Math 463 HW4

Ian McConachie

Teen Gambling

First we can write a little bit of R code to retrieve the data that this problem is concerned with.

```
myfile <- "https://pages.uoregon.edu/dlevin/DATA/teengamb.txt"
teengamb = read.table(myfile, header = T)
teengamb$sex <- as.factor(teengamb$sex) # changing sex into a categorical variable
```

We want to make a regression model to predict gamble as a response variable based on this data, but we are unsure of what predictors to include in the model. We can make an analysis of variance table which will give us some indication of which variables contribute enough to predicting teen gambling as to justify their inclusion in the model. We can generate such an ANOVA table with the R code below:

```
diff_anova <- aov(gamble~income*sex + income*status, data=teengamb)
summary(diff_anova)
```

```
##               Df Sum Sq Mean Sq F value   Pr(>F)
## income         1  17681   17681  48.487 1.86e-08 ***
## sex            1   5227    5227  14.335 0.000491 ***
## status         1    202     202   0.553 0.461154
## income:sex     1   4115    4115  11.284 0.001699 **
## income:status  1   3514    3514   9.636 0.003451 **
## Residuals     41  14951     365
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

In the above table, the values that provide the most crucial information come in the last column. This column tells us the probability that we observe the F value seen in the previous column under the null hypothesis that there is no difference between response means for different values of that variable. In other words, the F test is testing to see if adding a new term to our model provides sufficient information to justify its addition or it would simply contribute to the noise in the data.

To analyze the above table, we can note that generally speaking, the smaller the F value, the more justifiable it is to add that variable to linear regression model. Here we can use a standardized, arbitrary value alpha, which will tell us if the F test was significant enough to include a variable; for the sake of consistency with the scientists in the room, we will choose our alpha to be 0.05.

With this alpha value, we would conclude that our model should include income and sex as predictors, with an interaction term between income and sex in the model. The status variable has large p-values associated with it which suggests that it is likely that any difference in population means between different levels of status can be attributed to random variation rather than actual difference in the response. The status variable failing to contribute to this model could be caused by its correlation to income–I'm not sure how "status" was measured in this data, but those two variables are highly correlated in this country–which would make the information it provided relatively close to noise when income is already considered.

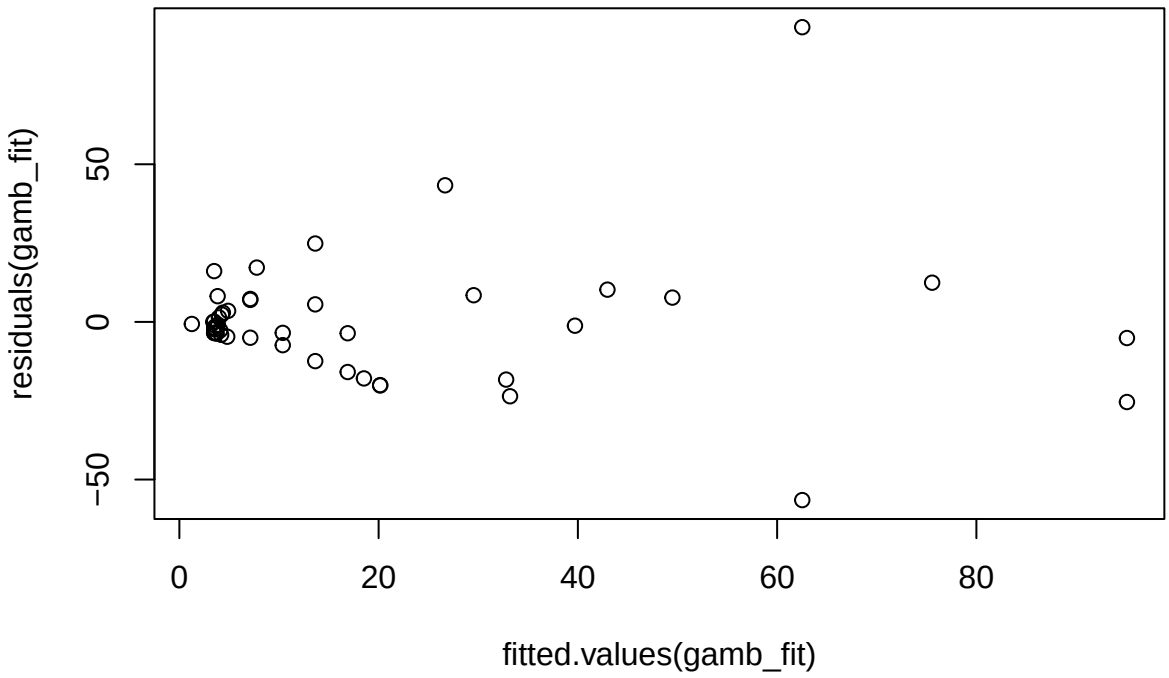
With the above analysis of variance table in mind, we can produce the following regression model to fit onto this data:

