Simulation for Simple Regression Models

Read in the data

We will use the *homework-education-gpa.csv* data.
The `nrow()` function counts the number of rows.

```
> jun_high = read.csv("~/homework-education-gpa.csv")


> head(jun_high)

  gpa parentEd homework
1  78       13        2
2  79       14        6
3  79       13        1
4  89       13        5
5  82       16        3
6  77       13        4


> nrow(jun_high)

[1] 100
```

Obtain Regression Coefficients

Use matrix algebra to obtain the regression coefficients

```
# Vector of the outcome
> y = jun_high$gpa


# Predictor (X) matrix
> x = matrix(c(rep(1, 100), jun_high$homework), ncol = 2)


# Solve for b
> b = solve(t(x) %*% x) %*% t(x) %*% y
> b

          [,1]
[1,] 74.289677
[2,]  1.214209
```

Prediction

To predict from the model, we substitute an *X*-value into the prediction equation. For example, say we wanted to predict the GPA for a student that spends 6 hours per week on homework.

$$\begin{aligned}\hat{\text{GPA}} &= 74.3 + 1.2(6) \\ &= 81.5\end{aligned}$$

Remember **y-hats are averages**, so this is the predicted average GPA for all students that spend 6 hours per week on homework.

```
# Predict
> 74.289677 + 1.214209 * 6

[1] 81.57493


# Can also index the estimated values from b directly
> b[1] + b[2] * 6

[1] 81.57493
```

The underlying data generating process (DGP) in OLS regression is:

$$Y_i = \beta_0 + \beta_1(X_i) + \epsilon_i$$

where β_0 and β_1 are fixed and

$$\epsilon_i \sim \mathcal{N}\left(0, \sigma_\epsilon\right)$$

Our best guesses for the β_0 and β_1 parameters are the estimated coefficients from the regression model. Similarly, we can use the RMSE as an estimate for σ_ε.

There are two sources of uncertainty that arise from the model (assuming that *X* is measured perfectly). The first source of uncertainty is due to **individual differences**. The second source of uncertainty is due to **sampling error**. These are typically referred to as **prediction uncertainty** and **sampling uncertainty**.

Prediction Uncertainty

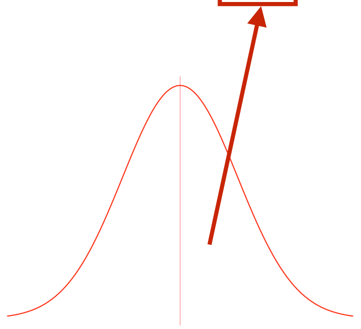
Prediction uncertainty arises because we know that, in practice, cases with the same X value have different Y values…there are individual differences. The goal of quantifying this uncertainty is to say how much individuals with the same X value differ in their Y values. The model is expressed as:

$$Y_i = \beta_0 + \beta_1(X_i) + \epsilon_i$$

where β_0, β_{1}, and σ_ε are all estimated from the data.

The prediction uncertainty for a given value of X, can be visually thought of as:

$$Y_i = \beta_0 + \beta_1(X_i) + \boxed{\epsilon_i}$$



Algorithm for Simulating Prediction Uncertainty

1. Obtain the beta estimates.

2. Use the beta estimates to produce a fitted value (y-hat) for a given value of *X*.

3. Obtain the estimate for RMSE.

4. Randomly generate an error value by sampling from a normal distribution with M = 0 and SD = RMSE.

5. Add the fitted value to the error to produce the individual's *Y* value.

Estimate RMSE

Use matrix algebra to obtain the RMSE

```
# Compute residuals
> residuals = y - (x %*% b)

# Compute SSE
> sse = t(residuals) %*% residuals
> sse

         [,1]
[1,] 5136.371

# Compute RMSE
> df = 98
> mse = sse / df
> rmse = sqrt(mse)
> rmse

        [,1]
[1,] 7.23961
```

Sample an Error Term

To **randomly generate an error term** we can use the estimated coefficients and the `rnorm()` function.

