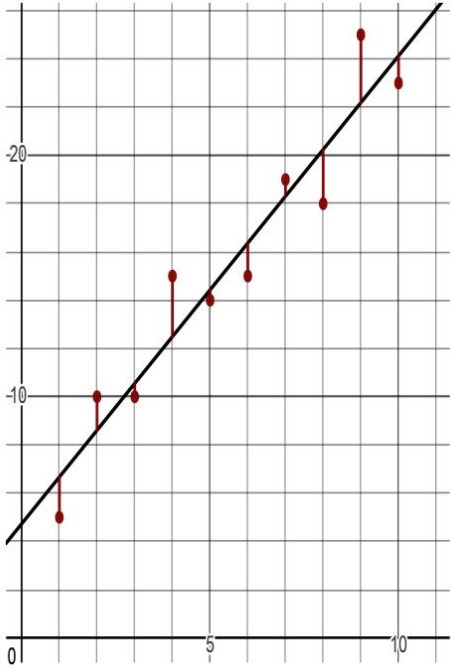
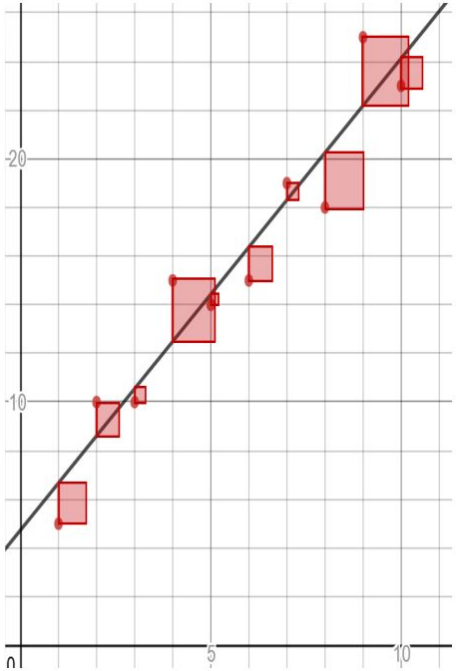
Linear regression

1. Finding the best fitting line

First we need to define a criterion based on which we will find the best fitting line. The most common criterion is the sum of squares.





Now that we have a way to measure the goodness of fit/loss function (sum of squares), we can look at the most common methods of finding the best fitting line based on the sum of squares. There are 5 main methods:

1.1.closed form equation

1.2.matrix inversion

1.3.matrix decomposition

1.4.gradient descent

1.5.stochastic gradient descent

1.1. Closed form equation

This is just a simple equation, but this only works for simple linear regression (exactly one independent variable) and this method is computationally slow for large amounts of data.

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

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

1.2. Matrix inversion

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

B will be the vector of the slopes (one slope for each independent variable). X is the matrix of observations (each column is an independent variable and each row is an observation). Y is the vector of our dependent variable. If we want to additionally get the value of the intercept we need to add a column on one's to our matrix X. If we have a lot of data we can make it easier for the computer and decompose matrix X into Q and R. The lowest equation shows how we can calculate the slope using Q and R.

$$X = Q \cdot R$$

$$b = R^{-1} \cdot Q^T \cdot y$$

1. Read the data
2. Extract input variables (X)
3. Add a column of ones to X (to calculate the intercept)
4. Extract the output column (Y)
5. Calculate the slope and intercept
6. Create a predicted y vector (optional)

Example 5-6. Using inverse and transposed matrices to fit a linear regression

```
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)
```

This is similar to the example before, but with the Q, R decomposition. In this step they don't make a prediction.

Example 5-7. Using QR decomposition to perform a linear regression

```
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 coefficents 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]
```

1.4. Gradient descent

Our goal is to minimize the residual sum of squares function. Because when the sum of squares is the smallest then our line best fits our data. If we want to find the minimum of a function we look for a point where the derivative (in other words the slope) is equal to 0. On the picture on the right there is an example (searching for the minimum of a quadratic function).

