CodeCademy – Machine Learning

Table des matières

[Introduction 5](#_Toc535151689)

[What is machine learning? 5](#_Toc535151690)

[Supervised Learning 6](#_Toc535151691)

[Script.py 6](#_Toc535151692)

[Script.py Output 7](#_Toc535151693)

[Unsupervised Learning 7](#_Toc535151694)

[Script.py 7](#_Toc535151695)

[Script.py Output 8](#_Toc535151696)

[Supervised VS Unsupervised 9](#_Toc535151697)

[Supervised Learning 9](#_Toc535151698)

[Unsupervised Learning 10](#_Toc535151699)

[Summary 10](#_Toc535151700)

[Scikit-Learn Cheatsheet 11](#_Toc535151701)

[Linear Regression 11](#_Toc535151702)

[Naive Bayes 11](#_Toc535151703)

[K-Nearest Neighbors 11](#_Toc535151704)

[K-Means 12](#_Toc535151705)

[Validating the Model 12](#_Toc535151706)

[Training Sets and Test Sets 12](#_Toc535151707)

[Linear Regression 13](#_Toc535151708)

[Introduction to Linear Regression 13](#_Toc535151709)

[Script.py 14](#_Toc535151710)

[Script.py Output 14](#_Toc535151711)

[Points and lines 15](#_Toc535151712)

[Script.py 15](#_Toc535151713)

[Script.py Output 15](#_Toc535151714)

[Loss 16](#_Toc535151715)

[Minimizing Loss 17](#_Toc535151716)

[Gradient Descent for Intercept 17](#_Toc535151717)

[Script.py 18](#_Toc535151718)

[Gradient Descent for Slope 18](#_Toc535151719)

[Script.py 18](#_Toc535151720)

[Put it Together 19](#_Toc535151721)

[Script.py 19](#_Toc535151722)

[Convergence 19](#_Toc535151723)

[Script.py 20](#_Toc535151724)

[Script.py Output 20](#_Toc535151725)

[Learning Rate 20](#_Toc535151726)

[Example with built in function 21](#_Toc535151727)

[Script.py 21](#_Toc535151728)

[Script.py Output 21](#_Toc535151729)

[MULTIPLE LINEAR REGRESSION 22](#_Toc535151730)

[Introduction to Multiple Linear Regression 22](#_Toc535151731)

[Script.py 22](#_Toc535151732)

[Script.py Output 23](#_Toc535151733)

[StreetEasy Dataset example 23](#_Toc535151734)

[Training Set vs. Test Set 24](#_Toc535151735)

[Script.py 24](#_Toc535151736)

[Script.py Output 25](#_Toc535151737)

[Training Set vs Validation Set vs Test Set 25](#_Toc535151738)

[TESTING OUR MODEL 25](#_Toc535151739)

[TRAINING SET VS VALIDATION SET 25](#_Toc535151740)

[HOW TO SPLIT 25](#_Toc535151741)

[N-FOLD CROSS-VALIDATION 25](#_Toc535151742)

[CHANGING THE MODEL / TEST SET 26](#_Toc535151743)

[Understanding the accuracy of your model is invaluable because you can begin to tune the parameters of your model to increase its performance. For example, in the K-Nearest Neighbors algorithm, you can watch what happens to accuracy as you increase or decrease K. You can also use this information to try to avoid overfitting or underfitting your data. 26](#_Toc535151744)

[Multiple Linear Regression: Scikit-Learn 27](#_Toc535151745)

[Script.py 27](#_Toc535151746)

[Script.py Output 27](#_Toc535151747)

[Multiple Linear Regression Equation 28](#_Toc535151748)

[Equation 6.1 28](#_Toc535151749)

[Equation 6.2 28](#_Toc535151750)

[Equation 6.3 28](#_Toc535151751)

[Correlations 29](#_Toc535151752)

[Script.py 29](#_Toc535151753)

[Script.py Output 30](#_Toc535151754)

[Evaluating the Model's Accuracy 30](#_Toc535151755)

[Script.py 31](#_Toc535151756)

[Script.py Output 31](#_Toc535151757)

# Introduction

## What is machine learning?

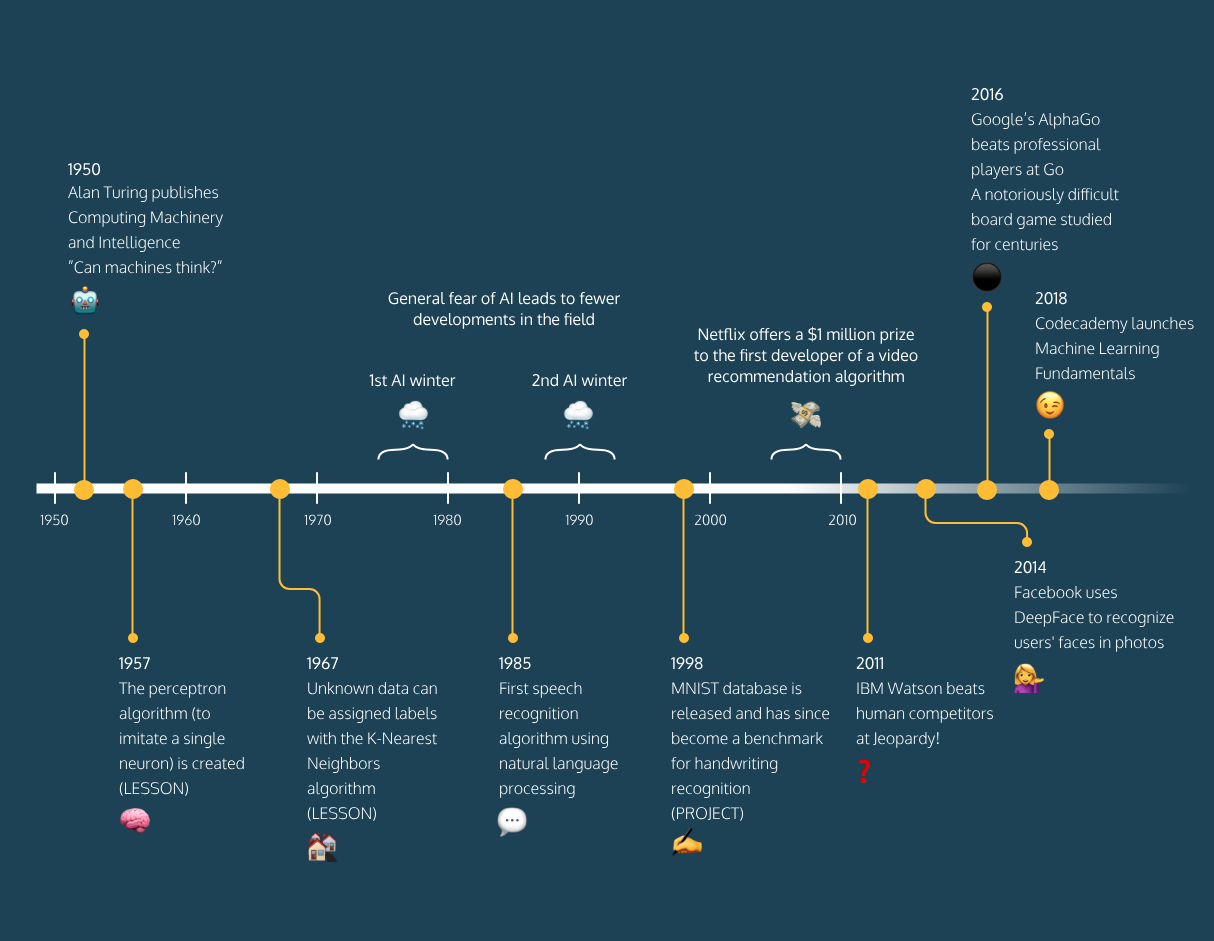
While at IBM, Arthur Samuel developed a program that learned how to play checkers. He called it — [*"the field of study that gives computers the ability to learn without being explicitly programmed"*](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.368.2254&rep=rep1&type=pdf)(1959).

What does this mean?

As programmers, we often approach problems in a methodical, logic-based way. We try to determine what our desired outputs should be, and then create the proper rules that will transform our inputs into those outputs.

Machine learning flips the script. We want the program itself to learn the rules that describe our data the best, by finding patterns in what we know and applying those patterns to what we don't know.