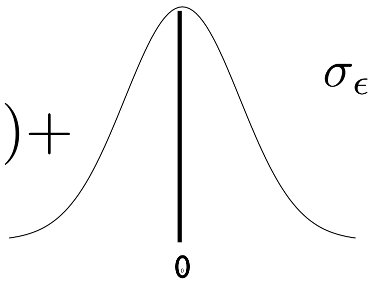
```
# Simulate from the DGP
> b[1] + b[2] * 6 + rnorm(n = 1, mean = 0, sd = rmse)

[1] 88.02932
```

Since the DGP incorporates a random element, the prediction will be different *every time you run the syntax*. The amount of variation in the predictions is referred to as **prediction uncertainty**.

Simulate Many Observations with Prediction Uncertainty

To obtain an estimate for the amount of prediction uncertainty (a quantification) we need to carry out many trials using the DGP. This is akin to generating many Y values for cases with the same X value. Then we can examine the variation in the randomly generated Y values.

$$Y_i = \beta_0 + \beta_1(X_i) +$$



$$\sigma_\epsilon \approx \text{RMSE}$$

Algorithm for Simulating Prediction Uncertainty

1. Obtain the beta estimates.
2. Use the beta estimates to produce a fitted value (y-hat) for a given value of X.
3. Obtain the estimate for RMSE.
4. Randomly generate an error value by sampling from a normal distribution with M = 0 and SD = RMSE.
5. Add the fitted value to the error to produce the individual's Y value.

Repeat Steps 4 and 5 many times.

There are many ways to have R carry out a set of computations several times. The method we will use is to (1) write a function that will make a prediction using the DGP, and (2) use the `replicate()` function to carry out this computation many times.

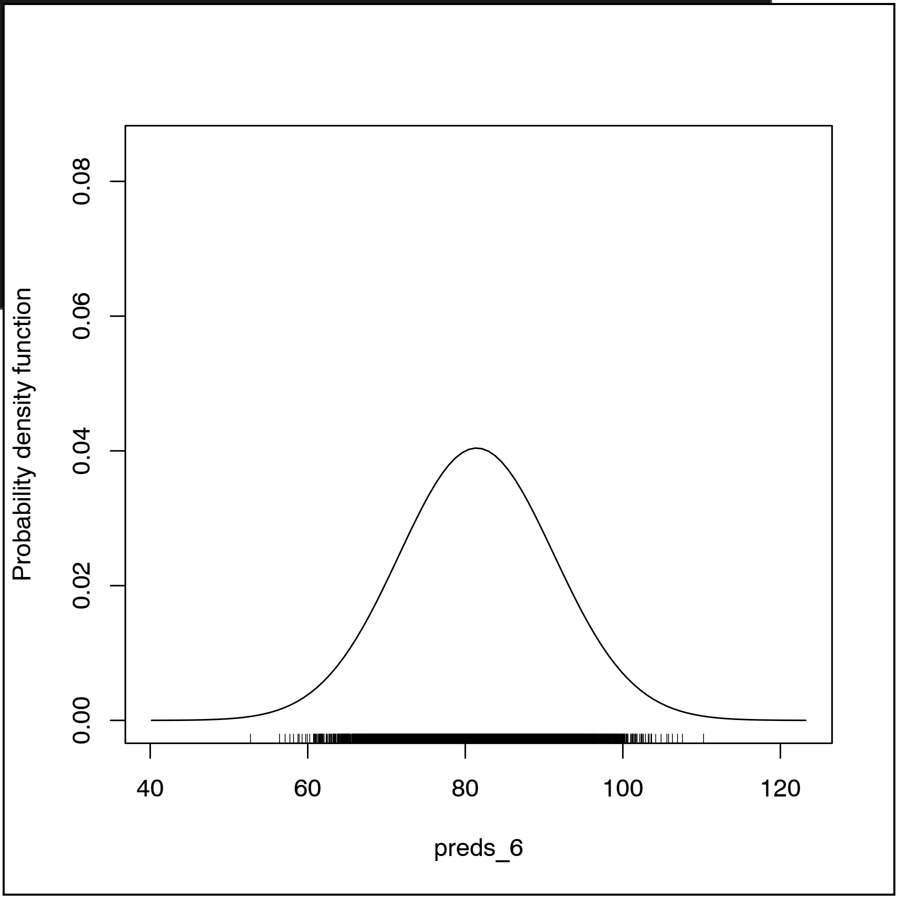
```
# Function to simulate from the DGP
> yhat = function(x){
    b[1] + b[2] * 6 + rnorm(n = 1, mean = 0, sd = rmse)
    }

# Use the function to predict for x = 25
> yhat(6)

[1] 78.00087
```

