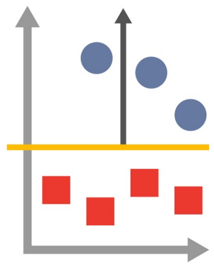
Learning with Linear Classifiers

What you'll do

- Apply linear machine learning algorithms to solve classification and regression problems.
- Identify the applicability, assumptions, and limitations of linear classifiers.
- Implement the perceptron algorithm for linear classification.
- Choose an appropriate loss function for a given data set.
- Use gradient descent to minimize loss functions.
- Build a linear classifier for email spam classification.



Course Description

Linear classifiers are a powerful class of machine learning algorithms. The first of these, the perceptron, was invented at Cornell University in 1957 and is the basis for modern neural network designs. These simple yet effective algorithms are useful in a number of settings, for both classification and regression problems.

In this course, Professor Weinberger introduces you to linear classifiers, highlighting assumptions, limitations, and implementations. You will have the chance to implement one of the first linear classifiers, the perceptron, yourself to investigate these concepts. You will further explore linear regression, logistic regression, and gradient descent and ultimately build your own email spam filtering classifier.

System requirements: This course contains a virtual programming environment that does not support the use of Safari, IE, Edge, tablets, or mobile devices. Please use Chrome or Firefox on a computer for this course.

Kilian Weinberger
Associate Professor
Computing and Information Science, Cornell University

Kilian Weinberger is an Associate Professor in the Department of Computer Science at Cornell University. He received his PhD from the University of Pennsylvania in Machine Learning under the supervision of Lawrence Saul and his undergraduate degree in Mathematics and Computer Science from the University of

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Oxford.

During his career, he has won several best paper awards at ICML (2004), CVPR (2004, 2017), AISTATS (2005), and KDD (2014, runner-up award). In 2011, he was awarded the Outstanding AAAI Senior Program Chair Award and in 2012 he received an NSF CAREER award. He was elected co-Program Chair for ICML 2016 and for AAAI 2018. In 2016, he was the recipient of the Daniel M Lazar '29 Excellence in Teaching Award.

Professor Weinberger's research focuses on Machine Learning and its applications. In particular, he focuses on learning under resource constraints, metric learning, machine-learned web-search ranking, computer vision, and deep learning. Before joining Cornell University, he was an Associate Professor at Washington University in St. Louis and previously worked as a research scientist at Yahoo! Research.

Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Q&A

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Live Labs: Building a Community of Learners

Building a Community of Learners

This course will include two live (synchronous) video sessions called Live Labs. Live Labs will be offered each week as an opportunity to connect with your peers and instructor to address questions about the coursework and dive deeper into the material. Each hour-long session will be scheduled in advance. You are highly encouraged to take advantage of this chance to connect with your community of learners to build relationships and make this course experience even more rewarding.

This short Q&A section of the course contains a discussion called **Live Labs Q&A** where you can post questions and interact with your peers at any time as you work through this course. Common questions and important topics brought up in this discussion may be addressed in the Live Labs sessions.

In order to review the session schedule and join a session, you'll need to navigate to the **Live Labs** link under the **Course Shortcuts** section in the navigation menu on the left.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Live Labs Q&A

The Live Labs Q&A discussion is a place for you to post questions and interact with your peers throughout your work in this course. Common questions and important topics brought up in this discussion will likely be addressed in a Live Labs session.

Instructions:

Read through existing posts to find out what other students are discussing. Upvote posts that you find interesting or contain questions that you also have by "liking" it with the thumbs-up button at the bottom of the post.
If you have something to contribute to an existing discussion, reply to the thread and get involved in the conversation.
If you have new ideas, questions, or issues, post them in this Live Labs discussion board. If you have potential answers or suggestions that may resolve a question thread, feel free to share your understanding in a reply. This will help guide the instructor's feedback and discussion during the live session.

The questions that are the most upvoted or replied to that haven't been resolved will likely be addressed at the scheduled time by the instructor.

Back to Table of Contents

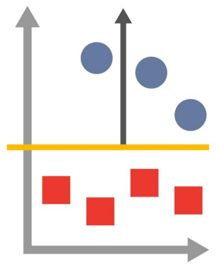
© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Module 1: Linear Classifiers

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Module Introduction: Using Linear Classifiers



One challenge in machine learning is to identify the algorithms that are best suited to solve your particular problem. In this module, you will examine how and when linear classifiers are best used for machine learning tasks. You will investigate the assumptions made by these models and will explore how linear classifiers are trained. You will see how the first linear classifier, the perceptron, works and will have the opportunity to implement it in Python.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Watch: Linear Classifiers

Linear classifiers can be used to split data into separate classes based on the features of the data set. Professor Weinberger describes the assumptions linear classifiers make and explains how they are trained in order to classify new data points.

Video Transcript

Linear classifiers make two basic assumptions on the data. The first one is that the data is binary; that means we have positive points and we have negative points. The second one is that these points are positioned such that we can place a hyperplane between them, such that the positive points lie on one side and the negative points lie on the other side. In this example, we have two-dimensional data; the positive points are the blue circles, the negative points are the red squares. And we can position a line between them such that the positive points lie on one side, the negative points on the other side. A hyperplane has two sets of parameters. The first one is a vector w, which determines the direction of the hyperplane. If you turn this w, the hyperplane changes, too; w is orthogonal to the hyperplane and always goes through the origin. The second parameter is the offset or the bias, and we call it b because of bias. And b determines how far off from the origin the hyperplane lies. If b equals 0, then the hyperplane loads exactly smack through the origin. The way linear classifiers work is that during training, we take the training data and we position the hyperplane such that it separates the two classes. And during testing, when a new test point comes in, we then determine which side of the hyperplane this point lies on. And if it's on the positive side, we predict it's a positive point; and if it's on the negative side, we predict it's a negative. The way we do this is we just compute the inner product; w transpose x plus the offset b. If that's positive, we say it's on the positive side; if it's negative, we say it's on the negative side. It seems like it's a strong assumption that we can position this as a hyperplane between the positive and negative points. What if, for example, one of the blue points was way down there at the bottom? It wouldn't be possible anymore. But it turns out — in low-dimensional spaces this is true; it is a strong assumption. But in high-dimensional spaces, it's not. If you have millions of dimensions, there's always some way of wedging a hyperplane between the positive and the negative points. And high-dimensional data is quite common; for example, text documents are represented as very high-dimensional vectors; sometimes even million- dimensional vectors. So typically we use linear classifiers as spam classifiers, to determine if an email is spam or not spam, or to classify news documents into politics or sports, for example.

Click to check your knowledge of linear classifiers.

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Watch: Incorporate Bias

Linear classifiers fit a hyperplane to separate data classes. Professor Weinberger shows you a few tricks that help simplify the math and avoid imposing restrictive assumptions on your classifier. The result is a method that ensures you can always fit a hyperplane through the origin that can separate points from different classes.