These algorithms are able to *learn*. Their performance gets better and better with each iteration, as it uncovers more hidden trends in the data.



## Supervised Learning

Machine learning can be branched out into the following categories:

* Supervised Learning
* Unsupervised Learning

[Supervised Learning](https://www.codecademy.com/articles/machine-learning-supervised-vs-unsupervised) is where the data is labeled and the program learns to predict the output from the input data. For instance, a supervised learning algorithm for credit card fraud detection would take as input a set of recorded transactions. For each transaction, the program would predict if it is fraudulent or not.

Supervised learning problems can be further grouped into regression and classification problems.

Regression:

In regression problems, we are trying to predict a continuous-valued output. Examples are:

* What is the housing price in Neo York?
* What is the value of cryptocurrencies?

Classification:

In classification problems, we are trying to predict a discrete number of values. Examples are:

* Is this a picture of a human or a picture of an AI?
* Is this email spam?

For a quick preview, we will show you an example of supervised learning.

### Script.py

**from** texts **import** text\_counter**,** text\_training

**from** sklearn**.**feature\_extraction**.**text **import** CountVectorizer

**from** sklearn**.**naive\_bayes **import** MultinomialNB

intercepted\_text **=** "This hot dog was awful!"

text\_counts **=** text\_counter**.**transform**([**intercepted\_text**])**

text\_classifier **=** MultinomialNB**()**

text\_labels **=** **[**0**]** **\*** 1000 **+** **[**1**]** **\*** 1000

text\_classifier**.**fit**(**text\_training**,** text\_labels**)**

final\_pos **=** text\_classifier**.**predict\_proba**(**text\_counts**)[**0**][**1**]**

final\_neg **=** text\_classifier**.**predict\_proba**(**text\_counts**)[**0**][**0**]**

**if** final\_pos **>** final\_neg**:**

**print(**"The text is positive."**)**

**else:**

**print(**"The text is negative."**)**

### Script.py Output

intercepted\_text **=** "This hot dog was awful!" >> the text is negative

intercepted\_text = "I love my governement" >> the text is positive

intercepted\_text = "42" >> the text is positive  
// This Naive Bayes classifier won't always get the sentiment correct!

## Unsupervised Learning

[Unsupervised Learning](https://www.codecademy.com/articles/machine-learning-supervised-vs-unsupervised) is a type of machine learning where the program learns the inherent structure of the data based on unlabeled examples.

Clustering is a common unsupervised machine learning approach that finds patterns and structures in unlabeled data by grouping them into clusters.

Some examples:

* Social networks clustering topics in their news feed
* Consumer sites clustering users for recommendations
* Search engines to group similar objects in one cluster

For a quick preview, we will show you an example of unsupervised learning.

### Script.py

**import** codecademylib3\_seaborn

**import** matplotlib**.**pyplot **as** plt

**import** numpy **as** np

**from** os**.**path **import** join**,** dirname**,** abspath

**from** mpl\_toolkits**.**mplot3d **import** Axes3D

**from** sklearn**.**cluster **import** KMeans

**from** sklearn **import** datasets

iris **=** datasets**.**load\_iris**()**

x **=** iris**.**data

y **=** iris**.**target

fignum **=** 1

# Plot the ground truth

fig **=** plt**.**figure**(**fignum**,** figsize**=(**4**,** 3**))**

ax **=** Axes3D**(**fig**,** rect**=[**0**,** 0**,** .95**,** 1**],** elev**=**48**,** azim**=**134**)**

**for** name**,** label **in** **[(**'Robots'**,** 0**),**

**(**'Cyborgs'**,** 1**),**

**(**'Humans'**,** 2**)]:**

ax**.**text3D**(**x**[**y **==** label**,** 3**].**mean**(),**

x**[**y **==** label**,** 0**].**mean**(),**

x**[**y **==** label**,** 2**].**mean**()** **+** 2**,** name**,**

horizontalalignment**=**'center'**,**

bbox**=**dict**(**alpha**=**.2**,** edgecolor**=**'w'**,** facecolor**=**'w'**))**

# Reorder the labels to have colors matching the cluster results

y **=** np**.**choose**(**y**,** **[**1**,** 2**,** 0**]).**astype**(**np**.**float**)**

ax**.**scatter**(**x**[:,** 3**],** x**[:,** 0**],** x**[:,** 2**],** c**=**y**,** edgecolor**=**'k'**)**

ax**.**w\_xaxis**.**set\_ticklabels**([])**

ax**.**w\_yaxis**.**set\_ticklabels**([])**

ax**.**w\_zaxis**.**set\_ticklabels**([])**

ax**.**set\_xlabel**(**'Time to Heal'**)**

ax**.**set\_ylabel**(**'Reading Speed'**)**

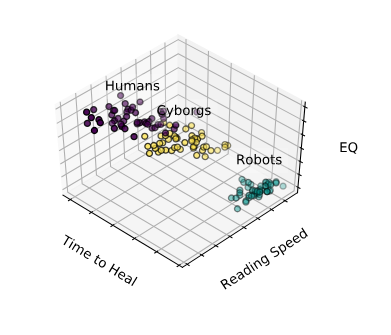
ax**.**set\_zlabel**(**'EQ'**)**

ax**.**set\_title**(**''**)**

ax**.**dist **=** 12

plt**.**show**()**

### Script.py Output



# Supervised VS Unsupervised

As humans, we have many different ways we learn things. The way you learned calculus, for example, is probably not the same way you learned to stack blocks. The way you learned the alphabet is probably wildly different from the way you learned how to tell if objects are approaching you or going away from you. The latter you might not even realize you learned at all!

Similarly, when we think about making programs that can learn, we have to think about these programs learning in different ways. Two main ways that we can approach machine learning are **Supervised Learning** and **Unsupervised Learning**. Both are useful for different situations or kinds of data available.

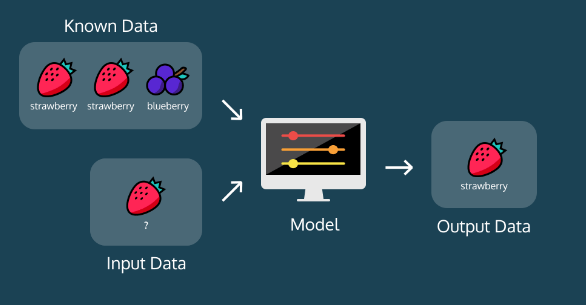
## Supervised Learning

Let's imagine you're first learning about different genres in music. Your music teacher plays you an indie rock song and says "This is an indie rock song". Then, they play you a K-pop song and tell you "This is a K-pop song". Then, they play you a techno track and say "This is techno". You go through many examples of these genres.

The next time you're listening to the radio, and you hear techno, you may think "This is similar to the 5 techno tracks I heard in class today. This must be techno!"

Even though the teacher didn't tell you about *this* techno track, she gave you enough examples of songs that were techno, so you could recognize more examples of it.

When we explicitly tell a program what we expect the output to be, and let it learn the rules that produce expected outputs from given inputs, we are performing supervised learning.



A common example of this is image classification. Often, we want to build systems that will be able to describe a picture. To do this, we normally show a program thousands of examples of pictures, with labels that describe them. During this process, the program adjusts its internal parameters. Then, when we show it a new example of a photo with an unknown description, it should be able to produce a reasonable description of the photo.

When you complete a [Captcha](https://en.wikipedia.org/wiki/CAPTCHA) and identify the images that have cars, you're labeling images! A supervised machine learning algorithm can now use those pictures that you've tagged to make it's car-image predictor more accurate.

## Unsupervised Learning

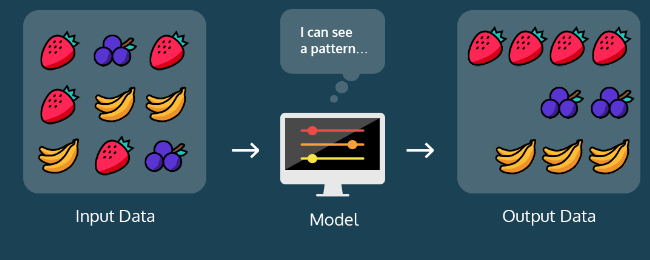
Let's say you are an alien who has been observing the meals people eat. You've embedded yourself into the body of an employee at a typical tech startup, and you see people eating breakfasts, lunches, and snacks. Over the course of a couple weeks, you surmise that for breakfast people mostly eat foods like:

* Cereals
* Bagels
* Granola bars

Lunch is usually a combination of:

* Some sort of vegetable
* Some sort of protein
* Some sort of grain

Snacks are usually a piece of fruit or a handful of nuts. No one explicitly *told* you what kinds of foods go with each meal, but you learned from natural observation and put the patterns together. In unsupervised learning, we don't tell the program anything about what we expect the output to be. The program itself analyzes the data it encounters and tries to pick out patterns and group the data in meaningful ways.



