



Linear regression

The two main families of algorithms and predictive machine learning are:

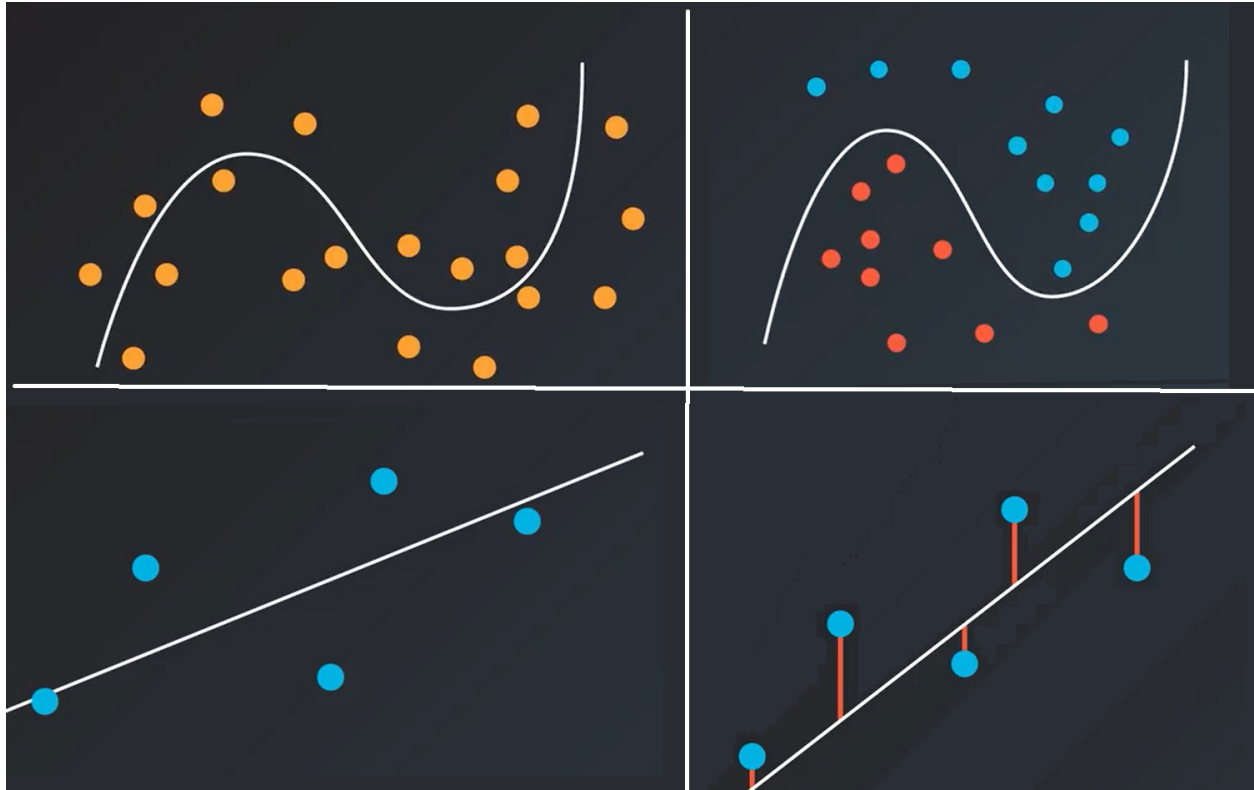
- **Classification**

- Classification answers questions of the form yes-no. For example, is this email spam or not, or is the patient sick or not.

- **Regression**

- Regression answers questions of the form how much. For example, how much does this house cost? Or how many seconds do expect someone to watch this video?

At the highest level, the entire topic of linear regression is based around the idea of trying to draw a line (called 'fitting') through an entire dataset of points. The algorithm uses the value of every point in the dataset to find the optimum line equation. Ultimately, the equation of that line can be used to plot new data. There are tricks and techniques for doing this. Sometimes this is perfectly clear and other times it is quite challenging, all based on the type of data in the dataset. Sometimes we want to split the data into different groups, other times we just want a line down the 'middle'. But rarely, if ever, is it perfect, much like data in real life. There are ways to determine how close you are getting, or whether your line is in the best possible place, in the best possible shape, and at the best possible slope. All pages in this lesson are centered around these ideas.



Linear Regression

Lesson outline

In this lesson, we will focus on regression. What do you need to know about regression? We will talk about each of the following throughout this lesson.

- Fitting a line through data
- Gradient descent
- Errors
- Linear regression in Scikit-learn
- Polynomial regression
- Regularization
- Feature scaling

Learning Objectives

By the end of the Linear Regression lesson, you should be able to

- Identify the difference between Regression and Classification
- Train a Linear Regression model to predict values
- Predict states using Logistic Regression

Fitting a line through data:

Here is a refresher on how we move lines by changing the parameters. Assume we have a line represented by the equation $y = w_1x + w_2$, where w_1 is a constant and represents the slope and w_2 is a constant and represents the y-intercept.

- If we increase w_1 , we increase the slope so the line rotates counterclockwise.
- If we decrease w_1 , we decrease the slope so the line rotates clockwise.
- If we increase w_2 , the line maintains its slope but moves up.
- If we decrease w_2 , the line maintains its slope but moves down.

Absolute Trick

The absolute trick works by starting with a point and a line.

- A point with coordinates (p, q)
- A line represented by $y = w_1x + w_2$.

To move the line closer to the point (p, q) the absolute trick has two steps:

- add to the y-intercept so that the line moves up
- add to the slope to make the line rotate in the direction of the point

The first possible step is to add 1 to the y-intercept and pp to the slope, giving us the equation $y = (w_1 + p)x + (w_2 + 1)$

This ends up being too large of a step, and we have over-corrected our line. Instead, we will utilize a small number called a **learning rate**, referred to as alpha (α), to take smaller steps.

We will add (1α) to the y-intercept, and $(p\alpha)$ to the slope. This gives us the equation

$$y = (w_1 + p\alpha)x + (w_2 + \alpha)$$

This works when the point (p,q) is above the line, but when the point is underneath the line we need to **subtract** in order to move the line appropriately.

$$y = (w_1 - p\alpha)x + (w_2 - \alpha)$$

Example

Let's say we have the point (5, 15) and the line $y = 2x + 3$ and a learning rate of 0.1.

- Update the slope: We take 5 and multiply it by 0.1 and add to the existing slope, giving us a new slope of 2.5
- Update the y-intercept by adding 0.1, giving us a new y-intercept of 3.1

This means our new equation is $y = 2.5x + 3.1$

If our point was (-5, 15) we would add 0.1 to the y-intercept to move the line upwards, but we update the slope by multiplying -5 by 0.1, or -0.5. This means our new equation is going to be $y = 1.5x + 3.1$.

| The absolute trick is used extensively in linear regression.