```
gamb_fit = lm(gamble~income*sex, data=teengamb)
print(gamb_fit)
```

```
##
## Call:
## lm(formula = gamble ~ income * sex, data = teengamb)
##
## Coefficients:
## (Intercept)       income         sex1  income:sex1
##      -2.660        6.518        5.800       -6.343
```

One way we can evaluate the fit of this model is by graphing its residuals against its fitted values and seeing if there is any obvious pattern. If the residual plot looks roughly random, this is some indication that the model does not violate assumptions of regression analysis. If the residual plot shows some strong pattern, this would be an indication that the way we are viewing the data is flawed in some systematic way.

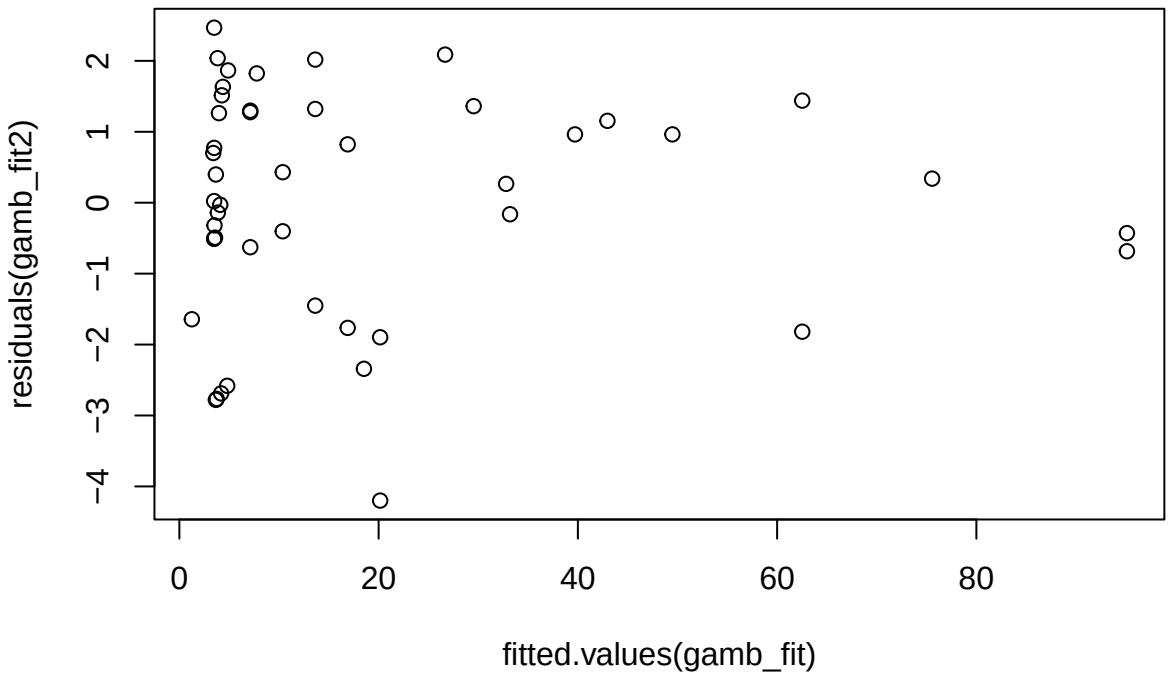
```
plot(residuals(gamb_fit)~fitted.values(gamb_fit), data=teengamb)
```



In the above two residual plots, we do see some indication of a pattern; namely, we can see a vague pattern of increased residual variation with higher fitted values. This pattern may be indication of flaws in our regression analysis, so a transformation on the data is probably a good move.

One possible transformation we could do on this data to potentially correct this issue is taking the log of the response variable (average amount gambled)—the gamble variable varies over several magnitudes which indicates that this kind of transformation could help our regression. We can check if our model improces by doing the transformation and re-plotting the residuals.

```
# This for loop makes a new variable in the data frame for the log of amount gambled
# I had to put it in a for loop because of the errors with log of 0
log_gamble = 1:length(teengamb$gamble)
for (i in 1:length(teengamb$gamble)) {
  if ((teengamb$gamble[i]) == 0) {
    log_gamble[i] = 0
  } else {
    log_gamble[i] = log(teengamb$gamble[i])
  }
}
teengamb$log_G <- log_gamble

# Now we fit the model with our transformed response parameter and plot the residuals
gamb_fit2 = lm(log_G~income*sex, data=teengamb)
plot(residuals(gamb_fit2)~fitted.values(gamb_fit), data=teengamb)
```



We can see from the above residual plots that taking the log of average amount gambled does appear to result in a more sound model as we see more randomness in the residuals.

Now we are asked if this transformation changes our answer for which variables should be included in this model in the first place. To answer this question, we can make another analysis of variance table, but this

time with log(gamble) as our response variable and see what our F tests say about the main effect and interaction terms this time:

```
diff_anova <- aov(log_G~income*sex + status*sex, data=teengamb)
summary(diff_anova)
```

```
##             Df Sum Sq Mean Sq F value  Pr(>F)
## income       1  30.22  30.220  10.976 0.00193 **
## sex          1  31.43  31.426  11.415 0.00161 **
## status       1   2.99   2.992   1.087 0.30331
## income:sex   1   7.27   7.268   2.640 0.11188
## sex:status   1   0.97   0.969   0.352 0.55627
## Residuals   41 112.88   2.753
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

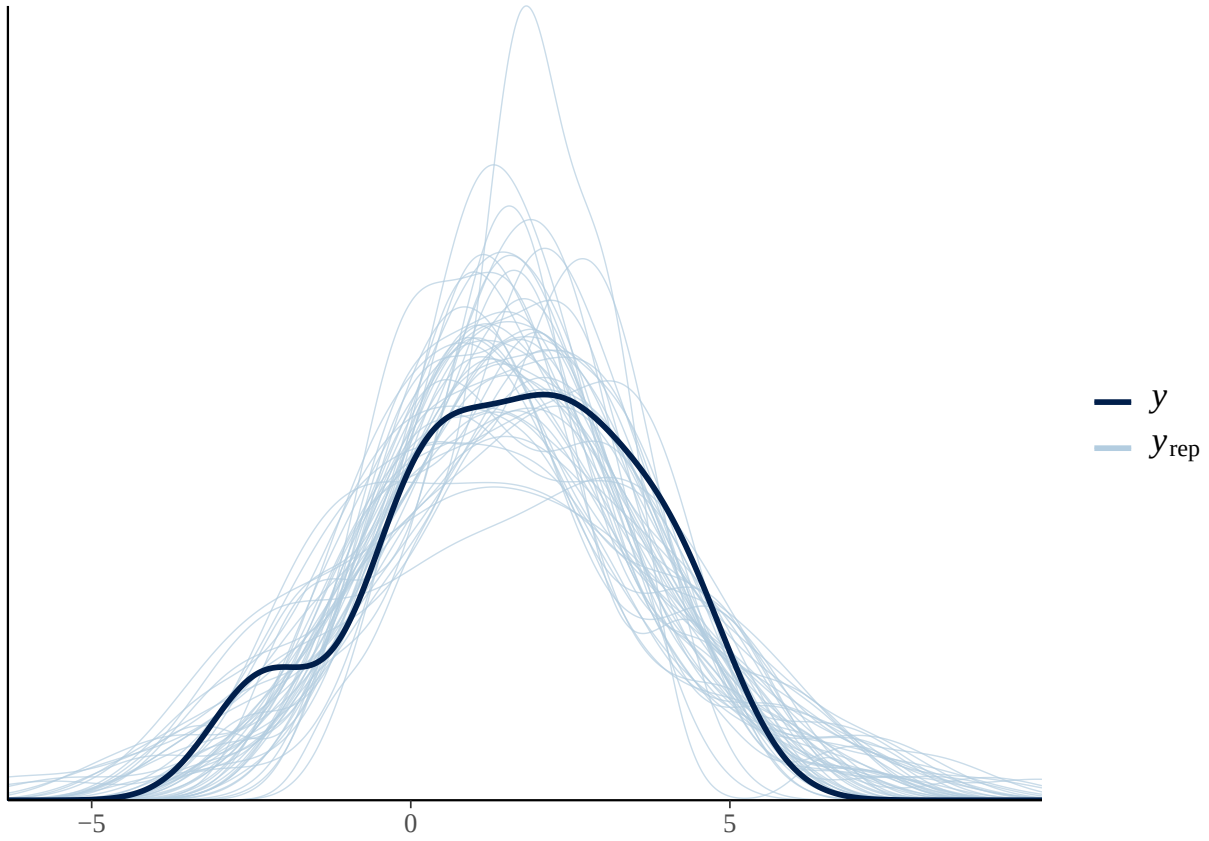
With this transformation, the only terms that are below our arbitrary alpha value for F tests are the main effect terms for income and sex–which suggests our model should use income and sex as predictors without any interaction terms between the two predictors.

All of the regression analysis above was based on classical, least-squares regression analysis, but we can also do some regression analysis from a Bayesian point of view to see which model is the best fit for this data. We can start off our Bayesian approach by producing 4 models using stan_glm:

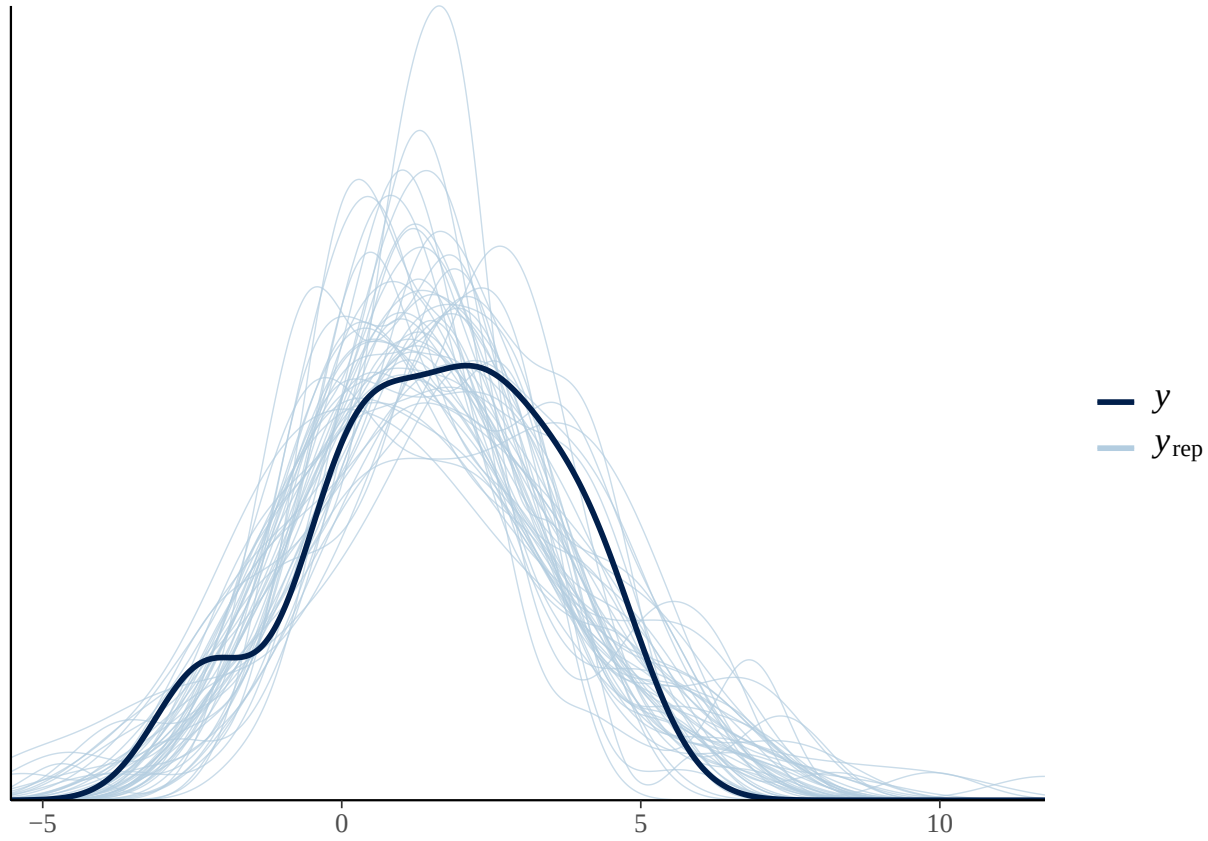
```
# This model has both sex and income as predictors with an interaction variable
g_fit1 = stan_glm(log_G~income*sex, data=teengamb, refresh=0)
# This model has income as a main effect with sex included in an interaction variable
g_fit2 = stan_glm(log_G~income + income:sex, data=teengamb, refresh=0)
# This model has sex as a main effect with income included only in an interaction variable
g_fit3 = stan_glm(log_G~sex + sex:income, data=teengamb, refresh=0)
# This model only has income as a predictor
g_fit4 = stan_glm(log_G~income, data=teengamb, refresh=0)
# This model only has sex as a predictor
g_fit5 = stan_glm(log_G~sex, data=teengamb, refresh=0)
```

We can then simulate predicted values in this dataset using all 5 of the above models and compare these to the real observed response values to see visually which model appears to produce the best fit. The R code below produces comparisons between the simulated predicted values and the observed response values for this dataset:

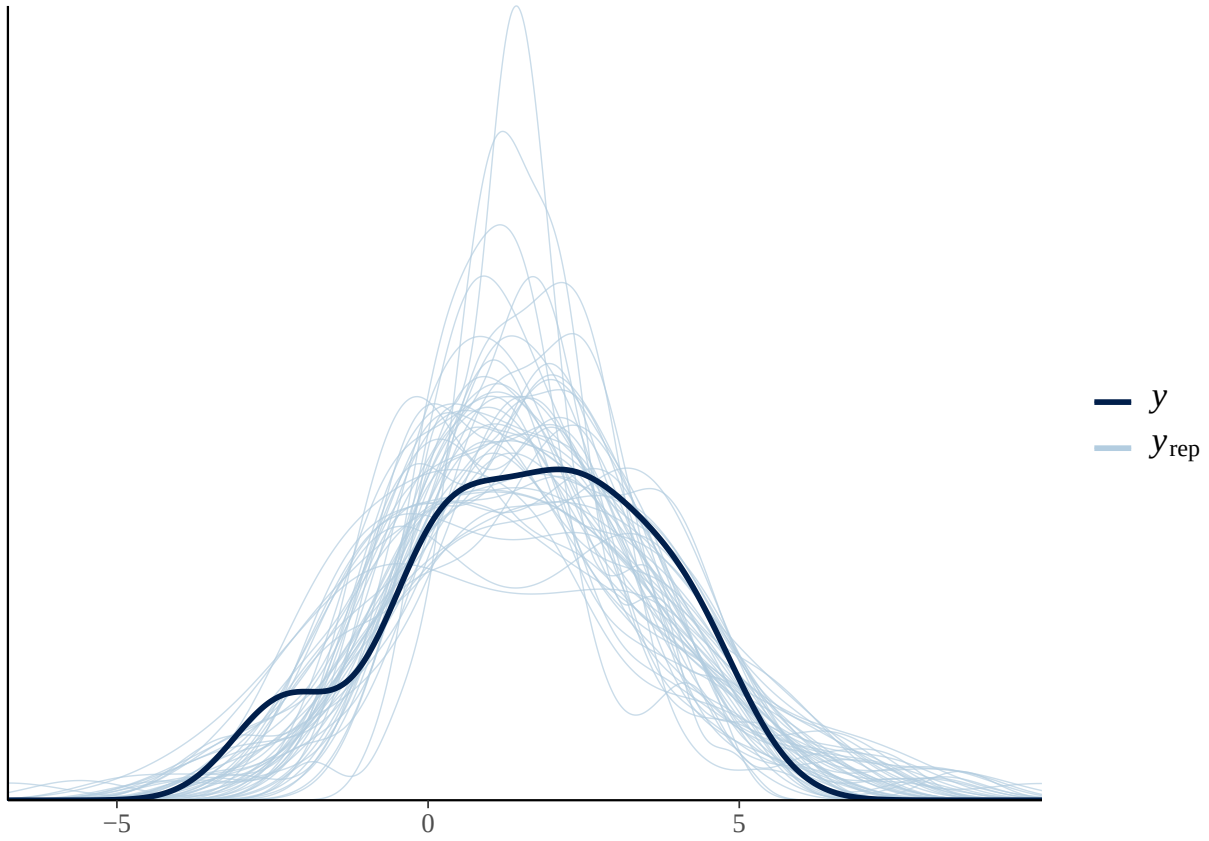
```
library(bayesplot)
# Making the simulated predicted values
rep1 <- posterior_predict(g_fit1)
rep2 <- posterior_predict(g_fit2)
rep3 <- posterior_predict(g_fit3)
rep4 <- posterior_predict(g_fit4)
rep5 <- posterior_predict(g_fit5)

# Displaying these simulations against the true observed values
ppc_dens_overlay(teengamb$log_G, rep1[1:50,])
```



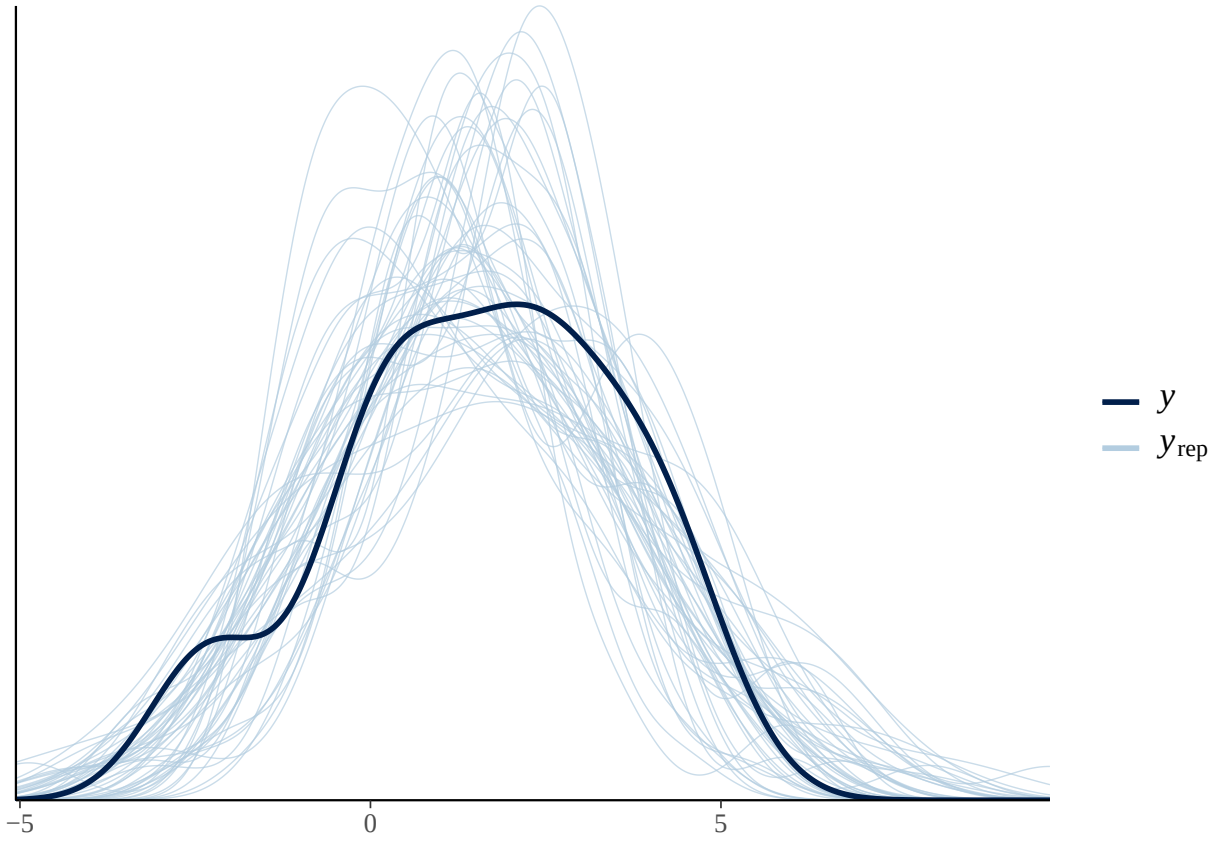

```
ppc_dens_overlay(teengamb$log_G, rep2[1:50,])
```



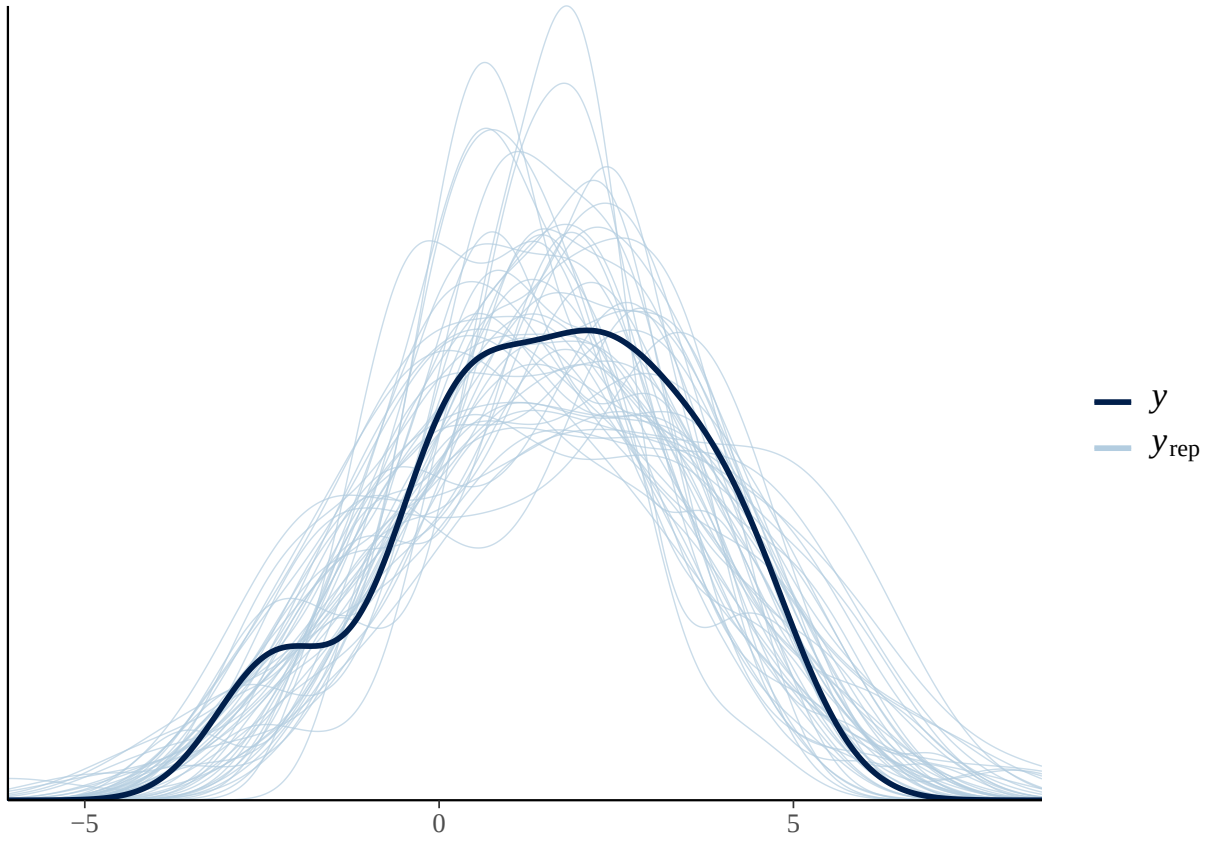

```
ppc_dens_overlay(teengamb$log_G, rep3[1:50,])
```




```
ppc_dens_overlay(teengamb$log_G, rep4[1:50,])
```




```
ppc_dens_overlay(teengamb$log_G, rep5[1:50,])
```



After running this R code several times, I have noticed that the results vary greatly, but on average, it seems like the 5 models aren't too different in their simulated predicted values in comparison to the actual observed values. However, among these similar-looking graphs, it appears that the first model (with income and sex as main effects and an interaction variable) has the best fit especially in the key areas of the mode (where it doesn't peak as much above the mode as other models) and the tail of the histogram below 0 (where this model follows the smoothed observed histogram more closely).

What these visualizations also tell us is that none of our 5 models are really great models because all of them have considerable error when comparing simulated predicted values to actual observed response values. However, among our imperfect models, it seems we should keep income as a main effect and sex should be included in the analysis in some manner (as a main effect and/or included in an interaction term).

11.2: Descriptive and Causal Inference

In section 7.1 we are given data for inflation adjusted growth in average personal income at the end of president's term and the incumbent party's vote share (among the votes for Republicans or Democrats) in the election following each of those terms. In this section, a linear model was fitted to the data with economic growth a predictor for the response variable, which was the incumbent party's vote share. With a Bayesian approach, we generated a model where the coefficient of economic growth had a median of 3.0 with a standard deviation of 0.7.