An example of this includes **clustering** to create segments in a business's user population. In this case, an unsupervised learning algorithm would probably create groups (or clusters) based on parameters that a human may not even consider.

## Summary

We have gone over the difference between supervised and unsupervised learning:

* **Supervised Learning**: data is labeled and the program learns to predict the output from the input data
* **Unsupervised Learning**: data is unlabeled and the program learns to recognize the inherent structure in the input data

# Scikit-Learn Cheatsheet

[Scikit-learn](http://scikit-learn.org/stable/) is a library in Python that provides many unsupervised and supervised learning algorithms. It's built upon some of the technology you might already be familiar with, like NumPy, pandas, and Matplotlib!

As you build robust Machine Learning programs, it's helpful to have all the sklearn commands all in one place in case you forget.

## [Linear Regression](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html)

**Import and create the model:**

**from** sklearn**.**linear\_model **import** LinearRegression

your\_model **=** LinearRegression**()**

**Fit:**

your\_model**.**fit**(**x\_training\_data**,** y\_training\_data**)**

* .coef\_: contains the coefficients
* .intercept\_: contains the intercept

**Predict:**

predictions **=** your\_model**.**predict**(**your\_x\_data**)**

* .score(): returns the coefficient of determination R²

## [Naive Bayes](http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html#sklearn.naive_bayes.MultinomialNB))

**Import and create the model:**

**from** sklearn**.**naive\_bayes **import** MultinomialNB your\_model **=** MultinomialNB**()**

**Fit:**

your\_model**.**fit**(**x\_training\_data**,** y\_training\_data**)**

**Predict:**

# Returns a list of predicted classes - one prediction for every data point predictions = your\_model.predict(your\_x\_data) # For every data point, returns a list of probabilities of each class probabilities = your\_model.predict\_proba(your\_x\_data)

## [K-Nearest Neighbors](http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html#sklearn.neighbors.KNeighborsClassifier)

**Import and create the model:**

**from** sklearn**.**neigbors **import** KNeighborsClassifier your\_model **=** KNeighborsClassifier**()**

**Fit:**

your\_model**.**fit**(**x\_training\_data**,** y\_training\_data**)**

**Predict:**

# Returns a list of predicted classes - one prediction for every data point predictions = your\_model.predict(your\_x\_data) # For every data point, returns a list of probabilities of each class probabilities = your\_model.predict\_proba(your\_x\_data)

## [K-Means](http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html)

**Import and create the model:**

**from** sklearn**.**cluster **import** KMeans your\_model **=** KMeans**(**n\_clusters**=**4**,** init**=**'random'**)**

* n\_clusters: number of clusters to form and number of centroids to generate
* init: method for initialization
  + k-means++: K-Means++ [default]
  + random: K-Means
* random\_state: the seed used by the random number generator [optional]

**Fit:**

your\_model**.**fit**(**x\_training\_data**)**

**Predict:**

predictions **=** your\_model**.**predict**(**your\_x\_data**)**

## [Validating the Model](http://scikit-learn.org/stable/modules/classes.html#sklearn-metrics-metrics)

**Import and print accuracy, recall, precision, and F1 score:**

**from** sklearn**.**metrics **import** accuracy\_score**,** recall\_score**,** precision\_score**,** f1\_score **print(**accuracy\_score**(**true\_labels**,** guesses**))** **print(**recall\_score**(**true\_labels**,** guesses**))** **print(**precision\_score**(**true\_labels**,** guesses**))** **print(**f1\_score**(**true\_labels**,** guesses**))**

**Import and print the confusion matrix:**

**from** sklearn**.**metrics **import** confusion\_matrix **print(**confusion\_matrix**(**true\_labels**,** guesses**))**

## [Training Sets and Test Sets](http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html)

**from** sklearn**.**model\_selection **import** train\_test\_split x\_train**,** x\_test**,** y\_train**,** y\_test **=** train\_test\_split**(**x**,** y**,** train\_size**=**0.8**,** test\_size**=**0.2**)**

* train\_size: the proportion of the dataset to include in the train split
* test\_size: the proportion of the dataset to include in the test split
* random\_state: the seed used by the random number generator [optional]

# Linear Regression

## Introduction to Linear Regression

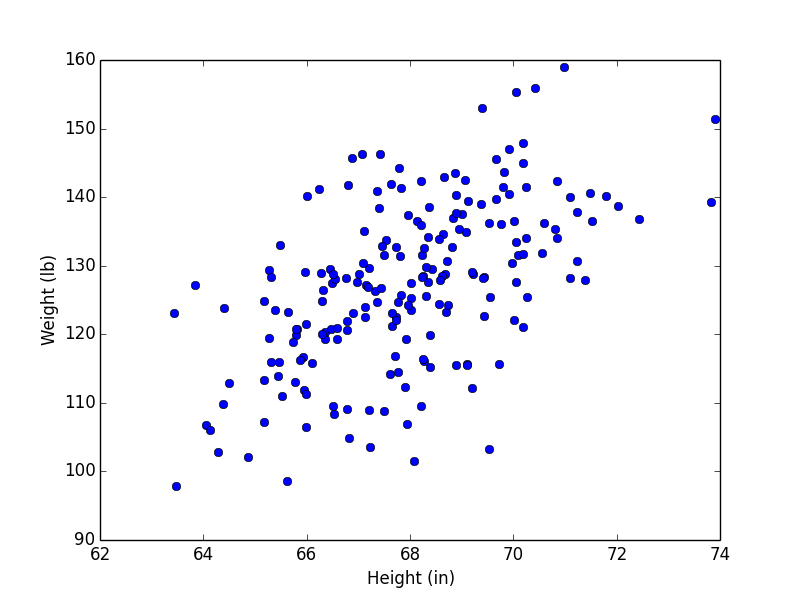
The purpose of machine learning is often to create a model that explains some real-world data, so that we can predict what may happen next, with different inputs.

The simplest model that we can fit to data is a line. When we are trying to find a line that fits a set of data best, we are performing Linear Regression.

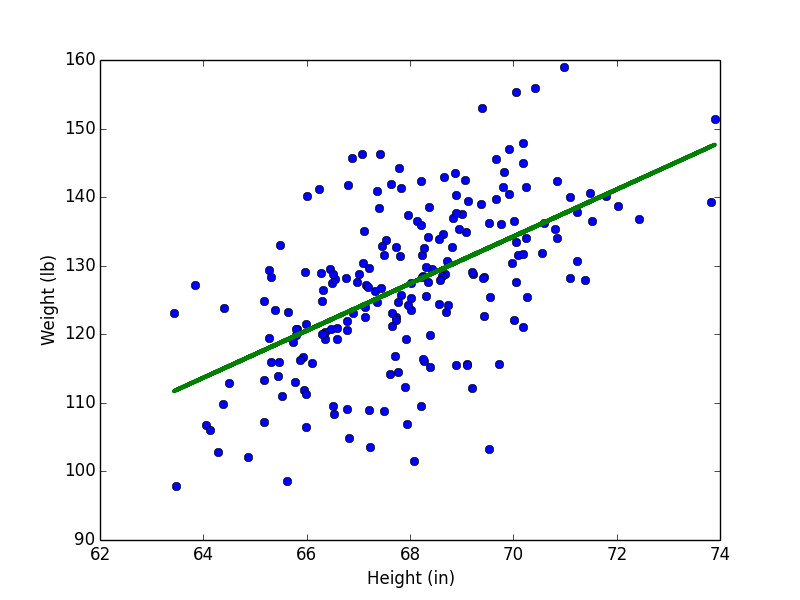
We often want to find lines to fit data, so that we can predict unknowns. For example:

* The market price of a house vs. the square footage of a house. Can we predict how much a house will sell for, given its size?
* The tax rate of a country vs. its GDP. Can we predict taxation based on a country's GDP?
* The amount of chips left in the bag vs. number of chips taken. Can we predict how much longer this bag of chips will last, given how much people at this party have been eating?