Square Trick

If we have a point that is close to a line, then the distance is small and we want to move the line very little. If the point is far from the line, they want to move the line a lot more. The absolute trick does not take into account how far the point is from the line.

The square trick addresses this.

Let's look at this vertical distance between the point and the line. The point over the line has coordinates (p,q) and the corresponding point on the line is (p, q')

The distance between the point and the line is $(q - q')$

We take this distance and multiply it into what we add to both the y-intercept and to the slope.

- Update the y-intercept by adding $\alpha(q - q')$
- Update the slope by adding $p\alpha(q - q')$

This gives us the equation

$$y = (w_1 + p(q - q')\alpha)x + (w_2 + (q - q')\alpha)$$

This trick automatically takes care of points that are under the line and we don't need two rules as we had on the absolute trick. We just have the same rule for both.

Example

Let's say we have the point $(5, 15)$ and the line $y = 2x + 3$ (which gives us $q' = 13$) and a learning rate of 0.01. Notice that this learning rate is smaller than the example that we used for the Absolute Trick.

- Update the slope: We take 5 and multiply it by 0.01, and then multiply this value by 2 before adding the result to the existing slope, giving us a new slope of **2.1**
- Update the y-intercept by adding $0.01 \cdot 2$, giving us a new y-intercept of **3.02**

This means our new equation is $y = 2.1x + 3.02$

That's the square trick.

Gradient Descent

The tricks we just learned still seem a little too magical, and we'd like to find their origin.

Let's say we have our points on our plan is to develop an algorithm that will find the line that best fits this set of points. And the algorithm works like this,

First, draw a random line, and calculate the error. The error is some measure of how far the points are from the line, in this drawing, it looks like it's the sum of these distances but it could be any measure that tells us how far we are from the points.

Now we're going to move the line around and see if we can decrease this error.

- We move in this direction and we see that the error kind of increases so that's not the way to go.
- We move in the other direction and see that the error decreased, so we pick this one and stay there.

Repeat these steps many times over and over every time descending the error a bit until we get to the perfect line.

To minimize this error, we're going to use something called **gradient descent**.

Descending from Mt. RainiError

To build your intuition of what gradient descent is, imagine we're standing on top of a mountain called Mt. RainiError as it measures how big our error is, and in order to descend from this mountain, we need to minimize our height.

On the left, we have a problem fitting the line to the data, which we can do by minimizing the error or the distance from the line to the points. Descending from the mountain is equivalent to getting the line closer to the points.

If we want to descend from the mountain we would look at the directions where we can walk down and find the one that makes us descend the most. Bit by bit, we descend a bit in this direction, this is equivalent to getting the line a little bit closer to the points.

So now our height is smaller because we're closer to the points since our distance to them is smaller. Again and again, we look at what makes us descend the most from the mountain.

Now we're at a point where we're descended from the mountain and on the right, we found the line that is very close to our points. Thus, we've solved our problem and that is gradient descent.

The mathematical definition

Consider the plot in two dimensions allowing a reality that the plot will be in higher dimensions. We have our weights on the x-axis and our error on the y-axis. And we have an error function. The way to descend is to actually take the derivative or gradient of the error function with respect to the weights. This gradient is going to point to a direction where the function increases the most. Therefore, the negative of this gradient is going to point down in the direction where the function decreases the most. So what we do is we take a step in the direction of the negative of that gradient, this means we are taking our weights w_i and changing them to w_i minus the derivative of the error with respect to w_i .

In real life, we'll be multiplying this derivative by the learning rate since we want to make small steps. This means the error function is decreasing and we're closer to the minimum. If we do this several times we get to either a minimum or a pretty good value where the error is small. Once we get to the point that we've reached a pretty good solution for our linear regression problem.

The two most common error functions for linear regression are:

- Mean absolute error
- Mean squared error

Mean Absolute Error

Assume we have a point with coordinates (x, y) and the line is called \hat{Y} since it is our prediction. The corresponding point on the line is (x, \hat{y}) , and the vertical distance from the point to the line is $(y - \hat{y})$. This is the **error**.

Our total error is going to be the sum of all these distances for all the points in our dataset.

$$Error = \sum_{i=1}^m |y - \hat{y}|$$

In some cases, we'll use the average or the mean absolute error, which is the sum of all the errors divided by m , the number of points in our dataset.

$$Error = \frac{1}{m} \sum_{i=1}^m |y - \hat{y}|$$

Using the sum or the average won't change our algorithms, since that would only scale our error by a constant, namely m .

Notice that we have an absolute value around $y - \hat{y}$, the reason is that if the point is on top of the line, the distance is $y - \hat{y}$, but if it's under the line then it's $\hat{y} - y$. We want the error to always be positive, otherwise negative errors will cancel with positive errors.

The graph has more dimensions but this is a two-dimensional simplification of that graph. As we descend in this graph using gradient descent, we get a better and better line until we find the best possible fit with the smallest possible mean absolute error.

Mean Squared Error

The Mean Squared Error is very similar to the Mean Absolute Error.

Assume we still have our point and our prediction, but instead of taking the distance between the point and the prediction, we're going to draw a square with this segment as its side. This area is $(y - \hat{y})^2$

Notice that this is always non-negative, so we don't need to worry about absolute values.

Our mean squared error is going to be the average of all these series of squares.

$$Error = \frac{1}{2m} \sum_{i=1}^m (y - \hat{y})^2$$

If the point is over the line or underneath the line the square is always going to be a non-negative number because the square of a real number is *always going to be non-negative*.

The one half is going to be there for convenience because later we'll be taking the derivative of this error.

2 ways to fit a line

We've learned two algorithms that will fit a line through a set of points.

- Using the absolute or square trick
- Minimizing any of the error functions namely the mean absolute error and the mean squared error.

The interesting thing is that these two are actually the exact same thing!

When we minimize the mean absolute error, we're using a gradient descent step and that gradient descent step is the exact same thing as the absolute trick. Likewise, when we minimize the squared error, the gradient descent step is the exact same thing as the square trick.

Mean squared error

Let's start with the mean squared error. Assume we have a point with coordinates (x, y) , and a line with the equation $\hat{y} = w_1 x + w_2$. So \hat{y} is our prediction, and it predicts the y coordinate of the point that matches the x coordinate of the point (x, y) , the point (x, \hat{y}) . The error for this point is $\frac{1}{2}(y - \hat{y})^2$, and the mean squared error for this set of points is the average of all these errors. Since the average is a linear function, whatever we do here applies to the entire error.