Video Transcript

Linear classifiers have two parameters: the direction vector w and the offset or bias b. When we position a hyperplane, we have to learn both of these parameters. It turns out that it's often convenient if this offset b was just 0 and would go away, just because of mathematically then we just have one term less to worry about. And it turns out that actually we can do this. So it seems very restrictive to say that b equals 0, because if we say b equals 0, that means the hyperplane must go through the origin, right? So now we made the assumption even more restrictive. But actually there's a simple trick how we can make it just as general as having a hyperplane with an offset, and that is if you just add one dimension. Let me walk you through it. So if we want to compute which side of a hyperplane the data point lies on, we have to compute w transpose x plus b. Here we need this b term, and it could be non-zero, so let's keep this general form. But here comes a simple trick. We could just add one more dimension to our data. So every single data point gets one additional dimension, and we just set the feature value of this dimension to 1. And then we concatenate the offset B as the parameter of this particular dimension to the w vector. So, w becomes w and a b concatenated to it, and x becomes x and a 1 concatenated to it. And now, if you see w, this new inner product between these two vectors is exactly w transpose x plus b. So, essentially what I just showed you is that if we add one more dimension to our data, we can always fit a hyperplane through the origin that's just as powerful as having a hyperplane that does not go through the origin to the original data. Let me show you an example. So imagine we have a one-dimensional data set. So that means everything lies on this one-dimensional line. Here we have a positive point and a negative point; we have a blue point and a red point. We would like to find a hyperplane between these two — if you want to cut between these, it wouldn't go through 0. But if we now add one more dimension that's a constant 1 — so basically both these points move up in this additional dimension to the value 1 — then we can just find a hyperplane that separates them, that goes through the origin 0. Similarly, if we originally did two dimensions — here again, we have the blue points and the red points. Again, this is a scenario where we would need a hyperplane with some offsets to separate the two. And if we add a third dimension, that means we move all the data

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

points in the third dimension into space by exactly 1. Now we can wedge a plane between these two sets of points. And this is what we will assume throughout. So we will basically from now on always talk about hyperplanes as if they always go to 0 and assume that the first thing we do is, as a pre-processing step when we get our data, we just add a constant dimension to all our data points, and then we are done with the offset or the bias term.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Watch: The Perceptron

The perceptron was the first linear classifier, invented at Cornell in 1957. Professor Weinberger describes the inner workings of the perceptron and how it updates the hyperplane if a sample is still misclassified. Through an example, you see how the perceptron update works to correctly position a hyperplane that separates two classes of data points.

Video Transcript

Arguably, the first modern machine learning algorithm was the perceptron, and it was invented in 1957 by Frank Rosenblatt right here at Cornell University. And when it came out, it was a huge deal; it was all over the news because it was the first time that someone created an artificial neuron that was really genuinely modeled after the human neuron. How does the perceptron work? The perceptron makes the assumption that our data is sampled from two classes, positive and negative; we call them plus 1 and minus 1. It takes an input vector x and assigns it either to the class plus 1 or minus 1. And how does that work? It essentially learns a weight w for every single dimension — so w1, w2, w3, etc. — and then just takes a weighted sum of all the input features times the corresponding weights. And if this weighted sum is positive, it assigns a plus 1; and if it's negative, it assigns a minus 1. So essentially what the perceptron is doing, it learns a weight vector w and then outputs the sign of the inner product of w with that input vector x. So that's exactly a linear classifier, right? So it's basically outputting are you on the positive side or on the negative side of this hyperplane? When Frank Rosenblatt initially trained the perceptron, he trained it to recognize the C of the Cornell logo, and every single input feature was a pixel of a camera. And whenever he showed it a C, it fired; that meant it would turn to plus 1, otherwise it would turn a minus 1. All right. So when we train a perceptron, we've given data points of the two classes, plus 1 and minus 1. Here the positive points are blue, the negative points are red. And we would like to position the hyperplane exactly between the two classes. So how do we do this? For this we need the perceptron update. And what the perceptron update does, it takes a hyperplane that is initially not well positioned and rotates it a little bit such that it's better positioned. And this principle behind it is quite simple. What we do is we take a hyperplane that's initially not a separating hyperplane. Because it's not a separating hyperplane, that means there's some point that is misclassified that's on the wrong side of the hyperplane. And what we do is, if this point is positive, we add it to w; and if it's negative, we subtract it from w. Let's go through an example. So here we have our positive points that are blue and our negative points that are red. And you can see there's one red point, for

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

example, is misclassified. Let's call it x. It's on the wrong side. This point x is really actually a vector that goes from 0, from the origin, to this point. There's now this orange vector. And because it's red because it's a negative point, to move it to the right side to do the perceptron update we have to subtract it from our w vector. So we take our vector w and subtract from it this orange vector; now it's minus x. And if we now move our new hyperplane to point exactly to the end point of these two vectors, then what you see is that this hyperplane is rotating such that this vector x is now on the correct side of the hyperplane. That's the perceptron update. The cool thing is here in this case actually now all the points are correctly classified. So we are done. In practice, it doesn't always happen that after one update, you're already done, But if you're not done, well, you just take another point that is misclassified and you do another update, and you keep doing this until there's no points left that are on the wrong side of the hyperplane. The amazing thing behind the perceptron is that this procedure always ends up with a separating hyperplane if such a hyperplane exists. And not only that, in fact, we can prove that you only need to do a finite number of updates, and that's a very, very strong guarantee and that's why people were so amazed when it came out. So just to recap, the perceptron assumes that our data comes from two classes, plus 1 and minus 1, and that there exists a linear hyperplane that can separate these two classes. If these two assumptions are made, the perceptron will output, in a finite number of steps, a hyperplane that separates these two classes.

Click to check your knowledge of the perceptron.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Watch: Apply the Perceptron to Visual Vectors

We can also use the perceptron on high dimensional data. In this video, Professor Weinberger shows how the perceptron can classify handwritten digits by treating the pixels of an image as dimensions of a feature vector.

Video Transcript

Let us do one more example of the perceptron algorithm; this time, not in two dimensions, but in 256 dimensions. 256 dimensions usually would be really hard to visualize. But in this case, we're using images, and images are great to visualize. Our data here consists of two different classes, handwritten digits: handwritten 0s, those are my positive class; and handwritten 7s, these are my negative class. This data was actually collected in the '80s by the U.S. Postal Service in the hope that someone eventually would build a zip code — automatic zip code reader, and that actually happened. All right. So, I'm showing you here some digits, and I'm also showing you the weight vector. Because the weight vector is just a vector, and this 256th-dimensional space, we can also visualize it as an image. On the y-axis here, you can see the inner product value. So images that move up on the y-axis have a positive inner product of the weight vector; images that are low have a negative inner product. Right now, everything is at 0 because the weight vector is all 0s. So everything multiplied with 0s is just 0. One thing that's cool about the space is that the weight vector can also be visualized and has a very nice interpretation. Every single weight is actually a weight on a pixel. So essentially what we're learning is we're basically assigning weights to pixels, and we're saying if it's positive, then that's indicative that it's a 0. And if it's negative, it's indicative that it's a 7. So let's go through this. The first digit is this handwritten 0, and it's misclassified; it should be positive. It's not positive. So what do we do? We add it to our weight vector, right? So now the weight vector actually — because initially it was 0 — becomes exactly that digit. So if you compute the inner products with all the other digits, you can see they're all becoming positive. And that makes sense because right now, everything is positive. That's great for the 0s, but the 7s are misclassified. We can look at the first 7; this inner product between the weight vector and the 7 should be negative. So we need to assign some negative weights around those pixels where the 7s typically has pixels. So what do we do? The second 7 is a negative example, so we have to subtract it from the weight vector. If you subtract the 7, you can see the new weight vector now is the positive 0 minus the negative 7. If you now assign — compute the inner products with all the digits, you can see that the 0s are positive because those align with the positive weights; and the 7s are all negative because they're aligned with negative weights. And now the whole

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

data set is classified correctly.