Imagine that we had this set of weights plotted against heights of a large set of professional baseball players:



To create a linear model to explain this data, we might draw this line:



Now, if we wanted to estimate the weight of a player with a height of 73 inches, we could estimate that it is around 143 pounds.

A line is a rough approximation, but it allows us the ability to explain and predict variables that have a linear relationship with each other. In the rest of the lesson, we will learn how to perform Linear Regression.

### Script.py

**import** codecademylib3\_seaborn

**import** matplotlib**.**pyplot **as** plt

months **=** **[**1**,** 2**,** 3**,** 4**,** 5**,** 6**,** 7**,** 8**,** 9**,** 10**,** 11**,** 12**]**

revenue **=** **[**52**,** 74**,** 79**,** 95**,** 115**,** 110**,** 129**,** 126**,** 147**,** 146**,** 156**,** 184**]**

plt**.**plot**(**months**,** revenue**,** "o"**)**

plt**.**title**(**"Sandra's Lemonade Stand"**)**

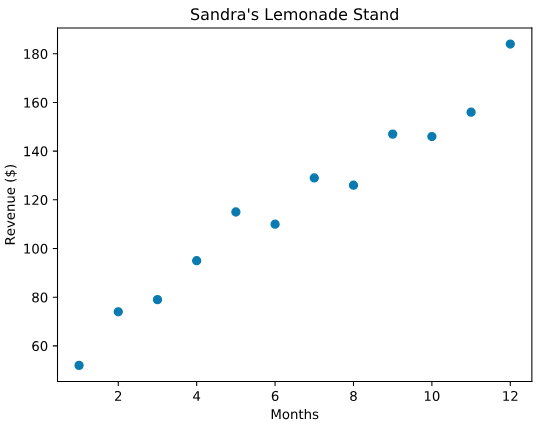
plt**.**xlabel**(**"Months"**)**

plt**.**ylabel**(**"Revenue ($)"**)**

plt**.**show**()**

# What do you think the revenue in month 13 would be?

### Script.py Output



## Points and lines

In the last exercise, you were probably able to make a rough estimate about the next data point for Sandra's lemonade stand without thinking too hard about it. For our program to make the same level of guess, we have to determine what a line would look like through those data points.

A line is determined by its *slope* and its *intercept*. In other words, for each point y on a line we can say: y = m x + by=mx+b

Where m is the slope, and b is the intercept. y is a given point on the y-axis, and it corresponds to a given x on the x-axis.

The slope is a measure of how steep the line is, while the intercept is a measure of where the line hits the y-axis.

When we perform Linear Regression, the goal is to get the "best" m and b for our data.

### Script.py

**import** codecademylib3\_seaborn

**import** matplotlib**.**pyplot **as** plt

months **=** **[**1**,** 2**,** 3**,** 4**,** 5**,** 6**,** 7**,** 8**,** 9**,** 10**,** 11**,** 12**]**

revenue **=** **[**52**,** 74**,** 79**,** 95**,** 115**,** 110**,** 129**,** 126**,** 147**,** 146**,** 156**,** 184**]**

#slope:

m **=** 12

#intercept:

b **=** 40

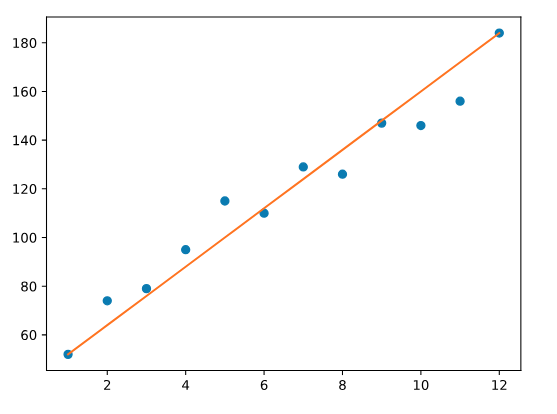
y **=** **[**m**\***x **+** b **for** x **in** months**]**

plt**.**plot**(**months**,** revenue**,** "o"**)**

plt**.**plot**(**months**,** y**)**

plt**.**show**()**

### Script.py Output



## Loss

When we think about how we can assign a slope and intercept to fit a set of points, we have to define what the best fit is.

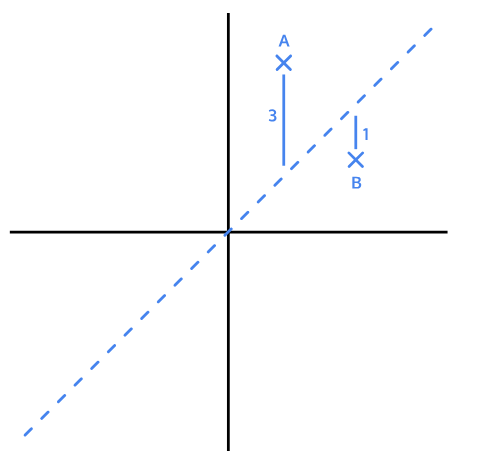
For each data point, we calculate loss, a number that measures how bad the model's (in this case, the line's) prediction was. You may have seen this being referred to as error.

We can think about loss as the squared distance from the point to the line. We do the squared distance (instead of just the distance) so that points above and below the line both contribute to total loss in the same way:

In this example:

* For point A, the squared distance is 9 (3²)
* For point B, the squared distance is 1 (1²)

So the total loss, with this model, is 10. If we found a line that had less loss than 10, that line would be a better model for this data.



## Minimizing Loss

*(use of :* [*https://s3.amazonaws.com/codecademy-content/programs/data-science-path/line-fitter/line-fitter.html*](https://s3.amazonaws.com/codecademy-content/programs/data-science-path/line-fitter/line-fitter.html) *)*

The goal of a linear regression model is to find the slope and intercept pair that minimizes loss on average across all of the data.

The interactive visualization in the browser lets you try to find the line of best fit for a random set of data points:

* The slider on the left controls the m (slope)
* The slider on the right controls the b (intercept)

You can see the total loss on the right side of the visualization. To get the line of best fit, we want this loss to be as small as possible.

To check if you got the best line, check the "Plot Best-Fit" box.

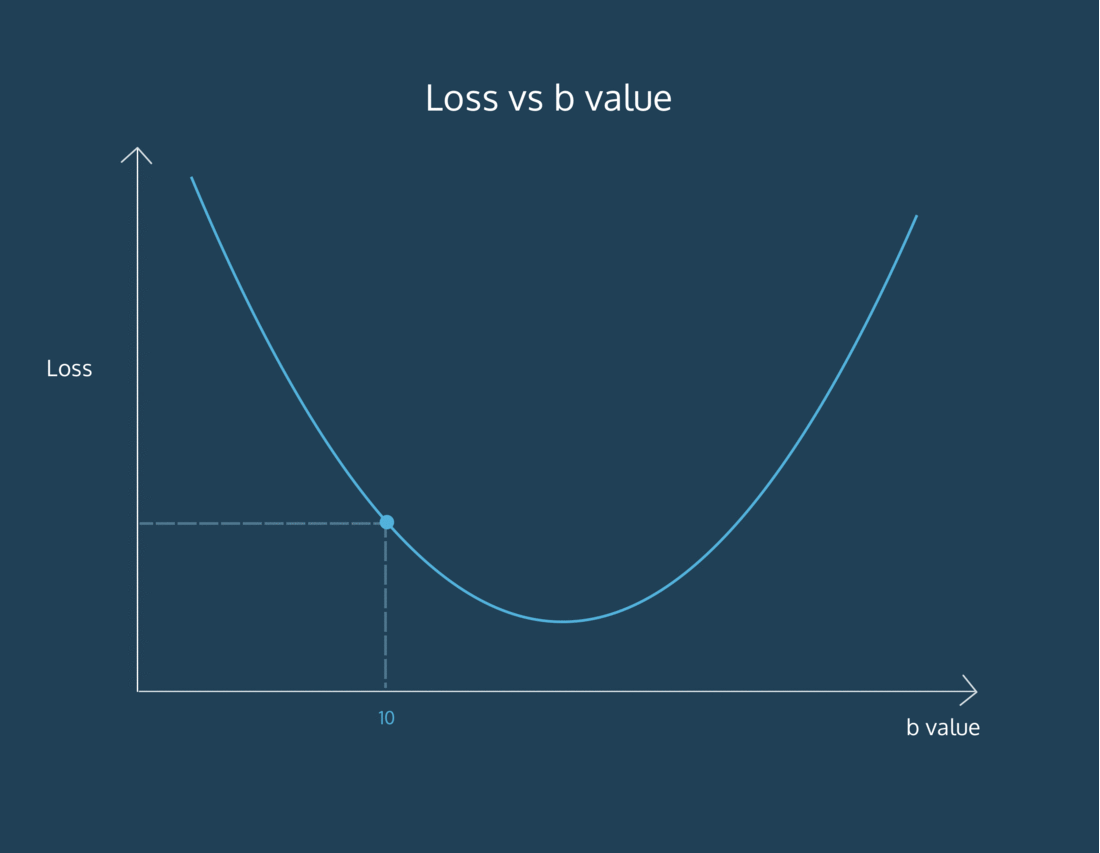
Randomize a new set of points and try to fit a new line by entering the number of points you want (try 8!) in the textbox and pressing randomize points

