model_fitting

April 26, 2020

1 Linear regression

Last session, we looked at correlation to assess whether two variables related to each other. Unfortunately, this did not tell us anything about the direction of causality: A correlation between variable X and Y could mean that X predicts Y, but it could also mean that Y predicts X, or even that a third variable predicts both X and Y.

This session's statistical tool allows you to be a bit more explicit about your predictions. In a **regression**, you use one or more variables to predict one or more other variables. This doesn't automatically mean that regressions tell you about the direction of causality, but at least you can use it to get a bit closer.

1.1 The basics

We'll start of simple, with a regression in which one variable X predicts another variable Y. This is roughly equivalent to a correlation. In order to make this a bit more exciting, let's look at some real data!

1.1.1 Background to our data

At the Winter Olympics, there is a sport called *speed skating*. If you're not from a select few countries, you might not have heard of it. It's essentially like running, but then you do it on ice, using special skates. In most disciplines, only two skaters compete against each other at a time. The times are compared between all athletes afterwards, and whoever had the best time wins. One of the more exciting races, is the 500 meter sprint. All competing athletes are *highly* trained, and focus on raw power and technique, as that is all that matters. Right?

Well, perhaps not. Perhaps coincidence or even bias might also play a role. You see, the starting procedure in speed skating is like most other racing sports. The referee says "Ready?", then waits for a bit, and then shoots their starting pistol. The regulations are clear on that "bit" of time: It should be 1-1.5 seconds from the moment athletes are in position, and hence it's regulated to be random.

Unlike in other racing sports, speed skaters race alone or against a single opponent, and times are compared between all skaters afterwards. (So the main competitors for the win might not directly face each other!) This means that every pair of racers starts with a different interval between their "Ready" cue and the starting shot.

Does this matter? In theory, the "Ready" signal could be considered an *alerting cue*, and we know from psychological research that the time between an alerting cue and a subsequent signal

affects how quickly people respond to that signal. In practice, an interval of 500 ms results in an optimal response time, and longer intervals result in higher response times.

Now, obviously, the Winter Olympics are not some psychological experiment. These aren't a bunch of students in some dank lab room, these are highly trained athletes competing under immense pressure, looked on by thousands of spectators in the stadium, and even more watching from home through live stream or television. Surely this alerting thing will not affect their actual performance?

That sounds like an empirical question!

1.1.2 Loading our data

The National Skating Union records not only finish times at 500 meters, but also measures athletes' time 100 meters into the race. Both of these numbers are freely available. Researchers from the Universities of Oxford (UK) and Utrecht (Netherlands) have collected data on the intervals between the onset of the "Ready" signal and the onset of the starting shot. The attached file, `speed_skating_all_races.csv`, has both the 100 and 500 meter times, and the ready-start intervals. In addition, there is a column that indicates whether an athlete fell or stumbled during a race.

You can load those data into Python using NumPy's `loadtxt` function:

```
In [ ]: import numpy
        from matplotlib import pyplot

        # Load the data from all individual races.
        data = numpy.loadtxt("speed_skating_all_races.csv", \
            delimiter=",", dtype=float, skiprows=1, unpack=True)
```

Let's do some convenience renaming on the columns in the data file:

```
In [ ]: # Get the ready-start interval data.
        interval = data[0,:]
        # Get the times at 100 and 500 meters.
        time_100m = data[1,:]
        time_500m = data[2,:]
        # Get the sex data, which is formatted as a 1 for male and
        # 0 for female. We can cast this into Booleans: True (1)
        # or False (0) for the question "Is this athlete male?"
        is_male = data[3,:].astype(bool)
        # The data on falls is formatted in the same way: 1 for a
        # fall/stumble, and 0 for a regular race. We can cast this
        # into Booleans too: True (1) of False (0) for "Did this
        # athlete fall?"
        fall = data[4,:].astype(bool)
```

1.1.3 Getting a feel for the data

Let's plot the data! Make sure to plot the (near) falls in a different colour, so we can see whether they really are different from the normal races.

```
In [ ]: # Plot the ready-start interval on the x-axis, and the 500
        # meter times on the y-axis.
        pyplot.plot(interval[fall==False], time_500m[fall==False], \
            'o', color="#FF69B4")
        # Also plot the races in which athletes fell or stumbled.
        pyplot.plot(interval[fall==True], time_500m[fall==True], \
            'o', color="#006900")

        # Add axis labels to make the plot clearer.
        pyplot.xlabel("Ready-start interval (sec)", fontsize=18)
        pyplot.ylabel("Finish time (sec)", fontsize=18)
```

So the first thing you see is that there are two REALLY slow races, both due to falls. Let's set a limit on the y-axis that will allow us to actually see the data.

```
In [ ]: # Plot the ready-start interval on the x-axis, and the
        # 500 meter times on the y-axis.
        pyplot.plot(interval[fall==False], time_500m[fall==False], \
            'o', color="#FF69B4")
        # Also plot the races in which athletes fell or stumbled.
        pyplot.plot(interval[fall==True], time_500m[fall==True], \
            'o', color="#006900")

        # Add axis labels to make the plot clearer.
        pyplot.xlabel("Ready-start interval (sec)", fontsize=18)
        pyplot.ylabel("Finish time (sec)", fontsize=18)

        # Set the axis limit.
        pyplot.ylim([34, 41])
```

It's quite clear there are two separate sub-groups in the data here... Let's see whether that corresponds with sex:

```
In [ ]: # MALE
        # Plot the ready-start interval on the x-axis, and the 500
        # meter times on the y-axis.
        pyplot.plot(interval[(is_male == True) & (fall==False)], \
            time_500m[(is_male == True) & (fall==False)], 'o', \
            color="#4e9a06", label="Men")
        # Also plot the races in which athletes fell or stumbled.
        pyplot.plot(interval[(is_male == True) & (fall==True)], \
            time_500m[(is_male == True) & (fall==True)], 'o', \
            color="#8ae234")

        # FEMALE
        # Plot the ready-start interval on the x-axis, and the 500
        # meter times on the y-axis.
        pyplot.plot(interval[(is_male == False) & (fall==False)], \
            time_500m[(is_male == False) & (fall==False)], 'o', \
```

```
    color="#c4a000", label="Women")
# Also plot the races in which athletes fell or stumbled.
pyplot.plot(interval[(is_male == False) & (fall==True)], \
    time_500m[(is_male == False) & (fall==True)], 'o', \
    color="#fce94f")

# Add axis labels to make the plot clearer.
pyplot.xlabel("Ready-start interval (sec)", fontsize=18)
pyplot.ylabel("Finish time (sec)", fontsize=18)

# Set the axis limit.
pyplot.ylim([34, 41])

# Add a legend.
pyplot.legend(loc="lower right")
```

It seems that on average, men were about 4 seconds quicker than women. That means we can't just compute a Pearson correlation on the entire dataset as one group: Clearly there are two separate underlying distributions. However, we don't really care about the difference between men and women here. Instead, we're only interested in whether or not there is an effect of ready-start interval on finish times.