Click to check your knowledge of visual vectors.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

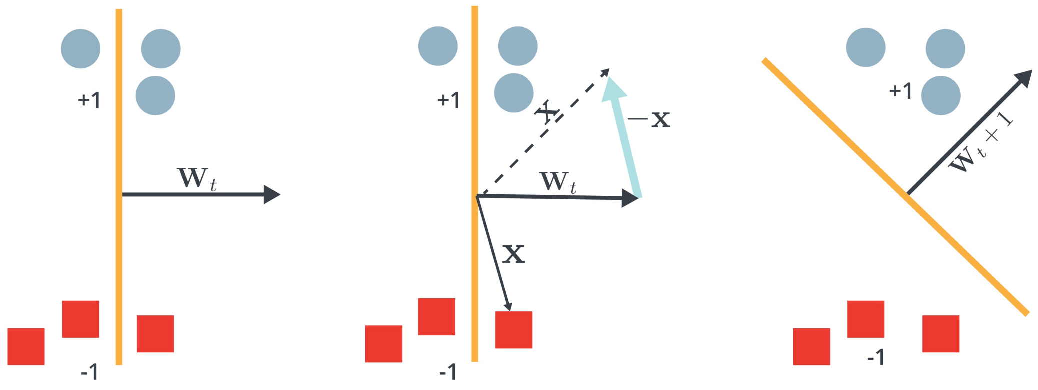
Read: Perceptron Update

The hyperplane is initially set to = 0.

The hyperplane is iteratively adjusted (or "tilted") using one misclassified point at a time.

The final hyperplane perfectly separates the two classes.

The perceptron algorithm aims to obtain a weight vector that defines a hyperplane that separates the data of two classes based on their feature values. So long as there exists a hyperplane that can separate the data completely, the perceptron algorithm will find it in a finite number of steps using repeated perceptron updates.



Above is an illustration of a perceptron update.

In the left diagram, the hyperplane defined by misclassified one red (-1) and one blue (+1) point. In the middle diagram, the rightmost red point is chosen and used for an update because it lies on the wrong side of the hyperplane. Its label is -1, so we need to subtract from to obtain a new weight vector. On the right, the updated hyperplane now separates the two classes and the perceptron algorithm has converged.

Notes:

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

We can “absorb” the bias into by appending a 1 to each data point, i.e.,

This means that the update step .dot( ) implicitly includes the addition of the bias term (which is simply).

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Activity: Calculate Perceptron Update

Exercise 1

Consider the following two-point 2D data set:

Positive class (+1):

- Negative class (-1):

Starting with , which is equivalent to a vertical hyperplane as on the previous page, how many updates will you have to perform to until convergence? Write down the sequence of each updated by iterating the data points in the order: .

When you think you know the answers, click the images to reveal the solutions

Click to reveal the answer

Answer: There are 5 updates as follows:

Exercise 2

Your friend Cüneyt comes to you, desperate for your perceptron expertise. His data set is massive, with more than 10 trillion training examples. After hours of training his perceptron until convergence, his code malfunctioned and did not save the final weight vector.

Thankfully, at every training iteration, the code saved which example was used for the update step. Surprisingly, only five of the more than 10 trillion training examples were ever misclassified. They are listed below, along with the number of times they were used in an update step.

Training Example	Times Used in an Update Step
(0, 0, 0, 0, 4), +1	2
(0, 0, 6, 5, 0), +1	1
(3, 0, 0, 0, 0), -1	1
(0, 9, 3, 6, 0), -1	1

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

(0, 1, 0, 2, 5), -1	1

What is the final weight vector of this perception (i.e. the weight vector that would have been saved if the code had not malfunctioned)?

Click to reveal the answer

Answer:

Because the final vector is equal to

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Tool: Perceptron Cheat Sheet

Use this Perceptron Cheat Sheet as a quick way to review the details of how the perceptron works.

This tool provides an overview of the perceptron for your reference. You'll find information about the applicability, underlying mathematical principles, assumptions, and other details of this classifier.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Watch: Proof of Convergence

The perceptron algorithm guarantees that if a hyperplane exists between the positive and negative classes, it will properly position the hyperplane after a finite number of updates. Professor Weinberger demonstrates how this works and explains the mathematical proof.

Video Transcript

One of the appealing aspects of the perceptron algorithm is the strong guarantees that come with it. In particular, there's a theorem that says that if a hyperplane exists between the positive and negative points, then the perceptron algorithm will find such a hyperplane within a finite number of updates. In fact, it can determine exactly an upper bound of how many updates the perceptron algorithm could possibly make. I can now walk you through the intuition behind this proof. In the notes, you'll see the details to every single step. We first make two assumptions. The first assumption is the assumption that you always make. with the perceptron, is that there exists such a hyperplane that separates the two classes. We don't have to know where it is; we just have to assume that it exists. Let's call it w*. The second assumption that we make is that all the data lies within the unit circle. That's a very mild assumption; basically just means we take all our data and we rescale it such that every point has a norm of at most 1. We just do this to make the math easier. Later on you can always rescale it back, and then you actually scale the hyperplane, too, so that's not really an assumption; it's just a mathematical trick to make it easier. All right. Finally let's have one more term, and we call this term . is the distance of this hyperplane w* that we assume exists and the closest point in either one of these two classes. All right; so this is just some term that is a function of the data. OK; so what we will do is we will show that the hyperplane w that we learned with the perceptron algorithm becomes in some sense more and more aligned with this hyperplane that separates the data sets w*. We measure this with w transpose w*. So, the first thing — and I will note details here in the notes — is that we show that w transpose w* grows by at least every single time we make an update. So this w transpose w* measures the alignment, and so that grows every single time. So, after M updates, this term is greater than M times . We can also take w transpose w* and decompose it just by the definition of the inner product. So, w transpose w* is the same thing as the norm of w times the cosine between the two vectors. If you look at the left-hand side — so we've now bounded that and said that's greater/equal than M times . We can now look at the right-hand side. Cosine of , that is at most 1, just by cosine is less/equal 1 for every single angle. And we can also look at the norm of w, and the norm of w essentially measures

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

how long that vector w is. So, think about it intuitively, right? So we say that w transpose w*, the angle between these two vectors — sorry; the alignment between these two vectors grows every single time. That can be because of two reasons. Number one is because they come closer. That means that the cosine becomes larger. The second thing is that if you could just scale w; that would not be very interesting. Because it would define the same hyperplane; it's just that w is longer. Luckily we can also bound that term. So, the norm of w, as it turns out, is just the square root of w transpose w. And we show in the notes that w transpose w grows by at most 1 every single time you make an update. So essentially after M updates, the norm of w is less/equal square root of M because it's the square root of w transpose w. So now we can put all this together. We get M times is less than w transpose w*, which equals the norm of w times the cosine. But these two terms can be bounded by 1 and by square root of M. So what we get is M times is less/equal square root of M. If we solve this inequality with respect to M, we get M is less/equal 1 over squared. And this is a beautiful result. It shows us two things. Number one, it shows us that M is the number of updates we've taken. Cannot be infinitely large; it must be less/equal than some term which is 1 over squared. So, you can only make that many updates, and then the algorithm has to stop. There is no more update to make. The second thing is if you look at 1 over squared, it has a very interesting, intuitive explanation, or interpretation, and that is that is the distance of the hyperplane that we assume exists to the closest point. If this closest point is very, very close to the hyperplane, then is very small. 1 over something very small is large, so that means that there's very little wiggle room for this hyperplane. Then we need many steps to find it. On the other hand, if is large, that means the closest point to the hyperplane w* is far away. That means there's a big margin between the positive and the negative points. There's a lot of wiggle room. Then one over squared is much smaller, and that means we have much fewer steps and the perceptron algorithm will converge very quickly to a separating hyperplane.

