Linear Regression

Machine learning, which is one of the most emerging field which is used for predictive modeling with minimizing the error of a model or making the most accurate predictions possible, at the expense of explainability.

> **Linear regression** is used to find the relationship between two variables by fitting a linear equation to observed data.

Linear regression is a linear model, e.g. a model that assumes a linear relationship between the input variables (x) and the single output variable (y). More specifically, that y can be calculated from a linear combination of the input variables (x).

> When there is a single input variable (x), the method is referred to as **simple linear regression**.
>
> When there are multiple input variables, literature from statistics often refers to the method as **multiple linear regression**.

Different techniques can be used to prepare or train the linear regression equation from data, the most common of which is called Ordinary Least Squares. It is common to therefore refer to a model prepared this way as Ordinary Least Squares Linear Regression or just Least Squares Regression.

The aim of linear regression is to find a mathematical equation for a continuous response variable Y as a function of one or more X variable(s). So that you can use this regression model to predict the Y when only the X is known.

This mathematical equation can be generalised as follows:

$$Y=\beta1+\beta2X+\epsilon$$

where, β1 is the **intercept** and β2 is the **slope**.

Collectively, they are called regression coefficients and ϵ is the **error term**, the part of Y the regression model is unable to explain.

In [1]:
```
# Import matplotlib.
import matplotlib.pyplot as plt
```

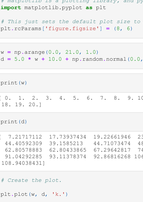
In [2]:
```
# Draw some axes.
plt.plot([-1, 10], [0, 0], 'k-')
plt.plot([0, 0], [-1, 10], 'k-')

# Plot the red, blue and green lines.
plt.plot([1, 1], [-1, 3], 'k:')
plt.plot([-1, 1], [3, 3], 'r:')

# Plot the two points (1,3) and (2,5).
plt.plot([1, 2], [3, 5], 'ko')
# Join them with an (extending) green line.
plt.plot([-1, 10], [-1, 21], 'g-')

# Set some reasonable plot limits.
plt.xlim([-1, 10])
plt.ylim([-1, 10])

# Show the plot.
plt.show()
```



Example

Businesses often use linear regression to understand the *relationship* between advertising spending and revenue.

For example, they might fit a *simple linear regression* model using advertising spending as the predictor variable and revenue as the response variable. The regression model would take the following form:

> **revenue = β0 + β1(ad spending)**

The coefficient **β0** would represent total expected revenue when ad spending is zero.

The coefficient **β1** would represent the average change in total revenue when ad spending is increased by one unit (e.g. one dollar).

If **β1** is negative, it would mean that more ad spending is associated with less revenue.

If **β1** is close to zero, it would mean that ad spending has little effect on revenue.

And if **β1** is positive, it would mean more ad spending is associated with more revenue.

Depending on the value of **β1**, a company may decide to either decrease or increase their ad spending.

Analysis

Here we'll import the Python libraries we need for or investigations below.

In [46]:
```
# Numpy efficiently deals with numerical multi-dimensional arrays.
import numpy as np

# matplotlib is a plotting library, and pyplot is its easy-to-use module.
import matplotlib.pyplot as plt

# This just sets the default plot size to be bigger.
plt.rcParams['figure.figsize'] = (8, 6)
```

In [15]:
```
w = np.arange(0.0, 21.0, 1.0)
d = 5.0 * w + 10.0 * np.random.normal(0.0, 5.0, w.size)
```

In [17]:
```
print(w)
```

```
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11. 12. 13. 14. 15. 16. 17.
 18. 19. 20.]
```

In [18]:
```
print(d)
```

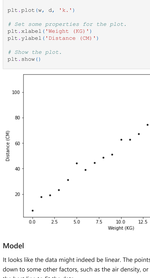
```
[  7.21717112  17.73937434  19.22661946  23.28940383  31.21462185
   44.40592309  39.1585213   44.71073474  48.45667017  50.99365
   62.80578883  62.80433665  67.29642817  76.41509741  72.9092546
   91.04292285  93.11378374  92.96816268 106.25291576 100.53468395
  108.94038431]
```

In [19]:
```
# Create the plot.

plt.plot(w, d, 'k.')

# Set some properties for the plot.
plt.xlabel('Weight (KG)')
plt.ylabel('Distance (CM)')

# Show the plot.
plt.show()
```



Model

It looks like the data might indeed be linear. The points don't exactly fit on a straight line, but they are not far off it. We might put that down to some other factors, such as the air density, or errors, such as in our tape measure. Then we can go ahead and see what would be the best line to fit the data.

Straight lines

All straight lines can be expressed in the form $y = mx + c$. The number m is the *slope* of the line. The slope is how much y increases by when x is increased by 1.0. The number c is the *y*-intercept of the line. It's the value of y when x is 0.

Fitting the model

To fit a straight line to the data, we just must pick values for m and c. These are called the parameters of the model, and we want to pick the best values possible for the parameters. That is, the best parameter values *given* the data observed. Below we show various lines plotted over the data, with different values for m and c.