## Gradient Descent for Intercept

As we try to minimize loss, we take each parameter we are changing, and move it as long as we are decreasing loss. It's like we are moving down a hill, and stop once we reach the bottom:

The process by which we do this is called gradient descent. We move in the direction that decreases our loss the most. Gradient refers to the slope of the curve at any point.

For example, let's say we are trying to find the intercept for a line. We currently have a guess of 10 for the intercept. At the point of 10 on the curve, the slope is downward. Therefore, if we increase the intercept, we should be lowering the loss. So we follow the gradient downwards.



We derive these gradients using calculus. It is not crucial to understand how we arrive at the gradient equation. To find the gradient of loss as intercept changes, the formula comes out to be: \frac{2}{N}\sum\_{i=1}^{N}-(y\_i-(mx\_i+b))N2​i=1∑N​−(yi​−(mxi​+b))

* N is the number of points we have in our dataset
* m is the current gradient guess
* b is the current intercept guess

Basically:

* we find the sum of y\_value - (m\*x\_value + b) for all the y\_values and x\_values we have
* and then we multiply the sum by a factor of -2/N. N is the number of points we have.

### Script.py

**def** get\_gradient\_at\_b**(**x**,** y**,** m**,** b**):**

diff **=** 0

**for** i **in** range**(**0**,** len**(**x**)):**

y\_val **=** y**[**i**]**

x\_val **=** x**[**i**]**

diff **+=** **(**y\_val **-** **((**m**\***x\_val**)+**b**))**

b\_gradient **=** **(-**2**/**len**(**x**))\***diff

**return** b\_gradient

## Gradient Descent for Slope

We have a function to find the gradient of b at every point. To find the m gradient, or the way the loss changes as the slope of our line changes, we can use this formula:

\frac{2}{N}\sum\_{i=1}^{N}-x\_i(y\_i-(mx\_i+b))N2​i=1∑N​−xi​(yi​−(mxi​+b))

Once more:

* N is the number of points you have in your dataset
* m is the current gradient guess
* b is the current intercept guess

To find the m gradient:

* we find the sum of x\_value \* (y\_value - (m\*x\_value + b)) for all the y\_values and x\_values we have
* and then we multiply the sum by a factor of -2/N. N is the number of points we have.

Once we have a way to calculate both the m gradient and the bgradient, we'll be able to follow both of those gradients downwards to the point of lowest loss for both the m value and the b value. Then, we'll have the best m and the best b to fit our data!

### Script.py

**def** get\_gradient\_at\_m**(**x**,** y**,** m**,** b**):**

diff **=** 0

**for** i **in** range**(**0**,** len**(**x**)):**

y\_val **=** y**[**i**]**

x\_val **=** x**[**i**]**

diff **+=** x\_val**\*(**y\_val**-((**m**\***x\_val**)+**b**))**

m\_gradient **=** **-**2**/**N **\*** diff

**return** m\_gradient

## Put it Together

Now that we know how to calculate the gradient, we want to take a "step" in that direction. However, it's important to think about whether that step is too big or too small. We don't want to overshoot the minimum error!

We can scale the size of the step by multiplying the gradient by a *learning rate*.

To find a new b value, we would say:

new\_b **=** current\_b **-** **(**learning\_rate **\*** b\_gradient**)**

where current\_b is our guess for what the b value is, b\_gradient is the gradient of the loss curve at our current guess, and learning\_rateis proportional to the size of the step we want to take.

In a few exercises, we'll talk about the implications of a large or small learning rate, but for now, let's use a fairly small value.

### Script.py

**def** step\_gradient**(**x**,** y**,** b\_current**,** m\_current**):**

b\_gradient **=** get\_gradient\_at\_b**(**x**,** y**,** b\_current**,** m\_current**)**

m\_gradient **=** get\_gradient\_at\_m**(**x**,** y**,** b\_current**,** m\_current**)**

b **=** b\_current **-** **(**0.01 **\*** b\_gradient**)**

m **=** m\_current **-** **(**0.01 **\*** m\_gradient**)**

**return** **[**b**,** m**]**

months **=** **[**1**,** 2**,** 3**,** 4**,** 5**,** 6**,** 7**,** 8**,** 9**,** 10**,** 11**,** 12**]**

revenue **=** **[**52**,** 74**,** 79**,** 95**,** 115**,** 110**,** 129**,** 126**,** 147**,** 146**,** 156**,** 184**]**

# current intercept guess:

b **=** 0

# current slope guess:

m **=** 0

b**,** m **=** step\_gradient**(**months**,** revenue**,** b**,** m**)**

**print(**b**,** m**)**

## Convergence

How do we know when we should stop changing the parameters mand b? How will we know when our program has learned enough?

To answer this, we have to define convergence. **Convergence** is when the loss stops changing (or changes very slowly) when parameters are changed.

Hopefully, the algorithm will converge at the best values for the parameters m and b.

### Script.py

**import** codecademylib3\_seaborn

**import** matplotlib**.**pyplot **as** plt

**from** data **import** bs**,** bs\_000000001**,** bs\_01

iterations **=** range**(**1400**)**

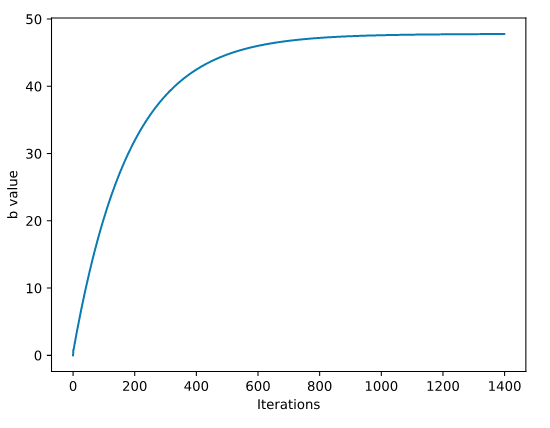
plt**.**plot**(**iterations**,** bs**)**

plt**.**xlabel**(**"Iterations"**)**

plt**.**ylabel**(**"b value"**)**

plt**.**show**()**

### Script.py Output



### Learning Rate

We want our program to be able to iteratively *learn* what the best mand b values are. So for each m and b pair that we guess, we want to move them in the direction of the gradients we've calculated. But how far do we move in that direction?

We have to choose a **learning rate**, which will determine how far down the loss curve we go.

A small learning rate will take a long time to converge — you might run out of time or cycles before getting an answer. A large learning rate might skip over the best value. It might *never* converge! Oh no!