Click to check your knowledge of proof of convergence.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Read: Proof of Convergence Details

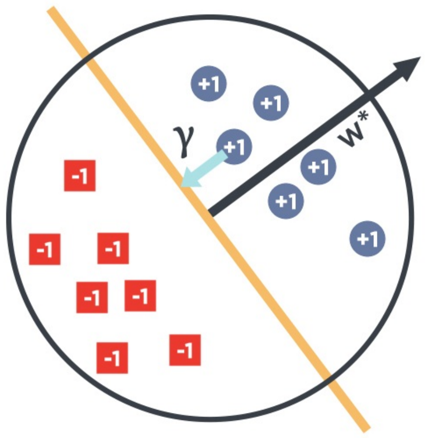
If a data set is linearly separable, the perceptron will find a separating hyperplane in a finite number of iterations.

The number of perceptron updates is finite (i.e. it is bounded from above by a constant).

The perceptron is arguably the first algorithm to have a strong formal guarantee: if a data set is linearly separable (i.e. the two classes can be perfectly split using a hyperplane in n-dimensional space), the perceptron will find a separating hyperplane in a finite number of iterations. If the data is not linearly separable, it will loop forever. To better understand why this is the case, let's take a look at the setup and proof of this guarantee.

Problem Setup

Take a look at the graphic on the right. Suppose there exists a hyperplane that separates the data. Formally, this means for all . For mathematical simplicity, suppose that you rescale each data point and your such that:



Unit Circle

Let us define the **margin of the hyperplane**

/math'> to be this is also the distance to the hyperplane.)

Setup summary:

All inputs live within the unit circle i.e. .
There exists a separating hyperplane defined by , with lies exactly on the unit circle).
is the distance from the separating hyperplane (in blue) to the closest data point.

Mathematical Proof

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Using the setup above, consider the effect of an update (becomes) on the two terms. First, let's establish two properties that must hold:

This holds because is misclassified by since an update wouldn’t occur otherwise. This holds because /mo /msup /math'> is a separating hyperplane and classifies all points correctly.

1) Consider the effect of an update on /mi /msup msup mrow class="MJX-TeXAtom-ORD" mi mathvariant="bold"w/mi /mrow mo#x2217;/mo /msup /math'>.

the distance from the hyperplane defined by /mo /msup /math'> to must be at least /mi /math'>. This follows from the definition of /mi /math'> and means that **for each update, grows by *at least* /mi /math'>**.

2) Consider the effect of an update on

The inequality follows from the fact that:

as an update was required, meaning was misclassified

This means that **for each update wTw grows by *at most* 1**.

Summary of Convergence

In summary, we conclude that after updates, two inequalities must hold:

(1) wTw* >- M gamma
(2) wTw <- M

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Further, we can establish that after updates, we must have:

This allows us to relate and directly in an inequality:

And hence, the number of updates is bounded from above by the constant .

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Read: Perceptron Pseudocode

The goal of your perceptron algorithm is to obtain a weight vector that will define a hyperplane that will separate the data. To do so, you must write code that will find such a hyperplane. Below, you will find pseudocode that illustrates the steps necessary to accomplish the task of finding such a hyperplane.

Pseudocode

```
Initialize $\vec{w} = \vec{0}$                          // Initialize $\vec{w}$. $\vec{w} = \vec{0}$ misclassifies everything.
while TRUE do                                           // Keep looping
    $m = 0$                                             // Count the number of misclassifications, $m$
    for $(x_i, y_i) \in D$ do                           // Loop over each (data, label) pair in the dataset, $D$
        if $y_i(\vec{w}^T \cdot \vec{x_i}) \leq 0$ then // If the pair $(\vec{x_i}, y_i)$ is misclassified
            $\vec{w} \leftarrow \vec{w} + y\vec{x}$     // Update the weight vector $\vec{w}$
            $m \leftarrow m + 1$                        // Counter the number of misclassification
        end if
    end for
    if $m = 0$ then                                     // If the most recent $\vec{w}$ gave 0 misclassifications
        break                                           // Break out of the while-loop
    end if
end while                                               // Otherwise, keep looping!
```

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Implement Perceptron Classifier

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Module Wrap-up: Using Linear Classifiers

In this module, you explored how linear classifiers are implemented to solve classification problems in machine learning. You saw the assumptions made with this type of classifier and how they work to classify data by positioning a hyperplane to separate data into classes. You examined the perceptron, how it works, and the formal guarantee it makes when data is linearly separable.

Back to Table of Contents

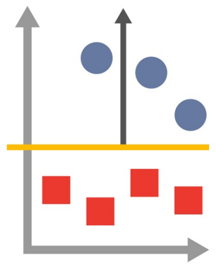
© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Module 2: Linear Regression

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Module Introduction: Linear Regression



Linear regression, also often referred to as ordinary least squares (OLS), is a model similar to the perceptron classifier; however, instead of classification, it can be used for regression. Professor Weinberger introduces the model assumptions and explains how a linear regression classifier can be trained from data. You will review how to find the closed form matrix solution to the minimization problem and will also see linear regression in action through a code demo.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Watch: Linear Regression

Linear regression models the relationship between your labels ( ) and the features ( ) as a linear function (that is learned from the data). Professor Weinberger explains linear regression with an example and illustrates that it uses a Gaussian noise model to account for cases where the linear assumption does not hold exactly.

Video Transcript

Linear regression uses a linear modeling assumption for regression. So in regression, we assume that the label of our data point x can take on any real value. For example, you're trying to predict the value of a house, or the height of a person, or something like this. A linear model describes, once again, a hyperplane, but this time you don't use a hyperplane to separate two classes; instead, we're saying that the label y actually lies on a hyperplane. Just as before, this hyperplane has two parameters; the direction and the offset. And just as before with linear classification, we can absorb the offset as one additional dimension in our x. So, essentially what we're saying is we're saying that the label y is a linear function of x, so y equals w transpose x. And what we would like to do is estimate w from some training samples x and y. Here's a little example: So here we have some data points for some xs, we know exactly the label y, and miraculously they lie exactly on a line. So we would like to find this line that's defined by w. In practice, of course, this is never the case. It's never the case that data lies exactly on a line, right? That would be too easy. There's always some noise, and noise could come from various things like measurement errors or from some factors that are not captured in our features; there may be some other additional things that we didn't actually account for. So what we do is we have a noise model, and what does a noise model do? A noise model basically says, OK, we make our modeling assumption, the data should lie on a line; it doesn't always lie on a line. And when it's not, it's off by a little bit, and we model that by some probability distribution. The simplest noise model is the Gaussian noise model, where you basically say we just put a Gaussian around this line in the vertical — you know, in the y axis, and we say it could be a little off — it could be a little off; it's concentrated around exactly the position on the line, and the Gaussian-based distribution says it can't be too far off because that would become very unlikely. Essentially, what we are doing is we are defining a probability distribution P of y given x and w, where we say that's a Gaussian distribution that's centered around x transpose w; that's the line, and it becomes less and less likely the further off we are from that line.

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Click to check your knowledge of linear regression.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Watch: Employ MLE for Linear Regression

Once we have established our assumptions (i.e. the label is a linear function of the features with additional noise), how do we find the slope of this linear function? Professor Weinberger explains how to use one method, maximum likelihood estimation, to estimate this slope by maximizing the likelihood of your data given the modeling assumption.