It is a good idea to first carry out a small number of trials to see that everything is working as you expect.

```
# Use replicate() to carry out a small number of trials of the simulation
> replicate(10, yhat(6))

[1] 85.60357 76.72832 83.08804 87.76909 82.92247 83.79610 87.01448
[8] 84.61612 76.62393 81.66562


# Carry out a large number of trials of the simulation; store them in an
object
> preds = replicate(1000, yhat(6))
> head(preds)

[1]  81.12418  78.99613  73.84564 100.62696  75.92292  90.95565
```

```
# Examine predictions
> sm.density(preds)


# Compute mean prediction
> mean(preds)

[1] 81.47567


# Compute prediction uncertainty
> sd(preds)

[1] 7.066781
```

The mean should be very close to our fixed estimate (where we did not add an error component). Examining the SD allows us to check that the simulation worked properly. It should be similar to the RMSE (after all that is how we specified the variation in the DGP!)



Interval Estimate of Prediction Uncertainty

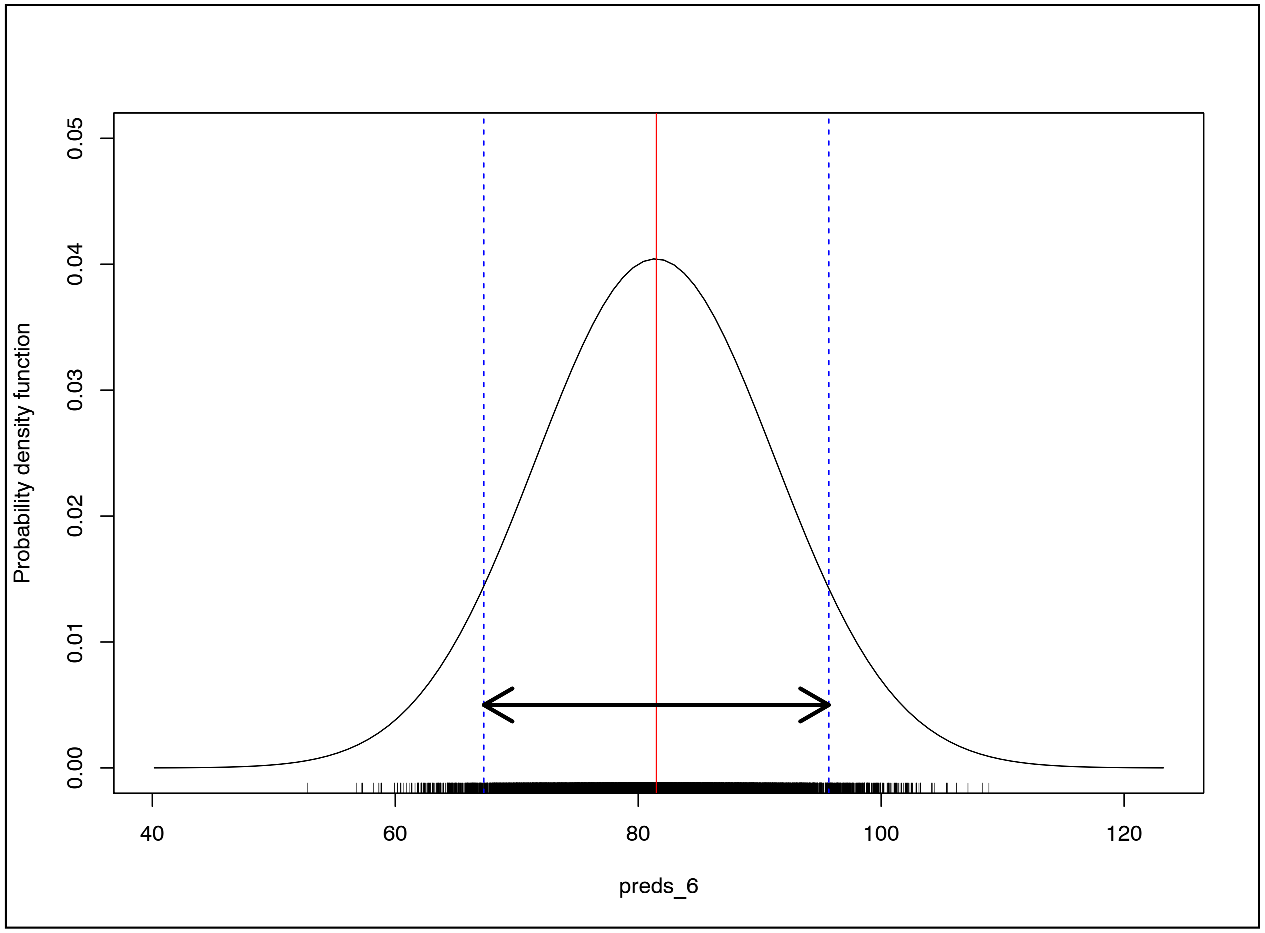
Interval Estimate Based on Simulated Mean and SD