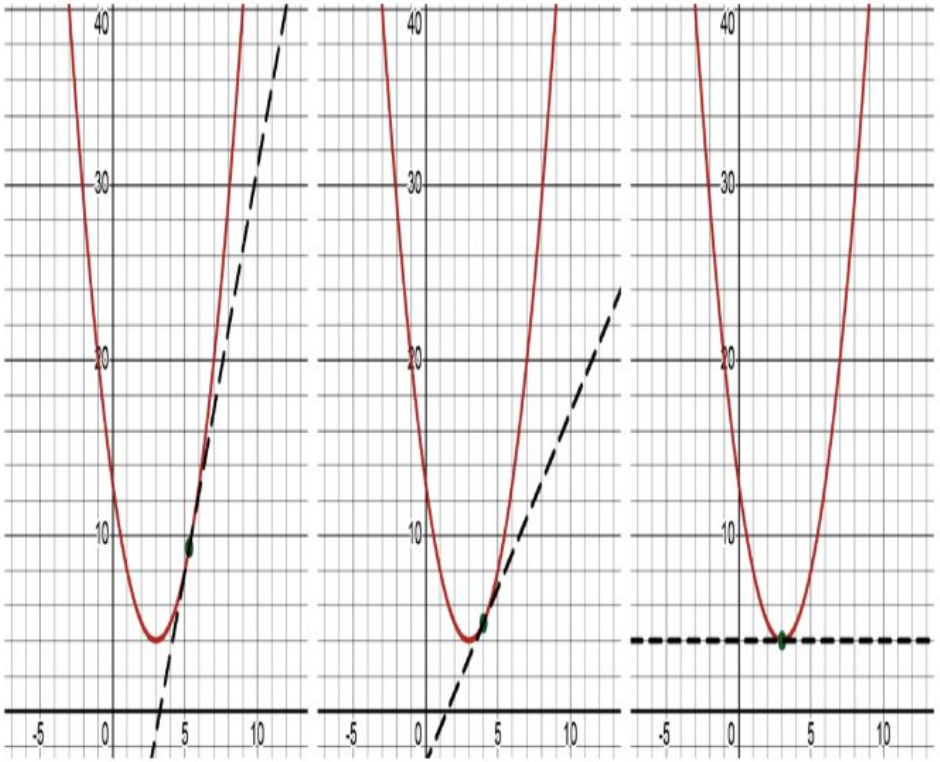


Figure 5-8. Stepping toward the local minimum where the slope approaches 0

Learning rate

As you can see in the right picture we ‘scan’ the function moving along its values by a given step size. This step size is called the learning rate. This is a hyperparameter (chosen by the user beforehand). If the learning rate (step size) is too small then the algorithm will take ages to find a solution, but that solution will be accurate. If the learning rate is to high, we will find a solution faster but it will be inaccurate (we could possibly miss a local minimum)

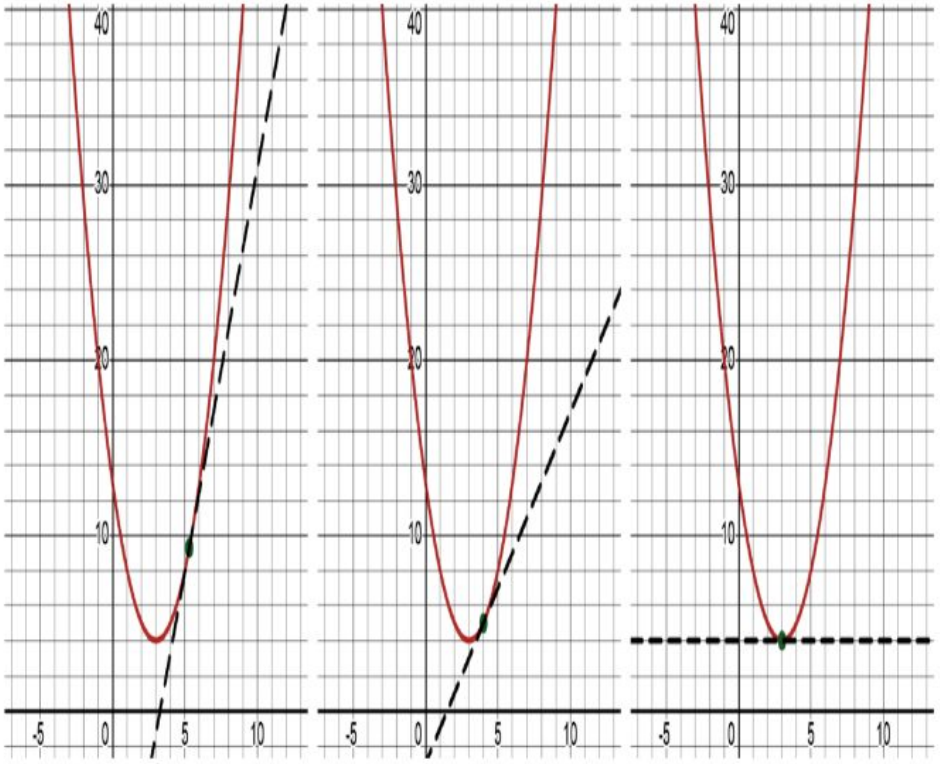


Figure 5-8. Stepping toward the local minimum where the slope approaches 0

We want to minimize the sum of squares function. To do this we first need to get its derivative. But remember that our primary goal is to find the parameters m and b (or b1 and b0) that best fit our data. So we have to calculate two separate derivatives. One derivative in respect to m and one derivative with respect to b. Then we will find the such an x where the derivative with respect to m is equal to 0 and in the next step we will find another x where the derivative with respect to b is equal to 0. The top equation is the sum of squares function and the bottom two equations are the respective derivatives.

$$e\left(x\right) = \sum_{i=0}^{n}((mx_i + b) - y_i\Bigg)^2$$

$$\frac{d}{dm}e\left(x\right) = \sum_{i=0}^{n} 2\left(b + mx_i - y_i\right)x_i$$

$$\frac{d}{db}e\left(x\right) = \sum_{i=0}^{n} \left(2b + 2mx_i - 2y_i\right)$$

This code shows how to calculate partial derivatives of the sum of squares function with respect to m and b.

Example 5-10. Calculating partial derivatives for m and b

```
from sympy import *

m, b, i, n = symbols('m b i n')
x, y = symbols('x y', cls=Function)

sum_of_squares = Sum((m*x(i) + b - y(i)) ** 2, (i, 0, n))

d_m = diff(sum_of_squares, m)
d_b = diff(sum_of_squares, b)
print(d_m)
print(d_b)

# OUTPUTS
# Sum(2*(b + m*x(i) - y(i))*x(i), (i, 0, n))
# Sum(2*b + 2*m*x(i) - 2*y(i), (i, 0, n))
```

This code shows how to perform gradient descent in Python. A lot of elements in the code are not clear to me. But generally you define initial m and b values and then iterate a given number of iterations. In each iteration you update the values of m and b.

Example 5-9. Performing gradient descent for a linear regression

```
import pandas as pd

# Import points from CSV
points = list(pd.read_csv("https://bit.ly/2KF29Bd").itertuples())

# Building the model
m = 0.0
b = 0.0


# The learning Rate
L = .001

# The number of iterations
iterations = 100_000

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

# Perform Gradient Descent
for i in range(iterations):

    # slope with respect to m
    D_m = sum(2 * p.x * ((m * p.x + b) - p.y) for p in points)

    # slope with respect to b
    D_b = sum(2 * ((m * p.x + b) - p.y) for p in points)

    # update m and b
    m -= L * D_m
    b -= L * D_b

print("y = {0}x + {1}".format(m, b))
# y = 1.9393939393939548x + 4.733333333333227
```

This is a graphical representation of the loss function (sum of squares). We want to get to the lowest point in the plain.

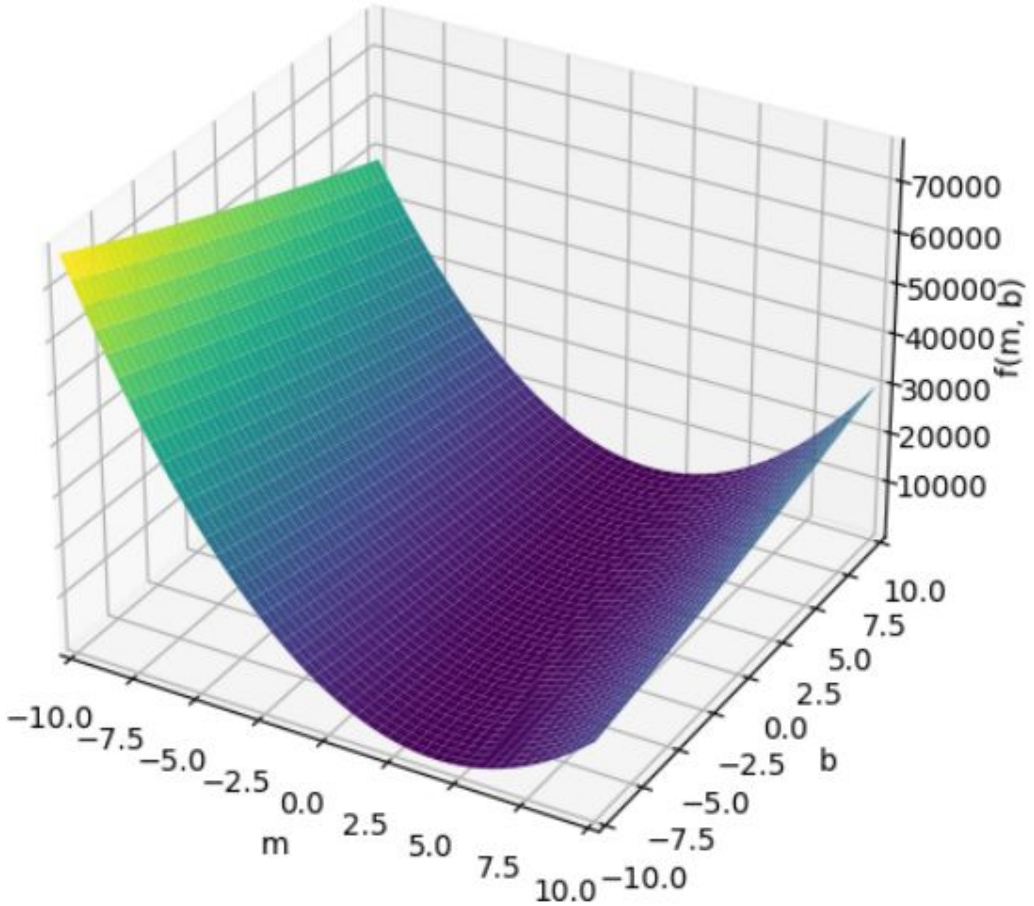


Figure 5-9. The loss landscape for a simple linear regression

overfitting

Why not just connect every data point? That would give us a sum of squares equal to 0 (an ideal fit). But the ultimate goal is not to fit the best line to our data, but to fit such a line that does the best job in predicting future observations. Obviously the function on the right is great for existing observations but would do a terrible job at predicting future observations (overfitting).

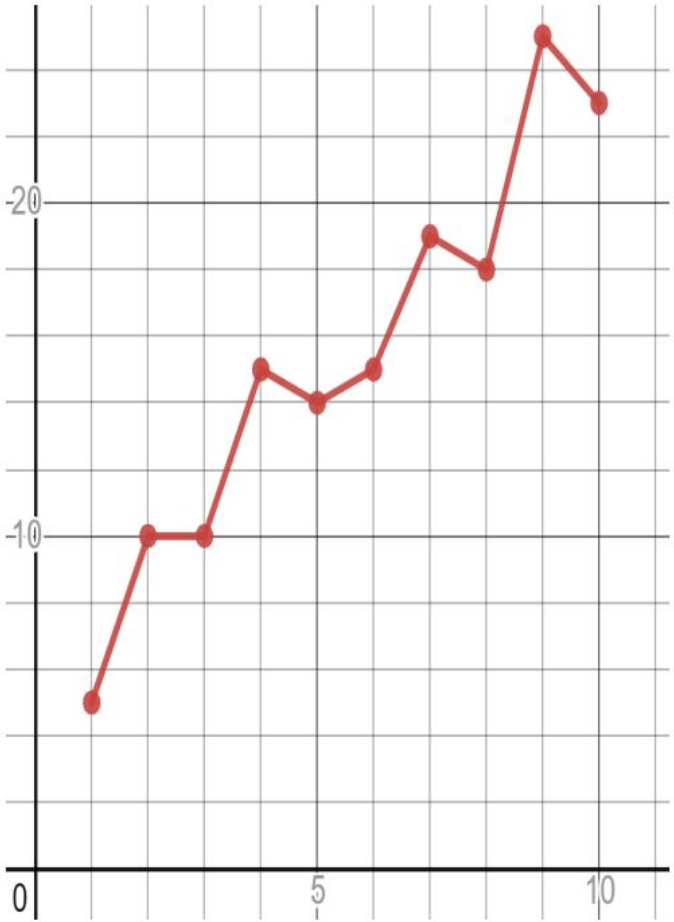
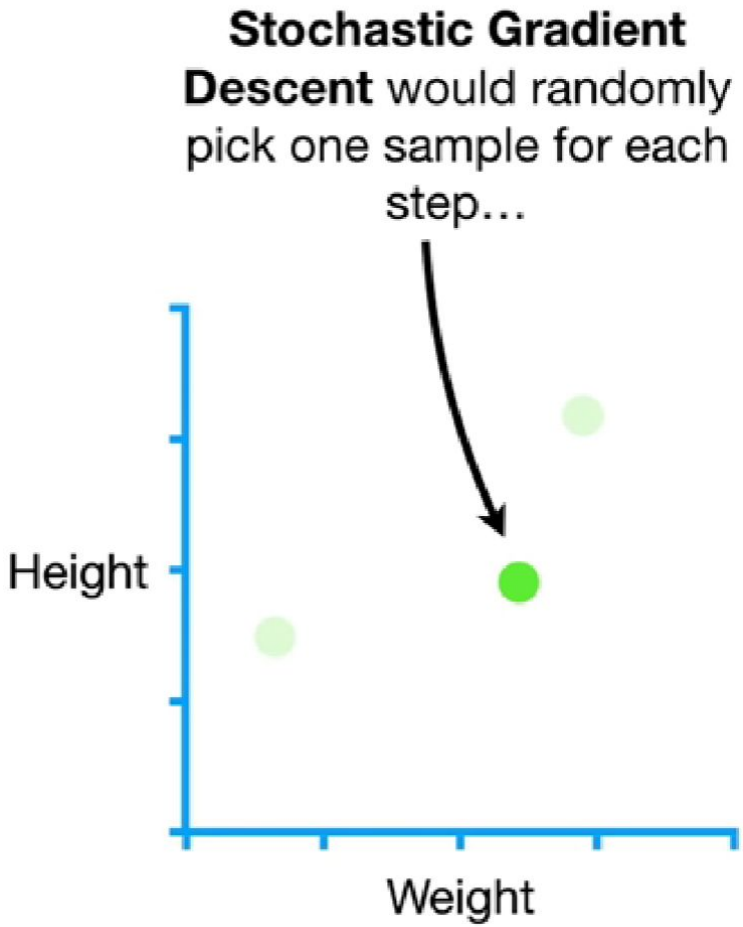


Figure 5-10. Performing a regression by simply connecting the points, resulting in zero loss

1.5. Stochastic gradient descent

In gradient descent we calculate the value of the derivative of the loss function (for example the least squares function) for every data point (every x). We do this two times, once with the derivative with respect to m and once with the derivative with respect to b. Normally we do many iterations (for example 1000). So if we had 1000 000 datapoint, we would need to make 1000 * 2 * 1000 000 calculations.

Stochastic gradient descent aims at drastically reducing the number of computations. This is especially useful in big datasets. In every iteration we only take one datapoint. So in the example from the previous slide we would only need to make 1000 * 2 * 1 iterations.



In addition Stochastic gradient descent prevents overfitting because we are training our model on a small subset of the data. Looking at only one datapoint introduces some randomness that prevents overfitting. Of Course stochastic gradient descent produces less accurate results than regular gradient descent. If we don’t only take one datapoint but for example 3 or 5, this is known as a ***mini-batch gradient descent***.

Code to run a stochastic gradient descent. The single data point is chosen in the first three lines of the for loop.

Example 5-13. Performing stochastic gradient descent for a linear regression

```
import pandas as pd
import numpy as np

# Input data
data = pd.read_csv('https://bit.ly/2KF29Bd', 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 = 1  # sample size
 L = .0001  # The learning Rate
 epochs = 1_000_000  # 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]

     # The current predicted value of Y
     Y_pred = m * x_sample + b

     # d/dm derivative of loss function
     D_m = (-2 / sample_size) * sum(x_sample * (y_sample - Y_pred))

     # d/db derivative of loss function
     D_b = (-2 / sample_size) * sum(y_sample - Y_pred)
     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("y = {0}x + {1}".format(m, b))
```

In the previous slides we have looked at 5 methods to fit a line to our data based on a loss function (in our example it was the sum of squares.) Once more, it is important to remember that we don’t want to find the line that best fits our data, but the line that is best for predicting new observations.

2. The Correlation coefficient

Correlation is a measure of the strength of the relationship between two variables. The Pearson's correlation coefficient can take on values from -1 to 1. -1 indicated a strong negative relationship, 0 indicates no relationship and 1 points to a strong positive relationship. Generally if there is a weak correlation, Linear Regression will not be a good model for our data, because the residuals will be high.

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

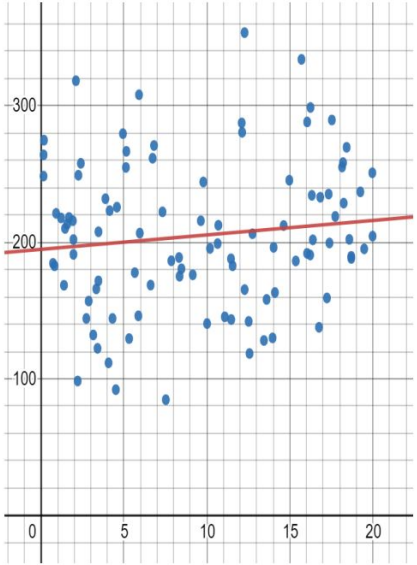


Figure 5-11. A scatterplot of data with high variance

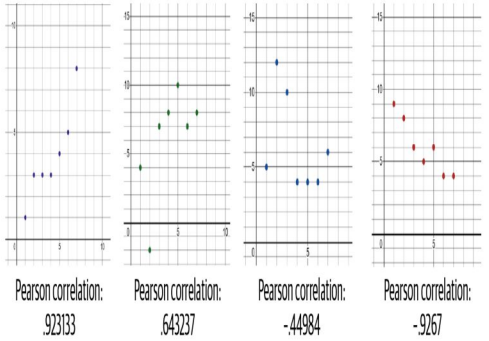


Figure 5-12. Correlation coefficients for four scatterplots

2.2. Statistical significance of the correlation coefficient

If we have a correlation coefficient we want to test how likely it is, we got this correlation did not occur by chance. To test this we perform a two tailed T test. Where n is our sample size. The next step is to confront our test results with the T distribution with a given confidence interval (in our case 95%) and our degrees of freedom. Now when we have our test score and our T distribution we can calculate the p-value, which tells us how statistically significant our correlation coefficient is. In most cases the more datapoint you have the lower your p-value.

$$t = \frac{r}{\sqrt{\frac{1-r^2}{n-2}}}$$

$$t = \frac{.957586}{\sqrt{\frac{1-.957586^2}{10-2}}} = 9.339956$$

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

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

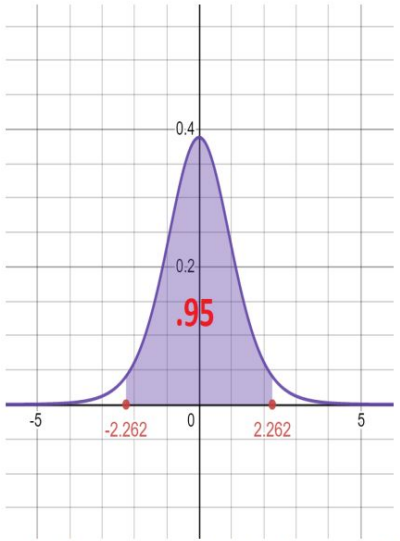


Figure 5-14. A T-distribution with 9 degrees of freedom, as there are 10 records and we subtract 1

Calculating the significance of the correlation coefficient in Python

First we define the H0 retention range based on the T distribution. Then we calculate our T test score based on our correlation coefficient r. Finally based on the T distribution and our test score we obtain the p-value.

Example 5-17. Testing significance for linear-looking data

```
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))
```

3. Coefficient of determination

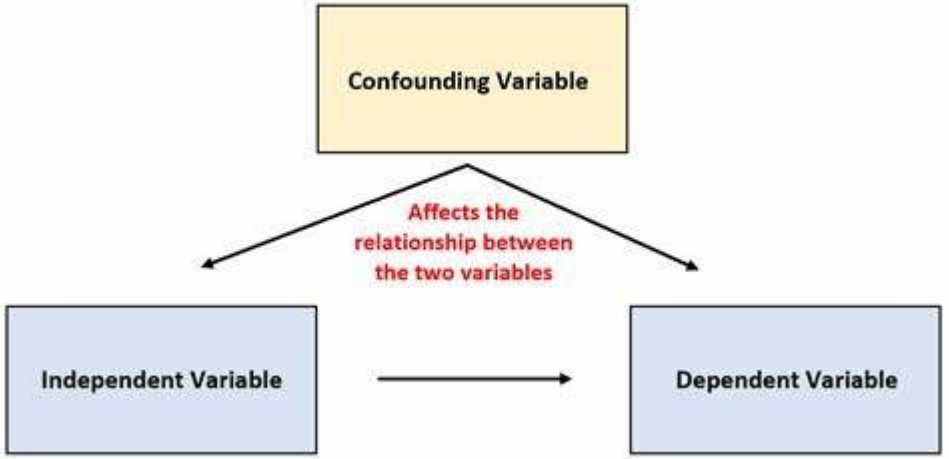
The coefficient of determination measure how much variation in one variable is explained by the variation of another variable. Mathematically this is just the square of the correlation coefficient (R^2), so it takes on values from 0 to 1. An R^2 of 0.9 means that 90% of the variance of x is explained by the variance of y and vice-versa. Meanwhile 10% of the variance of x (and y) is due to noise or some uncaptured variables.

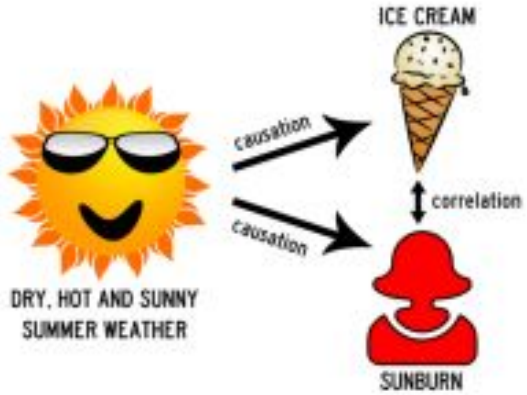
Example 5-18. Creating a correlation matrix in Pandas

```
import pandas as pd

# Read data into Pandas dataframe
df = pd.read_csv('https://bit.ly/2KF29Bd', delimiter=",")

# Print correlations between variables
coeff_determination = df.corr(method='pearson') ** 2
print(coeff_determination)

# OUTPUT:
#          x         y
# x  1.000000  0.916971
# y  0.916971  1.000000
```

Correlation is not causation

The fact that 90% of the variance of x is explained by the variance of y (and vice versa) doesn't mean that y has any effect on x and the other way around. The correlation could be due to a third variable z or due to random chance.