Video Transcript

So for linear regression, we made the assumption that the data is drawn from some line or hyperplane, plus some additional noise that is Gaussian and has some fixed variance. How do we find the slope of this hyperplane? And the way we do this is to use maximum likelihood estimation, which we've already seen before. So essentially our model is that we have this data and we would like to find this line with many, many little Gaussians. Every single data point basically corresponds to a point drawn from a Gaussian distribution around this hyperplane. We would like to position the hyperplane such that the data that we observed is as likely as possible. So in other words, every single one of the data points should be in the thick part of the Gaussian. So, what maximum likelihood estimation does is it basically says, OK, we take the product of all different points xi, yi, that we have in our training data set, and we want to maximize this term; We want to maximize the probability of yi given xi and with respect to w. So we treat w such that the data we observe becomes as likely as possible. As you've seen before several times already, when we take a product of many, many probabilities, that's awkward. So what we do is we just take the log of the function. Taking the log of a function that we are maximizing doesn't change anything. The maximum is still in the same location. So we take the log of this product; that becomes the sum of log probabilities. The probabilities are actually these Gaussian distributions, so we can just plug this in. The definition of a Gaussian distribution is basically the normalizer times e to the minus w transpose xi minus yi squared over the variance term squared. So we plug that in. And that's, again, a product of two terms. If we take the log of this, we get the sum of two logs. The first log term is log over 1 over square root of 2π sigma squared. That's actually just a constant, right; it does not contain w in any way. We are trying to maximize this whole expression with respect to w. This is independent of w, so we can just drop it. We can just maximize just the right-hand term with respect to w. The right-hand term is also interesting because it's a log of e to the power of something. Log of e to the power of something is actually just the "something" because log of e is the identity function; the inverse of each other. So what we get is just that we are maximizing the sum over all the inner products with a

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

different square differences between the inner products and their labels squared. And because maximizing a negative term is awkward, so we just flip it around and multiply the whole thing by minus 1 and minimize just the difference between these squares. And this actually now has a very nice interpretation. So essentially what we are doing — and we can add a 1 over n term in the front to make it an average — we are minimizing the average square difference of our predicted value and the label. And this comes out of our Gaussian assumption. So, if we look back at our data point, basically we have all these blue points that we are trying to fit, and the red line here is the difference between the actual label y, or the blue point, and the predicted one that we get along the line. And what we are trying to minimize is the sum of all these red lines squared. That's essentially what it is. Another interesting interpretation of this is actually that you could view these red lines as springs. If you kind of see every single data point as a steel ball and you had a line that is attached to the steel ball with string, then you would like to find the position where you have minimum amount of spring load. And that actually is exactly the same expression. So, what we are trying to do is we're going to minimize this expression that we also called the square loss function. And you will see in later videos how to do that.

Click to check your knowledge of MLE for linear regression.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Read: Linear Regression MLE

The maximum likelihood estimator maximizes the likelihood of the observed data given the model assumptions.

The MLE expression is mathematically equivalent to the minimization of the mean squared error.

As a starting point for linear regression, we assume that the labels are a linear function of the corresponding features with additive Gaussian noise :

In other words, we assume that the labels are sampled from the following conditional distribution:

Here, the Gaussian distribution samples the noise around the mean . The hyperparameters specify the variance of the noise distribution. Given such a probability distribution, we need to find the parameter vector that best explains our data. To this end, we can deploy MLE, which will seek the that maximizes the likelihood of our observed labels (given their feature vectors):

This derivation shows that the MLE estimate of is the minimizer of the following **loss function**:

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

This particular loss function is also known as the squared loss or **ordinary least squares (OLS)**. OLS can be optimized with gradient descent, Newton's method, or in closed form (i.e. solving for the exact analytical solution). It has a nice explanation: the linear prediction should minimize the squared error from the true label .

Regularization

To simplify models that have a large number of features, we can use regularization techniques such as **L1 regularization** (also known as LASSO regression) and **L2 regularization** (also known as Ridge regression). By adding these regularization terms, which are functions of the weights, we can mitigate overfitting by "punishing" a model with extreme values for any given weight.

L2 regularization introduces a regularization term to the loss function that is the sum of squares of all feature weights. L1 regularization introduces a term that penalizes less important features, reducing their coefficients to zero and effectively eliminating them.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Watch: Matrix Solution to Minimization Problem

We previously showed that linear regression involves minimizing the squared loss, the sum of squared differences between each prediction and the corresponding true label. The squared loss is a quadratic function (i.e. a paraboloid in high-dimensional space). Professor Weinberger demonstrates how we can derive a closed form solution if we change the loss to matrix notation.

Video Transcript

So in linear regression, we end up with a square loss, which is essentially a sum, over the squared difference, over every single prediction for a data point and the actual label. So the prediction is xi transpose w, and the label is yi. Turns out that's a sum over many quadratic functions, and it's itself a quadratic function. In a high-dimensional space, that's a parabola. And these are really nice to minimize. Why is that? Because, number one, they're convex; and number two, they have a single minimum that we can attain in closed form. And I can now show you how to do this. For this, we change our notation a little bit. We design a new matrix X, where every single row is one feature vector. So the ith row of X is the vector Xi. Then we create a vector y, where the ith entry of y is yi. So it's the ith label. So basically we take all the labels of our endpoints, concatenate them into one big vector. That's the vector y. So X contains all the features, y contains all the labels. And now we can rewrite the loss function in terms of matrix algebra as X times w minus y squared and averaged. And so we basically have this big matrix X, and when we multiply with w, for every single dimension, we compute Xi transpose w, and we subtract y from it. So how do we obtain the minimum of this function? Turns out, it's not very complicated. We write out the square, we take the first derivative, we set it to 0, and solve with respect to w. And that's exactly what we can do now. So the first thing is you write out the square, so then we get w transpose X transpose X times w, minus twice the cross-term y transpose Xw, plus y squared. Now, if you take the derivative, we get X transpose Xw, minus X transpose y. And if we equate it to 0, we can solve it with respect to w and obtain w equals the inverse of X transpose X, times X transpose y. And now we have the solution for w in a single line of number.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Read: Matrix Solution to Minimization Problem

If you use MLE to fit your model, you find the weight vector that minimizes the mean squared error of the predicted *y* values.

To find the minimum, take the gradient of this expression with respect to weight vector *w* and set it to zero.

Solving for *w*, you get a final expression in terms of *X* and *y* that can be easily evaluated with matrix multiplication in NumPy.

The MLE solution for linear regression, the weight vector , can be solved analytically using the label vector and data matrix . Recall that the objective with MLE was to find:

–

Before you can derive a closed form solution, you must first change this optimization to matrix notation. Stack your data points horizontally to form an /mo mid/mi /math'> data matrix and your labels into a single vector:

,

This notation allows us to rewrite the squared-loss objective function as:

–

Closed Form Solution

To start, let’s expand the objective function:

You can find the minimizer of this final term by taking the gradient with respect to and equating the resulting expression with zero, namely:

Using some elementary algebra, you can obtain a closed form matrix solution for . If your multivariate calculus is a little bit rusty, you might find the matrix cookbook to be helpful.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Tool: Linear Regression Cheat Sheet

Use this Linear Regression Cheat Sheet as a quick way to review the details of how linear regression works.

This tool provides an overview of linear regression for your reference. You'll find information about the applicability, underlying mathematical principles, assumptions, and other details of this modeling approach.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Code: Linear Regression Demo

Linear Regression Demo

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Module Wrap-up: Linear Regression

Linear regression assumes that the label is a linear function of the feature vector with additive Gaussian noise. The slope of this linear function can be estimated with maximum likelihood estimation, which results in the minimization of the squared loss function. This function has a closed form solution that can be computed in just a few lines of NumPy code. Linear regression is a powerful regression tool that is often the first approach data scientists try when they are faced with a regression problem.