We know that the gradient descent step uses these two derivatives, namely the derivative with respect to the slope w_1 and the derivative with respect to the y-

intercept w_2 . If we calculate the derivatives, we get $\frac{\partial}{\partial w_1} Error = -(y - \hat{y})x$ for the one respect to the slope and $\frac{\partial}{\partial w_2} Error = -(y - \hat{y})$ for the one with respect to the y-intercept w_2 .

Notice that the length of this red segment is precisely $(y - \hat{y})$ and the length of this green segment is precisely x .

Square trick

If you remember correctly, the square trick told us that we have to update the slope by adding $(y - \hat{y})x\alpha$, and updating the y-intercept by adding $(y - \hat{y})\alpha$. But that is precisely what this gradient descent step is doing.

Think to yourself about how this is the exact same calculation.

So this is why the gradient descent step utilize when we minimize the mean squared error is the same as the square trick.

Absolute trick

We can do the same thing with the absolute trick. The procedure is very similar except we have to be careful about the sign. Our error is $|y - \hat{y}|$ and the derivatives of the error with respect to w_1 and w_2 are $\pm x$ and ± 1 based on if the point is on top of or underneath the line. Since the distance is x , then you can also check that this is precisely what the gradient descent step does when we minimize the mean absolute error.

That is why minimizing these errors with gradient descent is the exact same thing that using the absolute and the square tricks.

Development of the derivative of the error function

Notice that we've defined the squared error to be

$$Error = \frac{1}{2}(y - \hat{y})^2.$$

Also, we've defined the prediction to be

$$\hat{y} = w_1 x + w_2.$$

So to calculate the derivative of the Error with respect to w_1 , we simply use the chain rule:

$$\frac{\partial}{\partial w_1} Error = \frac{\partial Error}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_1}.$$

The first factor of the right-hand side is the derivative of the Error with respect to the prediction \hat{y} , which is $-(y - \hat{y})$.

The second factor is the derivative of the prediction with respect to w_1 , which is simply x .

Therefore, the derivative is

$$\frac{\partial}{\partial w_1} Error = -(y - \hat{y})x$$

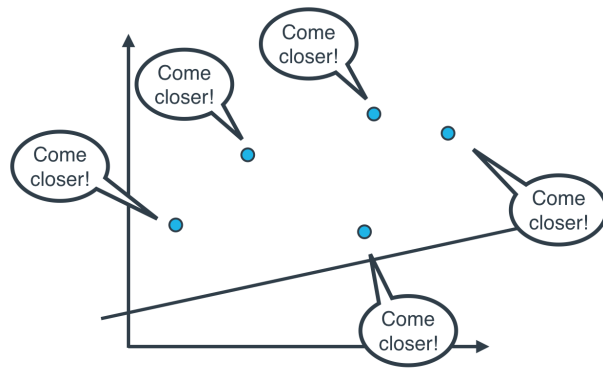
Batch vs Stochastic Gradient Descent

At this point, it seems that we've seen two ways of doing linear regression.

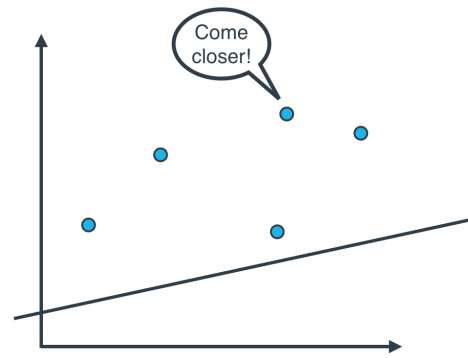
- By applying the squared (or absolute) trick at every point in our data *one by one*, and repeating this process many times.
- By applying the squared (or absolute) trick at every point in our data *all at the same time*, and repeating this process many times.

More specifically, the squared (or absolute) trick, when applied to a point, gives us some values to add to the weights of the model. We can add these values, update our weights, and then apply the squared (or absolute) trick on the next point. Or we can calculate these values for all the points, add them, and then update the weights with the sum of these values.

The latter is called *batch gradient descent*. The former is called *stochastic gradient descent*.



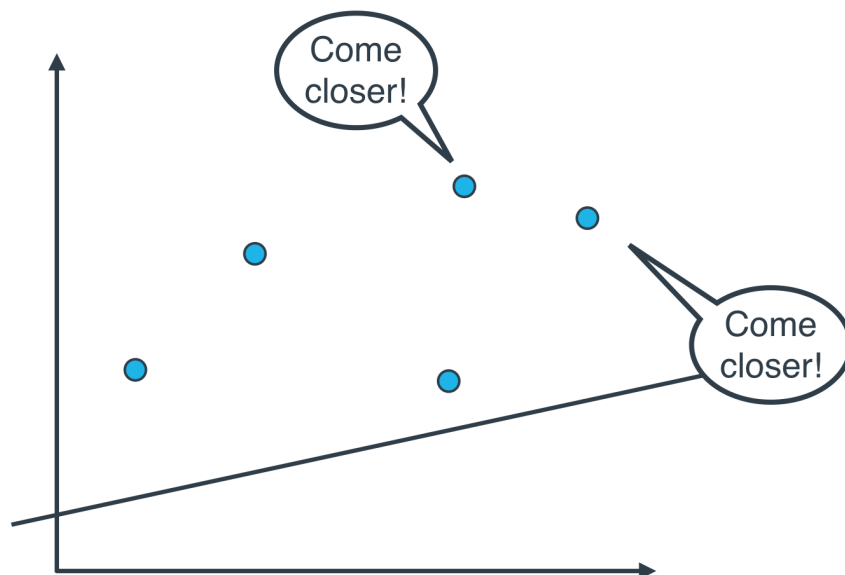
Batch



Stochastic

The question is, which one is used in practice?

Actually, in most cases, neither. Think about this: If your data is huge, both are a bit slow, computationally. The best way to do linear regression, is to split your data into many small batches. Each batch, with roughly the same number of points. Then, use each batch to update your weights. This is still called *mini-batch gradient descent*.