Finding the absolute best learning rate is not necessary for training a model. You just have to find a learning rate large enough that gradient descent converges with the efficiency you need, and not so large that convergence never happens.  
  
example: two new lists representing how the bvalue changed with different learning rates:

* bs\_000000001: 1400 iterations of gradient descent on bwith a learning rate of 0.000000001
* bs\_01: 100 iterations of gradient descent on b with a learning rate of 0.01

Change the plot to plot bs\_000000001 instead of bs : straight line  
Change the plot to plot bs\_01 instead of bs : constant line until 90 sudden jump

## Example with built in function

### Script.py

**from** sklearn**.**linear\_model **import** LinearRegression

**import** matplotlib**.**pyplot **as** plt

**import** numpy **as** np

temperature **=** np**.**array**(**range**(**60**,** 100**,** 2**))**

temperature **=** temperature**.**reshape**(-**1**,** 1**)**

sales **=** **[**65**,** 58**,** 46**,** 45**,** 44**,** 42**,** 40**,** 40**,** 36**,** 38**,** 38**,** 28**,** 30**,** 22**,** 27**,** 25**,** 25**,** 20**,** 15**,** 5**]**

plt**.**plot**(**temperature**,** sales**,** 'o'**)**

line\_fitter **=** LinearRegression**()**

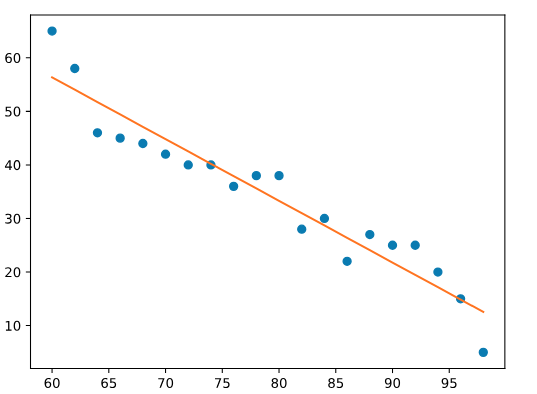
line\_fitter**.**fit**(**temperature**,** sales**)**

sales\_predict **=** line\_fitter**.**predict**(**temperature**)**

plt**.**plot**(**temperature**,** sales\_predict**)**

plt**.**show**()**

### Script.py Output



# MULTIPLE LINEAR REGRESSION

## Introduction to Multiple Linear Regression

Linear regression is useful when we want to predict the values of a variable from its relationship with other variables. There are two different types of linear regression models ([simple linear regression](https://www.codecademy.com/paths/data-science/tracks/dspath-supervised/modules/dspath-linear-regression/lessons/linear-regression)and multiple linear regression).

In predicting the price of a home, one factor to consider is the size of the home. The relationship between those two variables, price and size, is important, but there are other variables that factor in to pricing a home: location, air quality, demographics, parking, and more. When making predictions for price, our *dependent variable*, we'll want to use multiple *independent variables*. To do this, we'll use Multiple Linear Regression.

**Multiple Linear Regression** uses two or more independent variables to predict the values of the dependent variable. It is based on the following equation that we'll explore later on:



(on this example we will use this dataset: <https://streeteasy.com/blog/data-dashboard/> )

### Script.py

**import** matplotlib**.**pyplot **as** plt

**import** numpy **as** np

**import** pandas **as** pd

**from** mpl\_toolkits**.**mplot3d **import** Axes3D

**from** sklearn**.**linear\_model **import** LinearRegression

**from** sklearn**.**model\_selection **import** train\_test\_split

streeteasy **=** pd**.**read\_csv**(**"https://raw.githubusercontent.com/sonnynomnom/Codecademy-Machine-Learning-Fundamentals/master/StreetEasy/manhattan.csv"**)**

df **=** pd**.**DataFrame**(**streeteasy**)**

x **=** df**[[**'size\_sqft'**,**'building\_age\_yrs'**]]**

y **=** df**[[**'rent'**]]**

x\_train**,** x\_test**,** y\_train**,** y\_test **=** train\_test\_split**(**x**,** y**,** train\_size **=** 0.8**,** test\_size **=** 0.2**,** random\_state**=**6**)**

ols **=** LinearRegression**()**

ols**.**fit**(**x\_train**,** y\_train**)**

# Plot the figure

fig **=** plt**.**figure**(**1**,** figsize**=(**6**,** 4**))**

plt**.**clf**()**

elev **=** 43.5

azim **=** **-**110

ax **=** Axes3D**(**fig**,** elev**=**elev**,** azim**=**azim**)**

ax**.**scatter**(**x\_train**[[**'size\_sqft'**]],** x\_train**[[**'building\_age\_yrs'**]],** y\_train**,** c**=**'k'**,** marker**=**'+'**)**

ax**.**plot\_surface**(**np**.**array**([[**0**,** 0**],** **[**4500**,** 4500**]]),** np**.**array**([[**0**,** 140**],** **[**0**,** 140**]]),** ols**.**predict**(**np**.**array**([[**0**,** 0**,** 4500**,** 4500**],** **[**0**,** 140**,** 0**,** 140**]]).**T**).**reshape**((**2**,** 2**)),** alpha**=**.7**)**

ax**.**set\_xlabel**(**'Size (ft$^2$)'**)**

ax**.**set\_ylabel**(**'Building Age (Years)'**)**

ax**.**set\_zlabel**(**'Rent ($)'**)**

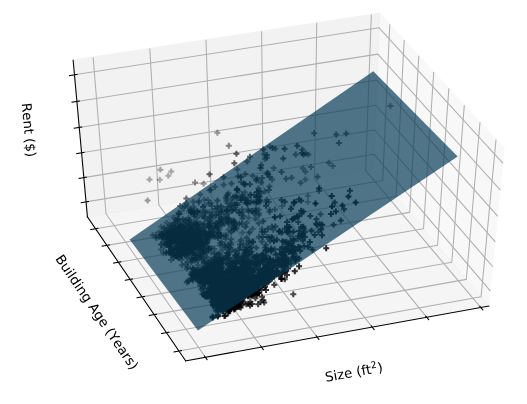
ax**.**w\_xaxis**.**set\_ticklabels**([])**

ax**.**w\_yaxis**.**set\_ticklabels**([])**

ax**.**w\_zaxis**.**set\_ticklabels**([])**

plt**.**show**()**

### Script.py Output



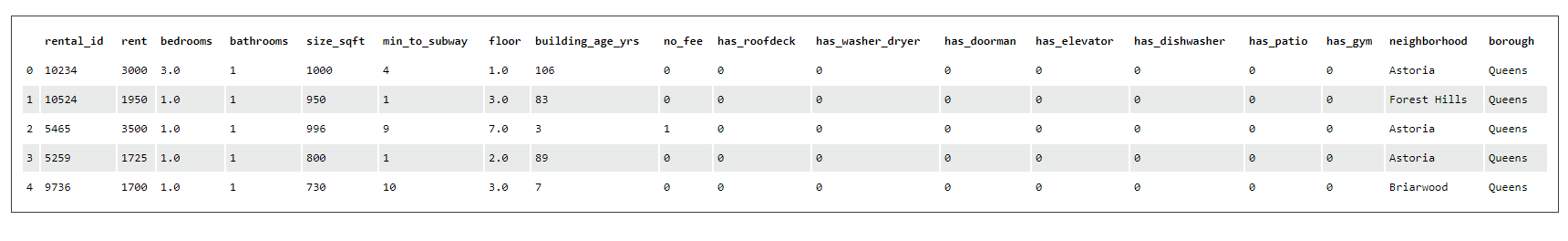
### StreetEasy Dataset example

**import** pandas **as** pd

streeteasy **=** pd**.**read\_csv**(**"https://raw.githubusercontent.com/sonnynomnom/Codecademy-Machine-Learning-Fundamentals/master/StreetEasy/queens.csv"**)**

df **=** pd**.**DataFrame**(**streeteasy**)**

**print(**df**.**head**())**



## Training Set vs. Test Set

As with most machine learning algorithms, we have to split our dataset into:

* **Training set**: the data used to fit the model
* **Test set**: the data partitioned away at the very start of the experiment (to provide an unbiased evaluation of the model)

In general, putting 80% of your data in the training set and 20% of your data in the test set is a good place to start.