Back to Table of Contents

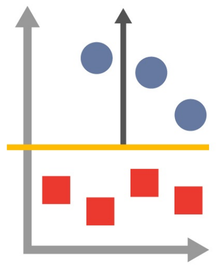
© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Module 3: Logistic Regression

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Module Introduction: Logistic Regression



Logistic regression is a linear classifier with modeling assumptions related to the multinomial Naive Bayes classifier. While fitting data in a linear regression model, logistic regression allows you to estimate the probability of a categorical label. You will review how multinomial Naive Bayes works and will be introduced to the sigmoidal function that turns real valued numbers into a "probability." Professor Weinberger explains the differences and similarities of logistic regression and multinomial Naive Bayes.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Watch: Linear Classification with Naive Bayes

You've seen the Naive Bayes classifier before. Professor Weinberger explains why the decision boundary generated by the Naive Bayes classifier makes it a linear classifier.

Video Transcript

So a while ago, we talked about the Naive Bayes classifier. And in the Naive Bayes classifier, we first estimated the probability of x given y for every single dimension separately, assuming some distribution, and we estimated P of y; we multiplied those together to get P of y given x. And depending on what distribution we assume — for example, if we use Gaussian or a multinomial distribution — it turns out that's actually a linear classifier. And I can quickly illustrate to you why that is the case. So, assume you have some data points, here red and blue, and what Naive Bayes does is, the first thing is it estimates the distribution of x given y in one dimension; so, for example, the horizontal dimension. And so here we make a Gaussian assumption. So this is Gaussian. Then it does the same thing in the vertical dimension. Again, we make a Gaussian assumption, so we fit a Gaussian distribution here. And then to get the high-dimensional estimate of P of x given y, we just multiply those together, and then we get the probability of x given y for any given point in this space. And so what you see here, these Easter eggs are really actually areas basically of one standard deviation of where you have equal probability of being drawn from that distribution. What's interesting is the point where it's exactly equally likely that you come from the red distribution or from the blue distribution, that's the decision boundary. That's basically — if you go a little bit to the right, you're classified as one class; and if you go a little bit to the left, you're classified as the other class. Turns out if you have two Gaussian distributions, this area is a hyperplane itself. So, what that means is that Naive Bayes, if you assume that you have a Gaussian equal variance in every single dimension, is a linear classifier. Similarly, that's the case for a multinomial distribution.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Watch: Logistic Regression

Linear regression is a linear classifier with modeling assumptions similar to the multinomial Naive Bayes classifier. Logistic regression allows you to estimate the probability of a label, given the data. You will review how multinomial Naive Bayes works and will be introduced to the sigmoidal function to turn any real value into a "probability". Professor Weinberger explains the differences and similarities of logistic regression and multinomial Naive Bayes.

Video Transcript

Logistic regression is highly related to Naive Bayes. In some sense, what we're doing is, we make similar modeling assumptions. However, instead of estimating P of x given y, we estimate P of y given x directly. And here's what happens. So, when you do Naive Bayes with the multinomial or with the Gaussian distribution, then, as we established earlier, you actually get a linear classifier. Let's just walk through this one more time, just to explain it. So, we have these — we estimate P of x given y, let's say in one dimension, using a Gaussian distribution. Assume we just have a one-dimensional data set; then we basically have P of x given y for both classes. And now for a new test point, I would like to know what's the probability that the class label is 1 or the class label is minus 1. And if you look at this, well. if I'm in the blue region, where all the blue points are, I'm much more likely to be drawn from the blue Gaussian, like the blue points, and therefore the probability of the negative label is much more likely. When I, however, go away from this and I get into the red territory, it's much, much more likely that my data point was drawn from the red distribution. And if we now look at the probability distribution of y given x — and let's say probability of y is positive given x — then you see that actually in the blue region this is almost 0, but as then we get closer and closer to the red region, this probability goes up and goes more and more closer to 1. And this function is a sigmoidal function. So, sigmoidal function basically is 0 on one limit and 1 on the other limit and kind of at some point has this S shape where it hits 0.5 exactly where the decision boundary is. There you are kind of undecided; it's equally likely that you have the red distribution over the blue distribution, and then you take off and become more and more likely that it's positive. The sigmoidal function is P of y given x is 1 over 1 plus e to the minus y w transpose x, where y is either plus 1 or minus 1. What we do in logistic regression is that we assume this function from the get-go. So instead of arriving there and basically doing our modeling assumptions for P of x given y and so on and at the end we arrive at the sigmoidal function, we assume the sigmoidal function from the get-go and now we just fit w with maximum likelihood estimation, similar to the linear regression model. So

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

essentially what we are saying, if you say, OK, well, we have a form of P of y given x, right, different from linear regression; we are no longer saying it's Gaussian. We are saying it has the following form; it has the sigmoidal form, 1 over 1 plus e to the minus y w transpose x, which is what you would get with the Naive Bayes classifier. But now we are tweaking w to make our data as likely as possible. And how we do this, well, we once again use maximum likelihood estimation. So, we go over our data and we maximize the product P of yi given xi for all our data points, and that is exactly what we do with logistic regression. The difference between logistic regression and Naive Bayes is that Naive Bayes makes a much, much stronger assumption about the data; basically assumes that P of x given y is, for example, Gaussian distributed. That can be good or bad. It can be good if you have very little data, because this way you can put some additional information into your model and it can help you learn quicker. If you have a lot of data, however, then it's probably a bad thing, because the probability of x given y is not exactly Gaussian, right? So, logistic regression, you actually learn P of y given x and you are much more flexible. So, if you have a lot of data, typically logistic regression works better. If you have very little data or you have a very precise idea what the distribution of x given y is, then Naive Bayes can be better.

Click to check your knowledge of logistic regression.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Watch: Generative vs. Discriminative Models

Logistic regression and Gaussian Naive Bayes lead to identical functional forms for the conditional label probabilities . However, logistic regression fits this term, , directly from the data whereas Gaussian Naive Bayes fits from the data (and uses Bayes Rule to then obtain). We call Naive Bayes a generative model and logistic regression a discriminative model. Professor Weinberger explains how these classifiers work for different sets of data.

Video Transcript

Logistic regression is highly related to Gaussian Naive Bayes. Both of them end up with very similar conditional probabilities of P of y given x; the probability of the label given x. At least these distributions have a very similar form. However, there's a crucial difference, and that is that Naive Bayes estimates P of x given y with maximum likelihood estimate, whereas logistic regression estimates P of y given x. We call the first one a generative model, the second one a discriminative model. Let me show you where this difference could come in. So, let me generate a data set here that I will then feed into the Gaussian Naive Bayes classifier. Here I'm drawing data points from the negative class and I'll make it nice and Gaussian. And now I draw data points from the positive class, and I make these also nice and Gaussian except I do something evil. I move two points, or maybe three points here, off to the side. That's extremely unlikely, right; if they really were drawn from a Gaussian distribution, this would never happen that three points are so far off, kind of all together. So our modeling assumption that probability of x given y is Gaussian doesn't really apply here. So what does Naive Bayes do in this kind of scenario? Well, if you run it, here is what you see. So, now you see all the red region is the region that Naive Bayes classifier classifies as positive; the blue region is the region that it classifies as negative. And what happens? It takes the data from these two different classes, estimates them both with a Gaussian in each dimension independently, and then it takes these two Gaussian distributions — ignores the data; just takes the two Gaussian distributions — and tries to find a hyperplane that separates these two Gaussian distributions from each other. That's what you get with a sigmoidal. So, the only thing we're estimating is the Gaussian distributions; after that, everything is on rails. After that, once you have two Gaussian distributions, there is a hyperplane that automatically basically specifies exactly when does one distribution become more likely than the other. Now, here's the key. When we estimate P of x given y, these outlier points don't really come in very much; they shift the Gaussian a little bit over into this region, but we will end up misclassifying these points. Because if you have a Gaussian here and a Gaussian here, the middle