We can use the mean and the SD from the simulation to compute an interval estimate of the **prediction uncertainty**.

$$81.5 \pm 2(7.1)$$

$$81.5 \pm 14.2$$

$$[67.3, 95.7]$$



Note that this range is **roughly the middle 95%** of the distribution of predictions.

Interval Estimate Using Percentile Interval Method

We can use the `quantile()` function to ask for the middle 95% of the distribution. We do this by finding the 2.5th-percentile and the 97.5th-percentile.

```
# Find the quantiles that demarcate the middle 95% of the distribution
> quantile(preds, props = c(0.025, 0.975))

    2.5%    97.5%
67.88128 95.59518
```

Interval Estimate Using Parametric Method

In 8251, we used the `predict()` function to estimate the **prediction uncertainty**. This is a parametric estimate of the interval.

Using the `predict()` function with the `interval="prediction"` argument produces the following output:

```
       fit      lwr     upr
1 81.57493 67.12236 96.0275
```

The difference from the intervals produced using simulation is negligible.

Comparison of Intervals

Using simulation and the formula:	[67.3, 95.7]
Percentile interval:	[67.9, 95.6]
Parametric method:	[67.1, 96.0]

The endpoints for all three the prediction intervals are quite similar. In practice these, depending on the data and degree to which assumptions of the model are met, the three intervals may be quite different.

For example, in the DGP for OLS regression, the residuals are assumed to have a normal distribution. When this is not the case, simulating from a normal distribution will be misleading, and will lead to a biased interval regardless of which method is used.

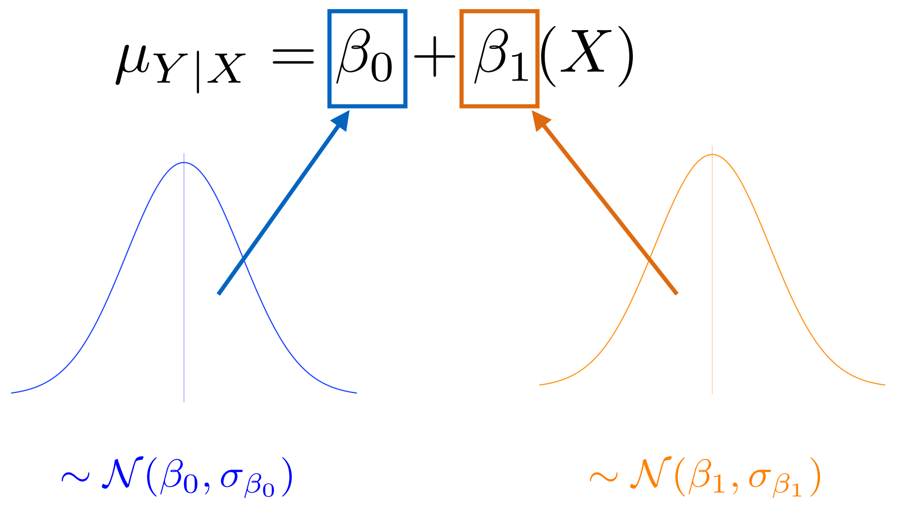
Sampling Uncertainty

Sampling uncertainty arises because we know that, in practice, different samples of data would produce different beta estimates. The goal of quantifying this uncertainty is to say how much the average value of Y varies because of this. It is important to note that since we are dealing with averages, we do not include the error term. The model is re-expressed using the conditional average rather than y-hats as:

$$\mu_{Y|X} = \beta_0 + \beta_1(X)$$

where β_0 and β_1 are estimated from the data.