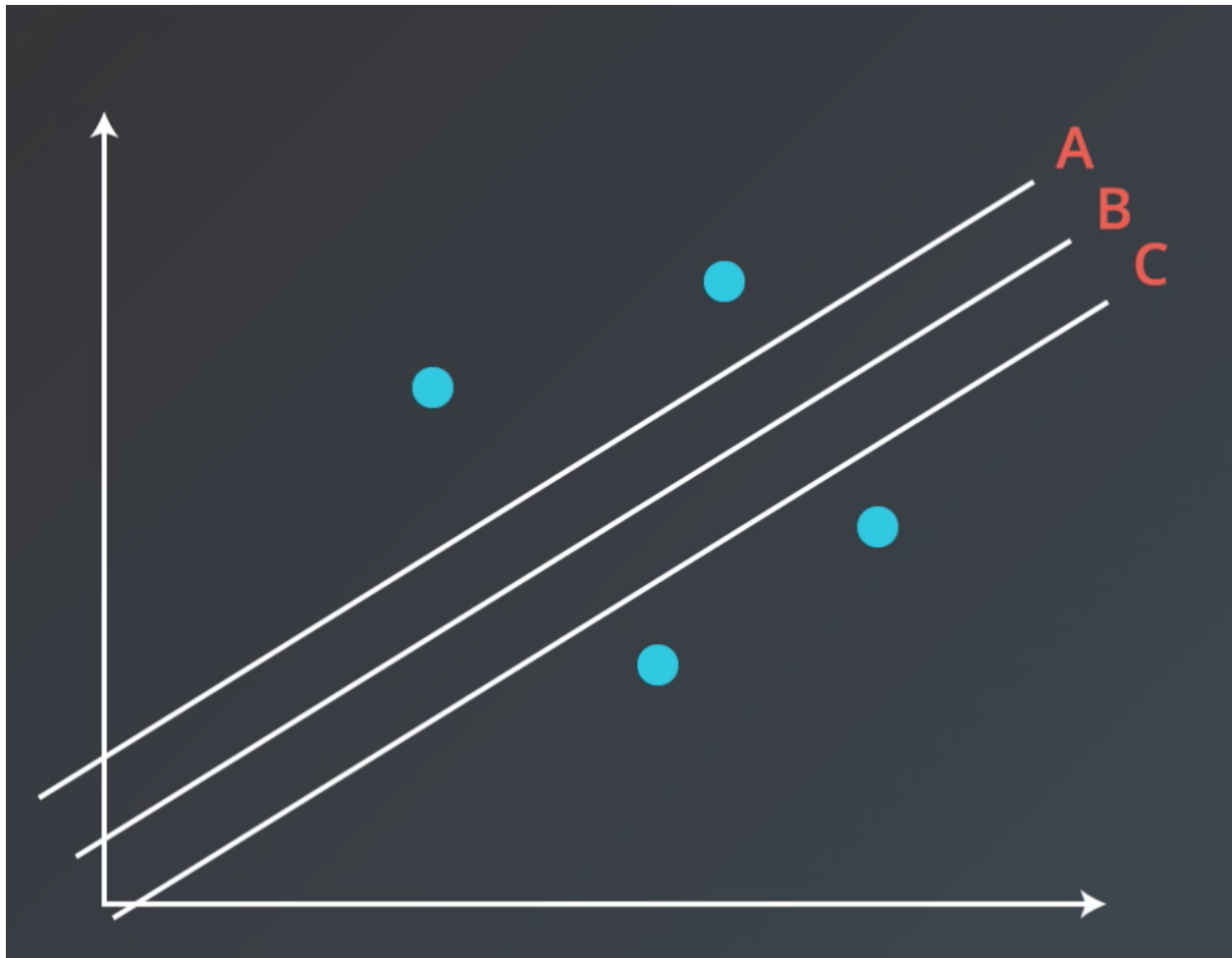
Mini-batch

Which is better: the mean absolute error or the mean squared error?

Both are used for a lot of different purposes, but here's one property that actually tells them apart.

Mean Absolute Error Quiz

With this set of data and we're going to try to fit it by minimizing the mean absolute error. We have line A, line B, and line C as seen below. And the question is, which one of these lines gives us a smaller mean absolute error?



QUESTION 1 OF 2

Which of the three lines gives you a smaller Mean Absolute Error?