In order to look at this, we could z-score the data within each group. This subtracts the group mean from every sample, and then divides it by the group standard deviation. The z-scored finish times for men and women should both have a mean of 0 and a standard deviation of 1, making the two groups directly comparable. (And, more importantly, combinable!)

```
In [ ]: from scipy.stats import zscore

        # Compute the z-scored finish times.
        z_time_500m_male = \
            zscore(time_500m[(is_male==True) & (fall==False)])
        z_time_500m_female = \
            zscore(time_500m[(is_male==False) & (fall==False)])

        # Combine the two vectors into one.
        z_time_500m = \
            numpy.hstack([z_time_500m_male, z_time_500m_female])

In [ ]: # Import the Pearson correlation function.
        from scipy.stats import pearsonr

        # Compute the correlation between interval and finish time.
        r, p = pearsonr(interval[fall==False], z_time_500m)

        pyplot.plot(interval[fall==False], z_time_500m, 'o')
```

```
print("R={}, p={}".format(round(r, ndigits=2), \
    round(p, ndigits=3)))
```

OK, so we now know that there is a *statistically* significant correlation between ready-start interval and finish time. However, we don't know whether this correlation is *practically* significant. Ideally, we would like to know exactly how a longer ready-start interval affects the finish time. In other words: If the referee waits 1 second longer to shoot the starting pistol, how much slower does an athlete become?

To answer this question, we can use *regression*.

1.2 Linear regression

One thing that wasn't mentioned about the 500 meter sprint in speed skating, is that athletes race twice. After the first round of races, pairs are mixed up, and all athletes race again. Their times of both races are combined, and whoever has the lowest summed time wins a gold medal.

This is not unlike an experimental manipulation: It sounds a bit like a researcher used two trials per participant to estimate the effect of ready-start interval on finish time. This is exactly how you can use the data!

For our each athlete, you can compute the difference in ready-start interval between both races. This will be your *predictor* variable `x`.

You can also compute the difference in finish times between both races. This will be your *outcome* variable `y`.

In a regression, you try to predict the outcome with one or more predictors. Or, in an equation: $y = \beta_0 + x_1\beta_1 + \epsilon$

Where y is the outcome variable, β_0 is the intercept (what is y when all x values are zero?), x_1 is the first predictor variable, β_1 is a free variable that determines how much x_1 affects y, and ϵ is the *error term* that determines how much was unaccounted for. Sometimes this is called *noise*, because it refers to all unpredicted things.

Or, in code:

```
from scipy.stats import linregress
slope, intercept, r_value, p_value, std_err = linregress(x, y)
```

First, we need to compute our predictor and outcome: The differences between the two races of each athlete in ready-start interval and finish time. We can load the data for individual races from the file `speed_skating_paired_races.csv`.

```
In [ ]: import numpy

        # Load the data from all individual races.
        data = numpy.loadtxt("speed_skating_paired_races.csv", \
            delimiter=",", dtype=float, skiprows=1, \
            usecols=range(1,9), unpack=True)

        # Rename the variables for our convenience.
        interval_1 = data[0,:]
        time_100m_1 = data[1,:]
        time_500m_1 = data[2,:]
        interval_2 = data[3,:]
```

```
time_100m_2 = data[4,:]
time_500m_2 = data[5,:]
is_male = data[6,:].astype(bool)
exclude = data[7,:].astype(bool)
```

Now we can compute the differences in ready-start interval and finish time between the two races. We'll exclude all the athletes who (nearly) fell, but we won't separate men and women again. This is because we have no reason to assume that alerting effects are any different between men and women, and thus our hypothesis should be that *all* athletes are affected, regardless of sex.

```
In [ ]: # Compute the interval difference between race 1 and 2.
        interval_d = \
            interval_1[exclude==False] - interval_2[exclude==False]
        # Compute the finish time difference between race 1 and 2.
        time_500m_d = \
            time_500m_1[exclude==False] - time_500m_2[exclude==False]
```

Let's do a quick check to see what our data looks like:

```
In [ ]: # Plot the values.
        pyplot.plot(interval_d, time_500m_d, 'o', color="#FF69B4")
        # Add axis labels.
        pyplot.xlabel("Ready-start interval difference (sec)", fontsize=18)
        pyplot.ylabel("Finish time difference (sec)", fontsize=18)
```

In the graph, we can see that ready-start interval differences lie between -1 and 1 seconds. We can also see that finish times are quite stable within each individual athlete: Most differences are between -0.4 and 0.4 seconds!

In addition, just by eyeballing the graph, it looks like there might be a positive correlation between the ready-start interval difference and the finish time difference. Let's quantify this relation by using a regression:

```
In [ ]: from scipy.stats import linregress

        slope, intercept, r, p, std_err = \
            linregress(interval_d, time_500m_d)

        print("R={}, p={}".format(round(r, ndigits=2), \
            round(p, ndigits=3)))
        print("slope=%.2f, intercept=%.2f" % (slope, intercept))
```

From the output, we can learn that there is a statistically significant correlation with a Pearson R of 0.35. In addition, we now know how to quantify the effect:

$y = 0.03 + 0.17x_1$

or:

$\Delta_{finish} = 0.03 + 0.17\Delta_{interval}$

In practice, this means that for every second the referee waits between "Ready" and the starting shot, they add (on average) 0.17 seconds to an atheletes finish time. You can draw this line into your graph:

```
In [ ]: # Select a bunch of x values that will cover the range of
        # the interval difference data.
        x = numpy.arange(-1, 1, 0.1)
        # Compute what the predicted y values would be for each
        # of these values.
        y = intercept + slope * x

        # Plot the line into a graph.
        pyplot.plot(x, y, '-', linewidth=3, color="#FF69B4")

        # Plot the measured values.
        pyplot.plot(interval_d, time_500m_d, 'o', color="#FF69B4", \
            alpha=0.5)

        # Add axis labels.
        pyplot.xlabel("Ready-start interval difference (sec)", \
            fontsize=18)
        pyplot.ylabel("Finish time difference (sec)", fontsize=18)
```

Is this a problem? Well, at Vancouver in 2010, the total difference between gold and silver was 0.16 seconds. It is not uncommen at all for speed skaters to get even closer: In 2014, the difference between gold and silver was 0.01 seconds, and between gold and bronze it was 0.15. In 2018, the difference between gold and silver was 0.01 seconds again.