The sampling uncertainty for a given value of X, can be visually thought of as:


$\mu_{Y|X} = \boxed{\beta_0} + \boxed{\beta_1}(X)$
$\sim \mathcal{N}(\beta_0, \sigma_{\beta_0})$
$\sim \mathcal{N}(\beta_1, \sigma_{\beta_1})$

We already have the beta estimates. We can compute the SEs for the beta estimates from the variance–covariance matrix for the coefficients.

```
# Variance-covariance matrix
> Vb = solve(t(x) %*% x)
> vc = as.numeric(rmse ^ 2) * Vb
> vc

           [,1]       [,2]
[1,]  3.7711939 -0.6379321
[2,] -0.6379321  0.1253305

# Compute SEs
> sqrt(diag(vc))

[1] 1.9419562 0.3540204
```

$$\hat{\beta}_0 = 74.28 \qquad \hat{\beta}_1 = 1.21$$

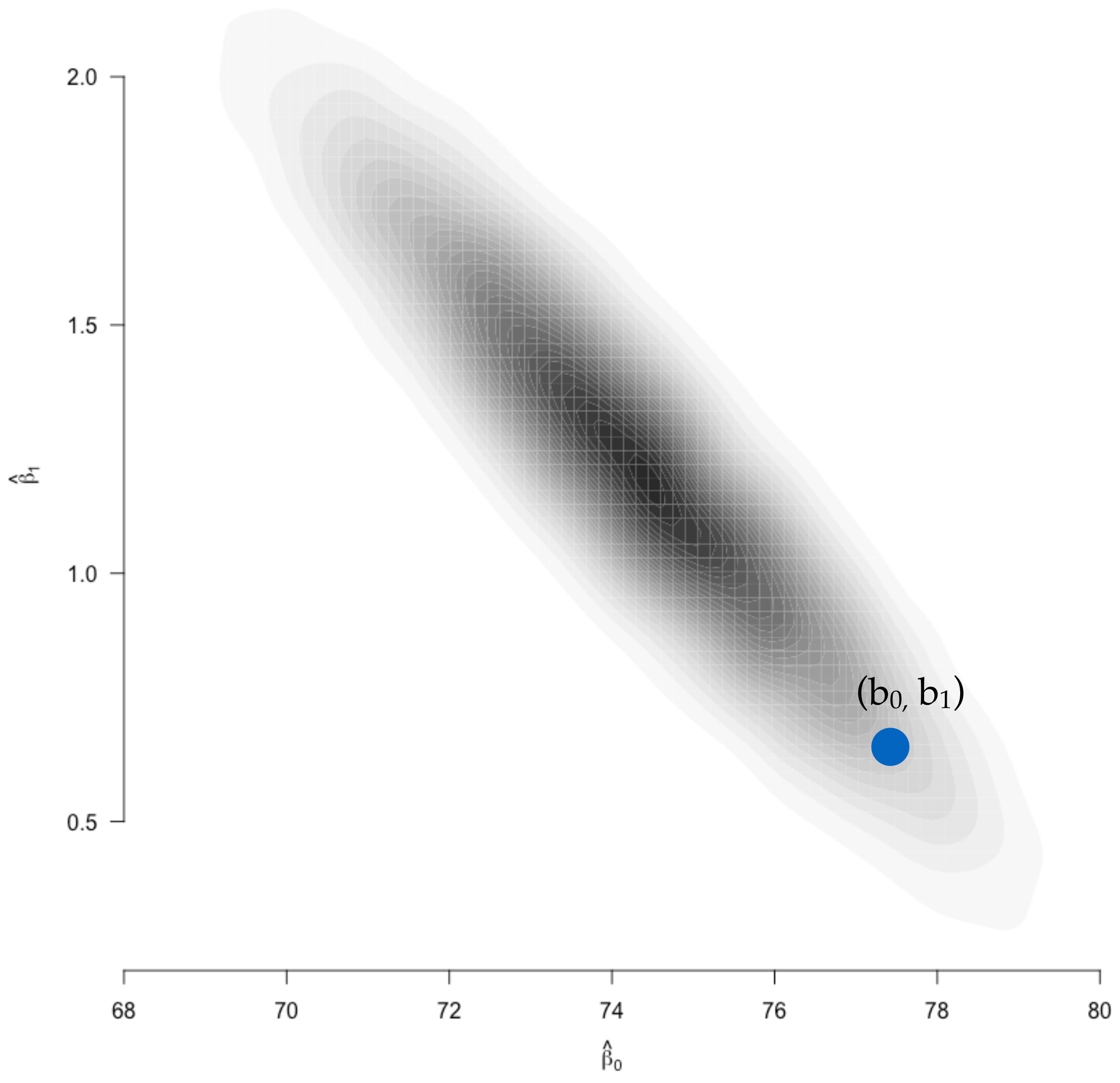
$$\text{SE}_{\hat{\beta}_0} = 1.94 \qquad \text{SE}_{\hat{\beta}_1} = 0.35$$

