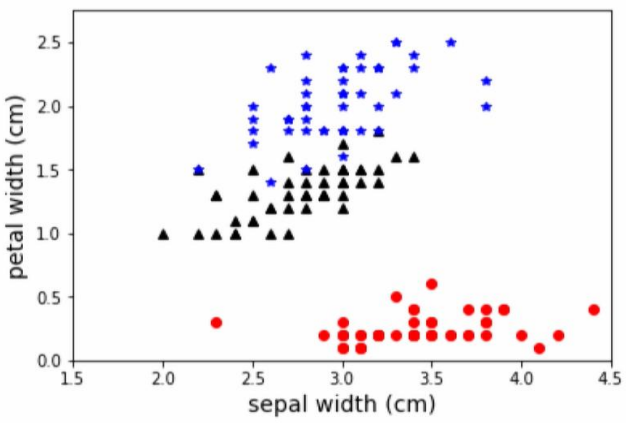
$$\xi \geq 0$$

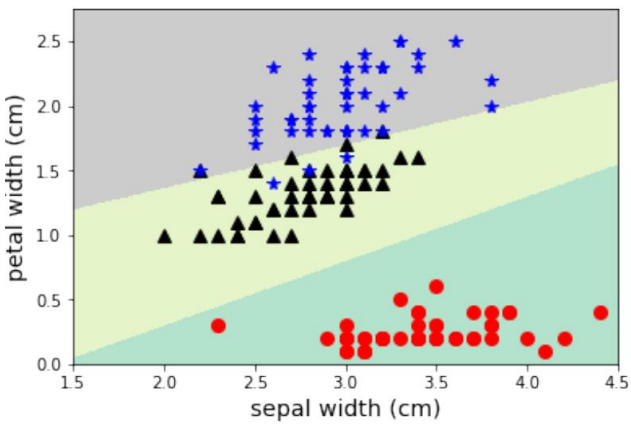
$w_{y^{(i)}} \cdot x^{(i)} + b_{y^{(i)}} \geqslant w_y \cdot x^{(i)} + b_y + 1 - \xi_i$

for $y \neq y^{(i)}$

Here is the convex program for multiclass support vectormachine. So as before, let's say we have a training set, x one y one through xn yn. As usual, we want to minimize the norm of w squared, but in this case we have k different ws so what we are minimizing is the sum of all of their squared norms. As usual with the soft margin SVM, we have a slack parameter for each data point. Now if you'll recall the function of this was to allow us to violate some of the constraints, because for example, it might not be possible to perfectly classify all of the training data. So the slack parameter allows us to violate some of these constraints, but we don't want too much violation. And so we include the sum of the slack variables in the function that we want to minimize. And that remains unchanged from before, from the soft margin SVM. And then, over here, we have the constraints that say that we want the data points to be classified correctly. So what exactly are these constraints? Let's take a look. Let's say we have a particular data point, xi, yi. Now there are k possible labels. In order for this point to be correctly classified, we need the linear function for the correct label to beat out the linear function for any of the incorrect labels. So for the linear function for ys of i, has to be greater than the linear function for the k minus one other labels. The ys that are not equal to ys of i. So if we write out this condition, what we have is that the correct linear function which is w of ys of i, the w for the correct label, dot x of i plus b of ys of i should exceed any of the wrong linear functions. So wy dot x of i plus by for any of the labels that is incorrect. This is the condition that says that all the points are perfectly classified. Now as we did before, we can equivalently insist, that actually the correct label exceeds the incorrect labels by one. And we can do this by simply rescaling the ws and because appropriately. So this is another way of saying that all the points are correctly classified.
Now of course it might not be possible to correctly classify all the points, so we do have to allow a little bit of slack, and that's where we include the slack variable xis of i. Okay so these are our final constraints, and this is exactly the inequality we have over here, just shifted around a little bit. So this is the convex program for the multiclass SVM. So let's see it at work.

Multiclass SVM example: iris





This is the iris data set which we've talked about before. I've chosen two of the four features. There are three classes and it's clear that the data is not linearly separable. In particular we would not be able to run the perceptron algorithm on this. Okay, but let's see what the multiclass, soft margin SVM does. This is the boundary it finds. It does a very good job. As usual, it is a very competent classifier.

总结:

Multiclass SVM

For each pt: k-1 constraints

Given training set $(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})$: total: $n(k-1)$.

$$\min_{w_1,\ldots,w_k \in \mathbb{R}^d, b_1,\ldots,b_k \in \mathbb{R}, \xi \in \mathbb{R}^n} \sum_{j=1}^{k} \|w_j\|^2 + C \sum_{i=1}^{n} \xi_i$$

$$w_{y^{(i)}} \cdot x^{(i)} + b_{y^{(i)}} - w_y \cdot x^{(i)} - b_y \geq 1 - \xi_i \quad \text{for all } i \text{ and all } y \neq y^{(i)}$$

$$\xi \geq 0$$

Once again, a convex optimization problem.

Question: how many variables and constraints do we have?

$w_1 \ldots w_k$: kd

$b_1 \ldots b_k$: k

$\xi_1 \ldots \xi_n$: n

$kd + k + n$

So let's end with a little bit of a, with a small question. So this was the convex program for the multiclass soft margin SVM. How many variables do we have here and how many constraints? Let's see if we can count these. So the variables are the ws. So w one through wk. How many parameters do we have there? Well each of them is a d dimensional vector, so it's a total of kd parameters. Our variables also include b1 through bk. Each of them is a number, so that's k parameters, and then we have the slack variables. xi one through xi m. So m parameters. So the total number of variables is kd plus k plus m. How about constraints? How many constraints do we have over here? Okay, so let's see, for each point we need the correct label to beat out the k minus one incorrect labels. So for each point we have k minus one constraints. So the total number of constraints since there are n data points is n times k minus one. And that's it.

Okay that's good for today. We've seen how we can extend our linear classification methods from binary to multiclass. It's really quite simple and in fell swoop we were able to take care of logistic regression, proceptron, and the support vector machine. See you next time.

POLL

How many of the linear functions are updated when the multiclass perceptron algorithm classifies a point incorrectly?

结果

1		22%
✓ 2		62%
3		8%
✗ 4		8%

总结: