Linear Regression

With Python

Simple Linear Regression

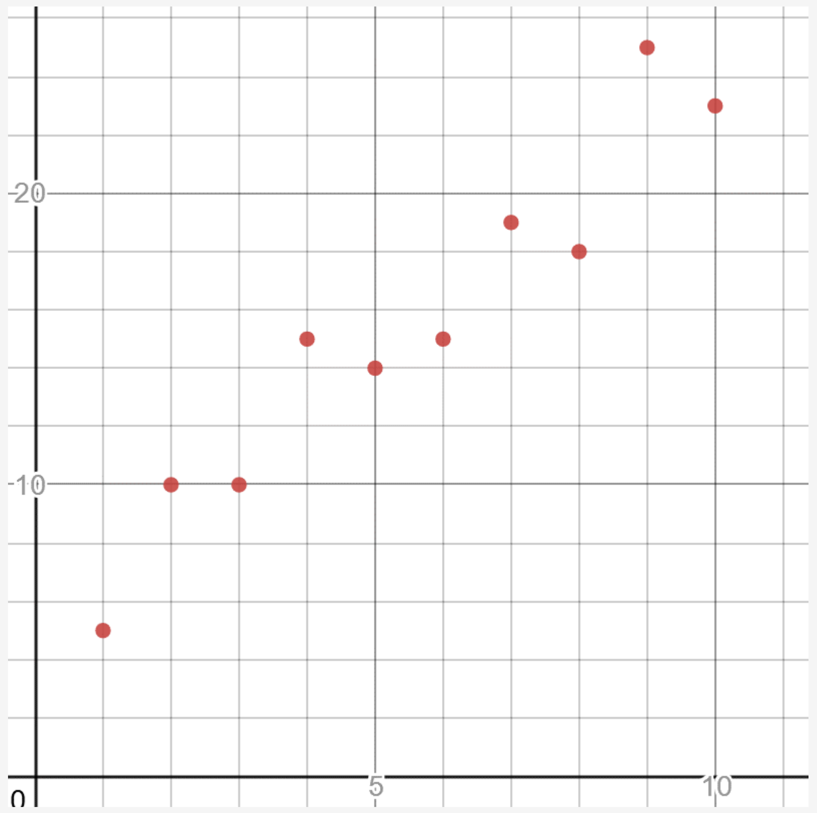
Simple Linear Regression

In practicality, machine learning is often about two tasks: regression and classification.

Let's start with regression, particularly the simplest one called **linear regression** which finds the best fit straight line through some points.

Here is a simple 2-dimensional plot of two variables, where x is independent and y is dependent.

- **Independent variables** are observed values that will serve as inputs into a function.
- **Dependent variables** are the outputted variables derived off the independent variables.



https://www.desmos.com/calculator/fmhotfn3qm

Simple Linear Regression

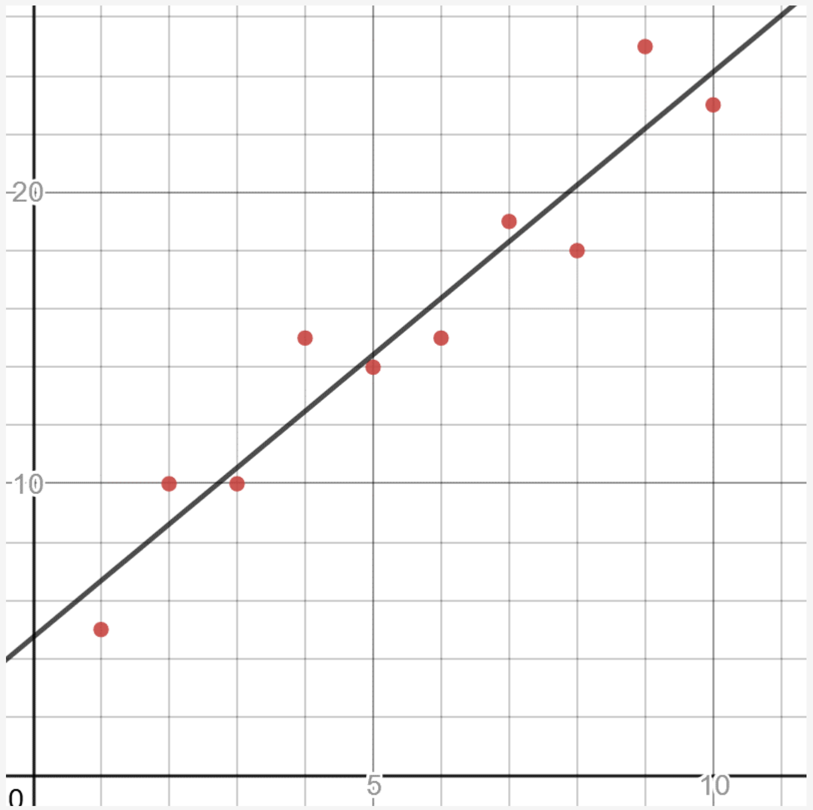
For a simple linear regression, we want to find a function $\boldsymbol{y} = \boldsymbol{mx} + \boldsymbol{b}$ that best fits to these points.

$$y = mx + b$$

We already know the $\boldsymbol{x}$ and $\boldsymbol{y}$ values from our existing data (the red points).

So the missing information is "what $\boldsymbol{m}$ and $\boldsymbol{b}$ values will create the best fit line"?

But before we solve for the $\boldsymbol{m}$ and $\boldsymbol{b}$ values, let's first ask "what defines a best fit anyway?"



https://www.desmos.com/calculator/fmhotfn3qm

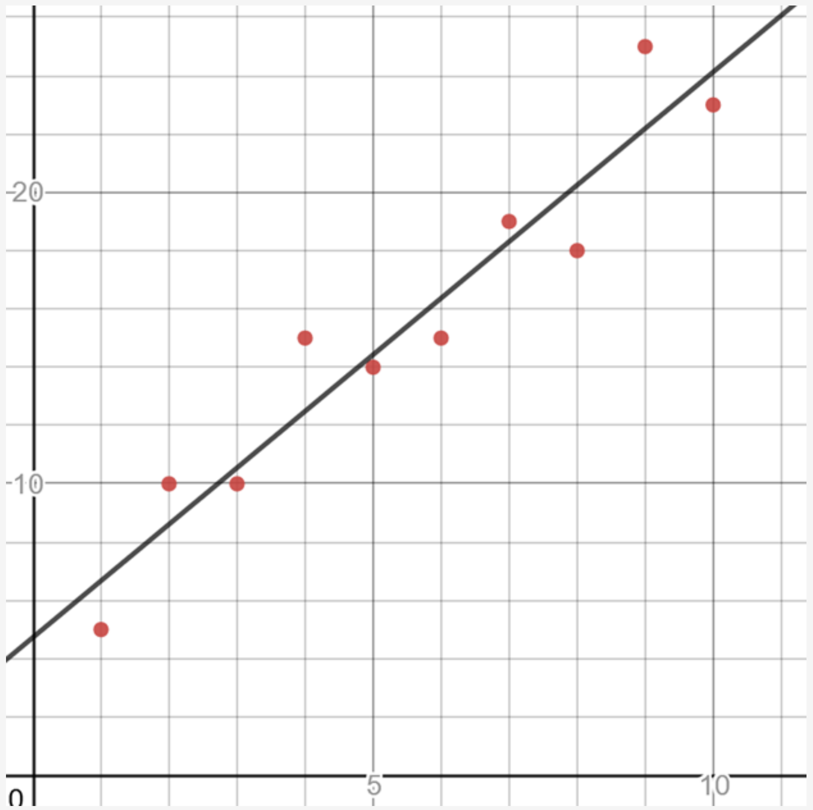
Simple Linear Regression

Pretend you drew any straight line through the points.

You will not be able to fit a perfect line, because the points do not exist on a straight line.

- Machine learning models are never perfect, as real-world data is never perfect.
- But you can fit a line to estimate a new y value for a given x value, even if there is an inevitable margin of error called loss.

Even if there is loss, it can be helpful in estimating predictions, such as how much y growth there will be at x time in the future.



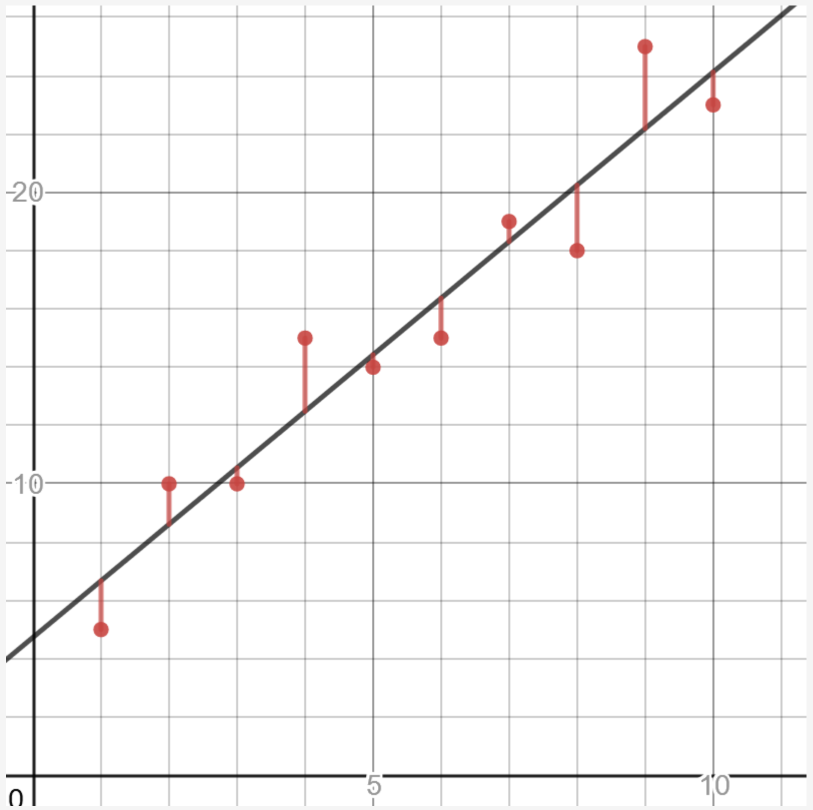
https://www.desmos.com/calculator/fmhotfn3qm

Simple Linear Regression

When you plot a given line, notice that there is a difference between the predicted y values and actual y values with our observed data.

These are called **residuals**, and in machine learning we want to minimize them.

So how do we minimize residuals in aggregate?



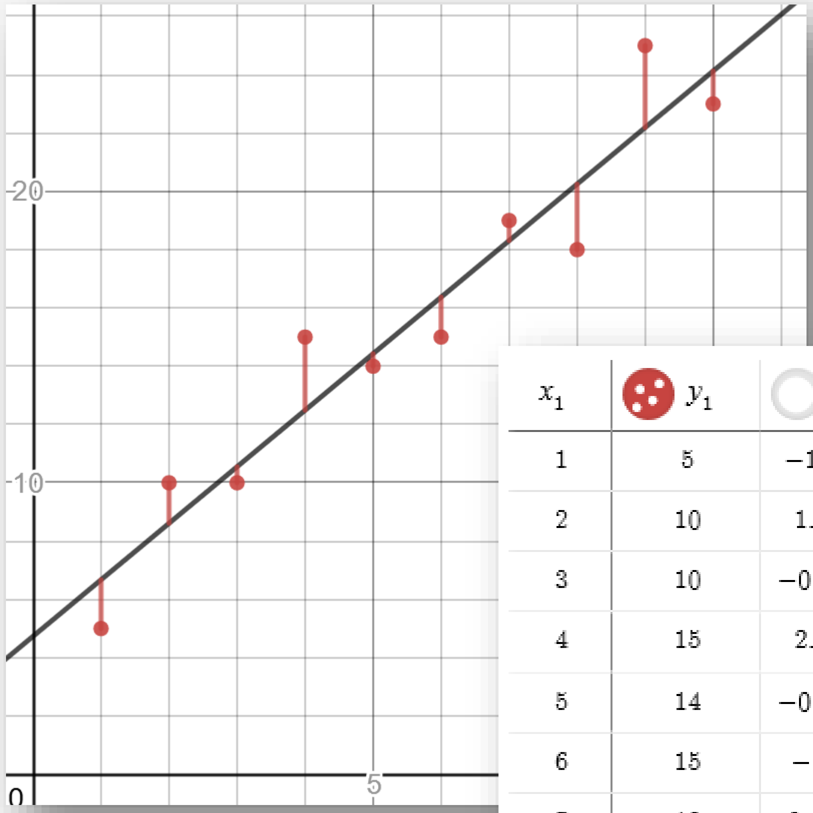
https://www.desmos.com/calculator/fmhotfn3qm

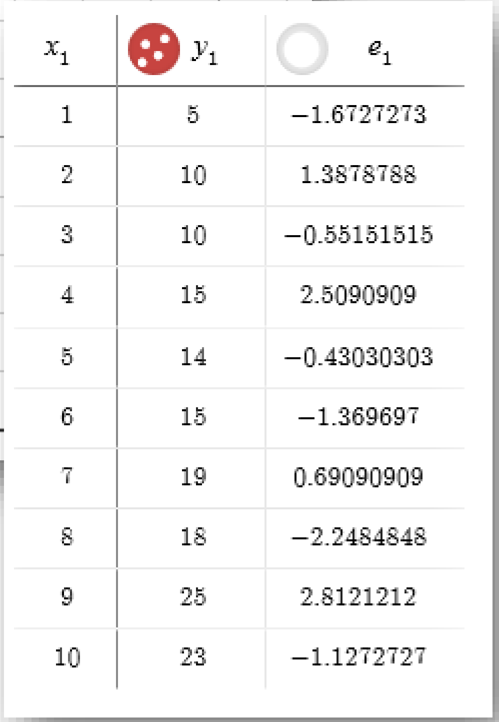
Sum of Least Squares

When we solve for $\boldsymbol{m}$ and $\boldsymbol{b}$ values, we need an objective to minimize total residuals using a loss function.

- We do not necessarily want to sum up the residuals, as the negatives will cancel out the positives.
- We can sum the absolute values, but that does not amplify larger residuals, and absolute values are mathematically difficult to work with.

So what is the best approach to total all the residuals to evaluate the quality of the fit?





x_1	y_1	e_1
1	5	−1.6727273
2	10	1.3878788
3	10	−0.55151515
4	15	2.5090909
5	14	−0.43030303
6	15	−1.369697
7	19	0.69090909
8	18	−2.2484848
9	25	2.8121212
10	23	−1.1272727

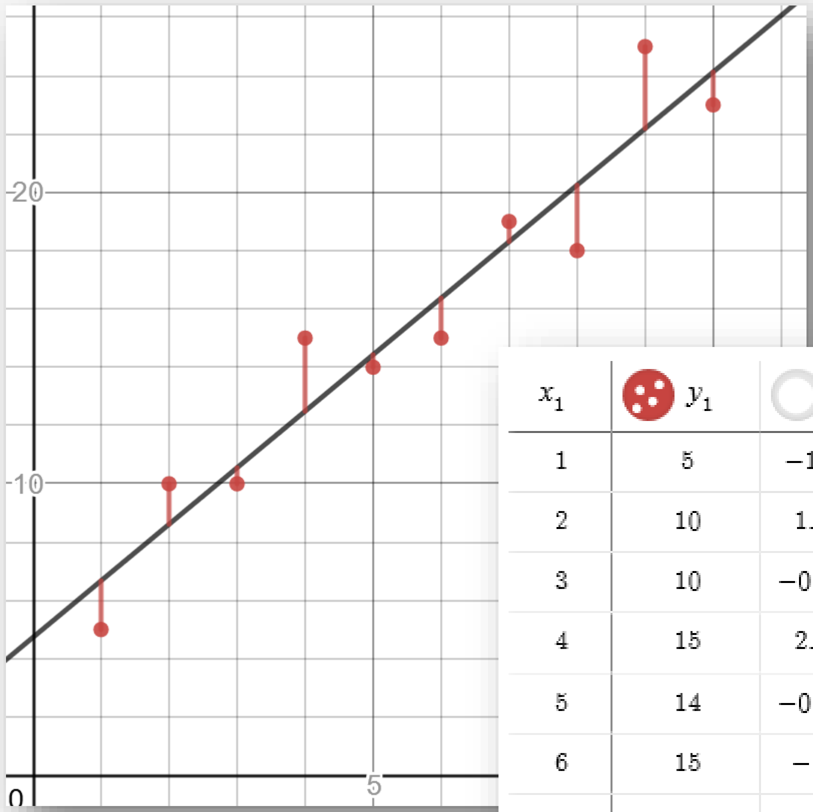
Sum of Least Squares

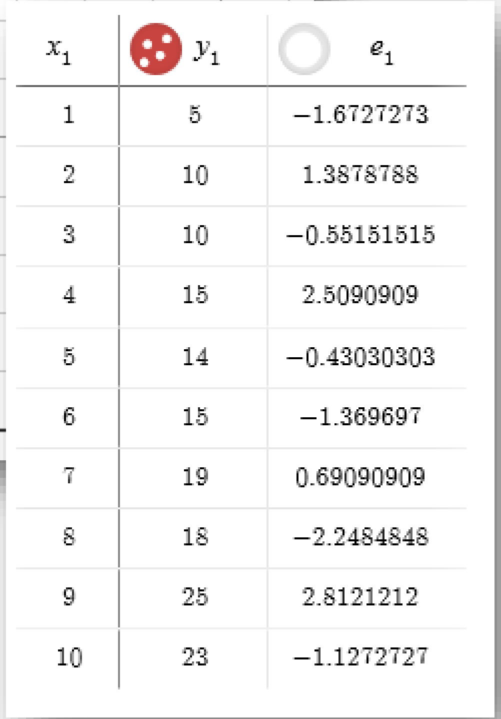
The best approach: we can square the residuals and sum them!

- Squaring will penalize large residuals by making them even larger!
- Squaring will also conveniently turn negatives into positives.
- Although we are not going to dive into calculus, squares are much easier to take the derivative of.

We can then find the $\boldsymbol{m}$ and $\boldsymbol{b}$ values that will find the sum of least squares.

https://www.desmos.com/calculator/fvrnuhw0hy

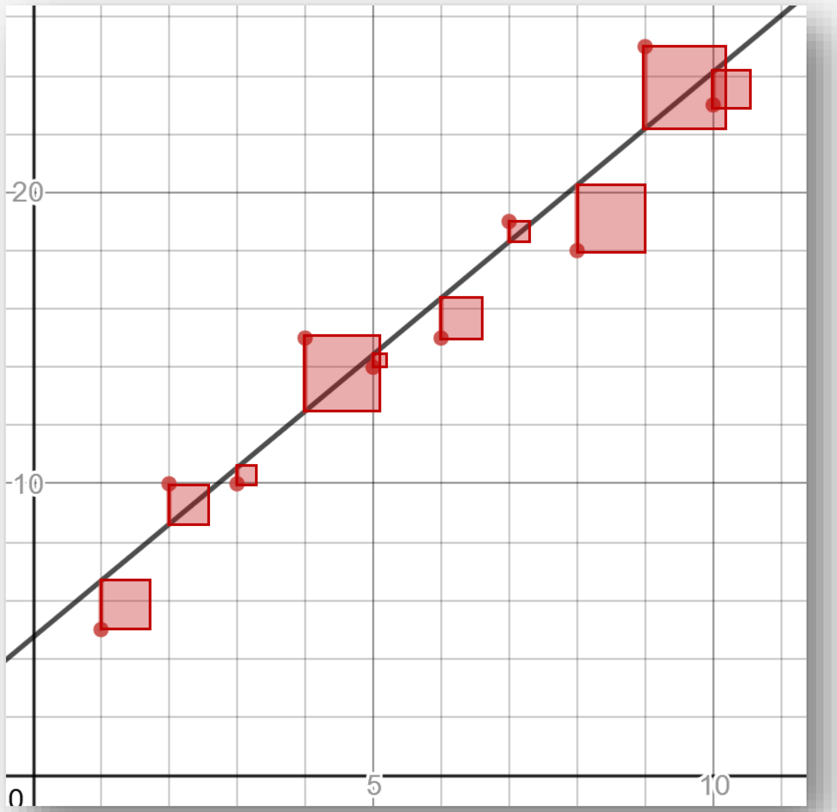




x_1	y_1	e_1
1	5	−1.6727273
2	10	1.3878788
3	10	−0.55151515
4	15	2.5090909
5	14	−0.43030303
6	15	−1.369697
7	19	0.69090909
8	18	−2.2484848
9	25	2.8121212
10	23	−1.1272727

Calculating Sum of Squares

m	b			
1.93939	4.73333			
x	y	y_predict (mx+b)	residual	residual squared
1	5	6.67272	1.67272	2.797992198
2	10	8.61211	-1.38789	1.926238652
3	10	10.5515	0.5515	0.30415225
4	15	12.49089	-2.50911	6.295632992
5	14	14.43028	0.43028	0.185140878
6	15	16.36967	1.36967	1.875995909
7	19	18.30906	-0.69094	0.477398084
8	18	20.24845	2.24845	5.055527402
9	25	22.18784	-2.81216	7.908243866
10	23	24.12723	1.12723	1.270647473
			Sum of Squares	28.0969697

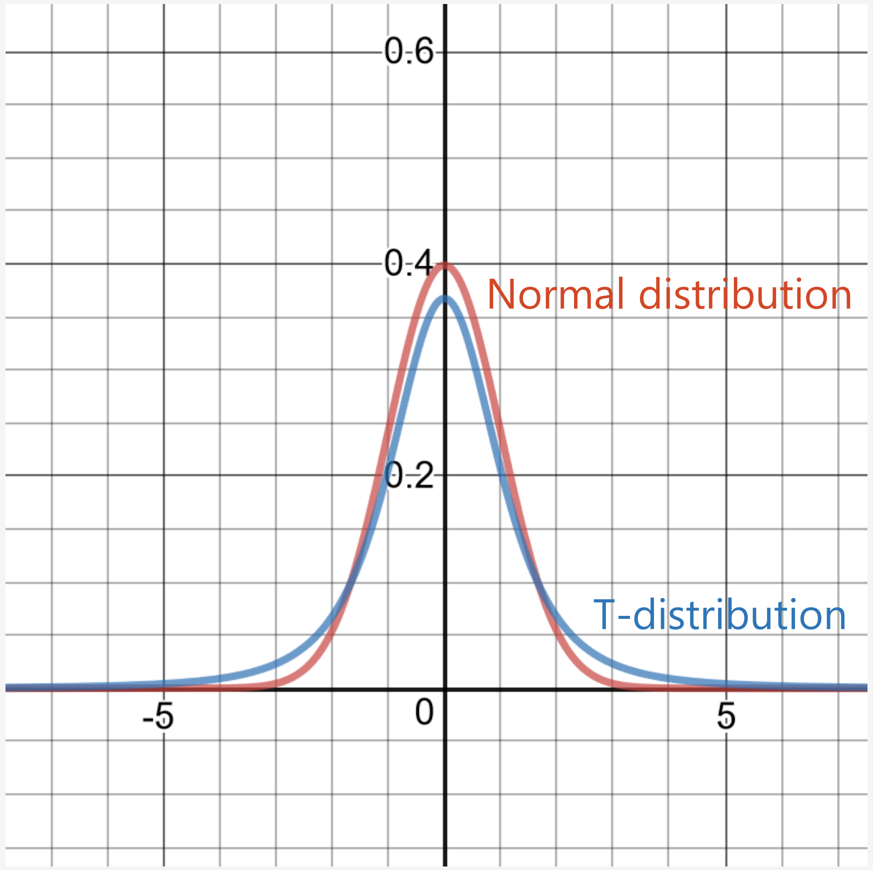


Solving for a Fit

Now that we have a loss function (the sum of squared residuals), how do we find the **m** and **b** values to minimize it?

- We randomly increase/decrease **m** and **b** with random values from a standard normal distribution, or even better a T-Distribution which has fatter tails.
- Fatter tails = more smaller/larger values = more diverse moves.

We can use these random adjustments to **m** and **b** as our moves in **hill climbing**.



https://www.desmos.com/calculator/xm56tvvalh

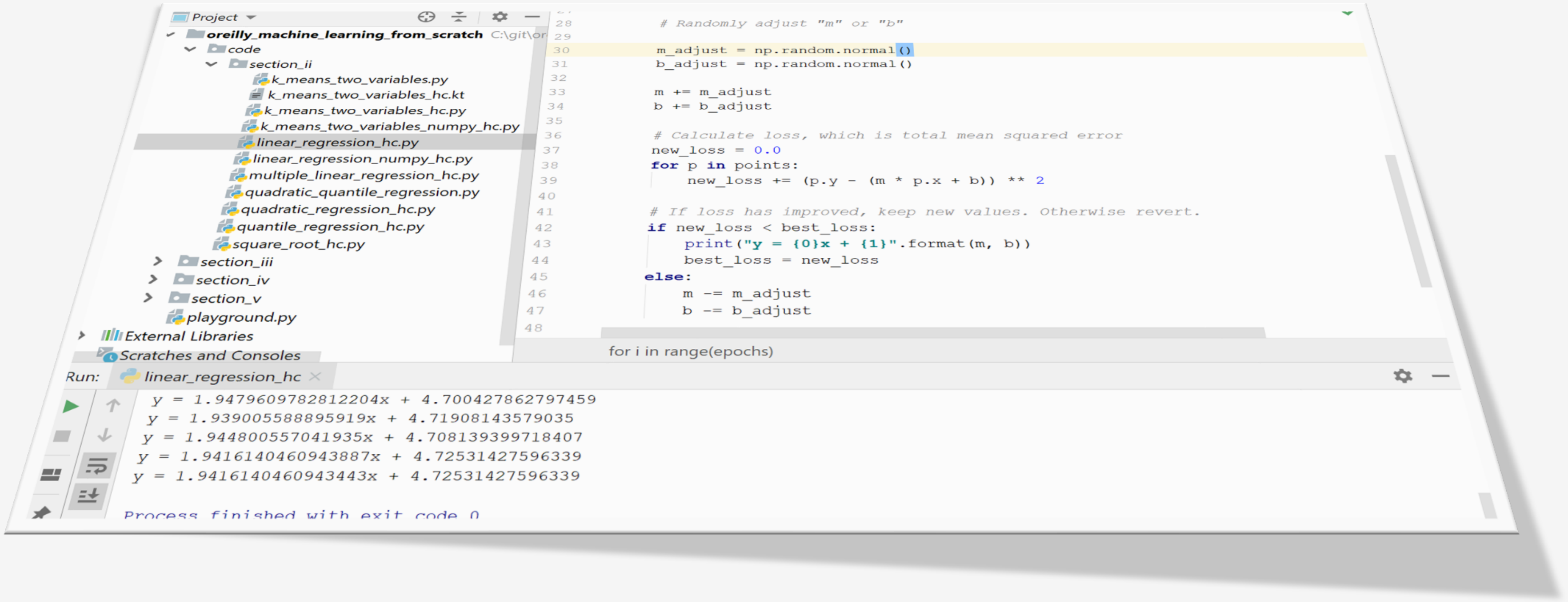
Hill Climbing – A Simple Optimization Algorithm

1 Start with a random or initial solution, even if it is poor quality.

2 Repeat the following steps for a number of iterations and/or until the solution cannot improve anymore.

1. Select a random part of the solution and change it.
2. If that results in an improvement, keep it.

Hands-On: Simple Linear Regression



```
Project
oreilly_machine_learning_from_scratch  C:\git\or
  code
    section_ii
      k_means_two_variables.py
      k_means_two_variables_hc.kt
      k_means_two_variables_hc.py
      k_means_two_variables_numpy_hc.py
      linear_regression_hc.py
      linear_regression_numpy_hc.py
      multiple_linear_regression_hc.py
      quadratic_quantile_regression.py
      quadratic_regression_hc.py
      quantile_regression_hc.py
      square_root_hc.py
    section_iii
    section_iv
    section_v
    playground.py
External Libraries
Scratches and Consoles
```

```python
        # Randomly adjust "m" or "b"
        m_adjust = np.random.normal()
        b_adjust = np.random.normal()

        m += m_adjust
        b += b_adjust

        # Calculate loss, which is total mean squared error
        new_loss = 0.0
        for p in points:
            new_loss += (p.y - (m * p.x + b)) ** 2

        # If loss has improved, keep new values. Otherwise revert.
        if new_loss < best_loss:
            print("y = {0}x + {1}".format(m, b))
            best_loss = new_loss
        else:
            m -= m_adjust
            b -= b_adjust

```

for i in range(epochs)

Run: linear_regression_hc

```
y = 1.9479609782812204x + 4.700427862797459
y = 1.939005588895919x + 4.71908143579035
y = 1.944800557041935x + 4.708139399718407
y = 1.9416140460943887x + 4.72531427596339
y = 1.9416140460943443x + 4.72531427596339

Process finished with exit code 0
```

Simple Linear Regression: Scikit-Learn

Performing a simple linear regression using Scikit-Learn is straightforward.

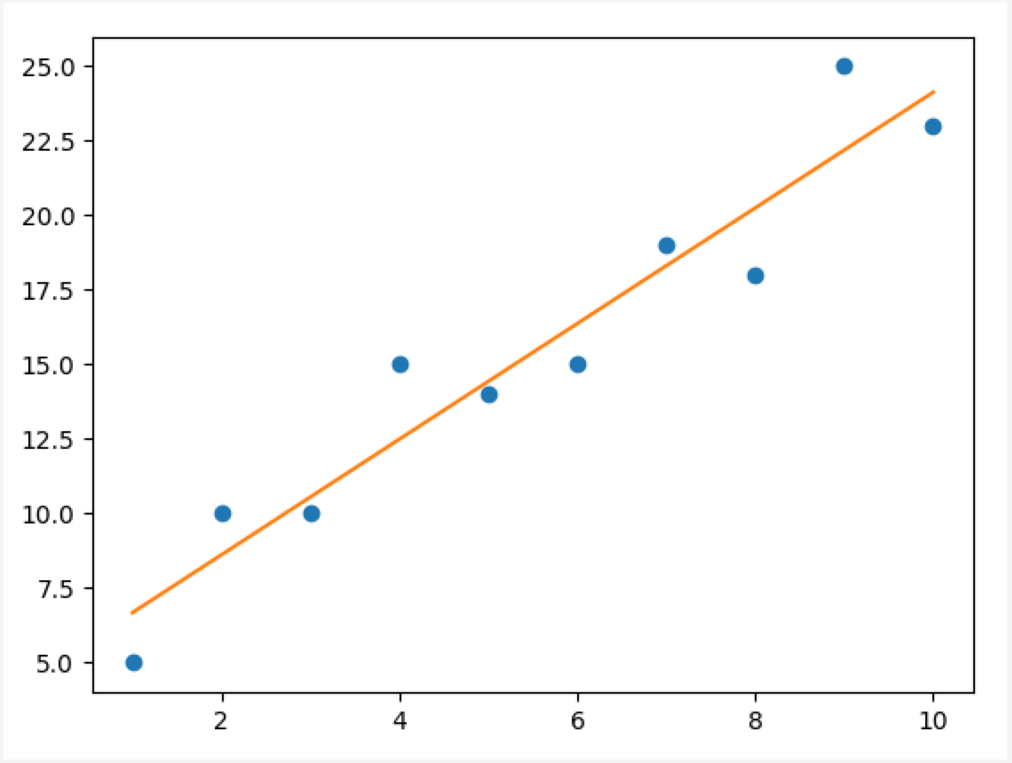
Creating a **`LinearRegression()`** and then calling its **`fit()`** function will return a trained model that can predict new values.

You can extract the coefficients for ***m*** and ***b*** by calling and flattening the **`coef_`** and **`intercept_`** properties.

```python
import pandas as pd
from sklearn.linear_model import LinearRegression

# Import points

df = pd.read_csv('https://bit.ly/3cIH97A', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Plain ordinary least squares
fit = LinearRegression().fit(X, Y)

# Print "m" and "b" coefficients
print("m = {0}".format(fit.coef_.flatten()))
print("b = {0}".format(fit.intercept_.flatten()))

# Predict a new "y" value for x = 3.5
print("x = 3.5, y = {0}".format(fit.predict([[3.5]]).flatten()))
```

Charting Using matplotlib

Matplotlib can be used to easily show a line with a scatterplot, effectively to display a linear regression.



```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression

# Import points
df = pd.read_csv('https://bit.ly/3goOAnt', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Fit a line to the points
fit = LinearRegression().fit(X, Y)

# m = 1.7867224, b = -16.51923513
m = fit.coef_.flatten()
b = fit.intercept_.flatten()
print("m = {0}".format(m))
print("b = {0}".format(b))

# show in chart
plt.plot(X, Y, 'o') # scatterplot
plt.plot(X, m*X+b) # line
plt.show()
```

Simple Linear Regression – Closed Form Equation

For a simple linear regression with only one input variable and one output variable, there is a simple formula to solve for ***m*** and ***b***.

$$m = \frac{n\sum xy - \sum x \sum y}{n \sum x^2 - (\sum x)^2}$$

$$b = \frac{\sum y}{n} - m\frac{\sum x}{n}$$

```
import pandas as pd

# Load the data
points = list(pd.read_csv('https://bit.ly/2KF29Bd', delimiter=",").itertuples())

n = len(points)

m = (n*sum(p.x*p.y for p in points) - sum(p.x for p in points) *
    sum(p.y for p in points)) / (n*sum(p.x**2 for p in points) -
    sum(p.x for p in points)**2)

b = (sum(p.y for p in points) / n) - m * sum(p.x for p in points) / n

print(m, b)
# 1.9393939393939394 4.7333333333333325
```

Inverse Matrix Technique

You can also use inverse matrices to perform a linear regression.

To get the coefficients as a vector *b* with slope(s) and intercept:

$$b = (X^T \cdot X)^{-1} \cdot X^T \cdot y$$

Note you will have to add a column of 1's to the input variable vector ***X***, so the intercept can be generated.

```python
import matplotlib.pyplot as plt
import pandas as pd
from numpy.linalg import inv
import numpy as np

# Import points
df = pd.read_csv('https://bit.ly/3goOAnt', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1].flatten()

# Add placeholder "1" column to generate intercept
X_1 = np.vstack([X, np.ones(len(X))]).T

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Calculate coefficents for slope and intercept
b = inv(X_1.transpose() @ X_1) @ (X_1.transpose() @ Y)
print(b) # [1.93939394 4.73333333]

# Predict against the y-values
y_predict = X_1.dot(b)
```

QR Decomposition Technique

A more computationally stable method is to use QR Decomposition.

Use NumPy's `qr()` function to break up a matrix into ***Q*** and ***R*** components.

Plug it into this formula to get the vector ***b*** containing the slope(s) and intercept.

$$X = Q \cdot R$$

$$b = (X^T \cdot X)^{-1} \cdot X^T \cdot y$$

```python
import matplotlib.pyplot as plt
import pandas as pd
from numpy.linalg import qr, inv
import numpy as np

# Import points
df = pd.read_csv('https://bit.ly/3goOAnt', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1].flatten()

# Add placeholder "1" column to generate intercept
X_1 = np.vstack([X, np.ones(len(X))]).transpose()

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# calculate coeffecents for slope and intercept
# using QR decomposition
Q, R = qr(X_1)
b = inv(R).dot(Q.transpose()).dot(Y)

print(b) # [1.93939394 4.73333333]
```

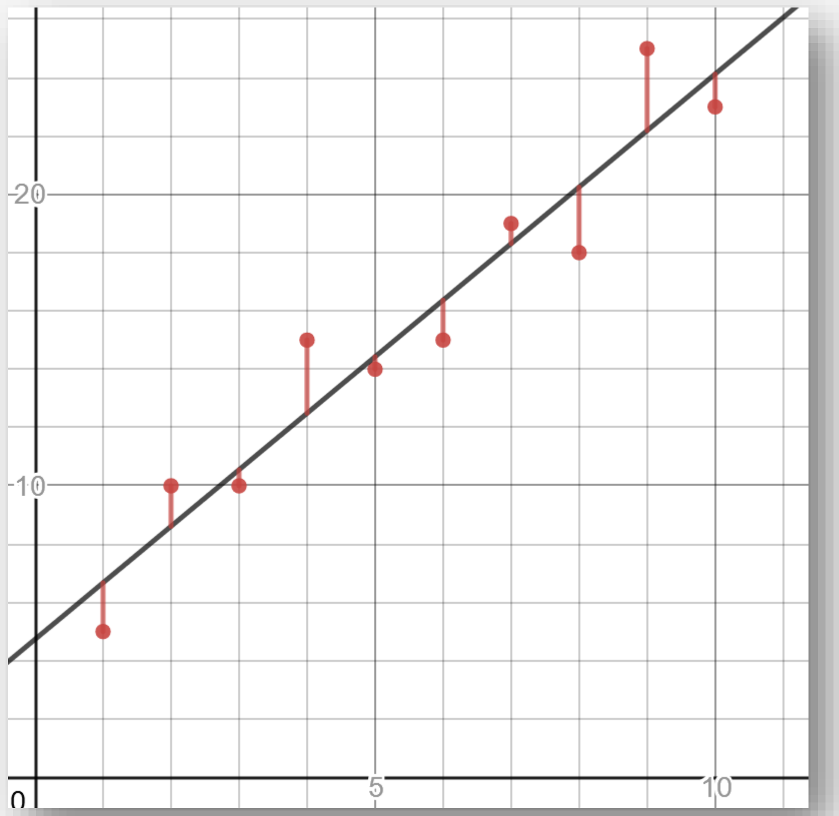
When Do We Use Linear Regression?

Data should somewhat have the following qualities for a linear regression to be appropriate:

1) Input variables are continuous, not binary or class (use logistic regression if latter)

2) Input variables follow a Gaussian (bell curve) distribution

3) Input variables are relevant to the output variables, and not highly correlated with each other (known as **collinearity**).



Multiple Linear Regression

Multiple Linear Regression

Solving for a function $\boldsymbol{y} = \boldsymbol{mx} + \boldsymbol{b}$ is elementary as it only has one independent variable $\boldsymbol{x}$.

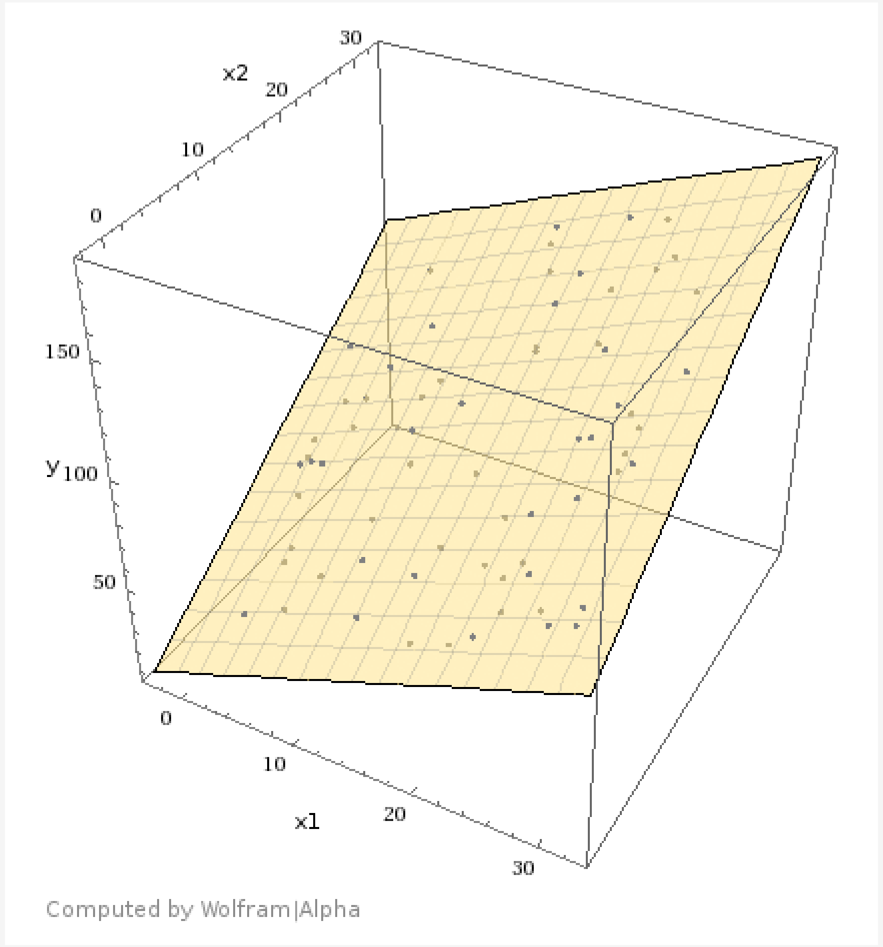
We can also solve for multiple independent variables, like x_1, x_2, x_3, and so on…

Let's say we have columns of independent variables x_1 and x_2, and a dependent variable y.

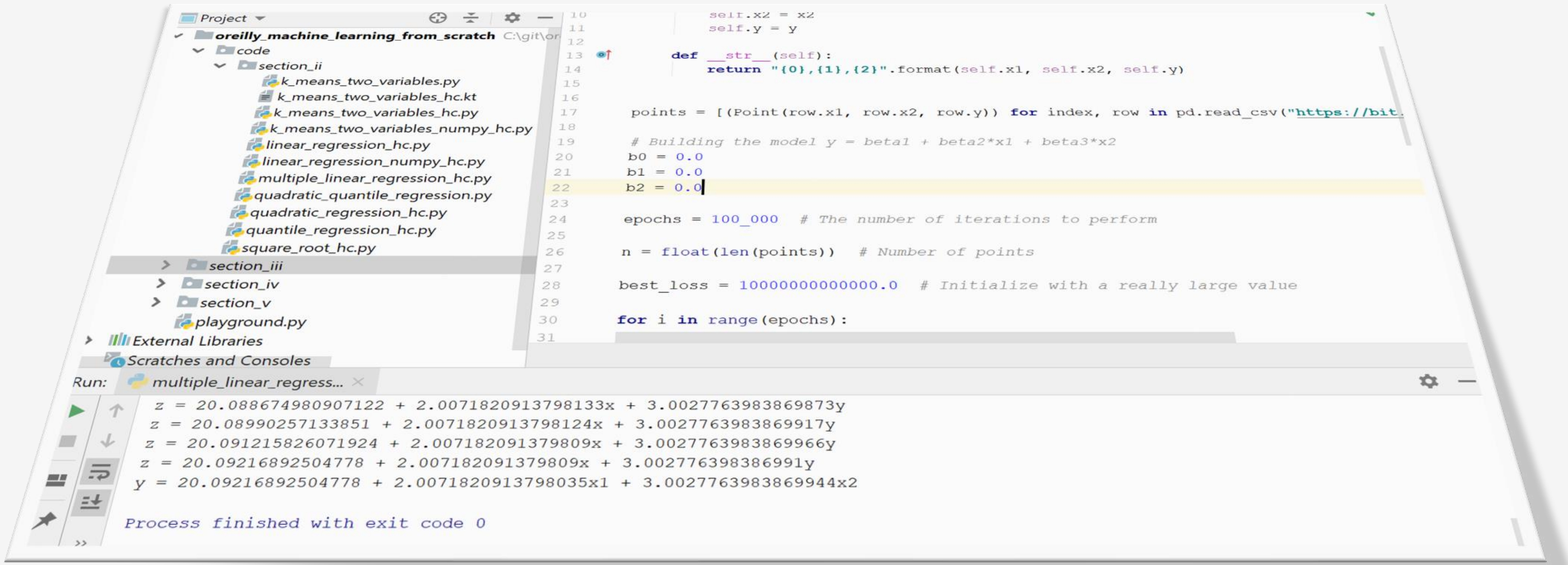
$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

Instead of $\boldsymbol{m}$ and $\boldsymbol{b}$ constants, we now need to solve for each β parameter, where β_0 is the y-intercept and β_1 and β_2 are slopes for the respective x variables.

We can use the same hill-climbing technique as before!



Hands-On: Multiple Linear Regression



Project
oreilly_machine_learning_from_scratch C:\git\or
code
section_ii
k_means_two_variables.py
k_means_two_variables_hc.kt
k_means_two_variables_hc.py
k_means_two_variables_numpy_hc.py
linear_regression_hc.py
linear_regression_numpy_hc.py
multiple_linear_regression_hc.py
quadratic_quantile_regression.py
quadratic_regression_hc.py
quantile_regression_hc.py
square_root_hc.py
section_iii
section_iv
section_v
playground.py
External Libraries
Scratches and Consoles
Run:
multiple_linear_regress...

```
        self.x2 = x2
        self.y = y

    def __str__(self):
        return "{0},{1},{2}".format(self.x1, self.x2, self.y)


points = [(Point(row.x1, row.x2, row.y)) for index, row in pd.read_csv("https://bit.

# Building the model y = beta1 + beta2*x1 + beta3*x2
b0 = 0.0
b1 = 0.0
b2 = 0.0

epochs = 100_000  # The number of iterations to perform

n = float(len(points))  # Number of points

best_loss = 10000000000000.0  # Initialize with a really large value

for i in range(epochs):
```

```
z = 20.088674980907122 + 2.0071820913798133x + 3.0027763983869873y
z = 20.08990257133851 + 2.0071820913798124x + 3.002776398386991 7y
z = 20.091215826071924 + 2.007182091379809x + 3.0027763983869966y
z = 20.09216892504778 + 2.007182091379809x + 3.002776398386991y
y = 20.09216892504778 + 2.0071820913798035x1 + 3.0027763983869944x2

Process finished with exit code 0
```

Multiple Linear Regression: Scikit-Learn

Just like a simple linear regression, Scikit-Learn supports multiple linear regression.

Fit the independent variables as the first argument to the **`fit()`** function, and the dependent variable to the second argument.

```
import pandas as pd
from sklearn.linear_model import LinearRegression

# Load the data
df = pd.read_csv('https://bit.ly/2X1HWH7', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)\
Y = df.values[:, -1]

# Plain ordinary least squares
fit = LinearRegression().fit(X, Y)

# Print "m" and "b" coefficients
print("Coefficients = {0}".format(fit.coef_))
print("Intercept = {0}".format(fit.intercept_))
print("z = {0} + {1}x + {2}y".format(fit.intercept_, fit.coef_[0], fit.coef_[1]))

# Predict a new “z" value for x = 3.5
print("x = 3.5, y = 4.5, z = {0}".format(fit.predict([[3.5, 4.5]])))
```

Quiz Time!

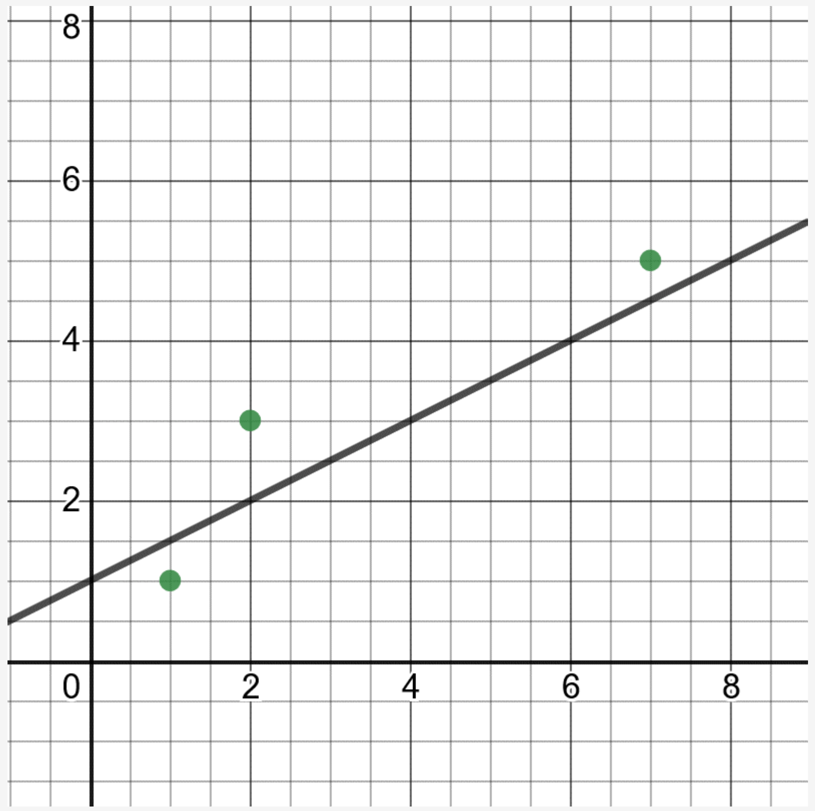
You have three points (1,1), (2,3), and (7,5). You are testing a fit for a line $y = \frac{1}{2}x + 1$. What is the loss calculated by sum of squares?

A) 1.5

B) 1.75

C) 1.0

D) -1.5



Quiz Time!

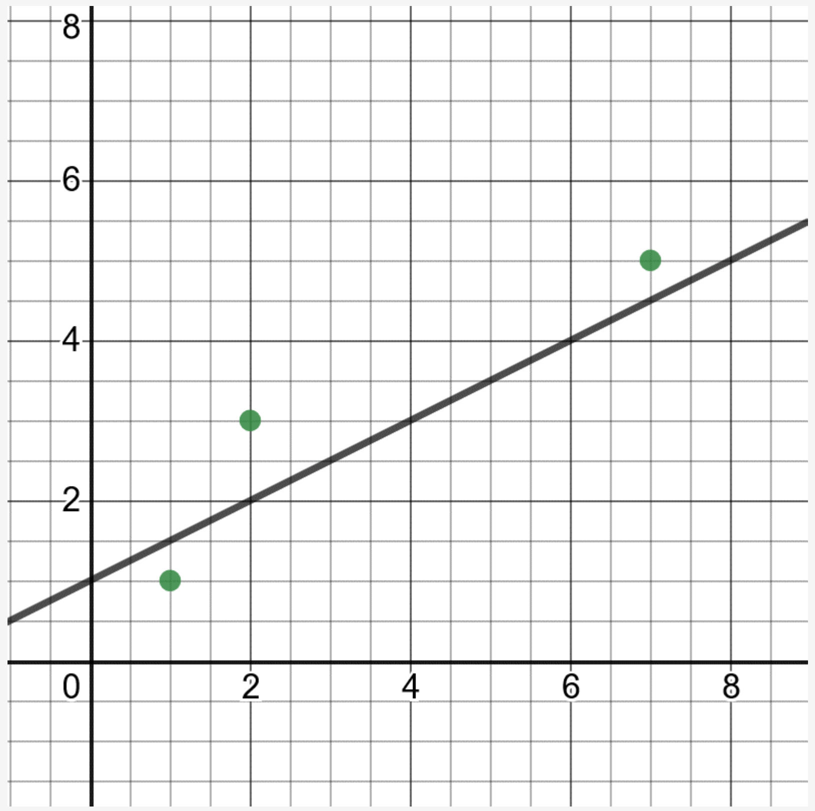
You have three points (1,1), (2,3), and (7,5). You are testing a fit for a line $y = \frac{1}{2}x + 1$. What is the loss calculated by sum of squares?

[✓ A) 1.5] $= (1 - (\frac{1}{2}1 + 1))^2 + (3 - (\frac{1}{2}2 + 1))^2 + 5 - (\frac{1}{2}7 + 1))^2$

B) 1.75

C) 1.0

D) -1.5



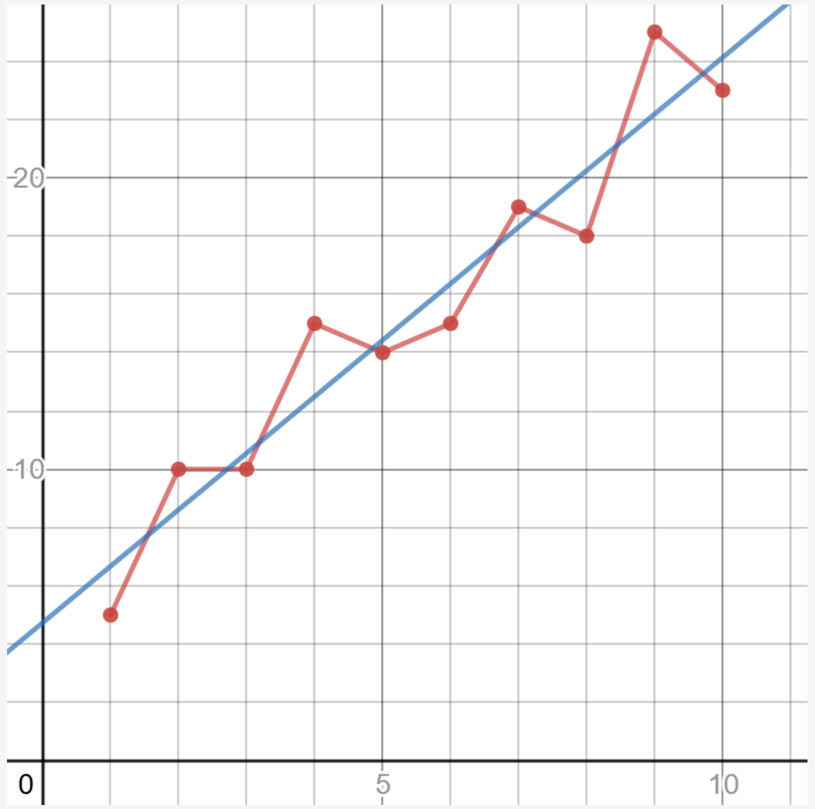
Separating Training/Testing Data

Overfitting

Overfitting means that our ML model works well with the data it was trained on but fails to predict correctly with new data.

- This can be due to many factors, but a common cause is the sampled data does not represent the larger population and more data is needed.
- The red line has high **variance**, meaning its predictions are sensitive to outliers and therefore can vary greatly.
- The blue line has high **bias**, meaning the model is less sensitive to outliers because it prioritizes a method (maintaining a straight line) rather than bend and respond to variance.

The red line to the right is likely overfitted (high variance, low bias), but the blue linear regression line (low variance, high bias) is less likely to be overfit.



https://www.desmos.com/calculator/wmwfolbvdk

Overfitting

Linear regression is a highly biased method and is resilient to overfitting.

There are other remedies to mitigate overfitting, the most basic being separating **training data** and **test data**.

- The model is fit to the training data, and then is tested with the test data.
- If the test data performs poorly compared to the training data, there is a possibility of overfit (or just no correlation altogether).

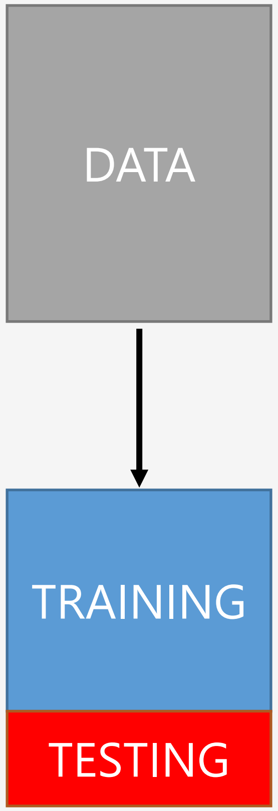
You can also try to train with more data as well as utilize cross-validation, regularization, bagging, boosting, and other techniques.

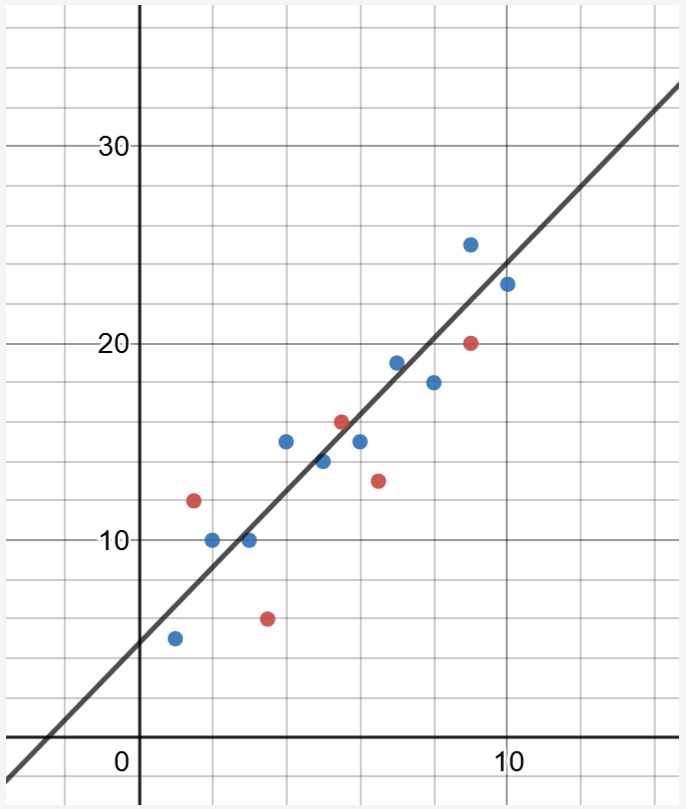
Training and Testing Data

A common practice to proactively prevent overfitting in machine learning is to separate training data and testing data.

- **Training data** is data used to fit a model and is typically 2/3 of the data.
- **Test data** is used to test the model and is the remaining 1/3 of the data.

By omitting the testing data from training, we see how well the model works on data it has not seen before and change our parameters accordingly.





Train/Test Split

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Load the data
df = pd.read_csv('https://bit.ly/3cIH97A', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)

Y = df.values[:, -1]

# Separate training and testing data to evaluate performance and reduce overfitting
# This leaves a third of the data out for testing
# Set a random seed just to make the randomly selected split consistent
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=.33, random_state=10)

model = LinearRegression()
model.fit(X_train, Y_train)
result = model.score(X_test, Y_test)
print("Accuracy Mean: %.3f" % result)
```

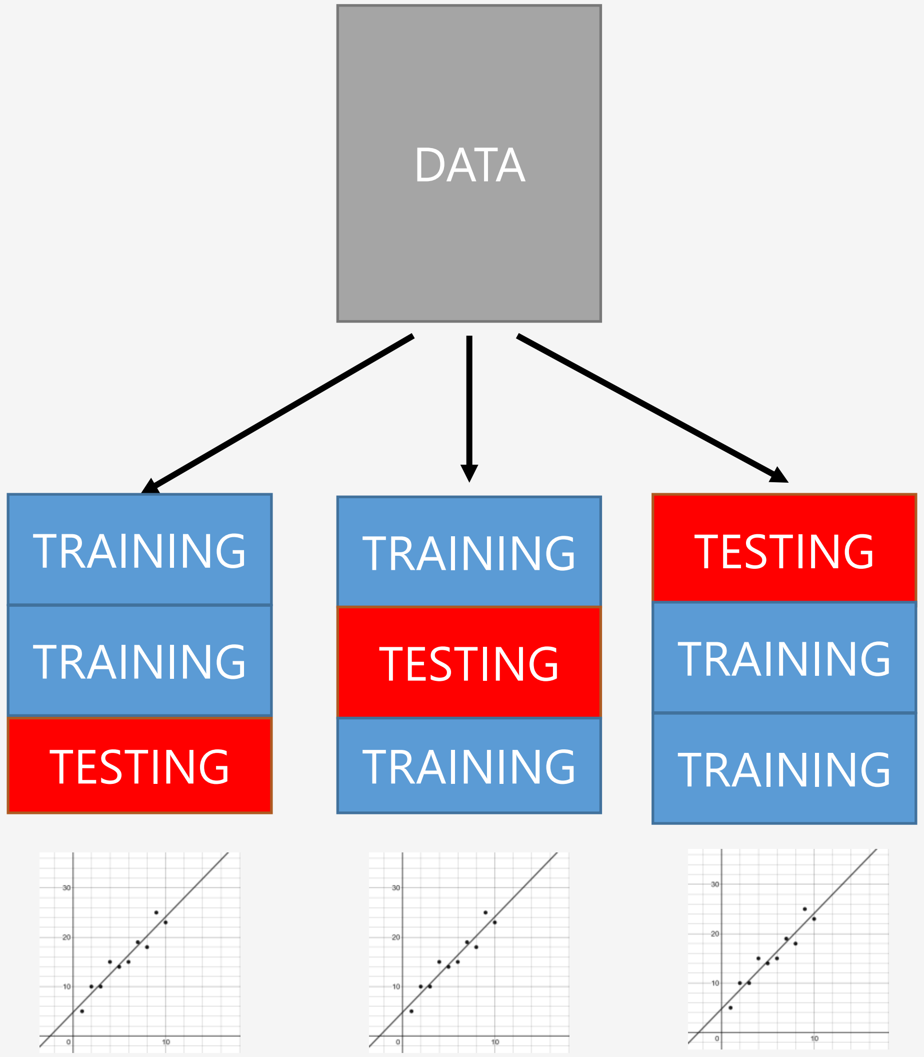
Cross Validation

We can take this concept of training/testing data a step further, and test different combinations of training and testing data.

This is known as **cross-validation**, the gold standard of validation techniques.

To the right we have **3-fold cross validation** which breaks the data into thirds and uses one of the pieces for testing.

We can then evaluate how well each of these perform, being able to compare different parameters and models (e.g. linear regression vs decision trees) and see which setup produces the best performance.



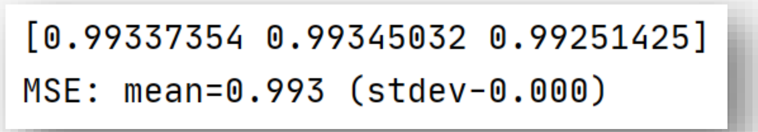
Cross Validation

To the right we see a typical **K-fold validation**, where k=3 resulting in 2/3 of the data being used for training and 1/3 being used for testing.

We can summarize the performance by taking the mean and standard deviation of each set on the performance metric we choose (which by default is R-square).

We will talk about performance metrics shortly.

```python
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold, cross_val_score

df = pd.read_csv('https://bit.ly/3clH97A', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)\
Y = df.values[:, -1]

# Perform a simple linear regression
kfold = KFold(n_splits=3, random_state=7, shuffle=True)
model = LinearRegression()
results = cross_val_score(model, X, Y, cv=kfold)
print(results)
print("MSE: mean=%.3f (stdev-%.3f)" % (results.mean(), results.std()))
```



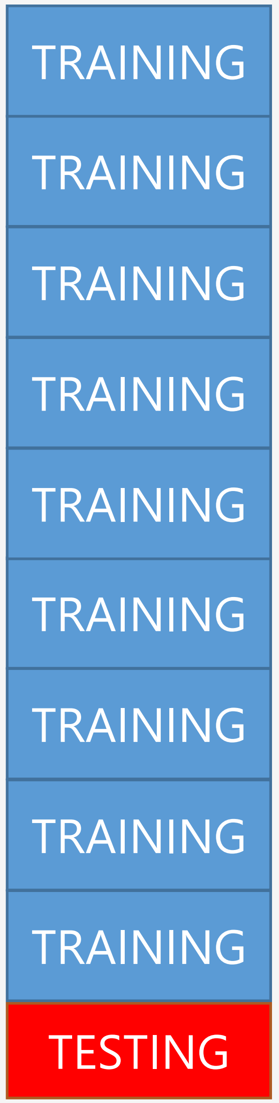
```
[0.99337354 0.99345032 0.99251425]
MSE: mean=0.993 (stdev-0.000)
```

Cross Validation

Note that **k-fold cross validation** allows us to slice our data into any number and not just 3 (typically 3, 5, or 10).

For example we can do 10-fold cross validation and validate 10 different combinations of training/test data.

The most extreme form of folding is **leave-one-out cross validation**, which omits one data record for testing and uses the remaining records for training, and this is done repeatedly.



Leave-one-out Cross-Validation in Scikit-Learn

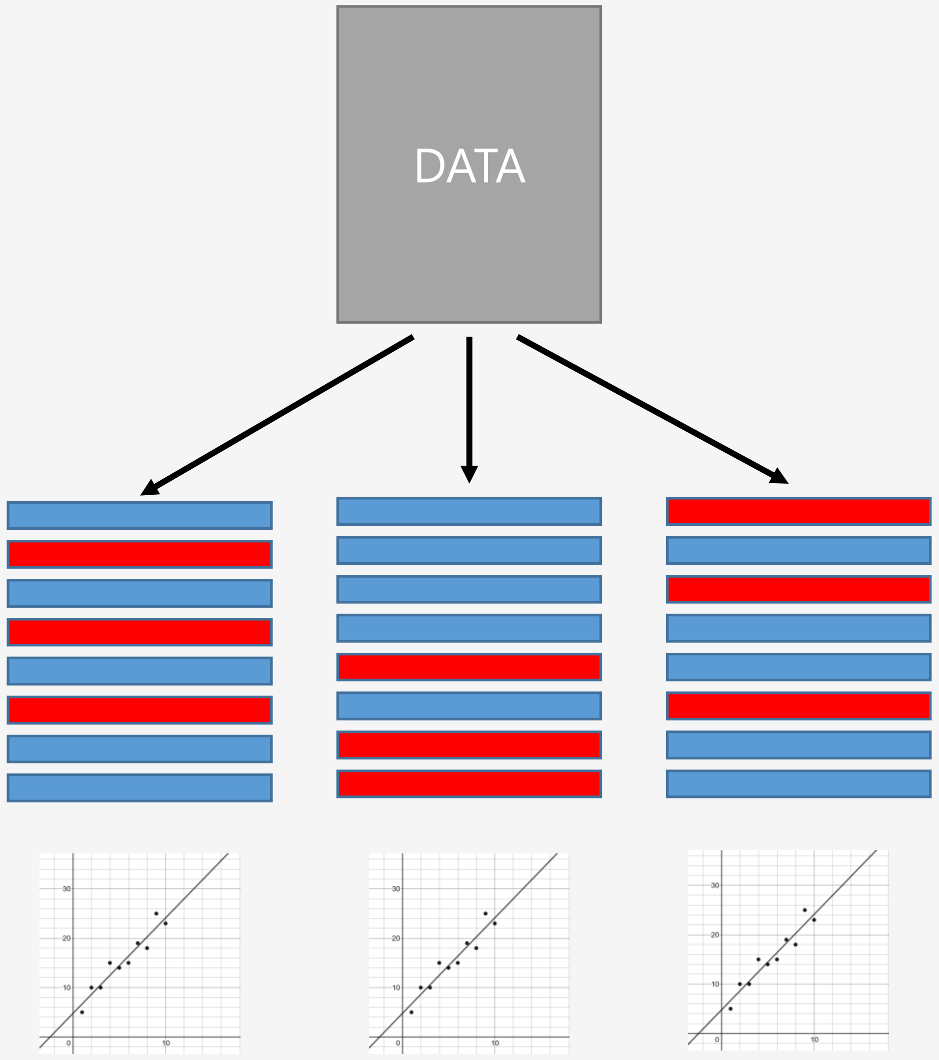
```python
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold, cross_val_score, LeaveOneOut

df = pd.read_csv('https://bit.ly/3cIH97A', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Perform a simple linear regression
kfold = KFold(n_splits=3, random_state=7, shuffle=True)
model = LinearRegression()

results = cross_val_score(model, X, Y, cv=LeaveOneOut())

print(results)
print("mean= %.3f (stdev= %.3f)" % (results.mean(), results.std()))
```

Random Fold Validation

As you may be noticing, machine learning often tries to overcome data variance with randomness.

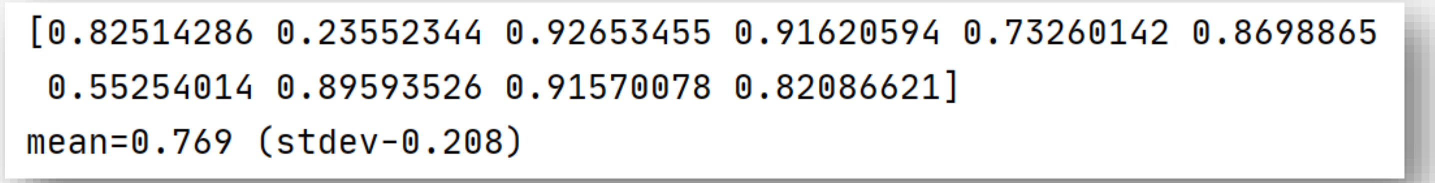
A variant with fold validation is **repeated random fold validation**, where we randomly shuffle the data and create random train/test folds as many times as desired.

This is helpful when we need to mitigate variance in the model.



Random Fold Validation

```python
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score, ShuffleSplit

df = pd.read_csv('https://bit.ly/38XwbeB', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)\
Y = df.values[:, -1]

# Perform a simple linear regression
kfold = ShuffleSplit(n_splits=10, test_size=.33, random_state=7)
model = LinearRegression()
results = cross_val_score(model, X, Y, cv=kfold)