Clearly, the margins are small, and thus this ready-start interval effect might have real-life consequences.

1.2.1 References

If you're interested in the background to the data, you can read the following two articles. They're very short, and not very technical:

- Dalmaijer, E.S., Nijenhuis, B.G., & Van der Stigchel, S. (2015). Life is unfair, and so are racing sports: Some athletes can randomly benefit from alerting effects due to inconsistent starting procedures. Frontiers in Psychology, 6(1618). doi:10.3389/fpsyg.2015.01618
- Dalmaijer, E.S., Nijenhuis, B.G., & Van der Stigchel, S. (2016). Commentary: Life is unfair, and so are racing sports: Some athletes can randomly benefit from alerting effects due to inconsistent starting procedures. Frontiers in Psychology, 7(119). doi:10.3389/fpsyg.2016.00119

1.3 But how does regression work?

In regression, you have an explicit *model* for your data. Specifically, it says that there is a linear relationship between variables y and x:

$y = \beta_0 + x_1\beta_1 + \epsilon$

You know what the values for y are, because you measured those. You also know what the values for x are, because you manipulated (or measured) those. But how do you know what the β values are? And what that ϵ is?

One way would be to simply try all possible values of each β, and see when the resulting line fits best. How do you know what set of β values fits best? Simple: It's when the difference between your predicted values of y and your measured values of y is the smallest.

Let's give this approach a go with the skating dataset. (As you've hopefully been doing with all snippets; read the comments for explanations of specific lines!)

```
In [ ]: # First, define the ranges along which we need to
        # search for the best fitting betas.
        b0_range = numpy.arange(0, 1, 0.01)
        b1_range = numpy.arange(-10, 10, 0.01)

        # Count the number of values we will try for each beta.
        n_b0 = len(b0_range)
        n_b1 = len(b1_range)

        # Second, define some starting values. The first are the
        # betas, which will be None to start with.
        beta = (None, None)
        # We also need to start with a difference between the y
        # values and the predicted y values. This
        # will start at infinitely high:
        min_s = numpy.inf

        # Finally, we loop through every possible combination of
        # b0 and b1.
        for i, b0 in enumerate(b0_range):
            for j, b1 in enumerate(b1_range):
                # Predict y using the current betas.
                y_pred = b0 + b1 * interval_d
                # Compute the difference between the predicted y
                # and the measured y for each observation.
                d = time_500m_d - y_pred
                # Compute the sum of squares of the differences
                # (residuals).
                s = numpy.sum(d**2)
                # Remember the current betas if the sum of squares
                # is lower than the previously lowest.
                if s < min_s:
                    betas = (b0, b1)
                    min_s = s

        print("Best fit: b0={}, b1={}".format(round(betas[0], ndigits=2), \
            round(betas[1], ndigits=2)))
```

As you might recall, these are the same values that you obtained through using the `linregress` function earlier!

The reason this works, is because you travelled through *parameter space*, and at each point computed the *sum of squares* of the difference between the actual and your predicted values. This

difference is called the *residuals*. You kept track of which point in parameter space was associated with the lowest *squared residuals*. This is called *least-squares regression*.

You can actually plot parameter space and the associated residual squares:

```
In [ ]: # First, define the ranges along which we need to search
        # for the best fitting betas.
        b0_range = numpy.arange(0, 10, 0.01)
        b1_range = numpy.arange(-10, 10, 0.01)

        # Count the number of values we will try for each beta.
        n_b0 = len(b0_range)
        n_b1 = len(b1_range)

        # Second, define some starting values. The first are the
        # betas, which will be None to start with.
        beta = (None, None)
        # We also need to start with a difference between the y
        # values and the predicted y values. This
        # will start at infinitely high:
        min_s = numpy.inf
        # Keep track of the residuals at every point in parameter
        # space. This starts as a matrix filled with NaN (not a
        # number), and one value will be added on every iteration.
        s = numpy.zeros((n_b0,n_b1), dtype=float) * numpy.NaN

        # Finally, we loop through every possible combination of
        # b0 and b1.
        for i, b0 in enumerate(b0_range):
            for j, b1 in enumerate(b1_range):
                # Predict y using the current betas.
                y_pred = b0 + b1 * interval_d
                # Compute the difference between the predicted y
                # and the measured y for each observation.
                d = time_500m_d - y_pred
                # Compute the sum of squares of the differences
                # (residuals).
                s[i,j] = numpy.sum(d**2)
                # Remember the current betas if the sum of squares
                # is lower than the previously lowest.
                if s[i,j] < min_s:
                    betas = (b0, b1)
                    min_s = s[i,j]

        print("Best fit: b0={}, b1={}".format(round(betas[0], ndigits=2), \
            round(betas[1], ndigits=2)))

        # Now plot the residual squares in parameter space:
        pyplot.imshow(s, cmap="hot")
```

```
        # Set the tick labels on the x and y axes.
        x_ticks = [0, n_b1//2, n_b1-1]
        x_tick_labels = numpy.round(b1_range[x_ticks])
        pyplot.xticks(x_ticks, x_tick_labels)
        pyplot.xlabel("Possible beta 1 values")
        y_ticks = [0, n_b0-1]
        y_tick_labels = numpy.round(b0_range[y_ticks])
        pyplot.yticks(y_ticks, y_tick_labels)
        pyplot.ylabel("Possible beta 0 values")

        # Draw an colour bar to show the resulting sums of residual
        # squares.
        pyplot.colorbar()
```