This coefficient means that, according to this data set, on average an increase of 1 percentage point in this particular heuristic of economic growth is associated with an observed 3 percent increase in vote share won by the incumbent party. This does NOT imply that with all other variables constant, a 1 percent increase in economic growth will lead to a 3 percent increase in vote share won by the incumbent party.

There are several difficulties in interpreting this regression coefficient as the "effect" of economic growth on

vote share won by the incumbent party. In particular, this regression model was fit to data that came from several elections, but the idea of "effect" implies applying estimates from this model to a single election where everything else is held constant outside of the predictor (economic growth) and the response (share won by the incumbent). Just because increased economic growth is associated with increased vote percentage to the incumbent party *on average*, this does not mean that in a specific election, increasing economic growth will lead to an increase in vote percentage going to the incumbent party.

Another assumption that does not necessarily hold when coming to the causal conclusion of economic growth "effecting" incumbent vote percentage is that it is economic growth and not some other variable correlated to economic growth that is leading to the change in vote percentage. For instance, unemployment is correlated to economic growth, so maybe this is what is affecting vote percentage going to the incumbent rather than economic growth itself–we simply do not have enough information to reach a reasonable conclusion on this front.

Both of the concerns addressed above prevent us from interpreting the conclusions drawn from this regression model in a causal way with economic growth "effecting" vote percentage going to the incumbent.

11.5: Residuals and Predictions

First we can write some R code to retrieve the data and process it into a format that is usable in R.

```
myfile <- "https://raw.githubusercontent.com/avehtari/ROS-Examples/master/Pyth/pyth.txt"
pyth = read.table(myfile, header = T)
pyth40 = pyth[1:40,]
```

Then we can generate a Bayesian Regression model predicting the response y from x_1 and x_2. The following R code does this for us using the stan_glm function.

```
y_fit = stan_glm(y~x1 + x2, data=pyth40, refresh = 0)
print(y_fit)
```

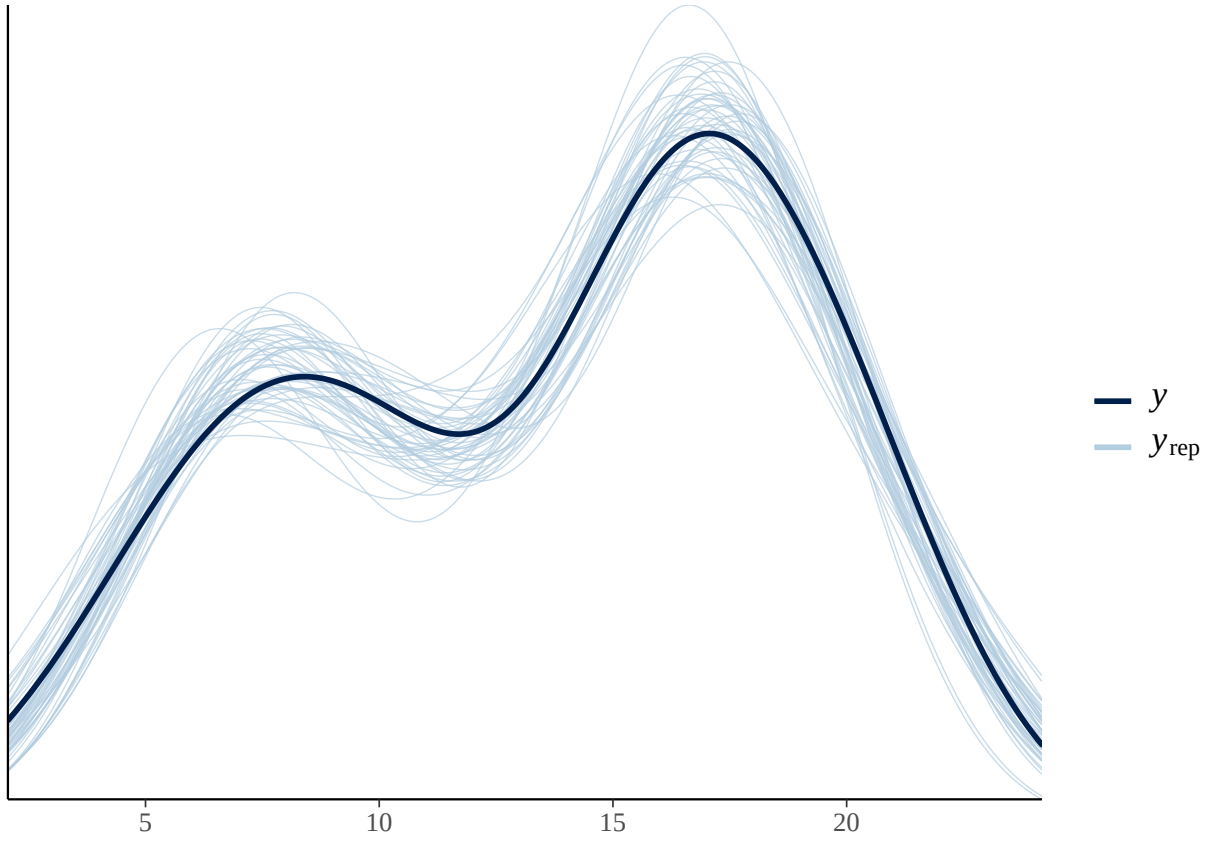
```
## stan_glm
##  family:       gaussian [identity]
##  formula:      y ~ x1 + x2
##  observations: 40
##  predictors:   3
## ------
##             Median MAD_SD
## (Intercept) 1.3    0.4
## x1          0.5    0.0
## x2          0.8    0.0
##
## Auxiliary parameter(s):
##       Median MAD_SD
## sigma 0.9    0.1
##
## ------
## * For help interpreting the printed output see ?print.stanreg
## * For info on the priors used see ?prior_summary.stanreg
```