print(results)
print("mean=%.3f (stdev-%.3f)" % (results.mean(), results.std()))
```



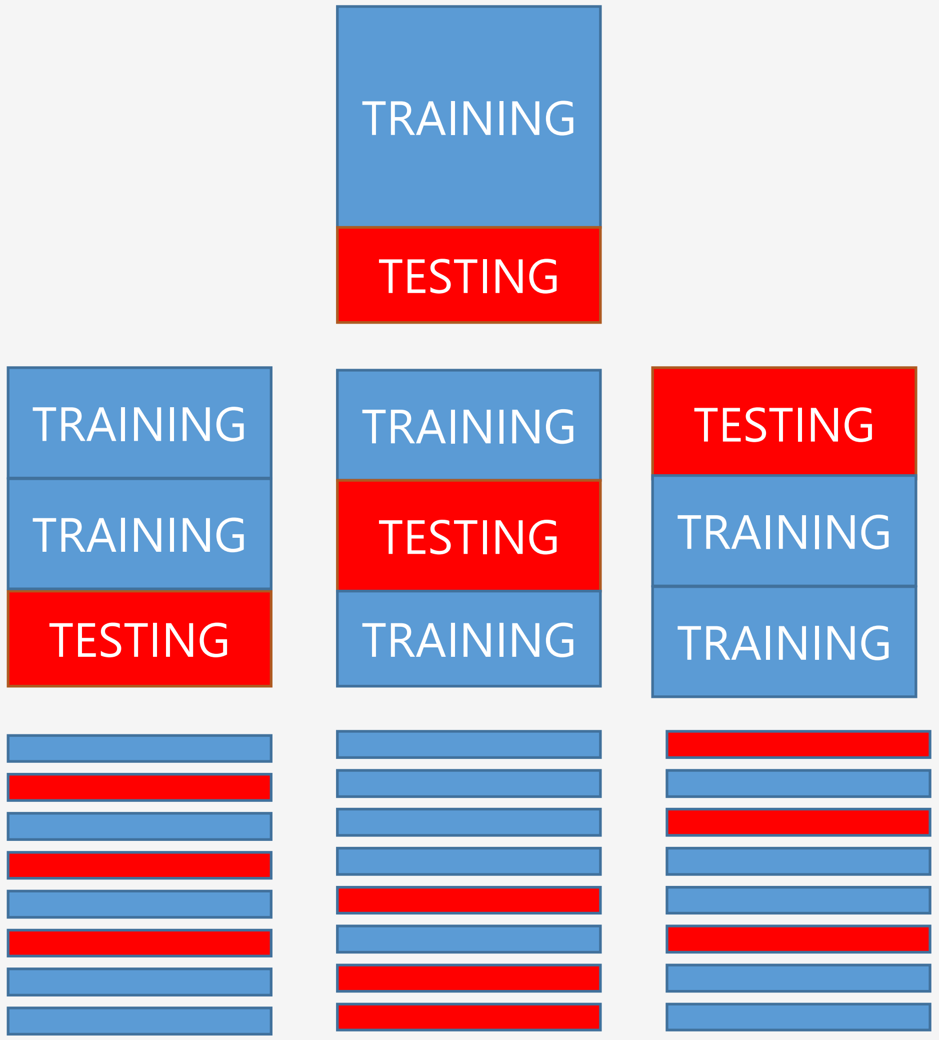
```
[0.82514286 0.23552344 0.92653455 0.91620594 0.73260142 0.8698865
 0.55254014 0.89593526 0.91570078 0.82086621]
mean=0.769 (stdev-0.208)
```

Which Validation to Use?

Generally, you will want to prefer k-fold validation as it is the gold standard.

A single train/test split might be warranted if performance of machine learning algorithm is slow and has enough data with lower bias.

Use the random fold split to mitigate variance in the model while balancing training speed and dataset sizes.

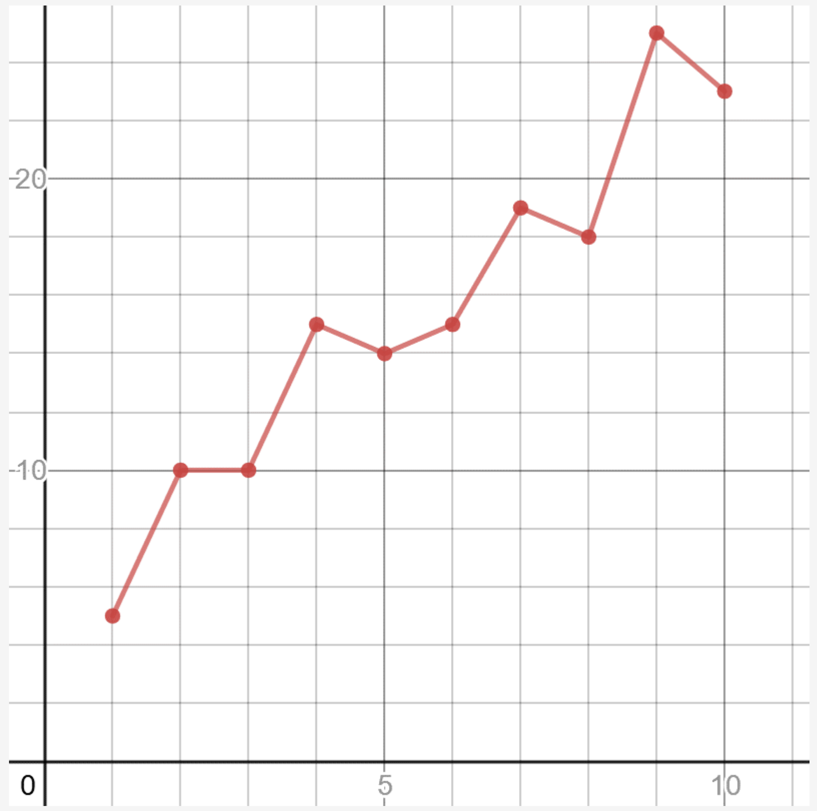


Quiz Time!

It is good to fit a regression as close to the points as possible, even if you just connect the points as shown to the right.

A) True

B) False



Quiz Time!

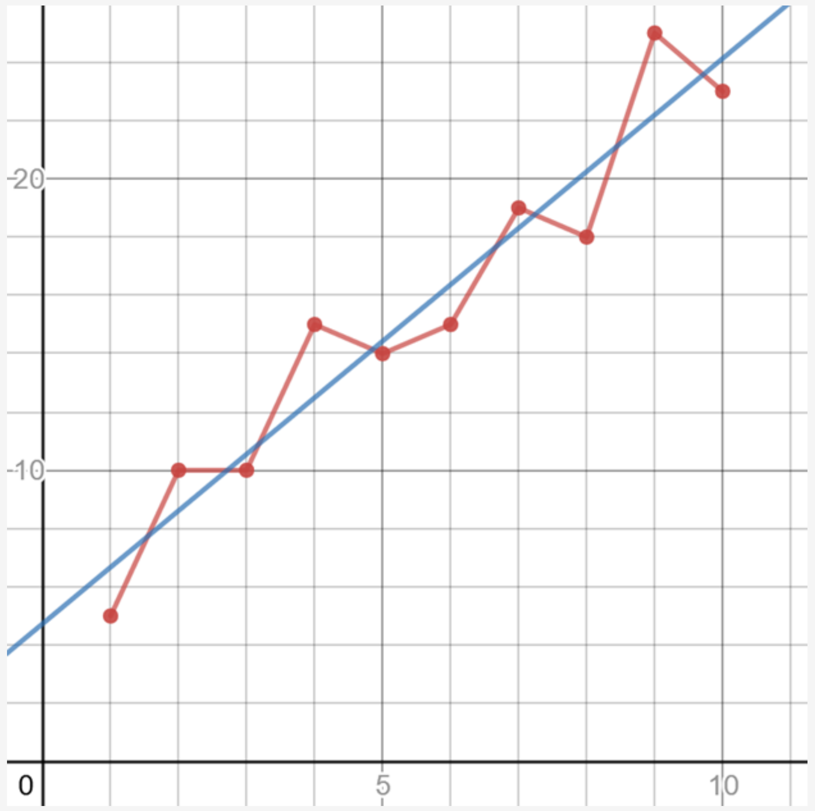
It is good to fit a regression as close to the points as possible, even if you just connect the points as shown to the right.

A) True

[✓ B) False]

Doing a regression is a balancing act between fitting exactly to your data (variance) versus leaving some wiggle room (bias) to predict new data that will be different. Otherwise you will risk overfitting even if you have zero loss.

You might be better doing a linear regression (blue line) as shown to the right, which has a loss but is more likely to predict new data accurately.

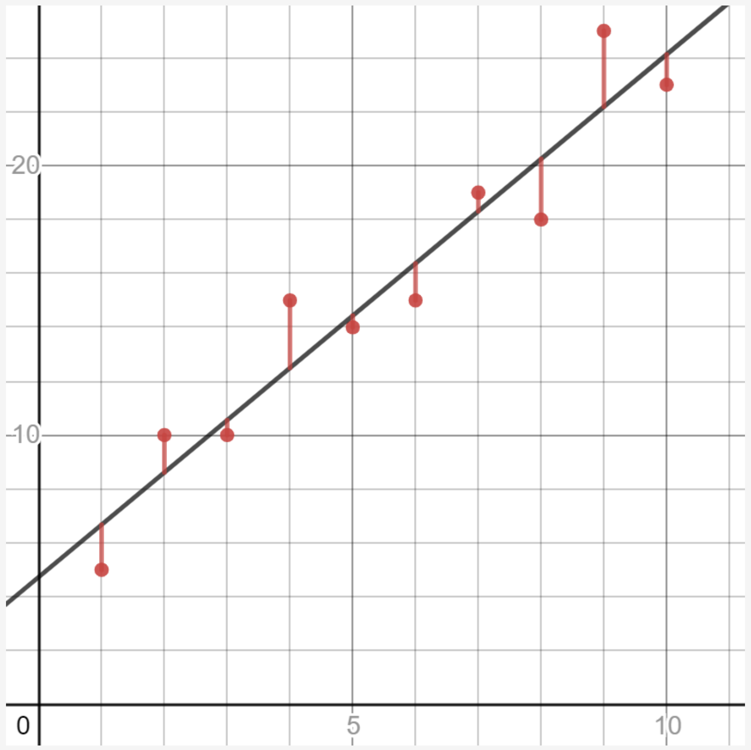


Quiz Time!

If we have a dataset and want to create a linear regression off it, we want to train it with the entire dataset

A) True

B) False



Quiz Time!

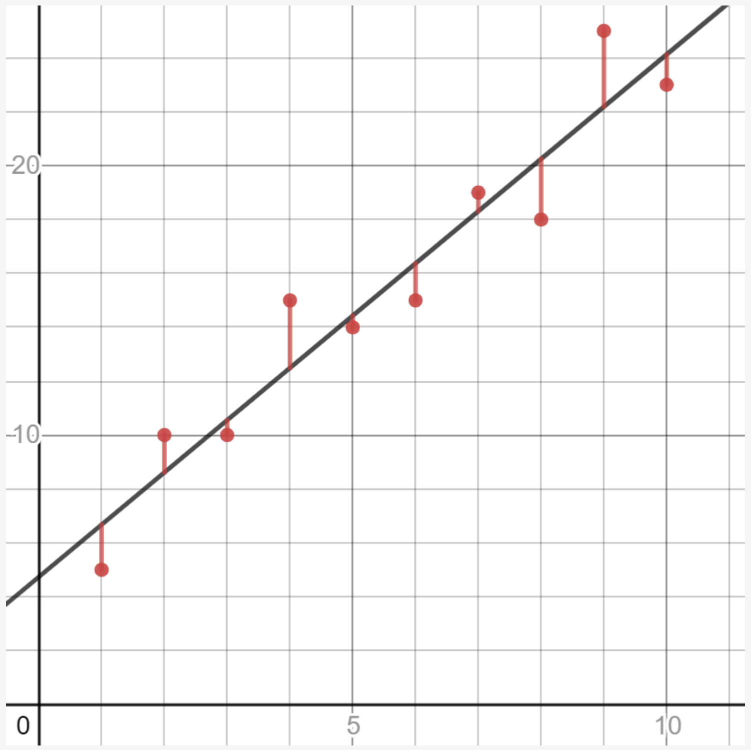
If we have a dataset and want to create a linear regression off it, we want to train it with the entire dataset

A) True

[✓ B) False]

We want to separate our training and testing data from a dataset, typically using a 3-fold where 2/3 is used for training and 1/3 is used for testing.

As we learned we can do more complicated fold strategies too.

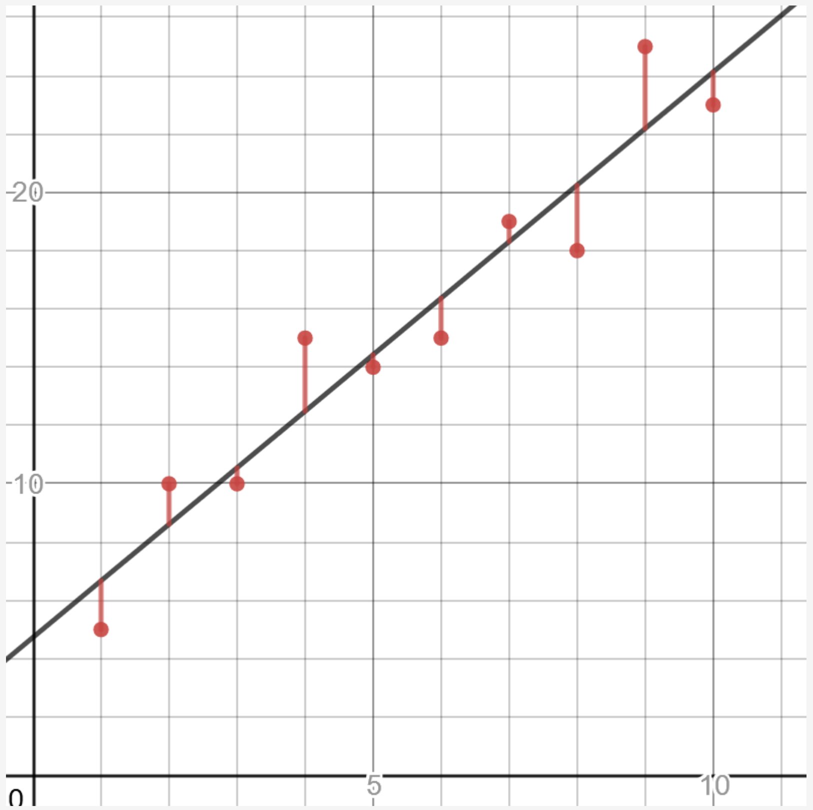


Performance Metrics

Evaluating Performance

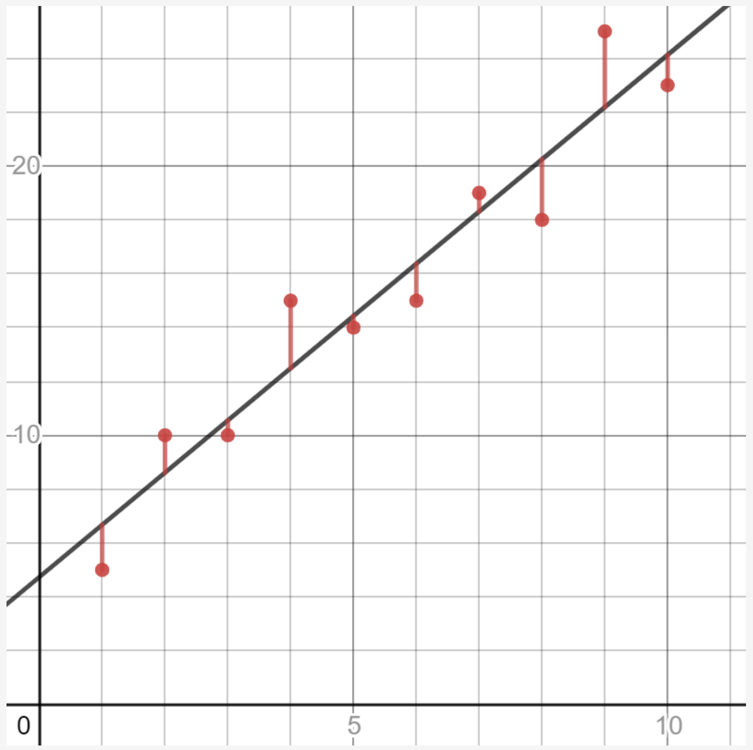
We learned how to separate training and testing data in a myriad of ways, but there are several ways we can measure the testing dataset.

Typically across the test dataset, or through multiple test datasets achieved through k-fold or random folds, we will average the performance metrics.



Mean Absolute Error

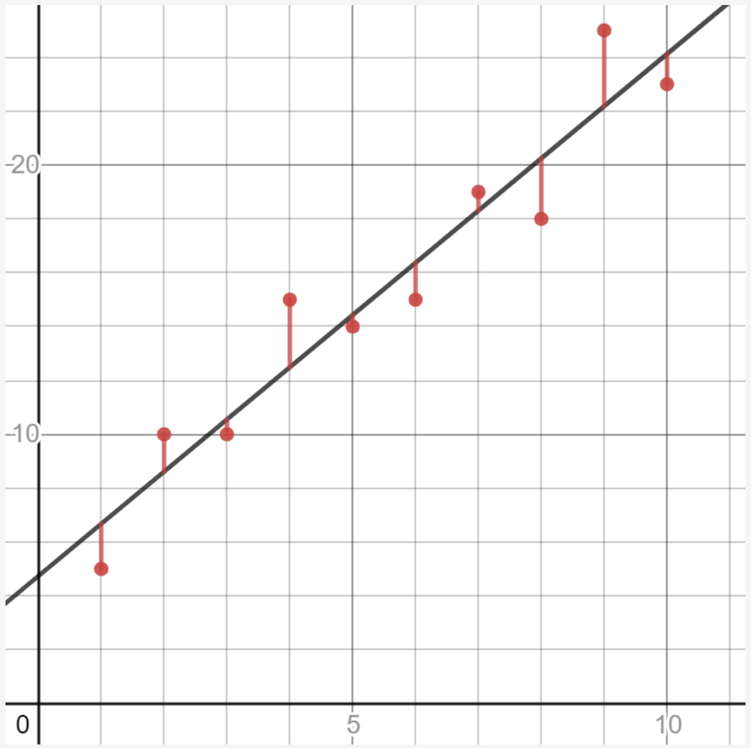
Mean absolute error (MAE) simply measures the sum of absolute values of residuals (difference between predicted values and actual values).



```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold, cross_val_score, validation_curve

df = pd.read_csv('https://bit.ly/3cIH97A', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)\
Y = df.values[:, -1]

# Perform a simple linear regression
kfold = KFold(n_splits=3, random_state=7, shuffle=True)
model = LinearRegression()

# Use negative mean absolute error
# It is negative because it is inverted
results = cross_val_score(model, X, Y, cv=kfold, scoring='neg_mean_absolute_error')

print("MAE: mean=%.3f (stdev-%.3f)" % (results.mean(), results.std()))
# prints MAE: mean=-2.446 (stdev-0.791)
```

Mean Squared Error

Mean squared error (MSE) simply measures the average squared values of residuals (difference between predicted values and actual values).

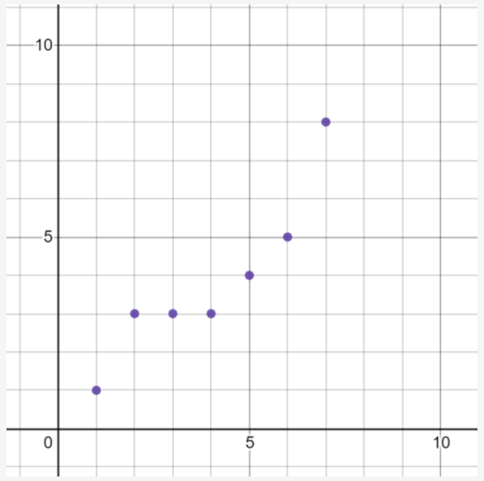


```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold, cross_val_score, validation_curve

df = pd.read_csv('https://bit.ly/3nJBdj9', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Perform a simple linear regression
kfold = KFold(n_splits=3, random_state=7, shuffle=True)
model = LinearRegression()

# Use negative mean squared error
# It is negative because it is inverted
results = cross_val_score(model, X, Y, cv=kfold, scoring='neg_mean_squared_error')

print("MAE: mean=%.3f (stdev-%.3f)" % (results.mean(), results.std()))
# prints MSE: mean=-7.664 (stdev-3.110)
```

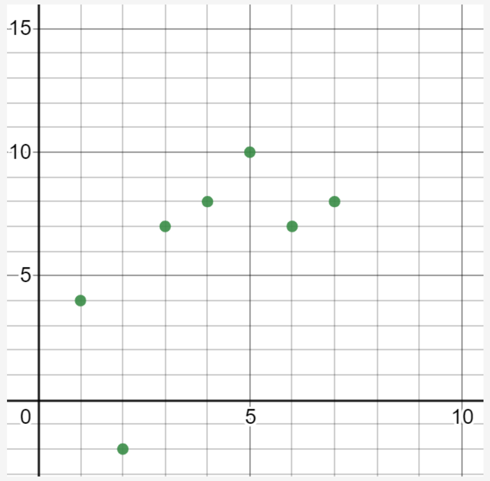
Pearson Correlation

Using a **Pearson correlation**, we can measure the strength of a correlation between two variables (from a range of -1.0 to 1.0).

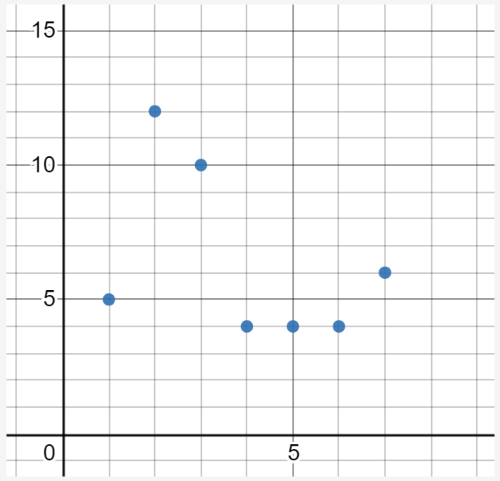
A correlation closer to 1.0 means a strong positive correlation, 0.0 means no correlation, and closer to -1.0 indicates negative correlation.



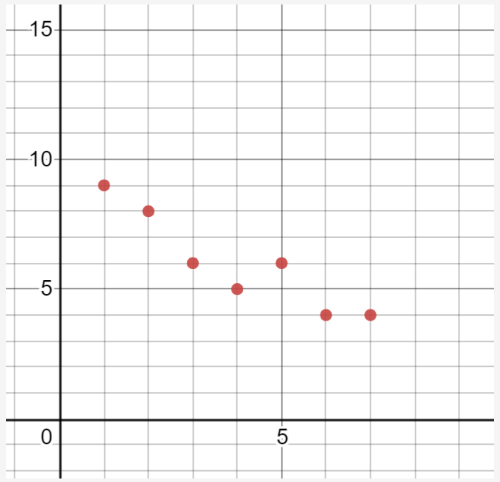
PEARSON CORRELATION: .923133



PEARSON CORRELATION: .643237



PEARSON CORRELATION: -.44984

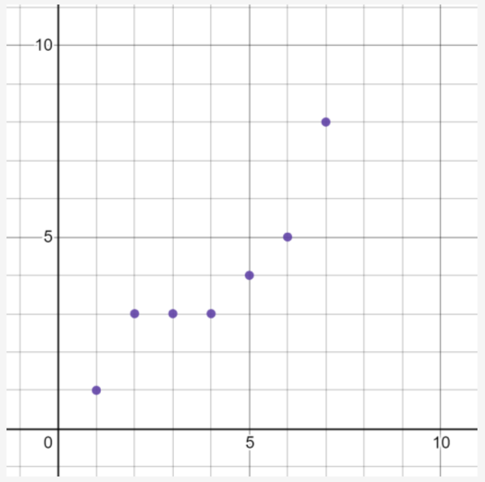


PEARSON CORRELATION: -.9267

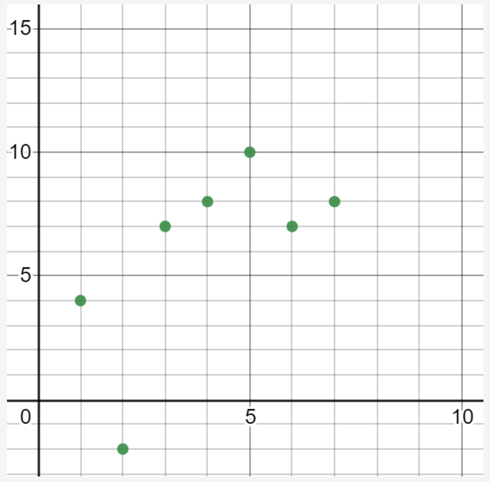
Pearson Correlation

The more data you collect, the more confident you can be in your correlation value.

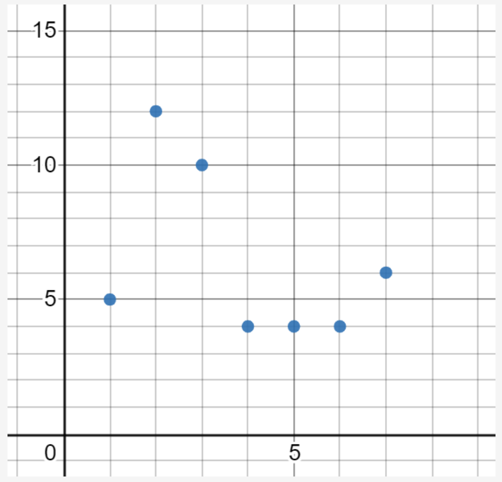
If our correlation is unlikely to have happened at random, that means we are more likely to have discovered something meaningful rather than by chance.



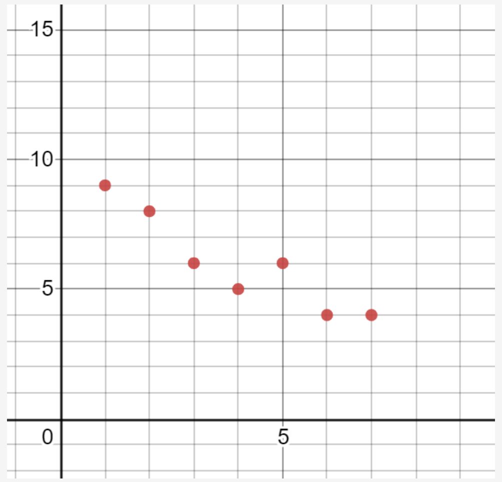
PEARSON CORRELATION: .923133



PEARSON CORRELATION: .643237

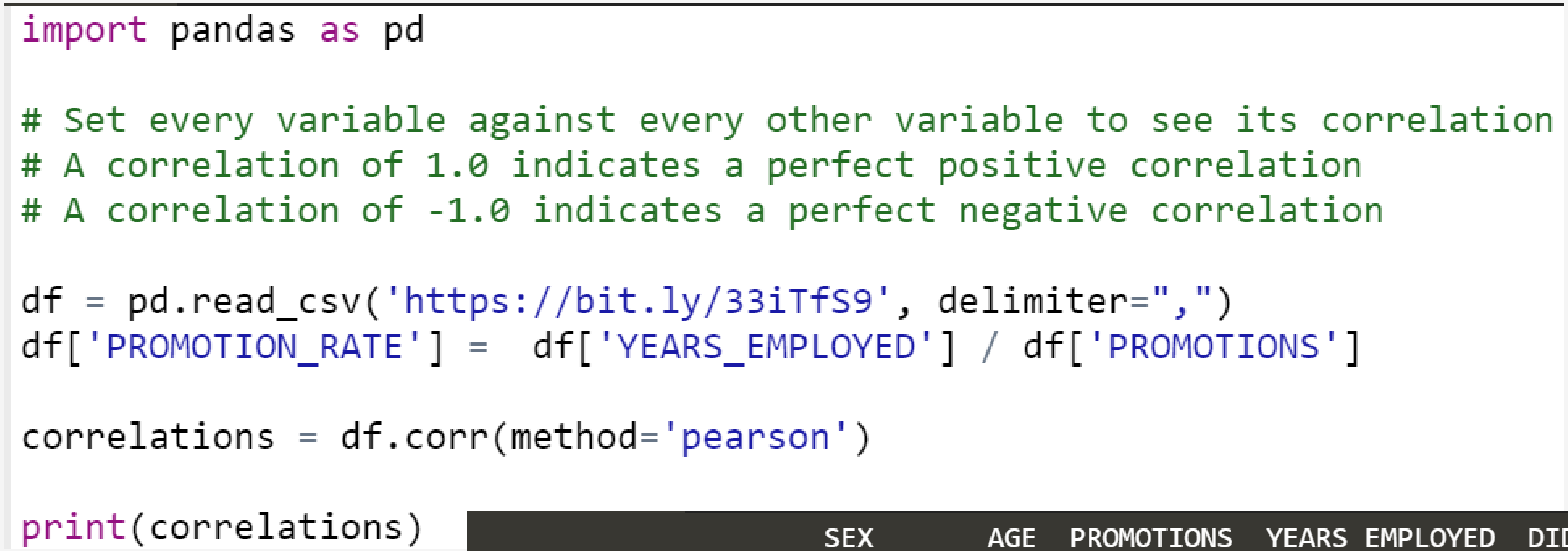


PEARSON CORRELATION: -.44984

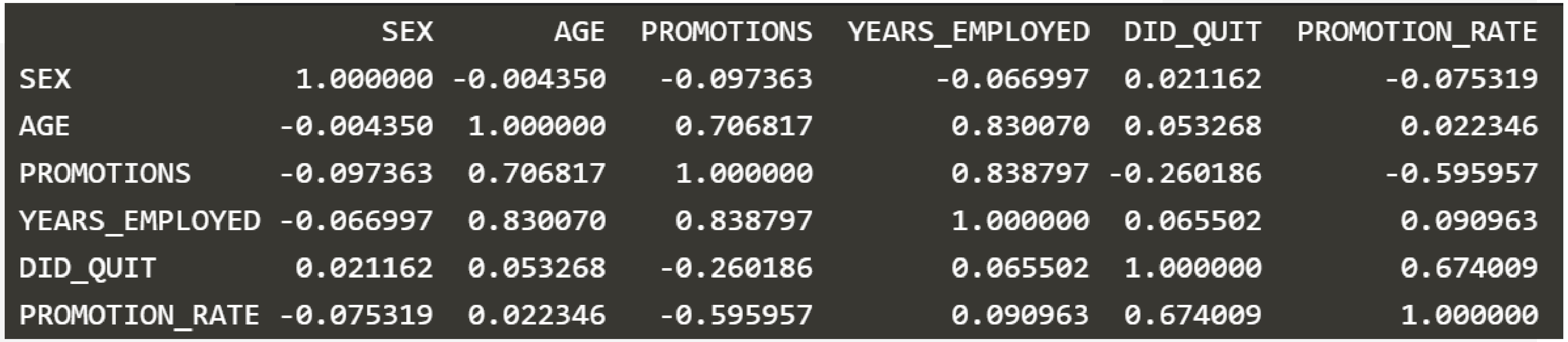


PEARSON CORRELATION: -.9267

Pearson Correlation Matrix