- A
- B

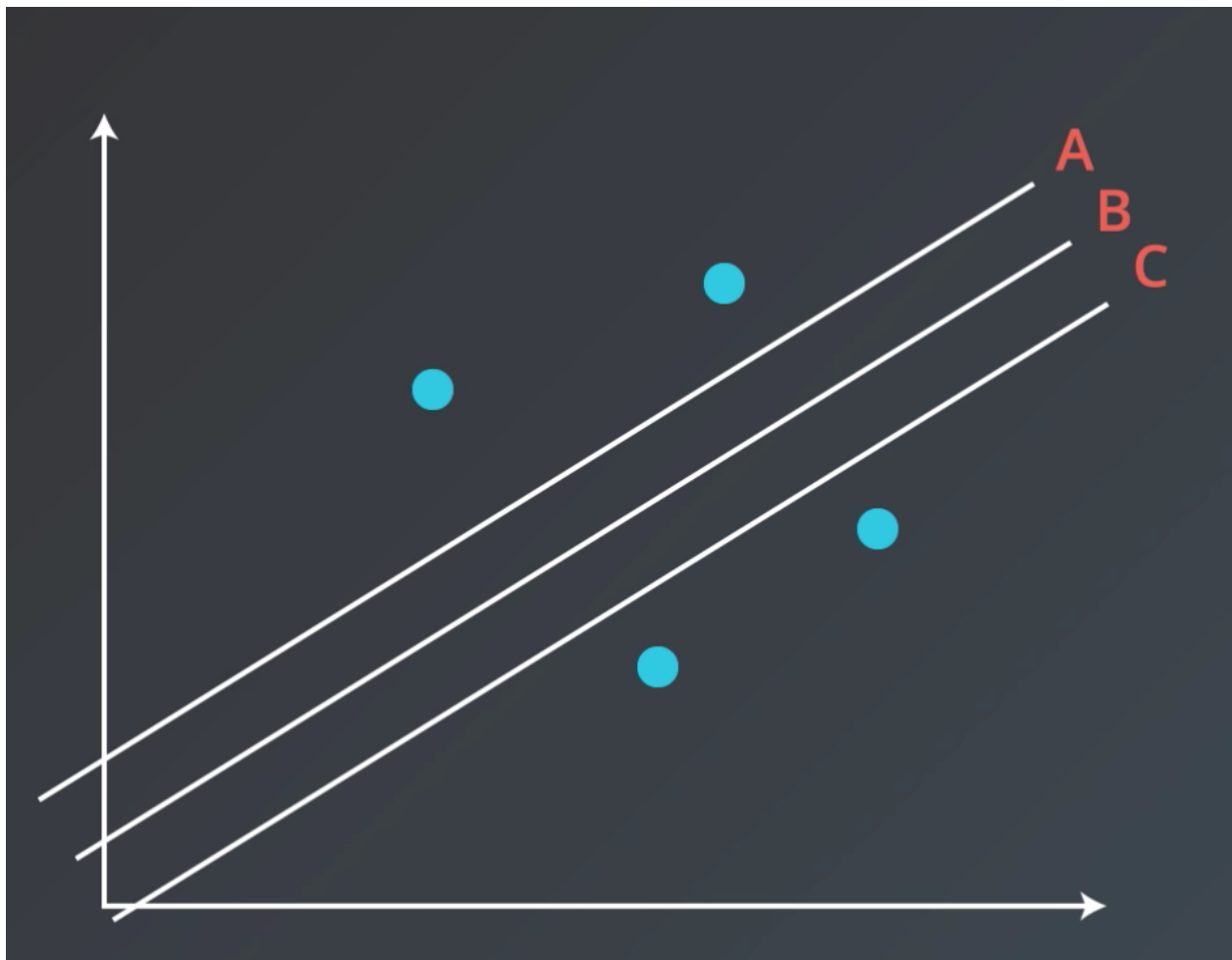
- C
- They all give the same error

Let's review this solution

If you said the same, then you were correct because as you can see, moving the line up and down actually keeps the mean absolute error the same. You can convince yourself by looking at the picture and checking that as you move a line up and down, you're adding a segment to two of the errors and removing a segment of equal length to the other two errors.

Mean Squared Error Quiz

Now, let's try the same quiz but with the **mean squared error**. The question is, which one of these three lines A, B, and C would give us a smaller mean squared error?



QUESTION 2 OF 2

Which of the three lines gives you a smaller Mean Squared Error?

- A
- B
- C
- They all give the same error

So this time the solution is B and the reason is more subtle.

Our mean squared error is actually a quadratic function and quadratic functions have a minimum at the point in the middle.

You can also check with your eyes or draw a small example and calculate the areas; line B would be giving you the smallest mean square error.

Higher Dimensions

Two-dimensions

In the previous example, we had a one-column input and one-column output. We had a two-dimensional problem.

1. The input was the size of the house
2. The output was the price.

Our prediction for the price would be a line and the equation would just be a constant times size plus another constant.

Three-dimensions

What if we had more columns in the input, for example, size and school quality?

That changes it to a three-dimensional graph because we have two dimensions for the input and one for the output. So now our points don't live in the plane, but they look like points flying in 3-dimensional space.

What we do here is we'll fit a plane through them instead of fitting a line, and our equation won't be $(constant_1 * variable + constant_2)$. It's going to be

$(constant_1 * SchoolQuality + constant_2 * Size + constant_3)$.

N-dimensions

We have $n - 1$ columns in the input and one in the output. For example, the inputs are size, school quality, number of rooms, et cetera. So for our input, we have $n - 1$ variable names: x_1, x_2 up to x_{n-1} and for the output of the prediction, we only have one variable \hat{y} . Our prediction would be an $n - 1$ dimensional hyperplane living in n dimensions. Since it's hard to picture n -dimensions, think of a linear equation in n variables, such as $\hat{y} = w_1 x_1 + w_2 x_2 + \dots + w_{n-1} x_{n-1} + w_n$.

In order to find the weights w_1 up to w_n , the algorithm is exactly the same as it is for two variables. We can either do the absolute or square root tricks, or we can calculate the mean absolute or square errors, and minimize using gradient descent.

Multiple Linear Regression

In the last section, you saw how we can predict life expectancy using BMI. Here, BMI was the **predictor**, also known as an independent variable. A predictor is a variable you're looking at in order to make predictions about other variables, while the values you are trying to predict are known as dependent variables. In this case, life expectancy was the dependent variable.

Now, let's say we get new data on each person's heart rate as well. Can we create a prediction of life expectancy using both BMI and heart rate?

Absolutely! As we saw in the previous video, we can do that using multiple linear regression.

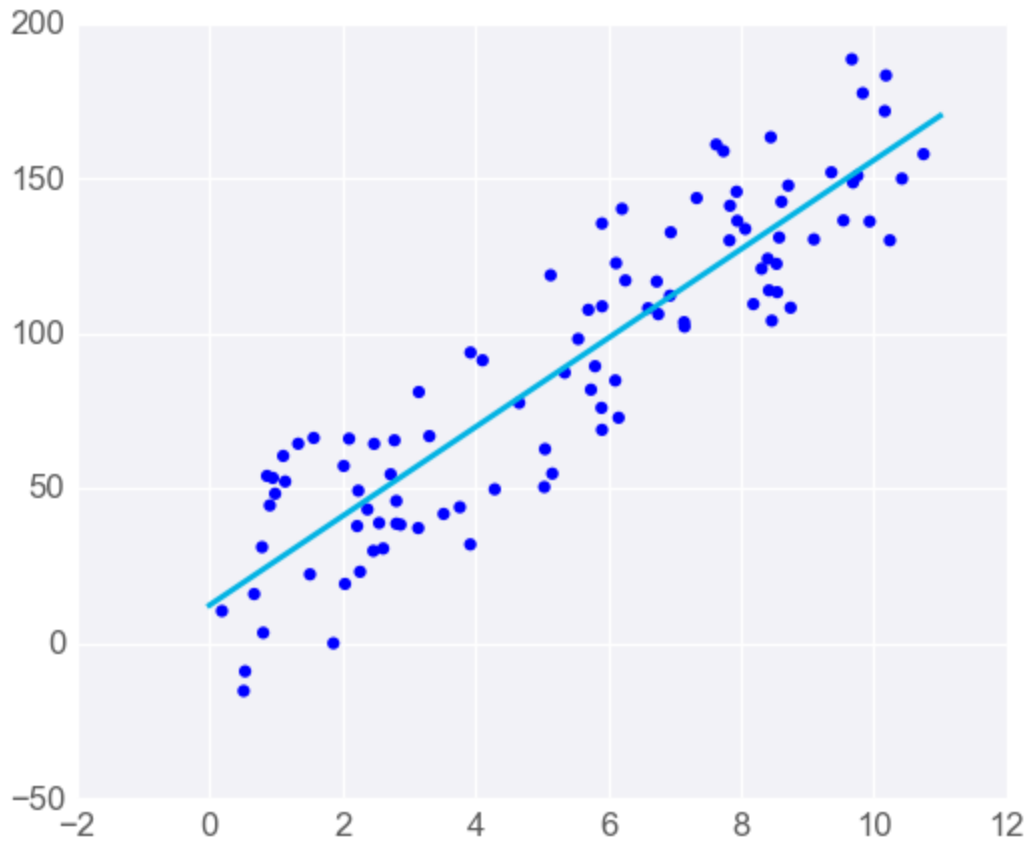
If the outcome you want to predict depends on more than one variable, you can make a more complicated model that takes this into account. As long as they're relevant to the situation, using more independent/predictor variables can help you get a better prediction.