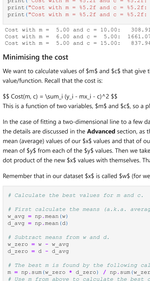
In [20]:
```
# Plot v versus d with black dots.
plt.plot(w, d, 'k.', label="Data")

# Overlay some lines on the plot.
x = np.arange(0.0, 21.0, 1.0)
plt.plot(x, 5.0 * x + 10.0, 'r-', label="$5x + 10$")
plt.plot(x, 6.0 * x +  5.0, 'g-', label="$6x +  5$")
plt.plot(x, 5.0 * x + 15.0, 'b-', label="$5x + 15$")

# Add a legend.
plt.legend()

# Add axis labels.
plt.xlabel('Weight (KG)')
plt.ylabel('Distance (CM)')

# Show the plot.
plt.show()
```



Calculating the cost

You can see that each of these lines roughly fits the data. Which one is best, and is there another line that is better than all three? Is there a "best" line?

It depends how you define the word best. Luckily, everyone seems to have settled on what the best means. The best line is the one that minimises the following calculated value.

$$ \sum_{i,y_i} (y_i - mx_i - c)^2 $$
Here (x_i, y_i) is the i^{th} point in the data set and $\sum_i$ means to sum over all points. The values of m and c are to be determined. We usually denote the above as $\cost(m, c)$.

Where does the above calculation come from? It's easy to explain the part in the brackets $(y_i - mx_i - c)$. The corresponding value to x_i in the dataset is y_i. These are the measured values. The value $m x_i + c$ is what the model says y_i should have been. The difference between the value that was observed (y_i) and the value that the model gives ($m x_i + c$), is $y_i - mx_i - c$.

Why square that value? Well note that the value could be positive or negative, and you sum over all of these values. If we allow the values to be positive or negative, then the positive could cancel the negatives. So, the natural thing to do is to take the absolute value $\mid y_i - m x_i - c \mid$. Well it turns out that absolute values are a pain to deal with, and instead it was decided to just square the quantity instead, as the square of a number is always positive. There are pros and cons to using the square instead of the absolute value, but the square is used. This is usually called *least squares* fitting.

In [21]:
```
# Calculate the cost of the lines above for the data above.
cost = lambda m,c: np.sum([(d[i] - m * w[i] - c)**2 for i in range(w.size)])

print("Cost with m = %5.2f and c = %5.2f: %8.2f" % (5.0, 10.0), cost(5.0, 10.0))
print("Cost with m = %5.2f and c = %5.2f: %8.2f" % (6.0,  5.0), cost(6.0,  5.0))
print("Cost with m = %5.2f and c = %5.2f: %8.2f" % (5.0, 15.0), cost(5.0, 15.0))

Cost with m =  5.00 and c = 10.00:   308.91
Cost with m =  6.00 and c =  5.00:  1661.07
Cost with m =  5.00 and c = 15.00:   837.94
```

Minimising the cost

We want to calculate values of m and c that give the lowest value for the cost value above. For our given data set we can plot the cost value/function. Recall that the cost is:

$$ \cost(m, c) = \sum_i (y_i - mx_i - c)^2 $$
This is a function of two variables, m and c, so a plot of it is three dimensional. See the **Advanced** section below for the plot.

In the case of fitting a two-dimensional line to a few data points, we can easily calculate exactly the best values of m and c. Some of the details are discussed in the **Advanced** section, as they involve calculus, but the resulting code is straight-forward. We first calculate the mean (average) values of our x values and that of our y values. Then we subtract the mean of x from each of the x values, and the mean of y from each of the y values. Then we take the *dot product* of the new x values and the new y values and divide it by the dot product of the new x values with themselves. Then gives us m, and we use m to calculate c.

Remember that in our dataset x is called w (for weight) and y is called d (for distance). We calculate m and c below.

In [23]:
```
# Calculate the best values for m and c.

# First calculate the means (a.k.a. averages) of w and d.
w_avg = np.mean(w)
d_avg = np.mean(d)

# Subtract means from w and d.
w_zero = w - w_avg
d_zero = d - d_avg

# The best m is found by the following calculation.
m = np.sum(w_zero * d_zero) / np.sum(w_zero * w_zero)
# Use m from above to calculate the best c.
c = d_avg - m * w_avg

print("m is %4.2f and c is %6.2f." % (m, c))
```

```
m is 4.965506 and c is 10.325747.
```

Note that numpy has a function that will perform this calculation for us, called polyfit. It can be used to fit lines in many dimensions.

In [25]:
```
np.polyfit(w, d, 1)
```

Out[25]:
```
array([ 4.96550363, 10.32574705])
```

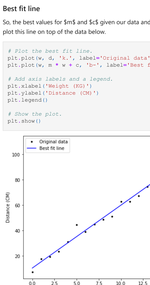
Best fit line

So, the best values for m and c given our data and using least squares fitting are about $4.955 for m and about $11.135 for c. We plot this line on top of the data below.

In [26]:
```
# Plot the best fit line.
plt.plot(w, d, 'k.', label='Original data')
plt.plot(w, m * w + c, 'b-', label='Best fit line')

# Add axis labels and a legend.
plt.xlabel('Weight (KG)')
plt.ylabel('Distance (CM)')
plt.legend()

# Show the plot.
plt.show()
```



Note that the $\cost$ of the best m and best c is not zero in this case.

In [27]:
```
print("Cost with m = %5.2f and c = %5.2f: %8.2f" % (m, c, cost(m, c)))
```

```
Cost with m =  4.97 and c = 10.33:   307.98
```