```python
import pandas as pd

# Set every variable against every other variable to see its correlation
# A correlation of 1.0 indicates a perfect positive correlation
# A correlation of -1.0 indicates a perfect negative correlation

df = pd.read_csv('https://bit.ly/33iTfS9', delimiter=",")
df['PROMOTION_RATE'] =  df['YEARS_EMPLOYED'] / df['PROMOTIONS']

correlations = df.corr(method='pearson')

print(correlations)
```



```
                    SEX       AGE  PROMOTIONS  YEARS_EMPLOYED  DID_QUIT  PROMOTION_RATE
SEX            1.000000 -0.004350   -0.097363       -0.066997  0.021162       -0.075319
AGE           -0.004350  1.000000    0.706817        0.830070  0.053268        0.022346
PROMOTIONS    -0.097363  0.706817    1.000000        0.838797 -0.260186       -0.595957
YEARS_EMPLOYED -0.066997  0.830070    0.838797        1.000000  0.065502        0.090963
DID_QUIT       0.021162  0.053268   -0.260186        0.065502  1.000000        0.674009
PROMOTION_RATE -0.075319  0.022346   -0.595957        0.090963  0.674009        1.000000
```

R^2

Probably the most useful performance metric is the R-square (R^2), which ratios the average y-value to the average of the residuals.

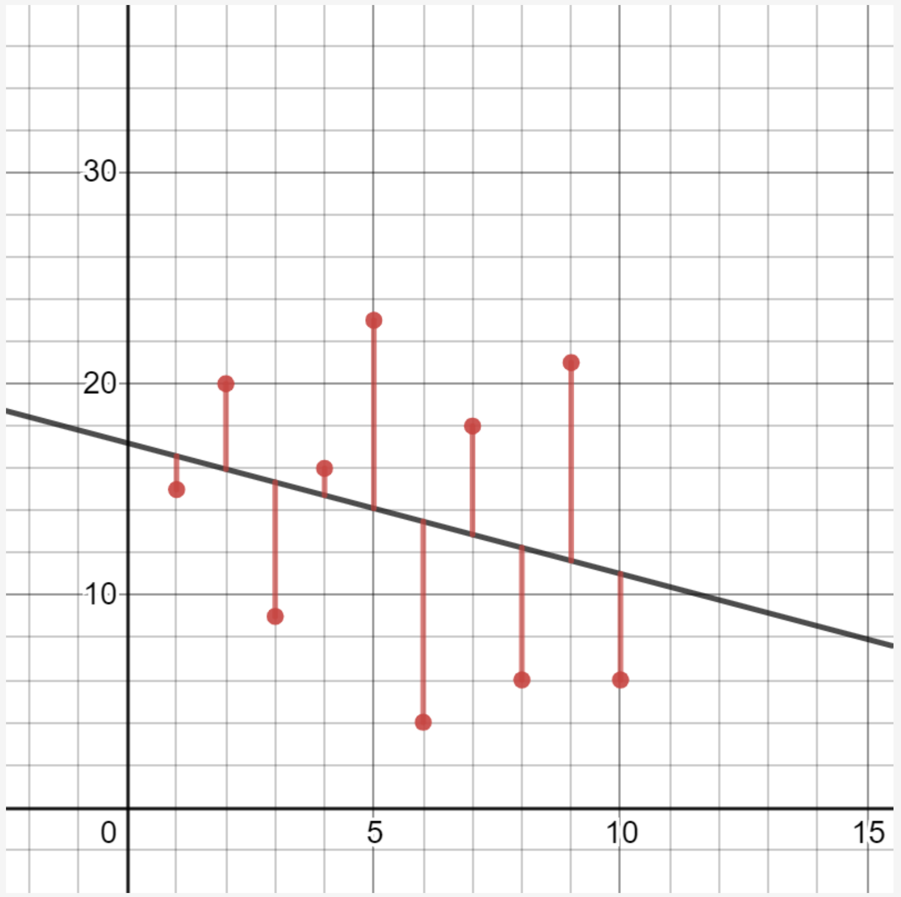
Note R^2 is also called the coefficient of determination and is the square of the Pearson correlation.

It's used to evaluate the quality of a model.

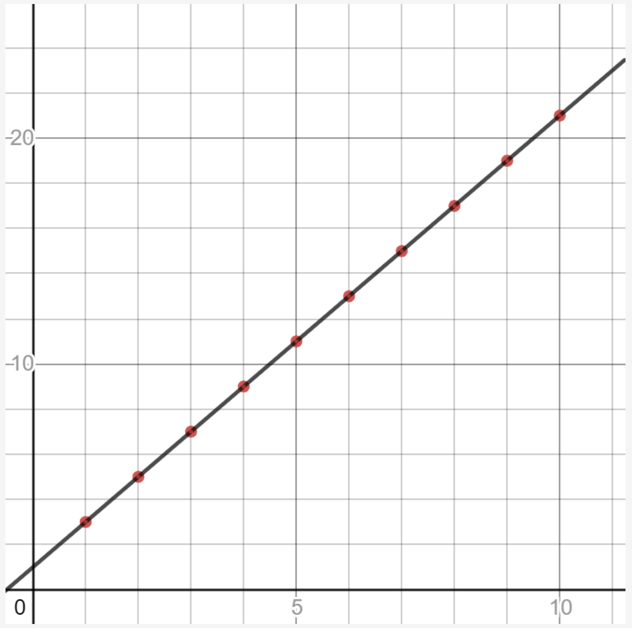
Think of R^2 as a measurement (between 0.0 and 1.0) of how well independent variables explain a dependent variable, rather than predicting with a simple average.

An R^2 of 1.0 indicates a variable perfectly explains a variable while 0.0 indicates there's no explanatory connection at all.

EXAMPLE: If ***x*** was calories consumed and ***y*** was number of pounds gained, and I got an R^2 of .85 on my test data, that means calories explains weight gain 85% better than just predicting with the average weight **y.**

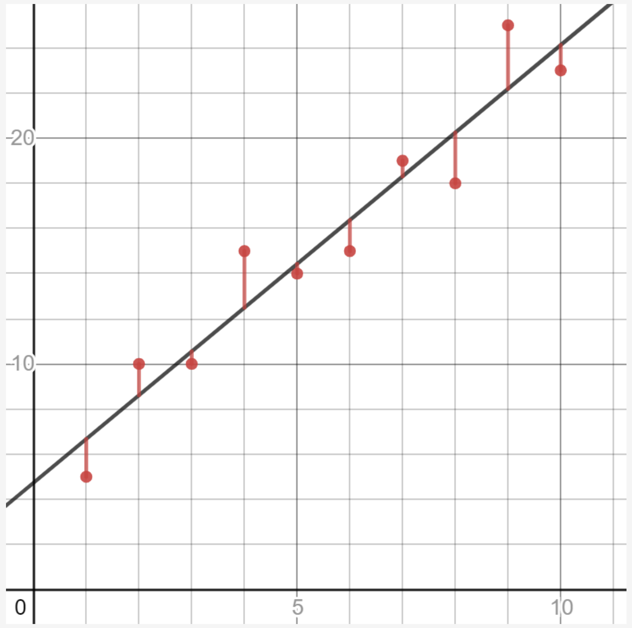


R^2 Visual Examples



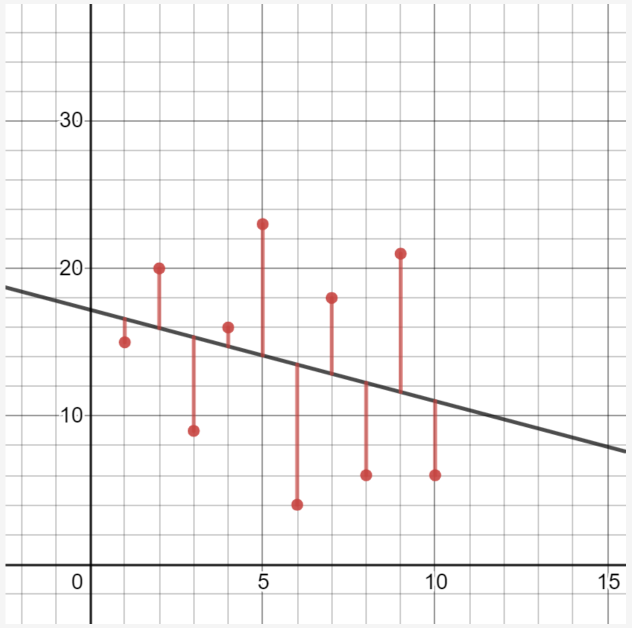
R^2 = 1.0

(PERFECT!)



R^2 = 0.917

(GOOD!)



R^2 = 0.071

(POOR!)

How to Calculate R^2 by Hand

1) Separate training and test data, perform linear regression on training data.

2) **Calculate sum of squares for regression:** using test data, subtract each predicted y value from each actual y value, square and sum

3) **Calculate sum of squares for average:** using test data, subtract the average y value from each actual y value, square and sum

4) Divide the sum of squares for regression by the sum of squares for average, subtract that from 1.0.

Now you calculated R^2 from scratch!

```python
# Calculating R-square from scratch

slope = 1.73333333
y_intercept = 5.49999999999999

x_test = [9, 3, 6, 7]
y_test = [25, 10, 15, 19]

y_test_avg = sum(y_test) / len(y_test)
# 17.25

y_test_predict = [(slope * x) + y_intercept for x in x_test]
# [21.099999969999992, 10.699999989999991, 15.89999997999999, 17.63333330999999]

sum_sq_regression = sum((y_test[i] - y_test_predict[i])**2 for i in range(0,len(y_test)))
sum_sq_total = sum((y_test[i] - y_test_avg)**2 for i in range(0,len(y_test)))

r_square = 1.0 - sum_sq_regression / sum_sq_total
# 0.8478030805337009

print(r_square)
```

Scikit-Learn and R^2

Using 3-Fold

```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold, cross_val_score,
train_test_split

df = pd.read_csv('https://bit.ly/3m20B31', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=.33)

# Perform a simple linear regression
model = LinearRegression()
model.fit(X_train, Y_train)

result = model.score(X_test, Y_test)
print(result) # prints 0.8478030825856914
```

Using Cross Validation

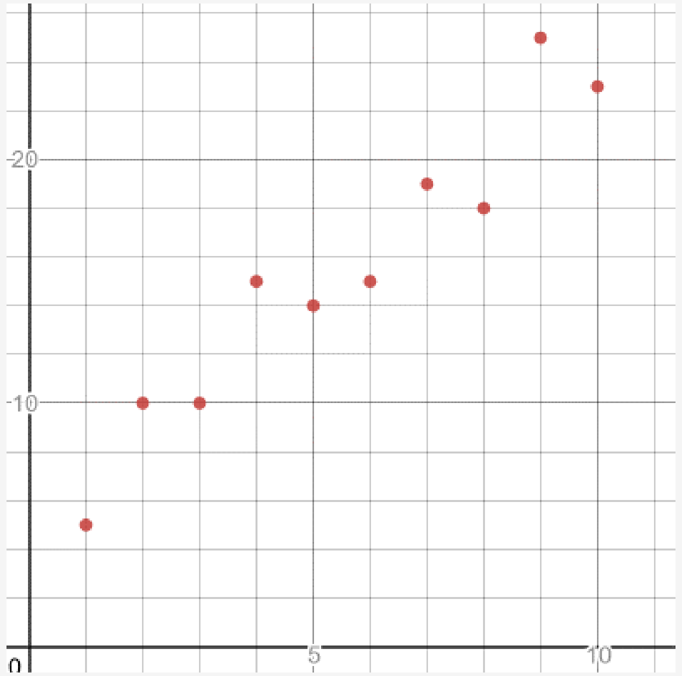
```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold, cross_val_score

df = pd.read_csv('https://bit.ly/3nJBdj9', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)\
Y = df.values[:, -1]

# Perform a simple linear regression
kfold = KFold(n_splits=3, random_state=7, shuffle=True)
model = LinearRegression()

# Use R-square to evaluate the performance
# An R-square of 0 is no-fit, and 1 is perfect fit
results = cross_val_score(model, X, Y, cv=kfold, scoring='r2')

print("R^2: mean=%.3f (stdev-%.3f)" % (results.mean(), results.std()))
# prints R^2: mean=0.590 (stdev-0.219)
```

P-Value and Statistical Significance

Is it possible I see a linear relationship in my data due to random chance? How can we be 95% sure the correlation between these two variables is real, not coincidental?

Our null hypothesis H_0 is that there is no relationship between two variables, or more technically the correlation coefficient is 0.

The alternative hypothesis H_1 is there is a relationship, and it can be a positive or negative correlation.



$$H_0 : \rho = 0 \text{ (implies no relationship)}$$

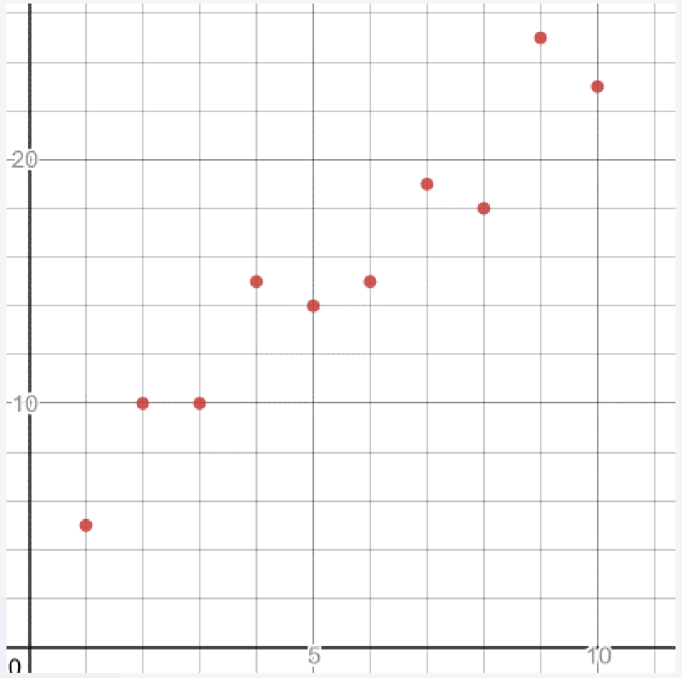
$$H_1 : \rho \neq 0 \text{ (relationship is present)}$$

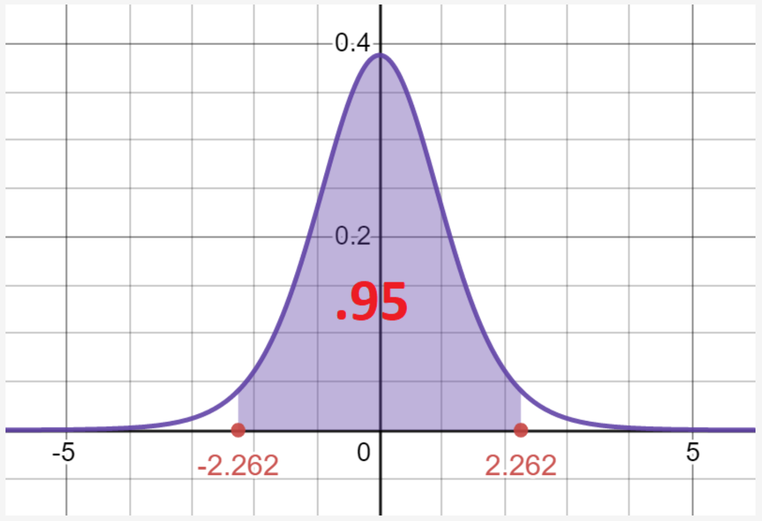
P-Value and Statistical Significance

Our data to the right has a correlation of 0.957586, which is strong but is it by random luck?

How can we be sure with 95% confidence?

Let's plot a T-distribution with a 95% critical value range, and since there are 10 data points that will be 9 degrees of freedom (10 -1 = 9).