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

point is somewhere here in the middle, right? And the Naive Bayes classifier will essentially sacrifice these three points. Logistic regression, on the other hand, will do something else. It estimates P of y given x for every single data point; it makes no explicit assumption that has to be Gaussian. And so what it will do, it goes through every single data point and say what's the probability that this data point really has the label that it has? Now, the current solution will actually be pretty bad, because these xs here are actually positive, but under the current model, they should be negative. Right? They are deep into the blue region. So this is not the optimal solution for logistic regression. So if we keep optimizing logistic regression, you will see that this decision boundary actually starts moving, and it moves around to get these viewpoints correctly classified and will find a separating hyperplane that actually classifies all the points correctly. So in summary, Naive Bayes and logistic regression are highly related. Both end up with a sigmoidal posterior distribution of P of y given x, w, but the way they find them is different. Naive Bayes models estimate in Gaussian distributions and then finds P of y given x from these Gaussian distributions, whereas logistic regression does it directly from the data. Naive Bayes is better if your modeling assumption is correct and if you have very little data; then actually this additional Gaussian helps you require less data to train the classifier. On the other hand, if your distribution via assumption is not met, if they're not truly Gaussian, or if you have a lot of data so that this modeling assumption is unnecessary, then logistic regression is more powerful and tends to get the better classifier.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Watch: Employ MLE for Logistic Regression

In logistic regression, we make a modeling assumption that is a logistic function. Once again we try to use maximum likelihood estimation to fit this hyperplane, finding the vector that makes the data we observe as likely as possible. Professor Weinberger explains why this time finding the MLE minimizer is not as easy as for linear regression.

Video Transcript

So in logistic regression, we make a modeling assumption over the probability of y, the label, given x and w. And we assume it's a logistic function. How do we fit this w? And once again we use maximum likelihood estimation. So essentially we try to find the w that makes the data that we observed as likely as possible. So essentially what we're doing is we take the product over all the other points; the product over all P of yi given xi, w. And you want to maximize this term. If you plug in our sigmoidal function and take the log, we get the following expression. And because we prefer minimizing terms as just a convention, we obtain that — the last function that we get is the minimum over the sum, over all these log terms, of 1 plus e to the minus yiw transpose x. This function is still a nice function to minimize, but it's not as nice as we had in a linear regression. In linear regression, if you remember, we could actually get a parabola if you could just jump right to the minimum. In this case we can't do this. Here it's still a sum over many different functions, but they are not quadratic. They look a little different; they kind of look like either this function or that function. And if you sum many, many — but each one of these is convex. If you sum up many, many convex functions, what you get is itself a convex function. And actually, the sum over these functions essentially ends up being a little bit like a salad bowl, where it's round at the bottom but kind of straight on the outsides. And how we minimize this function is not obvious right now, right? We know there's a single minimum. It's a convex function so it should be nice and behave if there is a minimum. But we can't just find it in closed form; we have to use something else, and we will learn this very soon.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Read: Logistic Regression MLE

As part of our modeling, we assume an expression for the probability of the observed labels given the data and parameter vector *w*.

We apply MLE and minimize the log-likelihood of the data.

The result is an expression that can be optimized using gradient descent.

Assume and we model as:

(Note: To make life easier, we assume that we absorb the parameter into through an addition constant dimension.)

To derive the MLE solution for logistic regression, we need to first construct the log-likelihood, namely, , where and (stacking the features horizontally so that each column is a different feature and each row is a data point). Using our assumption that our data points are independent and identically distributed, we know that the likelihood is:

Consequently, the log-likelihood becomes:

So, our MLE solution for logistic regression is:

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Unlike linear regression, logistic regression has no closed form solution (i.e. there is not an analytical expression for the optimal weight vector). Fortunately, the objective function is convex, so we can make use of gradient descent to find the optimal solution. (We will cover gradient descent in the next module.)

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Tool: Logistic Regression Cheat Sheet

Use this Logistic Regression Cheat Sheet as a quick way to review the details of how logistic regression works.

This tool provides an overview of logistic regression for your reference. You'll find information about the applicability, underlying mathematical principles, assumptions, and other details of this modeling approach.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Activity: Explore the Sigmoid Function

In logistic regression, we assume that the conditional label distribution takes on the following form:

where the sigmoid function is defined as:

/mi mo stretchy="false"(/mo mis/mi mo stretchy="false")/mo mo=/mo mfrac mn1/mn mrow mn1/mn mo+/mo msup mie/mi mrow class="MJX-TeXAtom-ORD" mo#x2212;/mo mis/mi /mrow /msup /mrow /mfrac /math'>

You will explore the mathematical behavior of the sigmoid and discover properties that will prove useful in the optimization step. Answer the two questions that follow below. Once you think you've determined the solutions, click the buttons to reveal the answers.

1. Show that the sigmoid function
 By proving this property, you show that you have a properly defined a probabilistic model, namely:

Click to reveal the answer

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

2. Show the following property for its first derivative: /mi mo#x2032;/mo /msup mo stretchy="false"(/mo mis/mi mo stretchy="false")/mo mo=/mo mi#x03C3;/mi mo stretchy="false"(/mo mis/mi mo stretchy="false")/mo mo stretchy="false"(/mo mn1/mn mo#x2212;/mo mi#x03C3;/mi mo stretchy="false"(/mo mis/mi mo stretchy="false")/mo mo stretchy="false")/mo /math'>.

Click to reveal the answer

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Where Are We Stuck?

Introduction/Scenario:

In the previous activity, you explored two different properties of the sigmoid function. Now you'll discuss the implications of these properties for optimization.

Discussion topic:

Explain why the loss is convex. (Take a look at the second derivative.)
What function does the loss approximate as   becomes large?

Instructions:

You are required to participate meaningfully in all course discussions.
Limit your comments to 200 words.
You are strongly encouraged to post a response to at least two of your peers' posts.

To participate in this discussion:

Click **Reply** to post a comment or reply to another comment. Please consider that this is a professional forum; courtesy and professional language and tone are expected. Before posting, please review eCornell's policy regarding **plagiarism** (the presentation of someone else's work as your own without source credit).

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Module Wrap-up: Logistic Regression

In this module, you were introduced to logistic regression and how it relates to the Naive Bayes classifier. Professor Weinberger discussed what assumptions these models make and under what circumstances one may be preferred over the other. He explained how maximum likelihood estimation is used to derive the logistic loss function to fit the separating hyperplane in logistic regression.

Back to Table of Contents

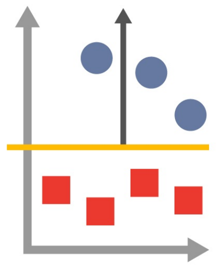
© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Module 4: Gradient Descent

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Module Introduction: Gradient Descent



Both linear regression and logistic regression have an associated loss function, which you can minimize with gradient descent. Professor Weinberger introduces gradient descent and provides an argument based on Taylor’s approximation to show why it minimizes the function. You will compute the gradient update of the logistic loss and will have the opportunity to apply what you have learned in this course in order to build a linear classifier for email spam classification.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Watch: Minimize a Function with Gradient Descent

Gradient descent is a simple, effective way to minimize a function with respect to some parameters. Gradient descent works best with a convex function, which has only one global minimum. Gradient descent converges if applied correctly. Professor Weinberger provides an example.