Suppose you have some values in x and some values in y:

**from** sklearn**.**model\_selection **import** train\_test\_split x\_train**,** x\_test**,** y\_train**,** y\_test **=** train\_test\_split**(**x**,** y**,** train\_size**=**0.8**,** test\_size**=**0.2**)**

Here are the parameters:

train\_size: the proportion of the dataset to include in the train split (between 0.0 and 1.0)

test\_size: the proportion of the dataset to include in the test split (between 0.0 and 1.0)

random\_state: the seed used by the random number generator [optional]

### Script.py

**import** pandas **as** pd

**from** sklearn**.**model\_selection **import** train\_test\_split

streeteasy **=** pd**.**read\_csv**(**"https://raw.githubusercontent.com/sonnynomnom/Codecademy-Machine-Learning-Fundamentals/master/StreetEasy/manhattan.csv"**)**

df **=** pd**.**DataFrame**(**streeteasy**)**

x **=** df**[[**'bedrooms'**,** 'bathrooms'**,** 'size\_sqft'**,** 'min\_to\_subway'**,** 'floor'**,** 'building\_age\_yrs'**,** 'no\_fee'**,** 'has\_roofdeck'**,** 'has\_washer\_dryer'**,** 'has\_doorman'**,** 'has\_elevator'**,** 'has\_dishwasher'**,** 'has\_patio'**,** 'has\_gym'**]]**

y **=** df**[[**'rent'**]]**

x\_train**,** x\_test**,** y\_train**,** y\_test **=** train\_test\_split**(**x**,** y**,** train\_size **=** 0.8**,** test\_size **=** 0.2**,** random\_state**=**6**)**

**print(**x\_train**.**shape**)**

**print(**x\_test**.**shape**)**

**print(**y\_train**.**shape**)**

**print(**y\_test**.**shape**)**

### Script.py Output

**(**2831**,** 14**)**

**(**708**,** 14**)**

**(**2831**,** 1**)**

**(**708**,** 1**)**

## Training Set vs Validation Set vs Test Set

### TESTING OUR MODEL

Supervised machine learning algorithms are amazing tools capable of making predictions and classifications. However, it is important to ask yourself how accurate those predictions are. After all, it’s possible that every prediction your classifier makes is actually wrong! Luckily, we can leverage the fact that supervised machine learning algorithms, by definition, have a dataset of pre-labeled datapoints. In order to test the effectiveness of your algorithm, we’ll split this data into:

**training set**

**validation set**

**test set**

### TRAINING SET VS VALIDATION SET

The training set is the data that the algorithm will learn from. Learning looks different depending on which algorithm you are using. For example, when using *Linear Regression*, the points in the training set are used to draw the line of best fit. In *K-Nearest Neighbors*, the points in the training set are the points that could be the neighbors.

After training using the training set, the points in the validation set are used to compute the accuracy or error of the classifier. The key insight here is that we know the true labels of every point in the validation set, but we’re temporarily going to pretend like we don’t. We can use every point in the validation set as input to our classifier. We’ll then receive a classification for that point. We can now peek at the true label of the validation point and see whether we got it right or not. If we do this for every point in the validation set, we can compute the validation error!

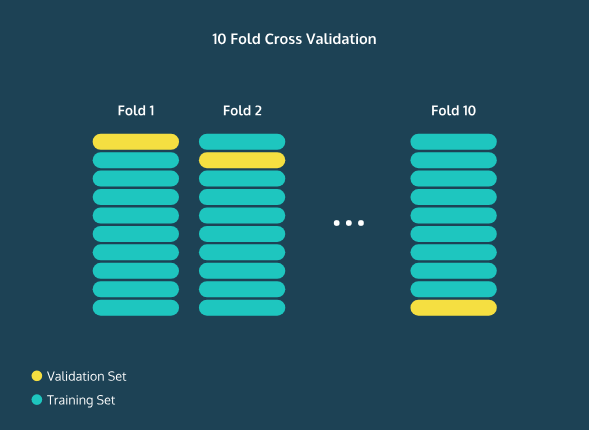
Validation error might not be the only metric we’re interested in. A better way of judging the effectiveness of a machine learning algorithm is to compute its [precision, recall, and F1 score](https://www.codecademy.com/content-items/1dd2cb55a893072f4dce2911004eeba2/exercises/accuracy).

### HOW TO SPLIT

Figuring out how much of your data should be split into your validation set is a tricky question. If your training set is too small, then your algorithm might not have enough data to effectively learn. On the other hand, if your validation set is too small, then your accuracy, precision, recall, and F1 score could have a large variance. You might happen to get a really lucky or a really unlucky split! In general, putting 80% of your data in the training set, and 20% of your data in the validation set is a good place to start.

### N-FOLD CROSS-VALIDATION

Sometimes your dataset is so small, that splitting it 80/20 will still result in a large amount of variance. One solution to this is to perform **N-Fold Cross-Validation**. The central idea here is that we’re going to do this entire process N times and average the accuracy. For example, in 10-fold cross-validation, we’ll make the validation set the first 10% of the data and calculate accuracy, precision, recall and F1 score. We’ll then make the validation set the second 10% of the data and calculate these statistics again. We can do this process 10 times, and every time the validation set will be a different chunk of the data. If we then average all of the accuracies, we will have a better sense of how our model does on average.



### CHANGING THE MODEL / TEST SET

### Understanding the accuracy of your model is invaluable because you can begin to tune the parameters of your model to increase its performance. For example, in the K-Nearest Neighbors algorithm, you can watch what happens to accuracy as you increase or decrease K. You can also use this information to try to avoid overfitting or underfitting your data.

Once you’re happy with your model’s performance, it is time to introduce the test set. This is part of your data that you partitioned away at the very start of your experiment. It’s meant to be a substitute for the data in the real world that you’re actually interested in classifying. It functions very similarly to the validation set, except you never touched this data while building or tuning your model. By finding the accuracy, precision, recall, and F1 score on the test set, you get a good understanding of how well your algorithm will do in the real world.

## Multiple Linear Regression: Scikit-Learn

### Script.py

**import** codecademylib3\_seaborn

**import** matplotlib**.**pyplot **as** plt

**import** pandas **as** pd

**from** sklearn**.**model\_selection **import** train\_test\_split

**from** sklearn**.**linear\_model **import** LinearRegression

streeteasy **=** pd**.**read\_csv**(**"https://raw.githubusercontent.com/sonnynomnom/Codecademy-Machine-Learning-Fundamentals/master/StreetEasy/manhattan.csv"**)**

df **=** pd**.**DataFrame**(**streeteasy**)**

x **=** df**[[**'bedrooms'**,** 'bathrooms'**,** 'size\_sqft'**,** 'min\_to\_subway'**,** 'floor'**,** 'building\_age\_yrs'**,** 'no\_fee'**,** 'has\_roofdeck'**,** 'has\_washer\_dryer'**,** 'has\_doorman'**,** 'has\_elevator'**,** 'has\_dishwasher'**,** 'has\_patio'**,** 'has\_gym'**]]**

y **=** df**[[**'rent'**]]**

x\_train**,** x\_test**,** y\_train**,** y\_test **=** train\_test\_split**(**x**,** y**,** train\_size **=** 0.8**,** test\_size **=** 0.2**,** random\_state**=**6**)**

# Add the code here:

mlr **=** LinearRegression**()**

mlr**.**fit**(**x\_train**,** y\_train**)**

y\_predict **=** mlr**.**predict**(**x\_test**)**

# Sonny doesn't have an elevator so the 11th item in the list is a 0

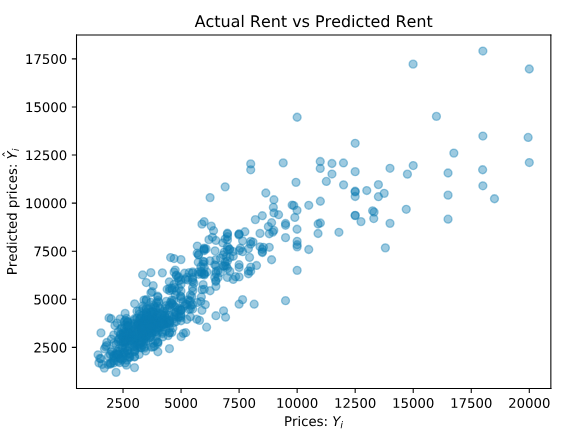
sonny\_apartment **=** **[[**1**,** 1**,** 620**,** 16**,** 1**,** 98**,** 1**,** 0**,** 1**,** 0**,** 0**,** 0**,** 1**,** 0**]]**

predict **=** mlr**.**predict**(**sonny\_apartment**)**

**print(**"Predicted rent: $%.2f" **%** predict**)**

### Script.py Output

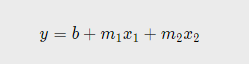
Predicted rent: $2451.48  
Using MathplotLib we can get graph like:



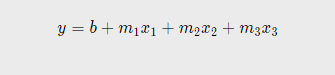
## Multiple Linear Regression Equation

Now that we have implemented Multiple Linear Regression, we will learn how to tune and evaluate the model. Before we do that, however, it's essential to learn the equation behind it.