```python
from scipy.stats import t

n = 10
lower_cv = t(n-1).ppf(.025)
upper_cv = t(n-1).ppf(.975)

print(lower_cv, upper_cv)
# -2.262157162740992 2.2621571627409915
```

P-Value and Statistical Significance

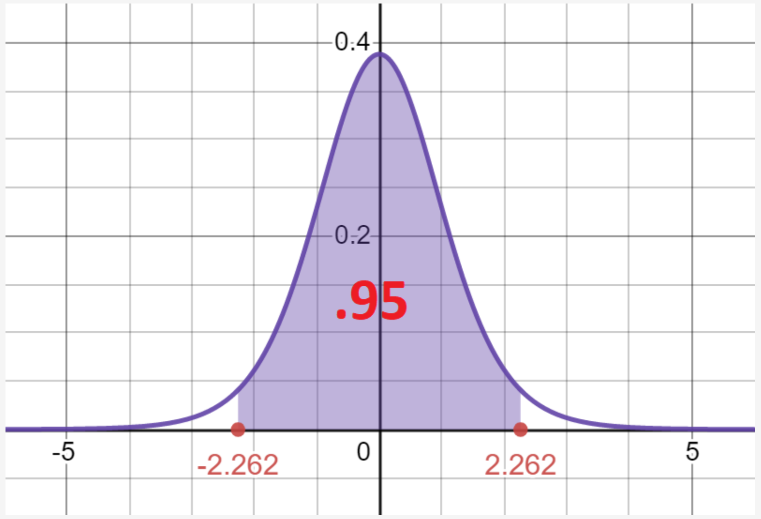
If our test value happens to fall outside this range of (-2.262, 2.262) we can reject our null hypothesis and say our correlation is real (not coincidental) with at least 95% confidence.

To calculate the test value *t*, we need to use this formula below, with *r* as the correlation coefficient and *n* as the sample size.

$$t = \frac{r}{\sqrt{\frac{1-r^2}{n-2}}} \qquad t = \frac{.957586}{\sqrt{\frac{1-.957586^2}{10-2}}} = 9.39956$$

The test value above is approximately 9.39956, which is outside the range of (-2.262, 2.262) so we can reject the null hypothesis and say our correlation is real.

The p-value is remarkably significant: .000005976.



```
from scipy.stats import t

n = 10
lower_cv = t(n-1).ppf(.025)
upper_cv = t(n-1).ppf(.975)

print(lower_cv, upper_cv)
# -2.262157162740992 2.2621571627409915
```

P-Value and Statistical Significance

```
from scipy.stats import t
from math import sqrt

# sample size
n = 10

lower_cv = t(n-1).ppf(.025)
upper_cv = t(n-1).ppf(.975)

# correlation coefficient
# derived from data https://bit.ly/2KF29Bd
r = 0.957586

# Perform the test
test_value = r / sqrt((1-r**2) / (n-2))

print("TEST VALUE: {}".format(test_value))
print("CRITICAL RANGE: {}, {}".format(lower_cv, upper_cv))

if test_value < lower_cv or test_value > upper_cv:
    print("CORRELATION PROVEN, REJECT H0")
else:
    print("CORRELATION NOT PROVEN, FAILED TO REJECT H0 ")
```

```
# Calculate p-value
if test_value > 0:
    p_value = 1.0 - t(n-1).cdf(test_value)
else:
    p_value = t(n-1).cdf(test_value)

# Two-tailed, so multiply by 2
p_value = p_value * 2
print("P-VALUE: {}".format(p_value))
```

Exercise

A dataset of red wine quality data is stored here, where the last column is a quality score: https://bit.ly/2Xov2ph

Perform a linear regression using cross-validation (3 splits) and test using R^2 as an evaluation metric.

If we use all independent variables does this linear regression perform well? Why or why not?

Exercise: ANSWER

When we train and perform a cross-validation, we don’t get a great fit.

Our R^2 is about .348 depending on your random seeding, which is much closer to 0 than 1.

It’s possible some variables are adding noise and useless information to our regression.

We can use Pearson correlation to try and remove these variables, as well as try Lasso regression which we will discuss later.

```python
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import KFold, cross_val_score

df = pd.read_csv('https://bit.ly/2Xov2ph', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)\
Y = df.values[:, -1]

# Perform a simple linear regression
kfold = KFold(n_splits=3, random_state=7, shuffle=True)
model = LinearRegression()

# Use R-square to evaluate the performance
# An R-square of 0 is no-fit, and 1 is perfect fit
results = cross_val_score(model, X, Y, cv=kfold, scoring='r2')

print("R^2: mean=%.3f (stdev-%.3f)" % (results.mean(), results.std()))
# R: mean=0.348 (stdev-0.015)
```

Transforming and Feature Engineering

Normalization

Normalization is scaling and converting the values of each variable, so they are relatively close together.

- Imagine you had a variable ***age*** whose values typically range in 0-99.
- But you had another variable ***income*** that typically ranges from 30,000 to 1,000,000.
- Because these ranges are so drastically different, fitting a model is not going to be productive until you transform them somehow.

There are a variety of techniques you can employ from linear scaling to fitting to a standard normal distribution.

Rescaling

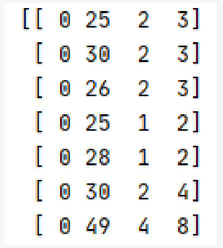
When you have different fields with varying scales, it can be helpful to rescale, or normalize, the data by compressing it between 0.0 and 1.0.

This can be helpful for optimization algorithms that perform training, making it easier to iterate parameters to fit the data more productively.

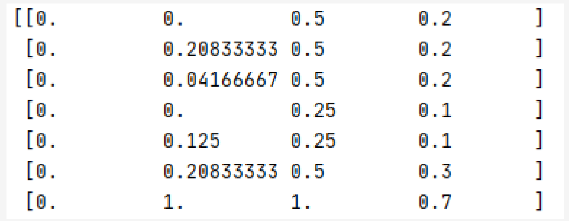
Note you do not always have to rescale, but sometimes it can give a better result especially for data with fields that vary in scale.

```python
import pandas as pd
from sklearn.preprocessing import MinMaxScaler

df = pd.read_csv('https://bit.ly/33iTfS9', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, 5th column)
Y = df.values[:, 4]

# Rescale all the input variables to be between 0 and 1
scaler = MinMaxScaler(feature_range=(0.0, 1.0))
rescaled_X = scaler.fit_transform(X)

print(rescaled_X)
```

Doing a simple rescale between 0.0 and 1.0 above in scikit-learn



```
[[ 0 25  2  3]
 [ 0 30  2  3]
 [ 0 26  2  3]
 [ 0 25  1  2]
 [ 0 28  1  2]
 [ 0 30  2  4]
 [ 0 49  4  8]
```



```
[[0.         0.         0.5        0.2       ]
 [0.         0.20833333 0.5        0.2       ]
 [0.         0.04166667 0.5        0.2       ]
 [0.         0.         0.25       0.1       ]
 [0.         0.125      0.25       0.1       ]
 [0.         0.20833333 0.5        0.3       ]
 [0.         1.         1.         0.7       ]
```

Standardization

Standardization is transforming numeric variables to fit a standard normal distribution (with a mean of 0 and standard deviation of 1) and express each value in standard deviations.

It can be preferable to rescaling, if the data can assume a normal distribution for each variable.

In other words, standardization is finding the z-scores for each variable.

```python
import pandas as pd
from sklearn.preprocessing import StandardScaler


df = pd.read_csv('https://bit.ly/3flZJSR', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Rescale all the input variables to be between 0 and 1
scaler = StandardScaler().fit(X)
rescaled_X = scaler.fit_transform(X)

print(rescaled_X)
```

Doing a standardization on all data fields

```
[[  6.    148.     72.    ...  33.6     0.627  50.   ]
 [  1.     85.     66.    ...  26.6     0.351  31.   ]
 [  8.    183.     64.    ...  23.3     0.672  32.   ]
```

```
[[ 0.63994726  0.84832379  0.14964075 ...  0.20401277  0.46849198
   1.4259954 ]
 [-0.84488505 -1.12339636 -0.16054575 ... -0.68442195 -0.36506078
  -0.19067191]
 [ 1.23388019  1.94372388 -0.26394125 ... -1.10325546  0.60439732
  -0.10558415]
```

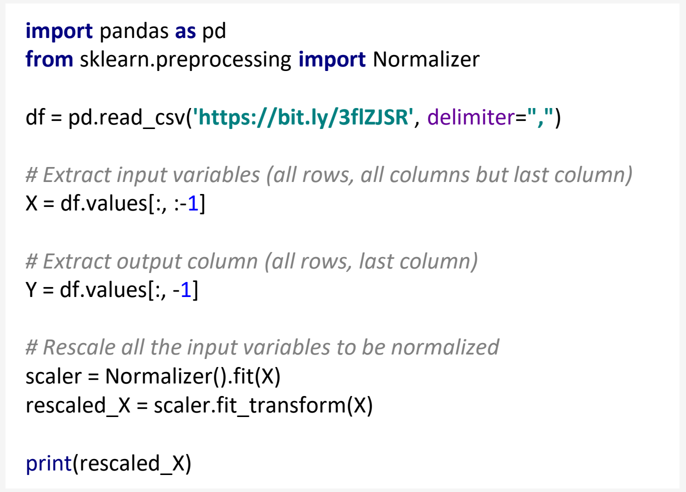
Unit Vector Normalization

Not to be confused with standardization, unit vector normalization is rescaling each data record to have a vector length of 1.0.

This is probably the most complicated transformation, as it involves some understanding of linear algebra.

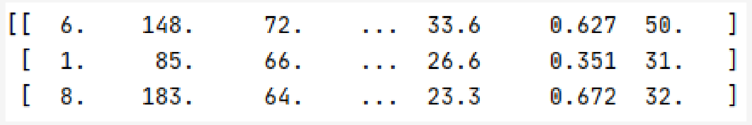
It works well for sparse data sets and it compresses the data into a smaller space, which is especially helpful for high-dimension datasets.

It is often used for neural networks, k-nearest neighbors, and a few other ML algorithms.

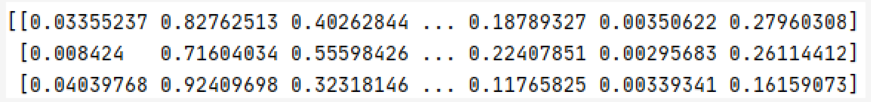


```python
import pandas as pd
from sklearn.preprocessing import Normalizer

df = pd.read_csv('https://bit.ly/3flZJSR', delimiter=",")

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

# Rescale all the input variables to be normalized
scaler = Normalizer().fit(X)
rescaled_X = scaler.fit_transform(X)

print(rescaled_X)
```

Doing a normalization on all data fields



```
[[  6.    148.     72.    ...  33.6     0.627  50.   ]
 [  1.     85.     66.    ...  26.6     0.351  31.   ]
 [  8.    183.     64.    ...  23.3     0.672  32.   ]
```



```
[[0.03355237 0.82762513 0.40262844 ... 0.18789327 0.00350622 0.27960308]
 [0.008424   0.71604034 0.55598426 ... 0.22407851 0.00295683 0.26114412]
 [0.04039768 0.92409698 0.32318146 ... 0.11765825 0.00339341 0.16159073]
```

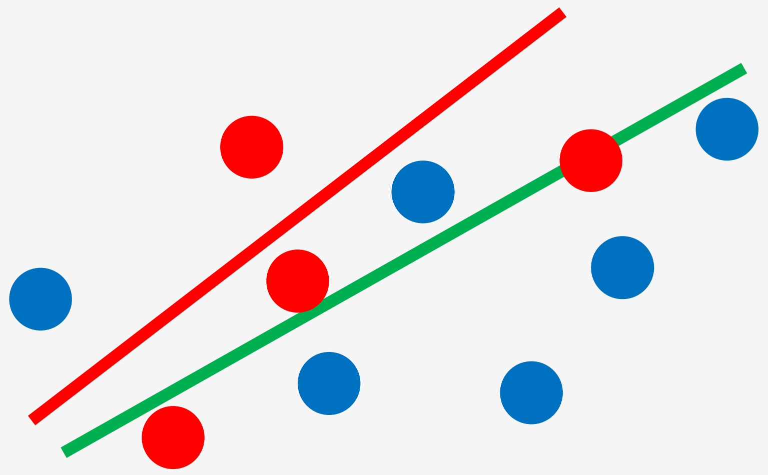
Regularization

Ridge Regression (L2)

Sometimes we want to increase the bias of our model, so it is less sensitive to variance in the training data.

This can especially be the case when we do not have much data and are worried about overfitting.

This is where **ridge regression** can be used, which minimizes not just on the sum/mean of squares but also a penalty value that is introduced.

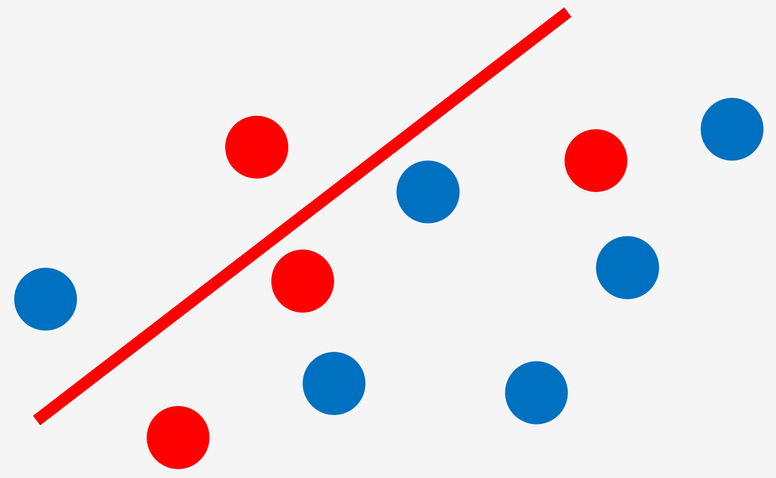


Ridge Regression (L2)

Notice to the right how we have trained our linear regression with the training data (red points) to the right.

But we are concerned with actual/testing data (blue points) which shows this linear regression is not predicting very well.

One thing we can do is increase the bias of our linear regression and make it less sensitive to the training data.



Ridge Regression (L2)

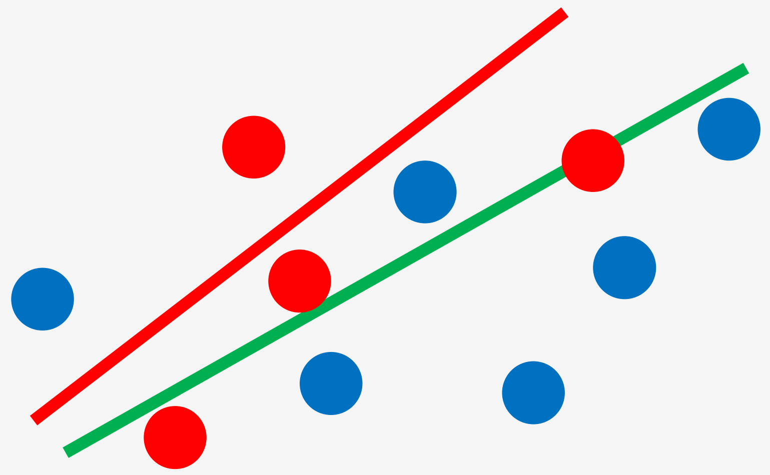
A ridge regression might adjust our linear regression as shown with the green line to the right.

This way it reduces the fit to our training data, increasing our bias and dropping our variance.

To implement our ridge regression, we modify our loss function as follows:

Loss = (Sum of squares) + $\lambda \times \text{slope}^2$

If we have multiple independent variables, each one gets its own $\lambda \times \text{slope}^2$ term.



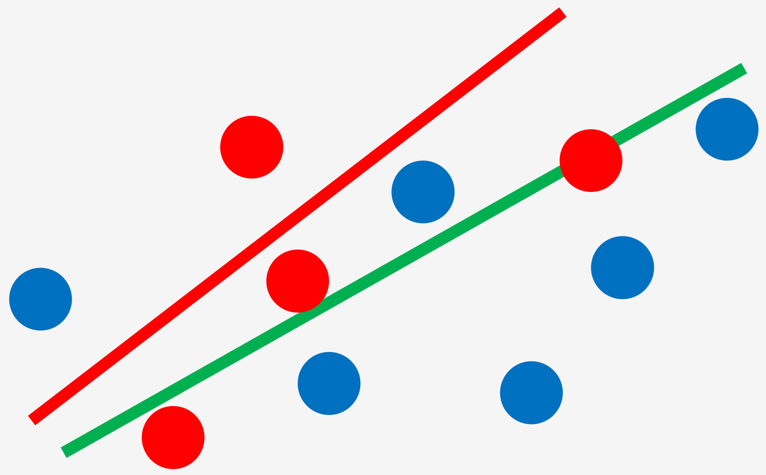
Ridge Regression (L2)

Loss = (Sum of squares) + $\lambda \times$ slope^2

This $\lambda \times$ slope^2 is a penalty to the least squares which prevents overfitting.

The lambda λ is a parameter to define how severe our penalty is (higher = less overfitting), and often optimized based on cross validation.

When we execute training on our data, we will get a less overfitted result.



Ridge Regression (L2)

```
import pandas as pd
from sklearn.linear_model import Ridge, LinearRegression
from sklearn.model_selection import KFold, cross_val_score

# Wine quality data
df = pd.read_csv('https://bit.ly/39uuK7J', delimiter=",")
print(df)

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

kfold = KFold(n_splits=3, random_state=7, shuffle=True)