Video Transcript

A simple yet very effective way to minimize a function ℓ with respect to some parameters w is gradient descent, often also referred to as steepest descent. Gradient descent works best if a function is convex. By "convex" we mean that it only has global minimum. So, you can imagine the function as kind of bathtub-shaped. If you were to take this function and fill it with water and put a little hole at the lowest point, all the water would drain. In contrast, a non-convex function could have local minimum. And if you were to fill it with water and drain it, water would stay trapped in these local minimum. If the function is convex, that means it does not have any local minimum, then gradient descent will provably converge to the global minimum if applied correctly. I'll walk you through how it works in a little example. So let's say we have this function that's kind of a bathtub shape in 3D space. We can look at it from above and then we can create a contour plot. A contour plot shows us lines that all have the same depth. So, for example, if you were to fill your bathtub with water, these lines will be water levels. Gradient descent works as follows: We initially create some parameter vector w that we just make up. That's our initial parameter guess. So, for example, you could just say w is all 0s. If the function is convex, it doesn't matter what w is initially because it will always converge to the global minimum. What gradient descent does is that any given point we compute the gradient of the function ℓ; so the gradient of ℓ with respect to w. The gradient points the direction of maximum ascent. So that's the direction you're going to have to follow if you want to maximize the function; if you want to go uphill. That's not what we want; we want to go downhill. So what we do is we go in the opposite direction. That's the negative gradient. And now we update our parameters; once we know which direction points downhill, we just say, well, w becomes w minus alpha times the gradient. So essentially we're taking a step down that direction. We update our w, and alpha is the step size. So if alpha is large, that means we take a large step downhill; and if alpha is small, we take a tiny little step. Gradient descent converges if alpha is small enough, and then you have take many, many teeny little steps until you get to the minimum. However, if alpha is too small, it can take a really, really long time. In contrast, if you make alpha larger — and you as a data scientist can choose what your alpha is. If alpha is larger, then you will get to the

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

vicinity of the minimum quicker, but there's the danger that at the minimum we would just bounce back and forth between the different sides of the bathtub and never actually get to the minimum. And therefore a good rule is to lower alpha as you keep optimizing. A good rule is to set alpha equals 1 over t, where t is the number of steps you've taken so far. This way, you take large steps at the beginning until you get close to the minimum, and then your steps get smaller and smaller then converge very nicely.

Click to check your knowledge of gradient descent.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Read: Gradient Descent Formalized

Use a Taylor expansion to show that stepping in the direction of steepest descent always reduces the loss function.

Setting the learning rate parameter α too high prevents the algorithm from converging.

A popular option is to decrease the step-size α with each step

In this section, our goal is to minimize a convex continuous and differentiable loss function . One way to achieve this is using gradient descent (aka "steepest descent"). By Taylor's expansion, we can approximate , which is the function value after we take a small update from our previous location along the direction (but note this only works if the step is small):

Where is the gradient of the function

.

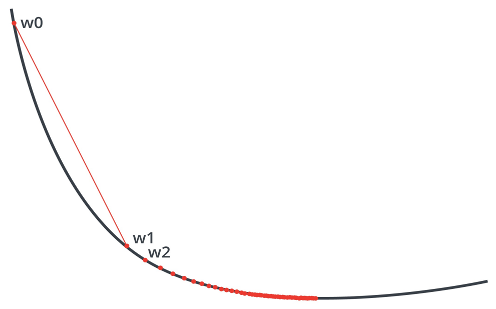
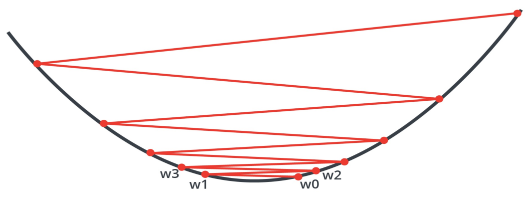
Essentially, we assume that behaves linearly around and we want to take a step so mig/mi mo stretchy="false"(/mo mrow class="MJX-TeXAtom-ORD" mi mathvariant="bold"w/mi /mrow mo stretchy="false")/mo /math'>, for some small /mi /math'>0. It is straightforward to prove that taking a step along this direction reduces the loss value, namely,

is not always an exact science. This proof only holds if the learning rate is sufficiently small such that the gradient descent will converge. (See the first figure below.) If it is too large, Taylor's approximation no longer holds and the algorithm can easily *diverge* out of control. (See the second figure below.) A safe (but sometimes slow) choice is to set

Note: Setting the learning rate

, which guarantees that it will eventually become small enough to converge (for any initial).

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Gradient Descent Convergence	Gradient Descent Divergence
	

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Tool: Gradient Descent Cheat Sheet

Use this Gradient Descent Cheat Sheet as a quick way to review the details of how gradient descent works.

This tool provides an overview of gradient descent for your reference. You'll find information about the applicability, underlying mathematical principles, assumptions, and other details of this algorithm.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Build a Spam Email Classifier

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Perceptron as a Spam Filter

In the previous activity, you used gradient descent to build an email spam filter. Now we'll consider the viability of using a linear classifier introduced previously: the perceptron.

Discussion topic:

Would the perceptron be well suited for an email spam filter? Why or why not? (*Hint: Think about how the different algorithms deal with misclassified points.*)

Instructions:

You are required to participate meaningfully in all course discussions.
Limit your comments to 200 words.
You are strongly encouraged to post a response to at least two of your peers' posts.

To participate in this discussion:

Click **Reply** to post a comment or reply to another comment. Please consider that this is a professional forum; courtesy and professional language and tone are expected. Before posting, please review eCornell's policy regarding **plagiarism** (the presentation of someone else's work as your own without source credit).

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Module Wrap-up: Gradient Descent

In this module, you saw how gradient descent minimizes loss functions in a simple, effective way. It is particularly effective for convex functions with only global minima. You also reviewed Taylor’s approximation to see why gradient descent reduces the loss with each step. Then you implemented your own logistic regression classifier and used it to build an email spam filter.

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Read: Thank You and Farewell

Kilian Weinberger
Associate Professor
Computing and Information Science

Cornell University

Congratulations on completing "Learning with Linear Classifiers."

Linear classifiers are extremely powerful — especially with high dimensional data, such as text data. I hope that you now recognize the power and potential that machine learning offers and the material covered here has met your expectations. Hopefully you will find many applications of machine learning in your own work context.

From all of us at Cornell University and eCornell, thank you for participating in this course.

Sincerely,

Kilian Weinberger

Back to Table of Contents

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.

Glossary

Gaussian Naive Bayes

A specific instance of a Naive Bayes classifier that that models the conditional distribution $P(x|y)$ of a feature x given the label y with a Gaussian distribution for each of the features (continuous-valued).

Gradient descent (steepest descent)

An optimization algorithm that finds the minimum of a function by moving in the direction of the steepest descent (negative gradient).

Global minimum

A lowest (and typically optimal) point of a loss function.

Hyperplane

An (n-1)-dimensional subspace of a d-dimensional vector space that is the solution to a linear equation and is used to separates classes within a data set.

Linear classifier

A machine learning classifier that separates two classes with a hyperplane.

Linear regression

A regression model for predicting the value of y for a given feature vector $\mathbf{x}$ for continuous data, using a straight line.

Logistic regression

A linear classifier for binary classification problems. The separating hyperplane is obtained by minimizing the logistic loss (log-loss). One advantage of Logistic Regression is that it outputs well calibrated probabilistic certainties about a predicted class label.

Multinomial Naive Bayes

A specific instance of a Naïve Bayes classifier that uses a multinomial distribution for each of the features (categorical).

Perceptron

The first linear classifier, invented at Cornell in 1957 by Frank Rosenblatt. It positions a hyperplane if one exists and corrects the position with a finite number of updates.

© 2019 eCornell. All rights reserved. All other copyrights, trademarks, trade names, and logos are the sole property of their respective owners.