You might think, we can randomly sample from the distribution of coefficients as follows…

```
# Although this seems intuitive, it is INCORRECT
> b0 = rnorm(n = 1, mean = 74.289677, sd = 1.9419562)
> b1 = rnorm(n = 1, mean =  1.214209, sd = 0.3540204)

> b0 + b1 * 6

[1] 83.59767
```

The problem with this process is that the two coefficients are being sampled *independently*. This implies that the correlation between the coefficients is 0. While there are times when this is the case (e.g., when participants are randomly assigned to groups, and the coefficients represent group effects), typically regression coefficients are correlated with one another.

Here is a contour plot of the multivariate distribution of coefficients.



When we sample from this distribution, we draw a pair of coefficients (b_0, b_1), simultaneously.

This suggests, for example, that if you sample a pair that includes a high intercept the pair is likely to also have a lower slope.

We need to account for the correlation between b_0 and b_1 in our DGP.

Repeat Steps 3 and 4 many times.

Algorithm for Simulating Sampling Uncertainty

1. Obtain the beta estimates from the fitted model.

2. Obtain the variance–covariance matrix.

3. Randomly generate (new) beta estimates by sampling from the multivariate normal distribution having a mean vector of **b**, and a variance–covariance matrix $\sigma^2\mathbf{V}_\beta$.

4. Use the sampled beta estimates to produce a fitted value (y-hat) for a given value of *X*.

To sample from a multivariate normal probability distribution, we use the `mvrnorm()` function from the **MASS** library. This takes three arguments:

- The number of times you want to sample from the distribution;
- A vector that contains the estimated regression coefficients (means); and
- The variance–covariance matrix of the coefficients

```
# TWO THINGS WE NEED
# Vector of means (we earlier stored that in b)
# Variance-covariance matrix (we earlier stored that in vc)


# Sample from the multivariate normal distribution
> library(MASS)
> sampled_b = mvrnorm(1, b, vc)
> sampled_b

[1] 71.549635  1.759678


# Predict the average for HW=6 using the randomly generated coefficients
> sampled_b[1] + sampled_b[2] * 6

[1] 82.1077
```

Note that we do not use argument names in this function.

To carry this random generation out many times, we will put our computations in a function and then use `replicate()` to repeat carrying out the function.

```
# Function to simulate from the DGP
> yhat2 = function(x){
    sampled_b = mvrnorm(1, b, vc)
    sampled_b[1] + sampled_b[2] * 6
    }

# Use the function to predict for x = 6
> yhat2(6)

[1] 81.72071


# Use replicate() to carry out many trials
> preds2 = replicate(1000, yhat2(x = 6))


# 95% percentile interval
> quantile(preds2, probs = c(0.025, 0.975))

    2.5%    97.5%
79.97324 83.02916
```

Accounting for sampling uncertainty, it is likely that the average GPA for students who spend six hours on homework is between 80.0 and 83.1.