4. The Standard Error of the Linear Regression Estimate

$$S_e = \frac{\sum (y - \hat{y})^2}{n - 2}$$

The Standard Error of the Linear Regression Estimate is a way to measure the overall error of our linear regression. y_hat are the y values calculated based on the regression equation, y are the actual data point values. n is the sample size. We subtract two from n (n-2), because linear regression has two coefficients (m and b).

Example 5-19. Calculating the standard error of the estimate

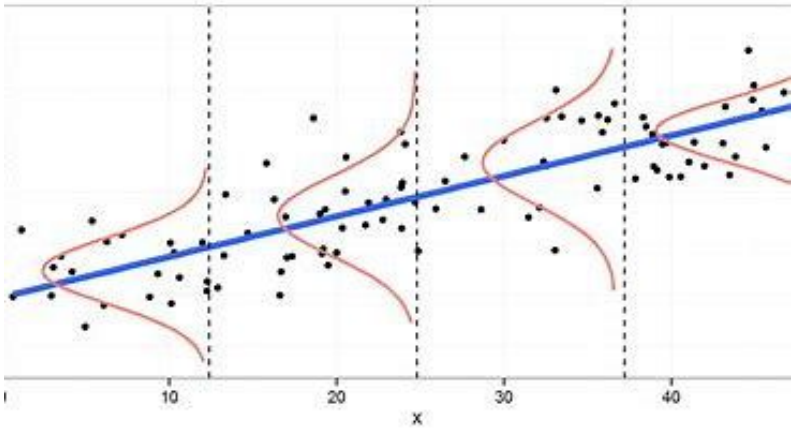
Here is how we calculate it in Python:

```
import pandas as pd
from math import sqrt

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

n = len(points)

# Regression line
m = 1.939
b = 4.733

# Calculate Standard Error of Estimate
S_e = sqrt((sum((p.y - (m*p.x +b))**2 for p in points))/(n-2))

print(S_e)
# 1.87406793500129
```

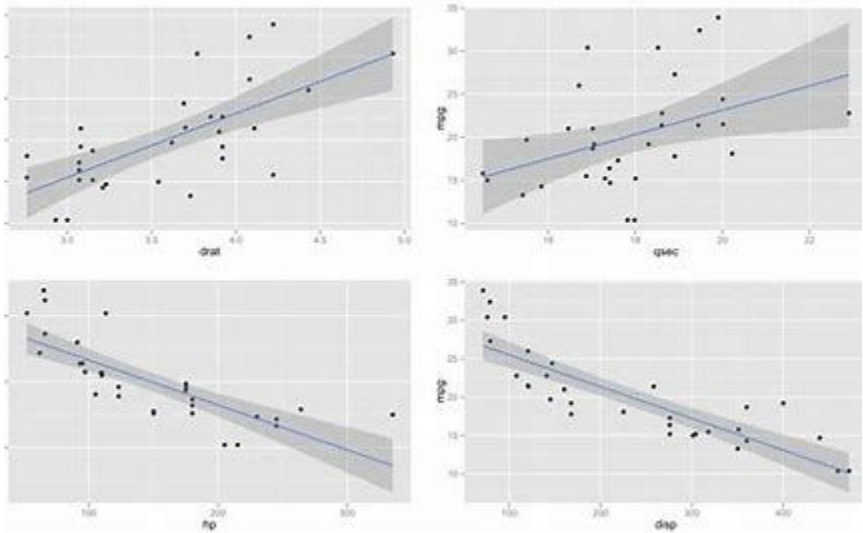
5. Prediction intervals

Linear Regression as a “moving normal distribution”

It is very useful to think of the regression line as a moving normal distribution. As seen on the picture on the right, the bell of the normal distribution moves along the regression line. Where the residuals are bigger, the bell is flatter (bigger residuals → more variance → a flatter bell). Where the residuals are smaller the bell is slimmer (smaller residuals → less variance → slimmer curve).



Prediction intervals will be explained on an example. Let's say that we have a regression equation y = mx + b, where y is the height and x is the weight. If we know the weight of a person we can plug this weight in as x, calculate y and say that a person of 70kg has a height of 175cm. But of course our linear regression is only a sample estimate. So it is safer to say that with 95% confidence a person with a weight of 70kg is taller than 170cm and shorter than 175cm. In our example the range [170cm, 175cm] is our 95% prediction interval. Prediction intervals aren't called confidence intervals, because we want to *predict* a new value. Prediction intervals are closely related to the moving normal distributions from the previous slide. They are broader the fewer observations we have in an area and they are narrower the more observations we have in a area.