# Ridge regression with 1.0 penalty
model = Ridge(alpha=1.0)

results = cross_val_score(model, X, Y, cv=kfold, scoring='neg_mean_squared_error')

print("Accuracy Mean: %.3f (stdev=%.3f)" % (results.mean(), results.std()))
```

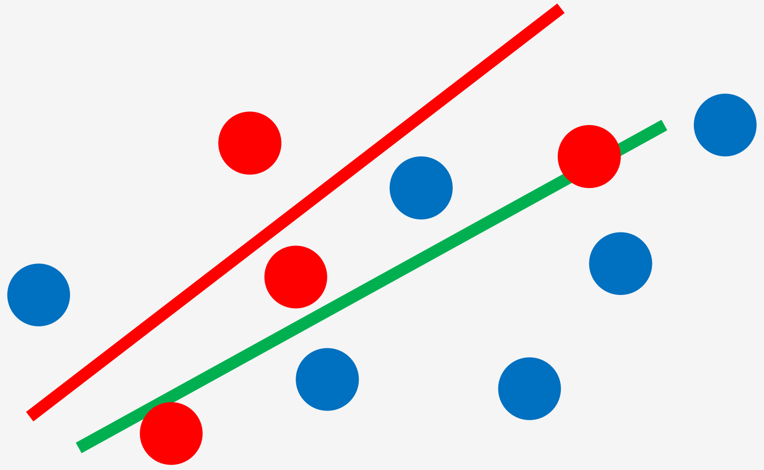
Lasso Regression (L1)

Lasso regression is similar to ridge regression, except it takes the absolute values of the slope(s) rather than square them.

Loss = (Sum of squares) + $\lambda \times |\text{slope}|$

If there are several independent variables, each variable gets its own $\lambda \times |\text{slope}|$

Lasso regression is handy when you may have variables that create noise and are irrelevant, and it will exclude those useless variables.



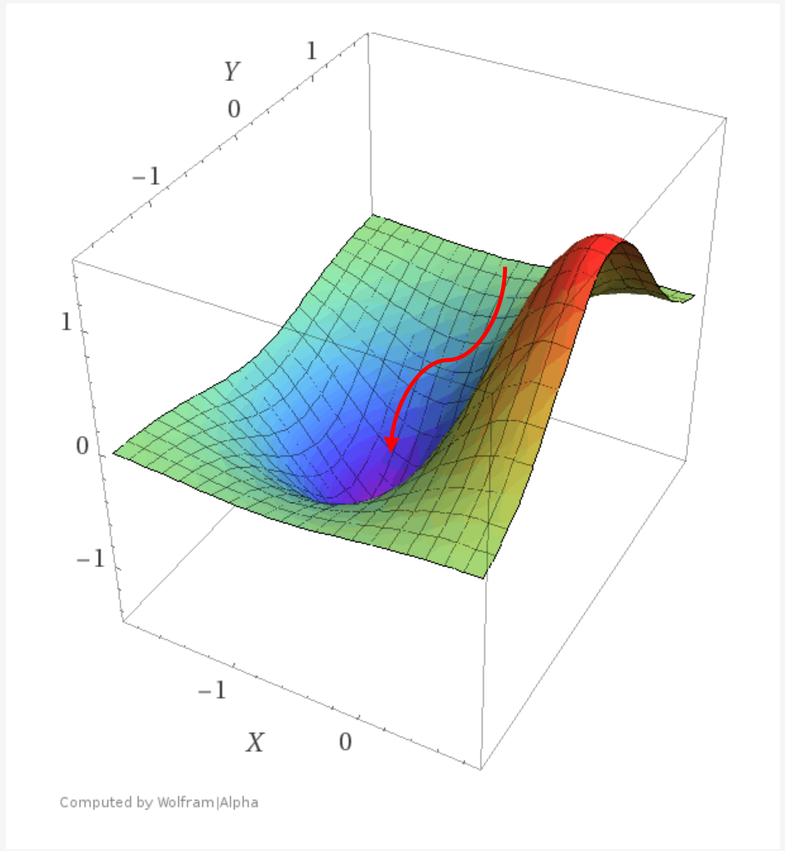
Lasso Regression (L1)

```
import pandas as pd
from sklearn.linear_model import Lasso
from sklearn.model_selection import KFold, cross_val_score

# Wine quality data
df = pd.read_csv('https://bit.ly/39uuK7J', delimiter=",")
print(df)

# Extract input variables (all rows, all columns but last column)
X = df.values[:, :-1]

# Extract output column (all rows, last column)
Y = df.values[:, -1]

kfold = KFold(n_splits=3, random_state=7, shuffle=True)

# Lasso regression with 0.1 penalty
model = Lasso(alpha=0.1)

results = cross_val_score(model, X, Y, cv=kfold, scoring='neg_mean_squared_error')

print("Accuracy Mean: %.3f (stdev=%.3f)" % (results.mean(), results.std()))
```

Stochastic Gradient Descent

Gradient Descent - Making Calculus Useful

- **Gradient descent** is a continuous/nonlinear optimization method that uses Calculus derivatives to find a local minimum/maximum.
- This methodology has become highly popular due to the need to optimize thousands and even millions of machine learning variables, like neural networks and their node weight values.
- While you can use simulated annealing to do these ML optimizations, gradient descent offers a more guided methodology that scales well with extremely large numbers of variables, at the cost of finding a better optimum.
- The main drawback of gradient descent is it is very mathy and easily gets stuck in local minima, which is why stochastic approaches are used to add randomness.



A Crash Course in Calculus Derivatives

A derivative tells the slope of a function (rate of change) at a given value for variable *x*.

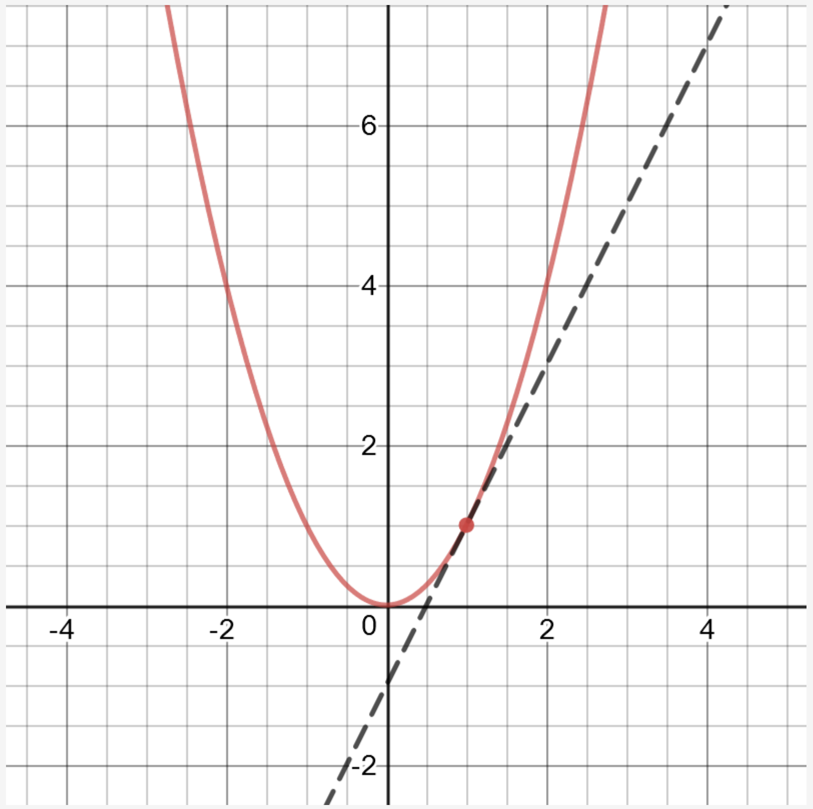
For example, the derivative of x^2 is $2x$.

$$\frac{\mathrm{d}}{\mathrm{d}x} x^2 = 2x$$

This means at $x = 1$, the slope of the function is 2.

- At $x = 2$, the slope is 4.
- At x = 3, it is 6.
- And so on…

Derivatives can be very useful for optimization. Can you imagine why? (Hint: where are we if the slope is 0?)



https://www.desmos.com/calculator/ruvtlp9zk6

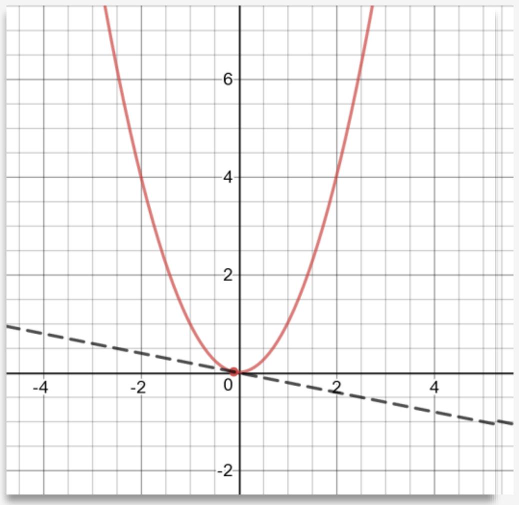
The Essence of Gradient Descent

Notice how when the slope is zero, we are at a local minimum/maximum!

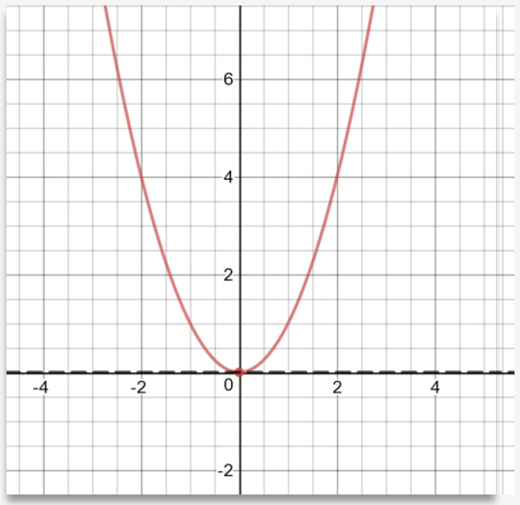
As the slope gets closer to zero, the closer we are to the local minimum /maximum.

Oppositely, the larger/steeper the slope the farther we are from the local minimum.

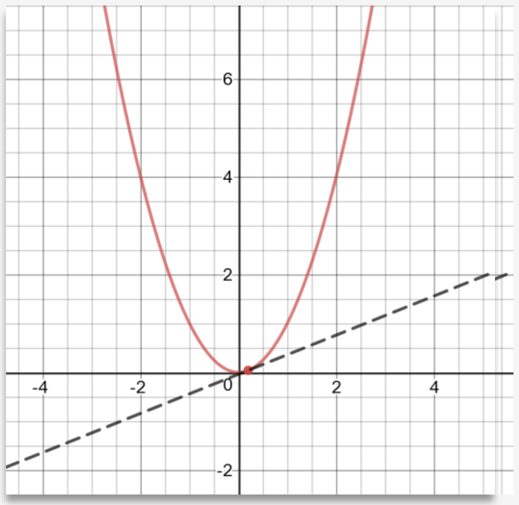
m =-0.4



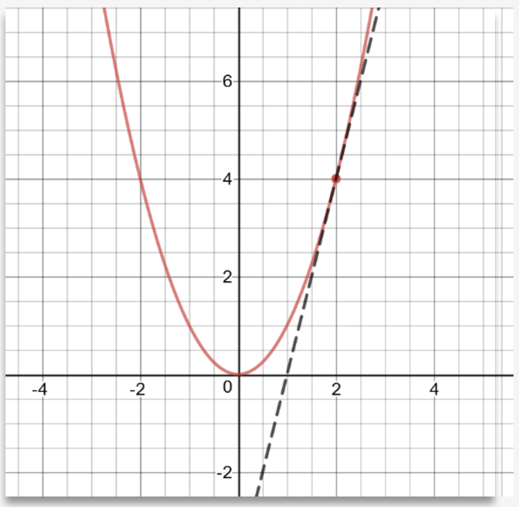
m = 0



m = 0.4



m = 4.0



https://www.desmos.com/calculator/ruvtlp9zk6

The Essence of Gradient Descent

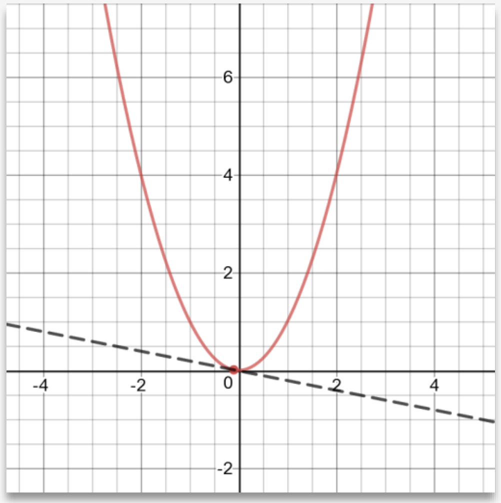
Starting our search in a random or arbitrary location, we want it to *step* towards the local minimum.

The slope is like a compass providing direction to the local minimum/maximum. We step in directions that make the slope smaller, leading it to 0.

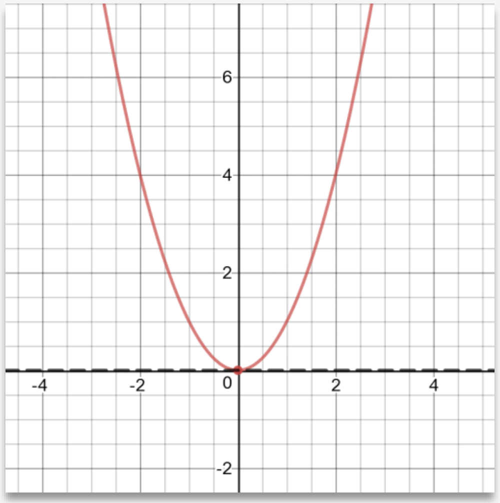
To make this process quicker, we can take bigger steps for bigger slopes, and smaller steps for smaller slopes.

This is the basic idea behind gradient descent.

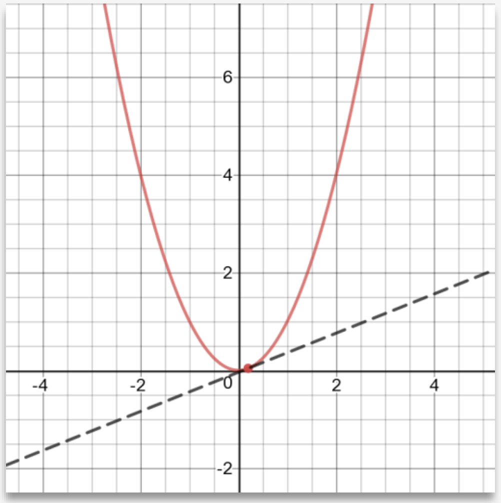
m =-0.4



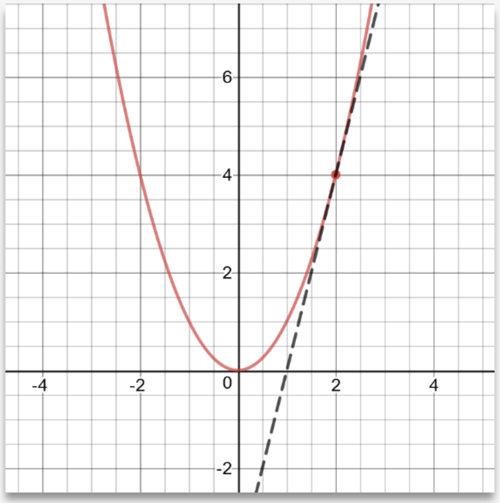
m = 0



m = 0.4



m = 4.0



https://www.desmos.com/calculator/ruvtlp9zk6

Gradient Descent for Linear Regression

For a simple linear regression fitting to *y* = *mx* + *b*, let's use the following mean of squares loss function:

$$E = \frac{1}{n}\sum_{i=0}^{n}(y_i - \bar{y}_i)^2$$

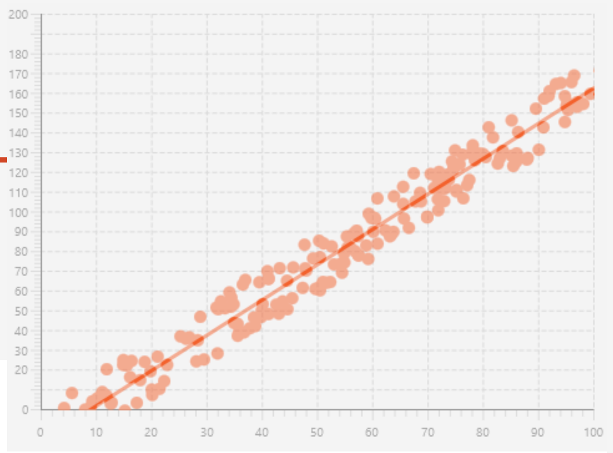
The derivative with respect to *m* is as follows:

$$D_m = \frac{-2}{n}\sum_{1=0}^{n} x_i(y_i - \bar{y}_i)$$

The derivative with respect to *b* is:

$$D_b = \frac{-2}{n}\sum_{i=0}^{n}(y_i - \bar{y}_i)$$

To the right is how we use these two partial derivatives to do gradient descent on a simple linear regression.



```python
import pandas as pd

# Input data
data = pd.read_csv('https://tinyurl.com/yaxgfjzt', header=None)
X = data.iloc[:, 0]
Y = data.iloc[:, 1]

# Building the model
m = 0.0
b = 0.0

L = .00001  # The learning Rate
epochs = 1000000  # The number of iterations to perform gradient descent

n = float(len(X))  # Number of elements in X

# Performing Gradient Descent
for i in range(epochs):
    Y_pred = m * X + b  # The current predicted value of Y
    D_m = (-2 / n) * sum(X * (Y - Y_pred))  # d/dm derivative of loss function
    D_b = (-2 / n) * sum(Y - Y_pred)  # d/dc derivative of loss function
    m = m - L * D_m  # Update m
    b = b - L * D_b  # Update b


print(m, b) # 1.786000956883716 -16.422581112886167
```

Gradient Descent

```
import pandas as pd

# Input data
data = pd.read_csv('https://bit.ly/3n1FKgE', header=0)
X = data.iloc[:, 0]
Y = data.iloc[:, 1]

# Building the model
m = 0.0
b =  0.0

L = .00001  # The learning Rate
epochs = 1000000  # The number of iterations to perform gradient descent

n = float(len(X))  # Number of elements in X

# Performing Gradient Descent
for i in range(epochs):
    Y_pred = m * X + b  # The current predicted value of Y
    D_m = (-2 / n) * sum(X * (Y - Y_pred))  # d/dm derivative of loss function
    D_b = (-2 / n) * sum(Y - Y_pred)  # d/dc derivative of loss function
    m = m - L * D_m  # Update m
    b = b - L * D_b  # Update b

    # print progress
    if i % 10000 == 0:
        print(i, m, b)

print(m, b)
```

Stochastic Gradient Descent

In practice, it can be expensive to process an entire data set at every iteration during training with gradient descent.

Fitting the entire training data on each iteration is also more likely to get stuck in a local minimum/maximum.

This is why **stochastic gradient descent** is how gradient descent is often used in practice (with neural networks, logistic regression, support vector machines, etc), as it randomly selects only a few training samples at every iteration, adding some randomized benefits to find a better local minimum as well as avoid saddles.

Learn more:
https://en.wikipedia.org/wiki/Stochastic_gradient_descent

Stochastic Gradient Descent

```
import pandas as pd
import numpy as np

# Input data
data = pd.read_csv('https://bit.ly/3n1FKgE', header=0)

X = data.iloc[:, 0].values
Y = data.iloc[:, 1].values

n = data.shape[0]  # rows

# Building the model
m = 0.0
b = 0.0

sample_size = 10  # sample size
L = .00001  # The learning Rate
epochs = 1000000  # The number of iterations to perform gradient descent

# Performing Stochastic Gradient Descent
for i in range(epochs):
    idx = np.random.choice(n, sample_size, replace=False)
    x_sample = X[idx]
    y_sample = Y[idx]

    Y_pred = m * x_sample + b  # The current predicted value of Y
    D_m = (-2 / sample_size) * sum(x_sample * (y_sample - Y_pred))  # d/dm derivative of loss function
    D_b = (-2 / sample_size) * sum(y_sample - Y_pred)  # d/dc derivative of loss function
    m = m - L * D_m  # Update m
    b = b - L * D_b  # Update b

    # print progress
    if i % 10000 == 0:
        print(i, m, b)

print(m, b)
```