Using the `predict()` function with the `interval="confidence"` argument produces an interval that ranges from 80.00 to 83.14, a negligible difference from the interval produced using simulation.

Accounting for Both Sampling and Prediction Uncertainty

To account for both sampling and prediction uncertainty, we sample coefficients *and* errors.

```
# Sample an error from the appropriate normal distribution
> e = rnorm(n = 100, mean = 0, sd = rmse)

# Use the vectors to produce predicted values
> sampled_b[1] + sampled_b[2] * 6 + e

[1] 100.4446
```

Note: The `predict()` function does not allow us to account for both sampling *and* prediction uncertainty. To do so, we must use simulation.

To carry this random generation out many times, we will put our computations in a function and then use `replicate()` to repeat carrying out the function.

However, before we do that, there is **one thing we haven't yet accounted for** in the simulation.

In this simulation the error term (e_i) was sampled from a distribution where the RMSE was fixed (it did not vary)…it was always 7.239.

We might want to randomly generate this value as well. The RMSE also plays a role in determining the variance–covariance matrix of the coefficients (V_b * RMSE^2). So, if we randomly generate a RMSE value, we should also use that generated value in the computation of the variance–covariance matrix.

Simulate the RMSE

Algorithm for Completely Simulating Sampling and Prediction Variation

1. Obtain the beta estimates from the fitted model.

2. Obtain the variance–covariance matrix.

3. Obtain the RMSE estimate.

4. Randomly generate the RMSE. We can do this using the formula:

$$\sigma = \hat{\sigma}\sqrt{\frac{n-k}{X}}$$

where $\hat{\sigma}$ is the RMSE from the initial fitted model, and X is a random draw from a χ^2 distribution with n–k degrees of freedom, where n is the sample size and k is the number of coefficients (including the intercept) in the model.

5. Use the randomly generated RMSE to then randomly generate the beta matrix from the multivariate normal distribution having mean vector $\mathbf{b}$, and a variance–covariance matrix $\sigma^2\mathbf{V}_{\beta}$.

$\left.\right\}$ Repeat Steps 4 and 5 many times.

```
# Randomly generate the RMSE
> rmse = 7.23961 * sqrt((100 - 2) / rchisq(n = 1, df = 98))

# Compute the variance-covariance matrix using the randomly generated RMSE
> vc_sim = as.numeric(rmse ^ 2) * Vb

# Randomly generate betas
> sampled_b = mvrnorm(n = 1, b, vc_sim)

# Randomly generate an error
> e = rnorm(n = 1, mean = 0, sd = rmse)

# Use the randomly generated parameters to produce predicted values
# Since we only generated one set of coefficients we index a vector
# instead of a matrix
> sampled_b[1] + sampled_b[2] * 6 + e

[1] 93.70448
```

To carry out many trials, we will again put these computations in a function and then use the `replicate()` function.

```
# Function to randomly generate RMSE, betas, and error then make a
prediction from a given value of x
> sim_y = function(x){
    n = nrow(jun_high)
    k = 2
    rmse = 7.23961 * sqrt((n - k) / rchisq(n = 1, df = n - k))
    vc_sim = as.numeric(rmse ^ 2) * Vb
    sampled_b = mvrnorm(n = 1, b, vc_sim)
    e = rnorm(n = 1, mean = 0, sd = rmse)
    sampled_b[1] + sampled_b[2] * x + e
    }


# Use the function to predict for a student with HW = 6
> sim_y(x = 6)

[1] 92.7184
```

```
# Use replicate() to carry out many trials
> yhat3 = replicate(1000, sim_y(x = 6))


# 95% percentile interval
> quantile(yhat3, probs = c(0.025, 0.975))

    2.5%    97.5%
67.93136 95.99950
```

Accounting for both sampling and prediction uncertainty, it is likely that a student who spend six hours on homework will have a GPA between 67.9 and 95.9.

#protip

The function `set.seed()` can be used to consistently generate replicable results.

```
# Set the simulation seed
> set.seed(314)


# Randomly generate something
> rnorm(n = 1, mean = 0, sd = 1)

[1] -1.288236
```