When there's just one predictor, the linear regression model is a line, but as you add more predictor variables, you're adding more dimensions to the picture.

When you have one predictor variable, the equation of the line is

$$y = mx + b$$

and the plot might look something like this:

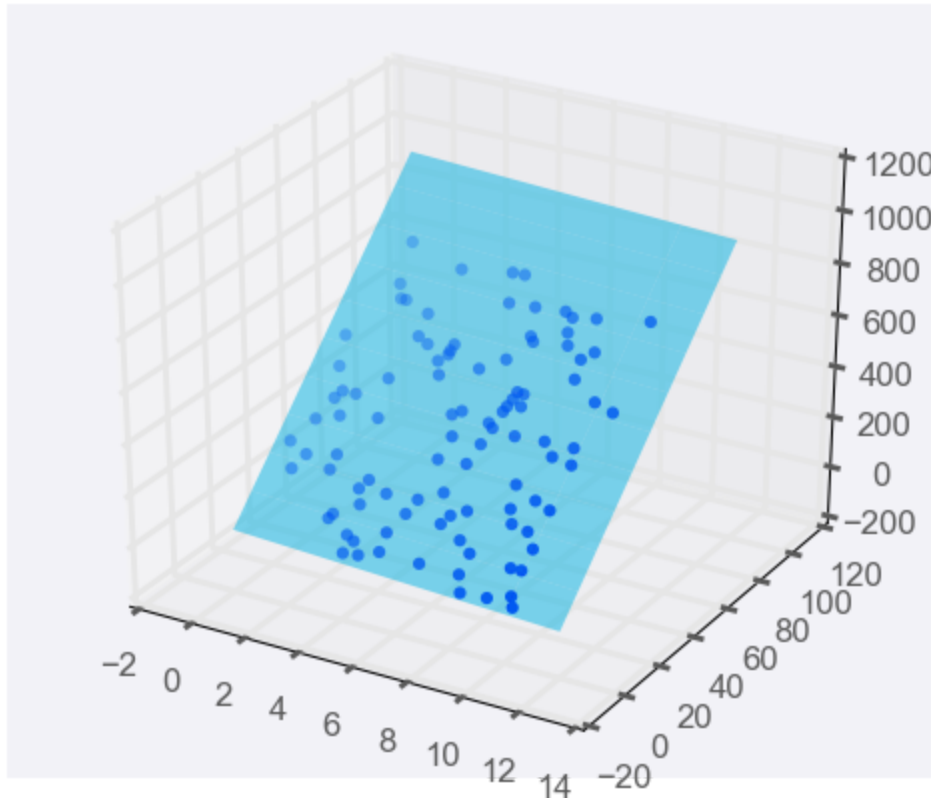


Linear regression with one predictor variable

Adding a predictor variable to go to two predictor variables means that the predicting equation is:

$$y = m_1x_1 + m_2x_2 + b$$

To represent this graphically, we'll need a three-dimensional plot, with the linear regression model represented as a plane:



Linear regression with two predictor variables

You can use more than two predictor variables - in fact, you should use as many as is useful! If you use n predictor variables, then the model can be represented by the equation

$$y = m_1x_1 + m_2x_2 + m_3x_3 + \dots + m_nx_n + b$$

As you make a model with more predictor variables, it becomes harder to visualise, but luckily, everything else about linear regression stays the same. We can still fit models and make predictions in exactly the same way - time to try it!

Closed Form Solution

Full derivation is available on the next page.

In order to minimize the mean squared error, we do not actually need to use gradient descent or the tricks.

We can actually do this in a closed mathematical form.

Here's our data (x_1, y_1) all the way to (x_m, y_m) ; and in this case, m is 5.

The areas of the squares represent our squared error. So our input is x_1 up to x_m and our labels are y_1 up to y_m , and our predictions are of the form $\hat{y}_i = w_1 x_i + w_2$, where w_1 is a slope of the line and w_2 is the y-intercept.

The mean squared error is given by this formula $E(w_1, w_2) = \frac{1}{2m} \sum_{i=1}^m (y - \hat{y})^2$.

The error is written as a function of w_1 and w_2 , since we can calculate the predictions and the error based on any given w_1 and w_2 . In order to minimize this error, we need to take the derivative with respect to the two input variables w_1 and w_2 and set them both equal to 0.

Information on how to solve the two equations using linear algebra is provided on the following page.

Linear Regression Warnings

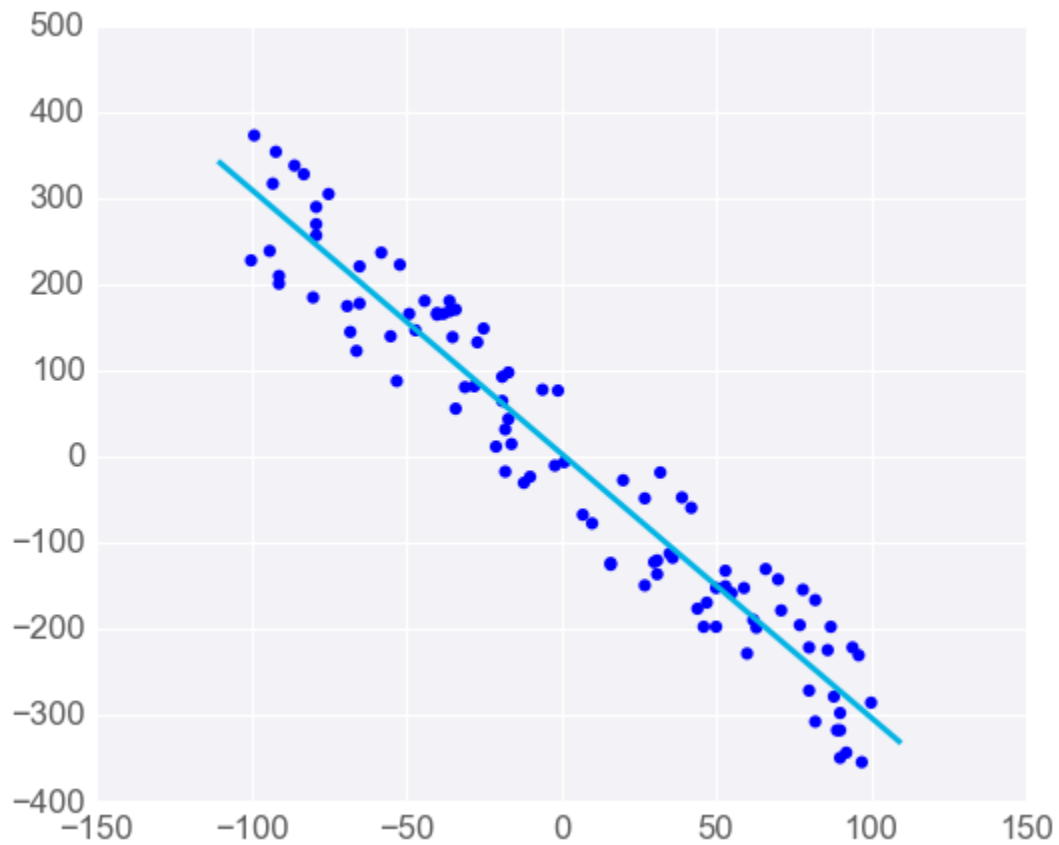
Linear regression comes with a set of implicit assumptions and is not the best model for every situation. Here are a couple of issues that you should watch out for.