From this plot, you can see that the optimal combination of betas is close to point (0,0). However, you can also see that we cast a *very* wide net. Perhaps it would have been better to choose a smaller search space. For example, instead of using ranges $[0, 10]$ for β_0 and $[-10, 10]$ for β_1, we could have used $[0, 0.1]$ and $[0, 0.5]$

```
In [ ]: # First, define the ranges along which we need to
        # search for the best fitting betas.
        b0_range = numpy.arange(0, 0.1, 0.001)
        b1_range = numpy.arange(0, 0.5, 0.001)

        # Count the number of values we will try for each beta.
        n_b0 = len(b0_range)
        n_b1 = len(b1_range)

        # Second, define some starting values. The first are the
        # betas, which will be None to start with.
        beta = (None, None)
        # We also need to start with a difference between the y
        # values and the predicted y values. This
        # will start at infinitely high:
        min_s = numpy.inf
        # Keep track of the residuals at every point in parameter
        # space. This starts as a matrix filled with NaN (not a
        # number), and one value will be added on every iteration.
        s = numpy.zeros((n_b0,n_b1), dtype=float) * numpy.NaN

        # Finally, we loop through every possible combination of
        # b0 and b1.
        for i, b0 in enumerate(b0_range):
            for j, b1 in enumerate(b1_range):
                # Predict y using the current betas.
                y_pred = b0 + b1 * interval_d
                # Compute the difference between the predicted y
                # and the measured y for each observation.
```

```
            d = time_500m_d - y_pred
            # Compute the sum of squares of the differences
            # (residuals).
            s[i,j] = numpy.sum(d**2)
            # Remember the current betas if the sum of squares
            # is lower than the previously lowest.
            if s[i,j] < min_s:
                betas = (b0, b1)
                min_s = s[i,j]

print("Best fit: b0={}, b1={}".format(round(betas[0], ndigits=2), \
    round(betas[1], ndigits=2)))

# Now plot the residual squares in parameter space:
pyplot.imshow(s, cmap="hot")
# Set the tick labels on the x and y axes.
x_ticks = [0, n_b1//2, n_b1-1]
x_tick_labels = numpy.round(b1_range[x_ticks], 2)
pyplot.xticks(x_ticks, x_tick_labels)
pyplot.xlabel("Possible beta 1 values")
y_ticks = [0, n_b0-1]
y_tick_labels = numpy.round(b0_range[y_ticks], 2)
pyplot.yticks(y_ticks, y_tick_labels)
pyplot.ylabel("Possible beta 0 values")

# Draw an colour bar to show the resulting sums of residual
# squares.
pyplot.colorbar()
```

Here, it's a lot clearer that the best fitting combination of predictor values is around (0.17, 0.03).

1.3.1 Doing your own least-squares regression

What you just did is a *full space estimate*. The advantage of such an approach is that you tried every possible point within a pre-defined grid. A significant downside, however, is that it takes ages to complete. Especially if you have no clue where your possible β values are going to be, and/or if you want a high resolution estimate (smaller step sizes between your points), you would have to try a very large number of combinations. In addition, if you want to add additional predictors, your grid will grow exponentially.

Fortunately, there are *minimisation algorithms*. These will walk through parameter space in a clever way. Most work by randomly starting at one particular point, computing the residual squares for that particular set of β values, and then they try a nearby point to compute the residual squares again. By using the slope between these points, the algorithm knows where to go: Because the best fitting solution is at the point with the lowest residual square, the algorithm simply has to follow the slope downwards until it reaches a point where it can no longer go down any further. This is the best fit!

Let's try one of these minimisation algorithms. It needs a function to minimise the value for (i.e. a function that computes the residual squares):

```
In [ ]: # Import the minimize function from SciPy.
        from scipy.optimize import minimize

        # Define a function to compute the sum of residual
        # squares based on our model.
        def residuals(betas, x, y):
            # Compute the predicted value of y.
            y_pred = betas[0] + betas[1] * x
            # Compute the residuals.
            res = y - y_pred
            # Compute the sum of squared residuals.
            s = numpy.sum(res**2)
            # Return the squared residuals.
            return s

        # Choose values that the algorithm uses as an initial guess.
        initial_guess = (0.0, 0.0)
        # Use the mimize function to compute the best fit.
        model = minimize(residuals, initial_guess, \
            args=(interval_d, time_500m_d), method="L-BFGS-B")

        # Report the betas.
        betas = model.x
        print("Best fit: b0={}, b1={}".format(round(betas[0], ndigits=2), \
            round(betas[1], ndigits=2)))
```

This is the same result as the full space estimate provided earlier. However, this method is **much** faster.

You could ask what the benefit of using `minimize` is over simply using `linregress`. Both functions allowed us to fit our data, both gave us the same answer, and both were very quick about it too. The neat thing about using the `minimize` approach is that it is very flexible. You could have used *any* model, regardless of how many predictors you would have liked. You could even have used a non-linear model. Or a different way of computing the "best fit", for example one that doesn't rely on residuals.

1.3.2 How good is a fit?

The one thing we didn't do yet, is computing how good a fit really is. The usual measure for this is the *coefficient of determination*, or R^2. This is computed by dividing the sum of squares of the residuals by the total sum of squares:

$R^2 = 1 - \frac{SS_{res}}{SS_{total}}$

Or, more scary-looking, but also more helpful:

$R^2 = 1 - \frac{\Sigma_{i=1}^{n}(y_i - f_i)^2}{\Sigma_{i=1}^{n}(y_i - \bar{y})^2}$

Where n is the number of observations, $\bar{y}$ is the mean of y, and f_i is the predicted value of y_i given the model. For example:

$f_i = \beta_0 + \beta_1 x_i$

Or, in code:

```
In [ ]: # Compute the predicted y values based on the fitted betas.
        y_pred = betas[0] + betas[1] * interval_d

        # Compute the residual sum of squares.
        ss_res = numpy.sum((time_500m_d - y_pred)**2)

        # Compute the total sum of squares.
        ss_tot = numpy.sum((time_500m_d - numpy.mean(time_500m_d))**2)

        # Compute R square.
        r_sq = 1.0 - (ss_res / ss_tot)

        print("R squared = {}".format(round(r_sq, ndigits=2)))
```

If R^2 is 1, all variance in outcome y is predicted by predictor x. If R^2 is 0, none of the variance in outcome y is explained by predictor x. Here, the ready-start interval differences can explain 12% of the variance in finish time difference.

In the case of high-level sporting events, the amount of variance explained by anything other than athlete's ability should ideally be 0%. Here, a random variation in pre-start time that is introduced by the person holding the starting pistol was 12%. That's probably not ideal.

1.4 Multi-variable regression

Now that you know how to do a regression with a single predictor, we can turn to regressions with multiple predictors. The general format of multi-variable regression looks very similar to the single-variable version:

$y = \beta_0 + x_1\beta_1 + ... + x_n\beta_n + \epsilon$

Where n is the number of variables you might have. For example, the equation for three predictors would look like this:

$y = \beta_0 + x_1\beta_1 + x_2\beta_2 + x_3\beta_3 + \epsilon$

Here, x_1, x_2, and x_3 are three different predictors. β_1, β_2, and β_3 indicate the magnitude and direction of the effect of each predictor on y. As before, ϵ captures the "noise": all variance in y that we could not explain using the predictors.

1.4.1 Another example dataset

Last session, we looked at a dataset that contained the number of minutes each participant listened to Taylor Swift, and their happiness ratings. We collected similar data, but now included a measure of IQ too. These data can be found in the attached file `taytay_revisited.csv`.

We can load the dataset using NumPy's `loadtxt` function:

```
In [ ]: import numpy
        from matplotlib import pyplot

        # Load the data.
        data = numpy.loadtxt("taytay_revisited.csv", dtype=float, \
            delimiter=",", skiprows=1, unpack=True)
```

```
# Create some easy variable names to point to the data.
tay_minutes = data[0,:]
happy = data[1,:]
iq = data[2,:]

# Use the two predictors together into a single variable.
predictors = numpy.vstack([iq, happy])
```

Our objective is to predict the number of minutes someone listens to Taylor Swift by using their IQ score and happiness rating. Or, in an equation:

$Swifting = \beta_0 + happiness * \beta_1 + IQ * \beta_2 + \epsilon$

Let's write a function to model this:

```
In [ ]: def residuals(betas, x, y):
            # Compute the predicted y values.
            y_pred = betas[0] + betas[1] * x[0,:] + \
                betas[2] * x[1,:]
            # Compute the residuals.
            res = y - y_pred
            # Compute the sum of squared residuals.
            s = numpy.sum(res**2)
            # Return the SSres
            return s
```

Now we can use SciPy's `minimize` function to fit out model to the data:

```
In [ ]: from scipy.optimize import minimize

        # Set an initial guess for the betas.
        initial_guess = [0.0, 0.0, 0.0]

        # Fit the model.
        model = minimize(residuals, initial_guess, \
            args=(predictors, tay_minutes), method="L-BFGS-B")

        # Report the betas.
        betas = model.x
        print("Best fit: b0={}, b1={}, b2={}".format( \
            round(betas[0], ndigits=2), round(betas[1], ndigits=2), \
            round(betas[2], ndigits=2)))
```

OK! So the best fit is the following:

$Swifting = 34.19 + happiness * 1.88 + IQ * 0.16 + \epsilon$

Our β values are 1.88 for happiness and 0.16 for IQ. Does that mean IQ is less important than happiness for determining Taylor Swift listening? It doesn't necessarily, because we're currently looking at *unstandardised coefficients*. The values for IQ are larger than the values for happiness: IQ, per definition, has a mean of 100 and a standard deviation of 15, whereas happiness was rated on a 0-10 scale. This difference in range alters the magnitudes of β values.

In order to directly compare the parameters, we'll need the *standardised coefficients*. You can compute those by simply z-scoring predictors (and outcomes!) **before** running your regression:

```
In [ ]: from scipy.stats import zscore
        from scipy.optimize import minimize

        # Use the two predictors together into a single variable.
        z_predictors = numpy.vstack([zscore(iq), zscore(happy)])
        z_minutes = zscore(tay_minutes)

        # Set an initial guess for the betas.
        initial_guess = [1.0, 1.0, 1.0]

        # Fit the model.
        stand_model = minimize(residuals, initial_guess, \
            args=(z_predictors, z_minutes), method="L-BFGS-B")

        # Report the betas.
        stand_betas = stand_model.x
        print("Best fit: b0={}, b1={}, b2={}".format( \
            round(stand_betas[0], ndigits=2), \
            round(stand_betas[1], ndigits=2), \
            round(stand_betas[2], ndigits=2)))
```

From this, we can really tell that happiness has a bigger effect on Taylor Swift listening than IQ does.

Let's compute how much of the variance we can explain with the current fit:

```
In [ ]: # Compute the predicted y values based on the fitted betas.
        y_pred = betas[0] + betas[1] * iq + betas[2] * happy

        # Compute the residual sum of squares.
        ss_res = numpy.sum((tay_minutes - y_pred)**2)

        # Compute the total sum of squares.
        ss_tot = numpy.sum((tay_minutes - numpy.mean(tay_minutes))**2)

        # Compute R square.
        r_sq = 1.0 - (ss_res / ss_tot)

        print("R squared = {}".format(round(r_sq, ndigits=2)))
```

An R^2 of 0.11 is pretty decent!

Or is it? We don't really know what it means in context, or whether it's statistically significant. Let's shelf that thought for a second, as we will return to it later.

1.4.2 Step-wise regression

Up until now, you have been using all predictors at the same time, without regard for whether they interact or not. One alternative to this, is to use *step-wise regression*. Here, you use one predictor at a time, and continue with the residuals after each predictor. This allows you to compute the relationship between two variables after accounting for another.

For example, let's regress out the effect of IQ on Taylor Swift listening first:

```
In [ ]: import numpy
        from matplotlib import pyplot
        from scipy.stats import linregress

        # Load the data.
        data = numpy.loadtxt("taytay_revisited.csv", dtype=float, \
            delimiter=",", skiprows=1, unpack=True)

        # Create some easy variable names to point to the data.
        tay_minutes = data[0,:]
        happy = data[1,:]
        iq = data[2,:]

        # Compute the slope and intercept for the regression of IQ
        # on Taylor Swift minutes.
        slope, intercept, r, p, std_err = linregress(iq, tay_minutes)

        print("IQ: R squared={}, p={}".format( \
            round(r**2, ndigits=2), round(p, ndigits=3)))

        # Predict tay_minutes based on the regression.
        y_pred_iq = intercept + slope * iq

        # Compute the residuals.
        residual_minutes = tay_minutes - y_pred_iq
```

We now have the residual minutes of Taylor Swift listening after accounting for the linear effect of IQ. Let's see if happiness predicts this residual score:

```
In [ ]: # Compute the slope and intercept for the regression
        # of IQ on Taylor Swift minutes.
        slope, intercept, r, p, std_err = \
            linregress(happy, residual_minutes)

        print("Happy: R squared={}, p={}".format(\
            round(r**2, ndigits=2), round(p, ndigits=3)))

        # Predict tay_minutes based on the regression.
        y_pred = intercept + slope * happy

        # Compute the residuals.
        residuals = residual_minutes - y_pred
```

The effect is roughly equal to what it was before (which you would only know if you had computed each individual variable's explained variance, btw). Also note that no additional variance is explained in a stepwise compared to the earlier multivariable regression.

In this dataset, IQ and happiness don't share much (or any) variance. When predictors are correlated between each other, regressions become harder to do.

Stepwise regression can be useful when you have a list of predictors, ordered from high to low priority according to your own theory. It can also be useful if you're aiming to regress out the effect of one predictor to assess whether a second predictor can predict any unique variance beyond the first predictor. This is particularly important when those two predictors are correlated.

One example to illustrate the above: Let's say you're interested in the effect of potential reward on saccadic peak velocity. Or: Do people make faster eye movements when you offer them a higher reward for doing so? We know that saccadic amplitude is strongly related to peak velocity. Thus, people could (implicitly) have learned to make larger eye movements to increase their peak velocity. Hence, if you care about the effect of reward on peak velocity, you will need to regress out the effect of saccadic amplitude first. If reward can predict some of the variance in the residuals after accounting for saccadic amplitude, it means that reward had a unique effect on peak velocity.

2 Model fitting

So we just learned about linear regression and least-squares estimation. We can extend this topic is extended into more general *model fitting*. We will be thinking about scientific hypotheses as *models*, which are precise descriptions of relationships between variables. For example, according to Newton's second law of motion, the relationship between an object's force, mass, and acceleration is given by the equation $F = ma$

Models do not have to be linear, but can take all shapes and sizes. The examples we'll look at today will include an exponential function, and a probability density function that is a mixture of typical distributions.

In Experimental Psychology, entirely descriptive box-and-arrow drawings are often mistaken for models. Although they might be useful for describing phenomena, it can be argued that they lack predictive power. Here, we'll go by a more narrow definition, and only use the word *model* for hypotheses that are testable and precise.

2.1 Fitting a non-linear model with one free parameter

The first example today is in the field of short-term memory. Storing visual information in short-term memory requires that stimuli in the environment are processed into an internal representation, and that this representation is maintained temporarily (typically for up to a few seconds). In the past, researchers have wondered whether human's faced storage capacity limits for visual information. In addition, they wondered whether there are limitations on the processing capacity. In other words, researchers wanted to know whether there was a maximum limit on the amount of information a person can have in short-term memory, and whether there is a maximum bandwith available for encoding information into short-term memory.

One experiment used in the 1980s and 1990s is the *whole report* paradigm. In this type of experiment, participants are presented with a number of visual stimuli. For example, they could be presented with 8 different letters. The researcher would control how long the letters were visible before they were masked (replaced by a neutral stimulus). In this way, the researcher could control exactly how long participants could process the presented stimuli.

What is measured in whole report experiments is how many of the stimuli participants could remember. At lower exposure durations, one would expect fewer stimuli to be processed, especially if humans have a limited processing capacity. In addition, if human short-term memory is

of a limited storage capacity, one expects that subjects would struggle to recall all stimuli even under very long exposure durations.

More precisely, one would expect that the number of recalled stimuli rises as a function of the exposure duration (the slope determined by processing capacity), but that this rise is limited by an asymptote (storage capacity).

Let's define the things we just said a bit more precisely. If the processing capacity (let's call it C) is limited, it should be divided among the number of stimuli that a participant is trying to recall (let's call that number T). So our capacity per stimulus is:

$\frac{C}{T}$

To find how much of a stimulus was encoded, we need to multiply the processing capacity by the time that a participant was processing a stimulus (let's call that τ):

$\frac{C\tau}{T}$

This can be used to describe the *probability of a stimulus finishing processing* (let's call that s). The following equation does just that:

$s = 1 - \exp\left(\frac{-C\tau}{T}\right)$

Note that, if there was no limit on processing capacity, we wouldn't have to divide the total capacity by the number of stimuli. Which would look like this:

$s = 1 - \exp\left(-C\tau\right)$

For now, let's assume there were fewer stimuli than participants could remember. In that case, the storage capacity wouldn't matter. The number of recalled stimuli (let's call it $E(score)$) could thus be computed by simply multiplying the probability that any one stimulus finishes encoding (s) by the number of stimuli (T):

$E(score) = Ts$

OK, let's recap: We have two models. In the first model, the limited processing capacity has to be divided over all stimuli. In the second model, the processing capacity does not have to be divided at all. The equations for these two models look like this:

$E(score)_1 = T(1 - \exp\left(\frac{-C\tau}{T}\right))$

$E(score)_2 = T(1 - \exp\left(-C\tau\right))$

Note that we know T, as this is the number of stimuli in our experiment. We also know τ, as this is the time participants were given to encode the stimuli.

In an experiment, participants were always shown 3 stimuli. These stimuli were letters that participants were asked to type in after a brief delay. In each trial, stimuli were presented with a different exposure duration (time between the onset of the stimuli and the onset of the mask). These exposure durations could be 10, 20, 40, 80, 160, 320 milliseconds.

The average number of stimuli that each participant could recall after each exposure duration has already been computed for you. You can find these data in the attached file `whole_report.csv`. Load it using NumPy's `loadtxt` function:

```
In [ ]: import numpy

        # Load the data from the data file.
        data = numpy.loadtxt("whole_report.csv", dtype=float, \
            delimiter=",", skiprows=1, unpack=True)

        # Create a variable that codes the number of
        # stimuli used in each trial of this experiment.
        n_stimuli = 3
```

```
        # Create a vector with all exposure durations from
        # the experiment.
        expdur = numpy.array([0.01,0.02,0.04,0.08,0.16,0.32], \
            dtype=float)

        # Convenience renaming of columns in the data file.
        # We won't actually use this, it's more to show you
        # what is in the file if you didn't open it outside
        # of this workbook.
        score_10ms =  data[0,:]
        score_20ms =  data[1,:]
        score_40ms =  data[2,:]
        score_80ms =  data[3,:]
        score_160ms = data[4,:]
        score_320ms = data[5,:]
```

Now that we have the data, let's visualise it to have a look:

```
In [ ]: from matplotlib import pyplot

        # Count the number of participants.
        n_participants = data.shape[1]

        # Loop through all participants, and plot the data for each.
        for i in range(n_participants):
            # Plot the exposure duration on the x-axis, and the
            # average score for the current participant on the
            # y-axis.
            pyplot.plot(expdur, data[:,i], alpha=0.2)

        # Add axis labels to the plot.
        pyplot.xlabel("Exposure duration (seconds)", fontsize=16)
        pyplot.ylabel("Average number of recalled stimuli", fontsize=16)
```

Having all participants in one plot is a bit messy. Let's compute the average, and plot that instead.

```
In [ ]: # Compute the average score across all participants.
        m = numpy.mean(data, axis=1)
        # Compute the standard deviation.
        sd = numpy.std(data, axis=1, ddof=1)
        # Compute the standard error of the mean.
        sem = sd / numpy.sqrt(n_participants)

        # Plot the average and the standard error of the mean.
        pyplot.plot(expdur, m, color="#FF69B4")
        pyplot.fill_between(expdur, m-sem, m+sem, \
            color="#FF69B4", alpha=0.3)
```

```
        # Add axis labels to the plot.
        pyplot.xlabel("Exposure duration (seconds)", fontsize=16)
        pyplot.ylabel("Average number of recalled stimuli", fontsize=16)
```

Now that you have had a look at the data, it's time to start fitting the models. You would typically do this for every participant, but for now let's practice on the mean instead.