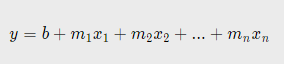
Equation 6.1 The equation for multiple linear regression that uses two independent variables is this:



Equation 6.2 The equation for multiple linear regression that uses three independent variables is this:



Equation 6.3 As a result, since multiple linear regression can use any number of independent variables, its general equation becomes:



Here, m1, m2, m3, ... mn refer to the coefficients, and b refers to the intercept that you want to find. You can plug these values back into the equation to compute the predicted y values.

Remember, with sklearn's LinearRegression() method, we can get these values with ease.

The .fit() method gives the model two variables that are useful to us:

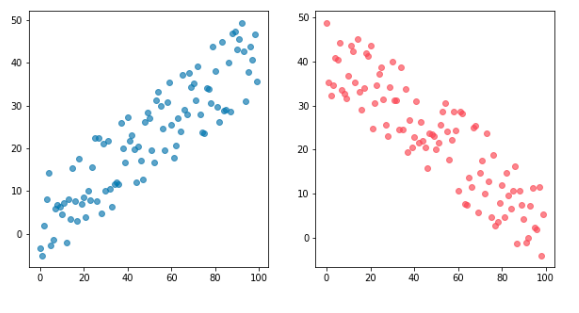
* .coef\_, which contains the coefficients
* .intercept\_, which contains the intercept

After performing multiple linear regression, you can print the coefficients using .coef\_.

Coefficients are most helpful in determining which independent variable carries more weight. For example, a coefficient of -1.345 will impact the rent more than a coefficient of 0.238, with the former impacting prices negatively and latter positively.

## Correlations

To see if there are any features that don't affect price linearly, let's graph the different features against rent. In regression, the independent variables will either have a positive linear relationship to the dependent variable, a negative linear relationship, or no relationship. A negative linear relationship means that as X values *increase*, Y values will *decrease*. Similarly, a positive linear relationship means that as X values *increase*, Y values will also *increase*. Graphically, when you see a downward trend, it means a negative linear relationship exists. When you find an upward trend, it indicates a positive linear relationship. Here are two graphs indicating positive and negative linear relationships:



### Script.py

**import** matplotlib**.**pyplot **as** plt

**import** pandas **as** pd

**from** sklearn**.**model\_selection **import** train\_test\_split

**from** sklearn**.**linear\_model **import** LinearRegression

streeteasy **=** pd**.**read\_csv**(**"https://raw.githubusercontent.com/sonnynomnom/Codecademy-Machine-Learning-Fundamentals/master/StreetEasy/manhattan.csv"**)**

df **=** pd**.**DataFrame**(**streeteasy**)**

# Input code here:

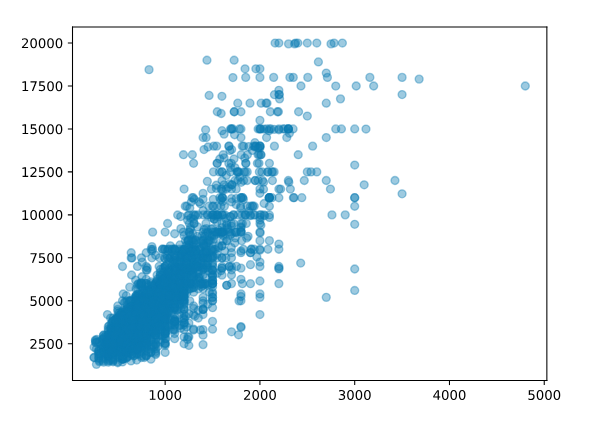
plt**.**scatter**(**df**[[**'size\_sqft'**]],** df**[[**'rent'**]],** alpha**=**0.4**)**

# plt.scatter(df[['floor']], df[['rent']])

# plt.scatter(df[['min\_to\_subway']], df[['rent']])

plt**.**show**()**

### Script.py Output



## Evaluating the Model's Accuracy

When trying to evaluate the accuracy of our multiple linear regression model, one technique we can use is **Residual Analysis**.

The difference between the actual value *y*, and the predicted value *ŷ* is the **residual***e*. The equation is:



In the StreetEasy dataset, *y* is the actual rent and the *ŷ* is the predicted rent. The real *y* values should be pretty close to these predicted *y* values.

sklearn's linear\_model.LinearRegression comes with a .score()method that returns the coefficient of determination R² of the prediction.

The coefficient R² is defined as:

1 - \frac{u}{v}

where *u* is the residual sum of squares and *v* is the total sum of squares (TSS).

The TSS tells you how much variation there is in the y variable.

R² is the percentage variation in y explained by all the x variables together. For example, say we are trying to predict rent based on the size\_sqft and the bedrooms in the apartment and the R² for our model is 0.72 — that means that all the x variables (square feet and number of bedrooms) together explain 72% variation in y (rent). Now let's say we add another x variable, building's age, to our model. By adding this third relevant x variable, the R² is expected to go up. Let say the new R² is 0.95. This means that square feet, number of bedrooms and age of the building *together* explain 95% of the variation in the rent. The best possible R² is 1.00 (and it can be negative because the model can be arbitrarily worse). Usually, a R² of 0.70 is considered good.

### Script.py

**import** matplotlib**.**pyplot **as** plt

**import** pandas **as** pd

**from** sklearn**.**model\_selection **import** train\_test\_split

**from** sklearn**.**linear\_model **import** LinearRegression

streeteasy **=** pd**.**read\_csv**(**"https://raw.githubusercontent.com/sonnynomnom/Codecademy-Machine-Learning-Fundamentals/master/StreetEasy/manhattan.csv"**)**

df **=** pd**.**DataFrame**(**streeteasy**)**

x **=** df**[[**'bedrooms'**,** 'bathrooms'**,** 'size\_sqft'**,** 'min\_to\_subway'**,** 'floor'**,** 'building\_age\_yrs'**,** 'no\_fee'**,** 'has\_roofdeck'**,** 'has\_washer\_dryer'**,** 'has\_doorman'**,** 'has\_elevator'**,** 'has\_dishwasher'**,** 'has\_patio'**,** 'has\_gym'**]]**

y **=** df**[[**'rent'**]]**

x\_train**,** x\_test**,** y\_train**,** y\_test **=** train\_test\_split**(**x**,** y**,** train\_size **=** 0.8**,** test\_size **=** 0.2**,** random\_state**=**6**)**

mlr **=** LinearRegression**()**

model**=**mlr**.**fit**(**x\_train**,** y\_train**)**

y\_predict **=** mlr**.**predict**(**x\_test**)**

# Input code here:

**import** codecademylib3\_seaborn

**import** matplotlib**.**pyplot **as** plt

**import** pandas **as** pd

**from** sklearn**.**model\_selection **import** train\_test\_split

**from** sklearn**.**linear\_model **import** LinearRegression

streeteasy **=** pd**.**read\_csv**(**"https://raw.githubusercontent.com/sonnynomnom/Codecademy-Machine-Learning-Fundamentals/master/StreetEasy/manhattan.csv"**)**

df **=** pd**.**DataFrame**(**streeteasy**)**

x **=** df**[[**'bedrooms'**,** 'bathrooms'**,** 'size\_sqft'**,** 'min\_to\_subway'**,** 'floor'**,** 'building\_age\_yrs'**,** 'no\_fee'**,** 'has\_roofdeck'**,** 'has\_washer\_dryer'**,** 'has\_doorman'**,** 'has\_elevator'**,** 'has\_dishwasher'**,** 'has\_patio'**,** 'has\_gym'**]]**

y **=** df**[[**'rent'**]]**

x\_train**,** x\_test**,** y\_train**,** y\_test **=** train\_test\_split**(**x**,** y**,** train\_size **=** 0.8**,** test\_size **=** 0.2**,** random\_state**=**6**)**

mlr **=** LinearRegression**()**

model**=**mlr**.**fit**(**x\_train**,** y\_train**)**

y\_predict **=** mlr**.**predict**(**x\_test**)**

# Input code here:

**print(**"Train score:"**)**

**print(**mlr**.**score**(**x\_train**,** y\_train**))**

**print(**"Test score:"**)**

**print(**mlr**.**score**(**x\_test**,** y\_test**))**

### Script.py Output

Train score: 0.7725460559817883 Test score: 0.8050371975357647