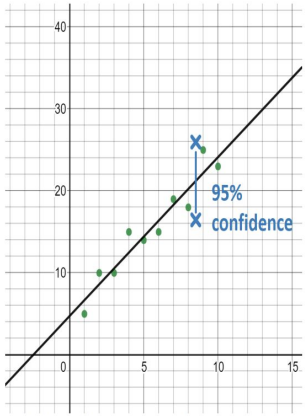


Figure 5-17. A prediction interval for a dog that is 8.5 years old with 95% confidence

$$E = t_{.025} * S_e * \sqrt{1 + \frac{1}{n} + \frac{n(x_0 + \bar{x})^2}{n(\sum x^2) - (\sum x)^2}}$$

The formula to calculate the prediction interval at x0. T0.025 is the critical value from the T distribution ans Se is the Standard Error of the linear regression (discussed in chapter 4.).

Below is the code to compute the prediction interval in Python.

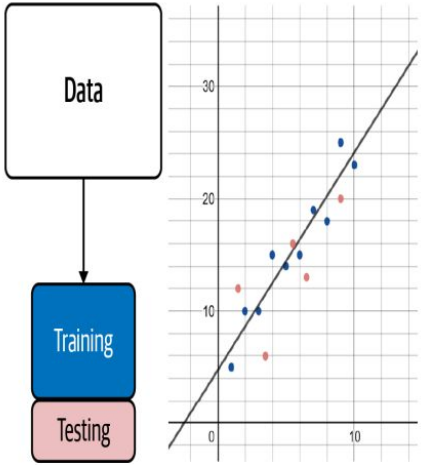
```
import pandas as pd
from scipy.stats import t
from math import sqrt

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

n = len(points)

# Linear Regression Line
m = 1.939
b = 4.733

# Calculate Prediction Interval for x = 8.5
x_0 = 8.5
x_mean = sum(p.x for p in points) / len(points)

t_value = t(n - 2).ppf(.975)

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

margin_of_error = t_value * standard_error * \
                  sqrt(1 + (1 / n) + (n * (x_0 - x_mean) ** 2) / \
                       (n * sum(p.x ** 2 for p in points) - \
                            sum(p.x for p in points) ** 2))

predicted_y = m*x_0 + b

# Calculate prediction interval
print(predicted_y - margin_of_error, predicted_y + margin_of_error)
# 16.462516875955465 25.966483124044537
```

6. Train test splits

machine learning vs linear regression

When we have very large amounts of multidimensional data (data with a lot of variables). Performing classic linear regression with all the p-values and prediction intervals is computationally hard. This is where machine learning comes in. Machine learning is less precise than linear regression but can handle the problems mentioned above. It can be said that linear regression is a scalpel and machine learning is a chainsaw. If we have enormous data that linear regression cannot handle, machine learning will give us at least a rough understanding of it.

This is a simple example of a train/test split, where ⅔ of the data are used to fit the best regression line and ⅓ of the data is used to evaluate that line. Below is a code how to perform this in Python.



Example 5-21. Doing a train/test split on linear regression

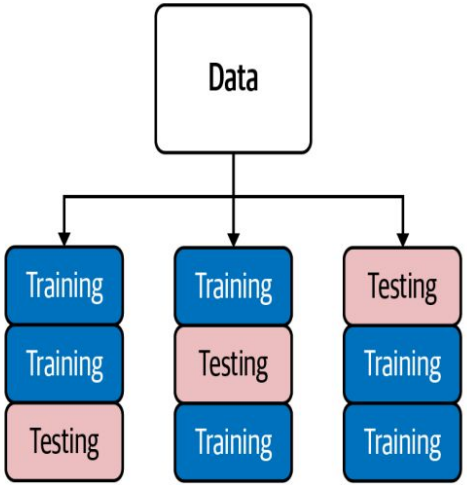
```
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# 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
# This leaves a third of the data out for testing
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
test_size=1/3)

model = LinearRegression()
model.fit(X_train, Y_train)
result = model.score(X_test, Y_test)
print("r^2: %.3f" % result)
```

k -fold cross validation

Another way of performing a train/test split is cross validation. We split our data into *k* parts (here 3 parts). We use one of the *k* parts for testing and the rest for training (here ⅓ vs ⅔). Then we repeat this process *k* times. ***Random shuffle cross validation*** adds the random shuffling (rearrangement) of our data in between folds. This is good for improving our model, but it is computationally expensive so it is not ideal for large amounts of data. If we have only a small amount of data we can perform a ***leave one out cross validation*** only one single observation is used for testing and the rest is used for training. The number of folds is equal to the number of datapoints. Below is a code to perform a 3-fold cross validation in Python.

Example 5-23. Using a random-fold validation for a linear regression

```
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()))
```



Example 5-22. Using three-fold cross-validation for a linear regression

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

a validation dataset

Sometimes it is a good idea to additionally test your model on a validation dataset. A validation dataset is such a dataset that was never part of neither the train nor the test dataset. In other words, the model has never seen this dataset ever before. Testing your model with the validation dataset should not be done instead of the classic train/test split but as an addition.