Fitting models to data works in much the same way as linear regression. First, you define a function to compute the residuals for each model:

```
In [ ]: # Model 1 was for a limited processing capacity.
        def model_1(parameters, T, x, y):
            # The predicted y values based on model 1:
            y_pred = T * (1 - numpy.exp((- parameters[0] * x) / T))
            return y_pred

        # Model 2 was for an unlimited processing capacity.
        def model_2(parameters, T, x, y):
            # The predicted y values based on model 2:
            y_pred = T * (1 - numpy.exp(- parameters[0] * x))
            return y_pred

        def residuals(parameters, T, x, y, model_nr):
            # Compute the predicted y values based on a model.
            if model_nr == 1:
                y_pred = model_1(parameters, T, x, y)
            elif model_nr == 2:
                y_pred = model_2(parameters, T, x, y)
            # Compute the difference between the measured outcome
            # and the predicted outcome (the residuals).
            res = y - y_pred
            # Compute the sum of squared residuals.
            s = numpy.sum(res**2)
            # Return the squared residuals.
            return s
```

Now that you've defined the models, you can find the best fitting parameters using a minimisation algorithm:

```
In [ ]: from scipy.optimize import minimize

        # Go through all models.
        for i, model in enumerate([model_1, model_2]):
            # Fit the current model.
            outcome = minimize(residuals, [0], \
                args=(n_stimuli, expdur, m, i+1), \
                method="L-BFGS-B", bounds=[(0,None)])
            # Get the best parameters for the best fit.
            C = outcome.x[0]
            # Compute the residual sum of squares.
```

```
            ss_res = residuals(outcome.x, n_stimuli, \
                expdur, m, i+1)

            print("Model {}: C={}, SSres={}".format( \
                i+1, round(C, ndigits=2), round(ss_res, ndigits=2)))

            # Compute the predicted outcome values.
            y_pred = model(outcome.x, n_stimuli, expdur, m)
            # Plot the predicted outcome.
            pyplot.plot(expdur, y_pred, '--', label="model %d" % (i+1))

        # Plot the average.
        pyplot.plot(expdur, m, 'o', color="#000000")
        # Add axis labels to the plot.
        pyplot.xlabel("Exposure duration (seconds)", fontsize=16)
        pyplot.ylabel("Average number of recalled stimuli", fontsize=16)
```

From this plot, which model would you say fits best?

Why, from a mathematical perspective, do you think the two models fit equally well? (Think about the relationship between parameters.)

2.2 Fitting a model with two free parameters

Did you notice how at exposure durations of 10 milliseconds, participants tend not to recall any stimuli? Perhaps our assumption that participants can use the full exposure duration to process information does not hold true. Maybe participants need a bit of initial time to visually process the stimuli before they can start encoding them into short-term memory?

This is an interesting suggestion, and one that can be written down as an equation. Recall our first model? It assumed that the processing time τ was the entire exposure duration (let's call that t).

$E(score)_1 = T(1 - \exp(\frac{-C\tau}{T}))$

where $\tau = t$

But perhaps there is a *minimally effective exposure duration*, i.e. a brief period during which no encoding into short-term memory occurs yet. Let's call that t_0. This value needs to be subtracted from the total exposure duration t:

$\tau = t - t_0$

This means that our first model could be written like this:

$E(score)_1 = T(1 - \exp(\frac{-Ct}{T}))$

And a version that does incorporate the minimally effective exposure duration could be written like this:

$E(score)_3 = T(1 - \exp(\frac{-C(t-t_0)}{T}))$

We can also incorporate this in the models that did not have a limited processing capacity:

$E(score)_2 = T(1 - \exp(-Ct))$

$E(score)_4 = T(1 - \exp(-C(t - t_0)))$

We should create functions for these new models too:

```
In [ ]: # Model 1 was for a limited processing capacity.
        def model_3(parameters, T, x, y):
```

```
        # The predicted y values based on model 3:
        y_pred = T * (1 - numpy.exp((- parameters[0] * \
            (x-parameters[1])) / T))
        return y_pred

    # Model 2 was for an unlimited processing capacity.
    def model_4(parameters, T, x, y):
        # The predicted y values based on model 4:
        y_pred = T * (1 - numpy.exp(- parameters[0] * \
            (x-parameters[1])))
        return y_pred

    def residuals(parameters, T, x, y, model_nr):
        # Compute the predicted y values based on a model.
        if model_nr == 1:
            y_pred = model_1(parameters, T, x, y)
        elif model_nr == 2:
            y_pred = model_2(parameters, T, x, y)
        elif model_nr == 3:
            y_pred = model_3(parameters, T, x, y)
        elif model_nr == 4:
            y_pred = model_4(parameters, T, x, y)
        # Compute the difference between the measured outcome
        # and the predicted outcome (the residuals).
        res = y - y_pred
        # Compute the sum of squared residuals.
        s = numpy.sum(res**2)
        # Return the squared residuals.
        return s
```

Now let's fit ALL the models!

```
In [ ]: from scipy.optimize import minimize

    # Go through all models.
    for i, model in enumerate([model_1, model_2, model_3, model_4]):
        # Fit the current model.
        outcome = minimize(residuals, [0, 0], \
            args=(n_stimuli, expdur, m, i+1), \
            method="L-BFGS-B", bounds=[(0,None), (0,None)])
        # Get the best parameters for the best fit.
        C = outcome.x[0]
        t0 = outcome.x[1]
        # Compute the residual sum of squares.
        ss_res = residuals(outcome.x, n_stimuli, \
            expdur, m, i+1)

        print("Model {}: C={}, SSres={}".format( \
```

```
            i+1, round(C, ndigits=2), round(ss_res, ndigits=2)))

        # Compute the predicted outcome values.
        y_pred = model(outcome.x, n_stimuli, expdur, m)
        # Plot the predicted outcome.
        pyplot.plot(expdur, y_pred, '--', label="model %d" % (i+1))

    # Plot the average.
    pyplot.plot(expdur, m, 'o', color="#000000")
    # Add axis labels to the plot.
    pyplot.xlabel("Exposure duration (seconds)", fontsize=16)
    pyplot.ylabel("Average number of recalled stimuli", fontsize=16)
```

Which of our models do you think fits best now?