Linear Regression Works Best When the Data is Linear Linear regression produces a straight line model from the training data. If the relationship in the training data is not really linear, you'll need to either make adjustments (transform your training data), add features (we'll come to this next), or use another kind of model.



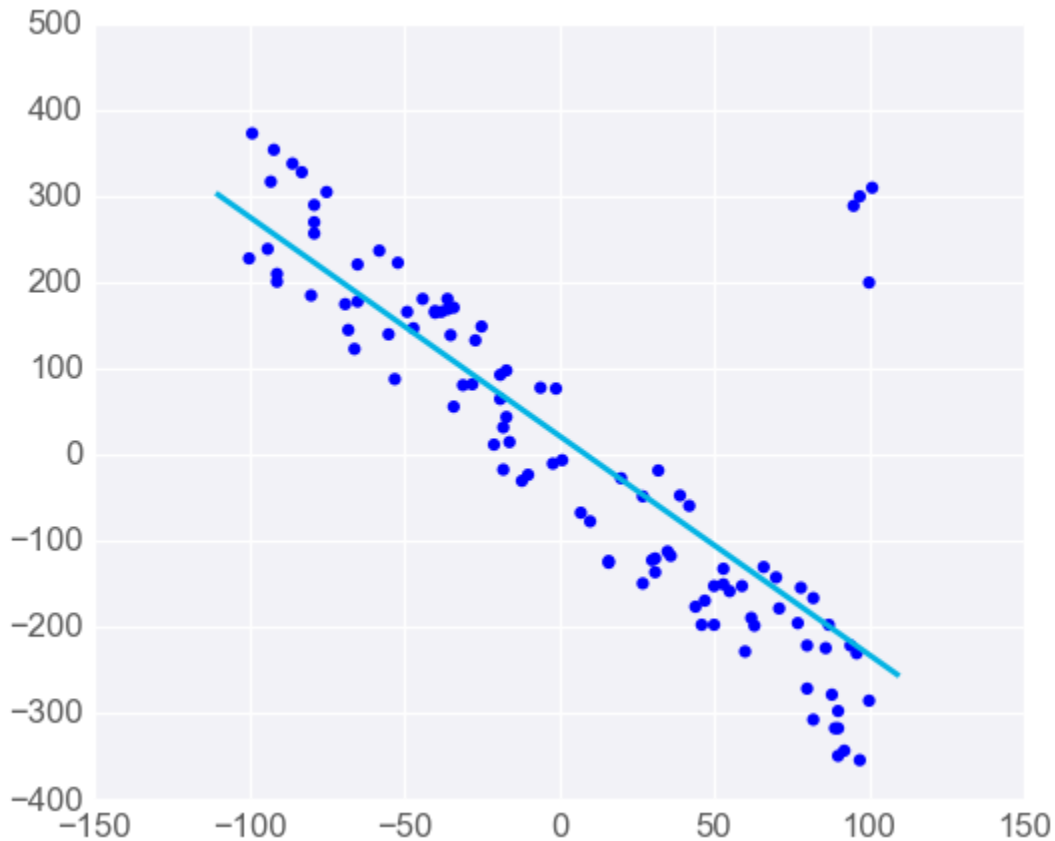
Linear Regression is Sensitive to Outliers Linear regression tries to find a 'best fit' line among the training data. If your dataset has some outlying extreme values that don't fit a general pattern, they can have a surprisingly large effect.

In this first plot, the model fits the data pretty well.



No outliers here

However, adding a few points that are outliers and don't fit the pattern really changes the way the model predicts.



Linear regression with outliers

In most circumstances, you'll want a model that fits most of the data most of the time, so watch out for outliers!

Polynomial Regression

What happens if we have data where a line won't do a good job fitting in? Maybe a curve, polynomial, such as $\hat{y} = 2x^3 - 8x^2 - 5x + 4$.

This can be solved using a very similar algorithm to linear regression.

Instead of considering lines, we consider **higher degree polynomials**. This would give us more weights to solve our problem.

For example, in this problem we would solve for four weights; w_1 , w_2 , w_3 , and w_4 . We take the mean absolute or squared error and take the derivative with respect to the four variables and use gradient descent to modify these four weights in order to minimize the error.

This algorithm is known as **polynomial regression**.

Regularization

Regularization through a classification problem

Note: as you will see, all the arguments here work with regression algorithms as well.

Regularization, it's a very useful technique to improve our models and make sure they don't overfit.

Let's look at some data and two models that classify this data.

- The first is a line
- The second is a higher degree polynomial curve.

Which one is better?

They both have pros and cons.

- The line makes a couple of mistakes.
- The curve makes zero mistakes but is a bit more complicated.

Example

Assume the equation of the line is something like $3x_1 + 4x_2 + 5 = 0$. The equation of the polynomial is more complex with high degree terms like $2x_1^3 - 2x_1^2x_2 - 4x_2^3 + 3x_1^2 + 6x_1x_2 + 4x_2^2 + 5 = 0$.

If we look at the equation of the line, it's much simpler than the polynomial equation. In particular, there are fewer coefficients, only three, four, five whereas the complex polynomial has many more.

A complex model will have a larger error and then a simple model.

If we add the coefficients of the line to its error, we get a slightly larger error. But what if we take all of the coefficients of the polynomial and add them to the error, we get a huge error. We can see that the modeling of the line is better because it has a smaller combined error. A simpler model has an edge over a complicated model.

| Simpler models have a tendency to generalize better.

L1 regularization

L1 regularization takes the absolute value of the coefficients of the model and adds them to the error.

For example, if we take our previous model $2x_1^3 - 2x_1^2x_2 - 4x_2^3 + 3x_1^2 + 6x_1x_2 + 4x_2^2 + 5 = 0$ we would add $(2 + 2 + 4 + 3 + 6 + 4) = 21$ (we don't add the constant). In the linear case ($3x_1 + 4x_2 + 5 = 0$), we see that we're only adding 3 and 4 which is 7. An error of 7 is much less than 21, so the complicated model gives us a much higher error.

L2 regularization

L2 regularization is similar to L1 Regularization however instead of adding the absolute values, we add the squares of the coefficients. So for the complicated case, we get $(2^2 + -2^2 + -4^2 + 3^2 + 6^2 + 4^2) = 85$. For the linear case, we get $(3^2 + 4^2) = 25$, which is much smaller than 85. So again, we see that the complex model gets punished a lot more than the simple model.

But what if we punish the complicated model too little, or punish it too much?

We can "tune," or alter the amount that we want to punish complex models by using a parameter called **lambda**.

The lambda (λ) Parameter

Using λ , we multiply the error that comes from the complexity of the model to adjust the overall error.

- With a small lambda, the error that comes from the complexity of the model is not large enough to overtake the errors in the simplified model misclassifying points, so we will choose the complex model.

- With a large value for λ , we're multiplying the complexity part of the error by a lot. This punishes the complex model more so the simple model wins.

If we have a large λ then we're punishing complexity by a large amount and we're picking a simpler model. Whereas if we have a small λ , we're punishing complexity by a small amount, so we're okay with having more complex models.

L1 or L2 regularization?

So here's a cheat sheet with some benefits for each one.

Efficiency

(con) L1 regularization is actually computationally inefficient even though it seems simpler because it has no squares, but actually, those absolute values are hard to differentiate.

(pro) Whereas, L2 regularization squares have very nice derivatives. So, these are easy to deal with computation.

Sparse data

(pro) L1 regularization is faster than L2 regularization. If you have a thousand columns of data but only 10 are relevant and the rest are mostly zeros, then L1 is faster,

(pro) L2 is better for non-sparse outputs which are when the data is more equally distributed among the columns.

Feature selection

(pro) L1 has one huge benefit which is that it gives us feature selection. So let's say, we have again, data in a thousand columns but really only 10 of the matters, and the rest are mostly noise. So, L1 will detect this and will make the relevant columns into zeroes.

(con) L2 on the other hand won't do this and it just takes the columns and treat them similarly.

Feature Scaling

What is feature scaling? Feature scaling is a way of transforming your data into a common range of values. There are two common scalings:

1. Standardizing
2. Normalizing

Standardizing

Standardizing is completed by taking each value of your column, subtracting the mean of the column, and then dividing by the standard deviation of the column. In Python, let's say you have a column in `df` called `height`. You could create a standardized height as:

```
df["height_standard"] = (df["height"] - df["height"].mean()) / df["height"].std()
```

This will create a new "standardized" column where each value is a comparison to the mean of the column, and a new, standardized value can be interpreted as the number of standard deviations the original height was from the mean. This type of feature scaling is by far the most common of all techniques (for the reasons discussed here, but also likely because of precedent).

Normalizing

A second type of feature scaling that is very popular is known as **normalizing**. With normalizing, data are scaled between 0 and 1. Using the same example as above, we could perform normalizing in Python in the following way:

```
df["height_normal"] = (df["height"] - df["height"].min()) / \
    (df["height"].max() - df["height"].min())
```

When Should I Use Feature Scaling?

In many machine learning algorithms, the result will change depending on the units of your data. This is especially true in two specific cases:

1. When your algorithm uses a distance-based metric to predict.
2. When you incorporate regularization.

Distance-Based Metrics

In future lessons, you will see one common supervised learning technique that is based on the distance points are from one another called Support Vector Machines (or SVMs).

Another technique that involves distance-based methods to determine a prediction is k-nearest neighbors (or k-nn). With either of these techniques, choosing not to scale your data may lead to drastically different (and likely misleading) ending predictions.

For this reason, choosing some sort of feature scaling is necessary with these distance-based techniques.

Regularization

When you start introducing regularization, you will again want to scale the features of your model. The penalty on particular coefficients in regularized linear regression techniques depends largely on the scale associated with the features. When one feature is on a small range, say from 0 to 10, and another is on a large range, say from 0 to 1 000 000, applying regularization is going to unfairly punish the feature with the small range. Features with small ranges need to have larger coefficients compared to features with large ranges in order to have the same effect on the outcome of the data. (Think about how $ab = ba$ for two numbers aa and bb .) Therefore, if regularization could remove one of those two features with the same net increase in error, it would rather remove the small-ranged feature with the large coefficient, since that would reduce the regularization term the most.

Again, this means you will want to scale features any time you are applying regularization.

- [A useful Quora post on the importance of feature scaling when using regularization.](#)

A point raised in the article above is that feature scaling can speed up convergence of your machine learning algorithms, which is an important consideration when you scale machine learning applications.

Use the quiz below to get some practice with feature scaling.



Lesson Review

In this lesson, you were introduced to linear models. Specifically, you saw:

- **Gradient descent** is a method to optimize your linear models.
- **Multiple Linear Regression** is a technique for when you are comparing more than two variables.

- **Polynomial Regression** for relationships between variables that aren't linear.
- **Regularization** is a technique to assure that your models will not only fit the data available but also extend to new situations.

Glossary

|  Key Term |  Definition |
|--|--|
| <u>Batch gradient descent</u> | The process of repeatedly calculating errors for all points at the same time and updating weights accordingly. |
| <u>Error</u> | The vertical distance from a given point to the predictive line. |
| <u>Feature scaling</u> | Transforming data into a common range of values using standardizing or normalizing. |
| <u>Gradient descent</u> | The reduction of the error by taking the derivative of the error function with respect to the weights. |
| <u>L1 Regularization</u> | Absolute values of the coefficients of the model are used for regularization. |
| <u>L2 Regularization</u> | Squares of the values of the coefficients of the model are used for regularization. |
| <u>Lambda</u> | The amount by which we punish complex models during the process of regularization. |
| <u>Learning rate</u> | The amount by which we adjust the weights of our equation. The larger the learning rate, the larger our adjustments. |
| <u>Mean absolute error</u> | The sum of the absolute value of all errors divided by the total number of points. |
| <u>Mean squared error</u> | The sum of the square of all errors divided by the total number of points. |
| <u>Regularization</u> | Taking into consideration the complexity of the model when evaluating regression models. |
| <u>Stochastic gradient descent</u> | The process of repeatedly calculating errors one point at a time and updating weights accordingly. |