Mind you, this is still a very narrow model: It just describes behaviour in a single task (although it was derived within a wider computational framework). In addition, there is a way to incorporate a storage capacity for short-term memory (which we have ignored here). For more info, you can read the following paper:

- Budesen, C. (1990). A theory of visual attention. *Psychological Review*, *97*(4), 523-547. doi:10.1037/0033-295X.97.4.523

2.3 Model comparison

One way to compare the models introduced above, is by measuring how much variance each explains. The logic behind this is that a better model should explain more variance. You could compute the coefficient of determination for each model, using the same equation that you used in linear regression last week:

$R^2 = 1 - \frac{\Sigma_{i=1}^{n}(y_i - f_i)^2}{\Sigma_{i=1}^{n}(y_i - \bar{y})^2}$

Where y is the measured outcome (correctly recalled number of stimuli) for each participant i, and f is the predicted number of stimuli for each participant.

Let's run the numbers:

```
In [ ]: # Go through all models.
        for i, model in enumerate([model_1, model_2, model_3, model_4]):
            # Fit the current model.
            outcome = minimize(residuals, [0,0], \
                args=(n_stimuli, expdur, m, i+1), \
                method="L-BFGS-B")
            # Get the best parameters for the best fit.
            C = outcome.x[0]
            # Compute the residual sum of squares.
            ss_res = residuals(outcome.x, n_stimuli, \
                expdur, m, i+1)
            # Compute the total sum of squares.
            ss_tot = numpy.sum((m - numpy.mean(m))**2)
            # Compute the coefficient of determination.
            r_sq = 1.0 - (ss_res / ss_tot)
```

```
        print("Model {}: R squared = {}".format( \
            i+1, round(r_sq, ndigits=3)))
```

2.3.1 Introducing the Bayesian Information Criterion

One downside of using R^2 is that it doesn't distinguish between models that have very few free parameters, and models that have very many. As a rule, the more free parameters in a model, the easier it is to fit it to some data. This is not a feature of how good your model is, but rather of how flexible it is.

To counteract this, we can turn to ways in which we can quantify the *goodness of fit*. One of these metrics is the *adjusted R squared*, which is computed from the R^2, the number of observations (n), and the number of free parameters (k):

$R^2_{adjusted} = 1 - \frac{(1-R^2)(n-1)}{n-k-1}$

Another measure for the goodness of fit is the *Bayesian Information Criterion* (BIC). One way of computing the BIC directly uses the residuals:

$BIC = n + n\ln(2\pi) + n\ln(\frac{SS_{res}}{n}) + \ln(n)(k+1)$

where n is the number of observations (in this case: number of exposure durations), and k is the number free parameters in a model

Both of these metrics do not only quantify how good a fit is, but also punish models as a function of how many parameters they require.

Let's use the BIC to compare our models one last time:

```
In [ ]: # Go through all models.
        for i, model in enumerate([model_1, model_2, model_3, model_4]):
            # Fit the current model.
            outcome = minimize(residuals, [0,0], \
                args=(n_stimuli, expdur, m, i+1), \
                method="L-BFGS-B")
            # Get the best parameters for the best fit.
            C = outcome.x[0]
            # Compute the residual sum of squares.
            ss_res = residuals(outcome.x, n_stimuli, \
                expdur, m, i+1)
            # Compute the total sum of squares.
            ss_tot = numpy.sum((m - numpy.mean(m))**2)
            # Compute the coefficient of determination.
            r_sq = 1.0 - (ss_res / ss_tot)

            # Count the number of observations.
            n = len(expdur)
            # Set the number of free parameters.
            if i in [0,1]:
                # The first two models have one free
                # parameter: C
                k = 1
            elif i in [2,3]:
                # The second two models have two free
```

```
            # parameters: C and t0
            k = 2
        # Compute the Bayesian Information Criterion.
        bic = n + n*numpy.log(2*numpy.pi) + \
            n*numpy.log(ss_res/n) + \
            numpy.log(n)*(k+1)

        print("Model {}: BIC = {}".format( \
            i+1, round(bic, ndigits=2)))
```

Choosing which model fits best is done by comparison of BICs. By convention, you choose the lowest BIC. This is the best fitting model. You then compute the differences between that BIC and the other models' BICs. The resulting values, ΔBIC, are usually interpreted using the guidelines of Raftery (1995), who considers ΔBIC values of 2-6 positive evidence in favour of the best fitting model, values of 6-10 as strong evidence, and values over 10 as very stong evidence.

Note that normally, you would fit a model to each participant, not to the average of the whole sample. You can then compute a BIC for each participant. The sum across all participants of the BIC values for a model is them compared against the other models' sums.

Such a procedure would look like this in code:

```
In [ ]: # Go through models 1 and 3.
        for i in [0,2]:
            # Get the model function for this model.
            if i == 0:
                model = model_1
            elif i == 2:
                model = model_3

            # Count the number of participants.
            n_participants = data.shape[1]

            # Maintain a list of BIC values for
            # all participants.
            bic = numpy.zeros(n_participants, dtype=float)

            # Go through all participants.
            for j in range(n_participants):

                # Fit the current model.
                outcome = minimize(residuals, [0,0], \
                    args=(n_stimuli, expdur, m, i+1), \
                    method="L-BFGS-B")

                # Get the best parameters for the best fit.
                C = outcome.x[0]
                t0 = outcome.x[1]

                # Compute the residual sum of squares.
```

```
        ss_res = residuals(outcome.x, n_stimuli, \
            expdur, m, i+1)
        # Compute the total sum of squares.
        ss_tot = numpy.sum((m - numpy.mean(m))**2)
        # Compute the coefficient of determination.
        r_sq = 1.0 - (ss_res / ss_tot)

        # Count the number of observations.
        n = len(expdur)
        # Set the number of free parameters.
        if i in [0,1]:
            # The first two models have one free
            # parameter: C
            k = 1
        elif i in [2,3]:
            # The second two models have two free
            # parameters: C and t0
            k = 2
        # Compute the Bayesian Information Criterion.
        bic[j] = n + n*numpy.log(2*numpy.pi) + \
            n*numpy.log(ss_res/n) + \
            numpy.log(n)*(k+1)

    print("Model {}: BIC sum = {}".format( \
        i+1, round(numpy.sum(bic), ndigits=3)))
```