From this Bayesian regression model, we can infer that there is a positive relationship between both predictors (x1, and x2) and the response variable y. Furthermore, we can infer that the model has some positive

intercept, meaning when either x1, x2, or both are zero (and the other is set to its mean when only one is 0), the value of the response variable y is positive.

After generating this model, we can check the fit of the model visually by simulating predicted response values given the predictors in pyth40 and then graphing the smoothed histograms of these simulations against a smoothed histogram of the actual observed response variables. This can be done with the following R code and produces a visual graph like the one seen below:

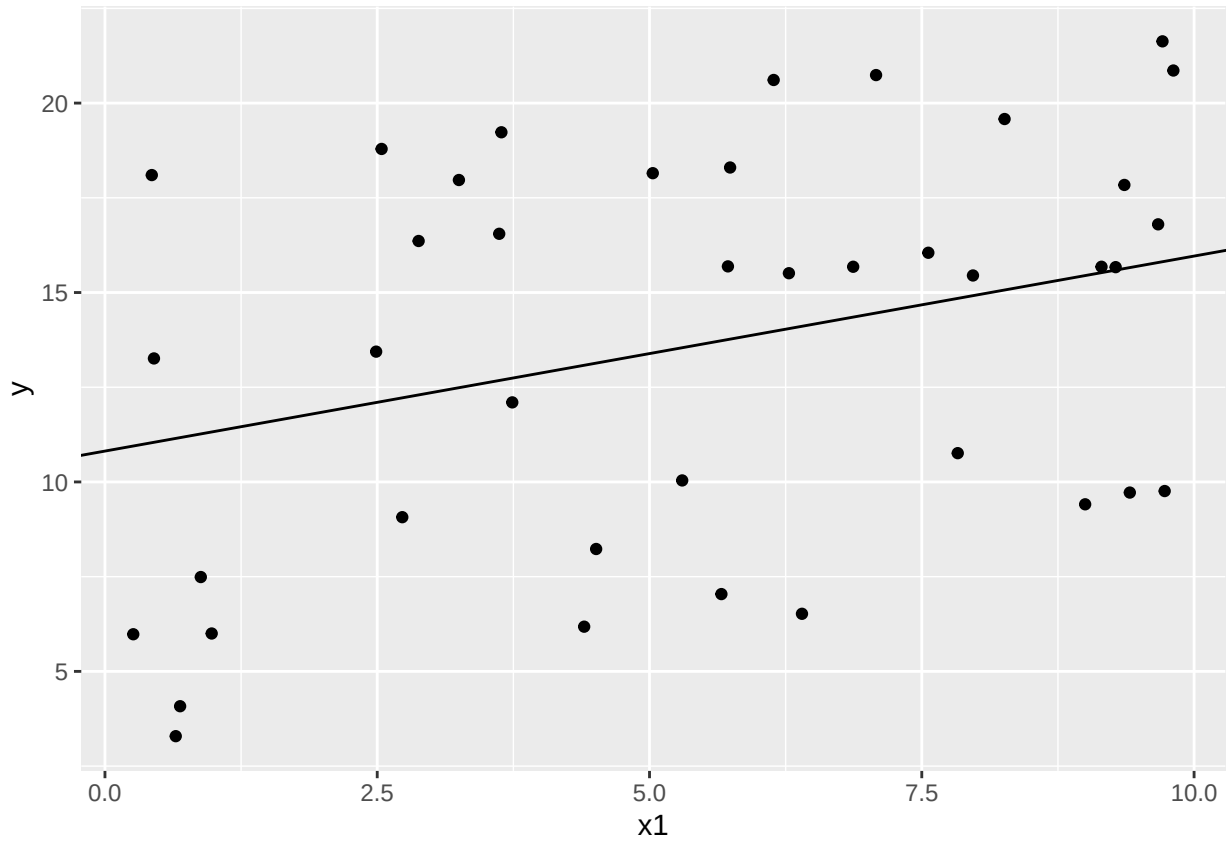
```
# Generating the predictions
rep <- posterior_predict(y_fit)
# Displaying these simulations against the true observed values
ppc_dens_overlay(pyth40$y, rep[1:50,])
```



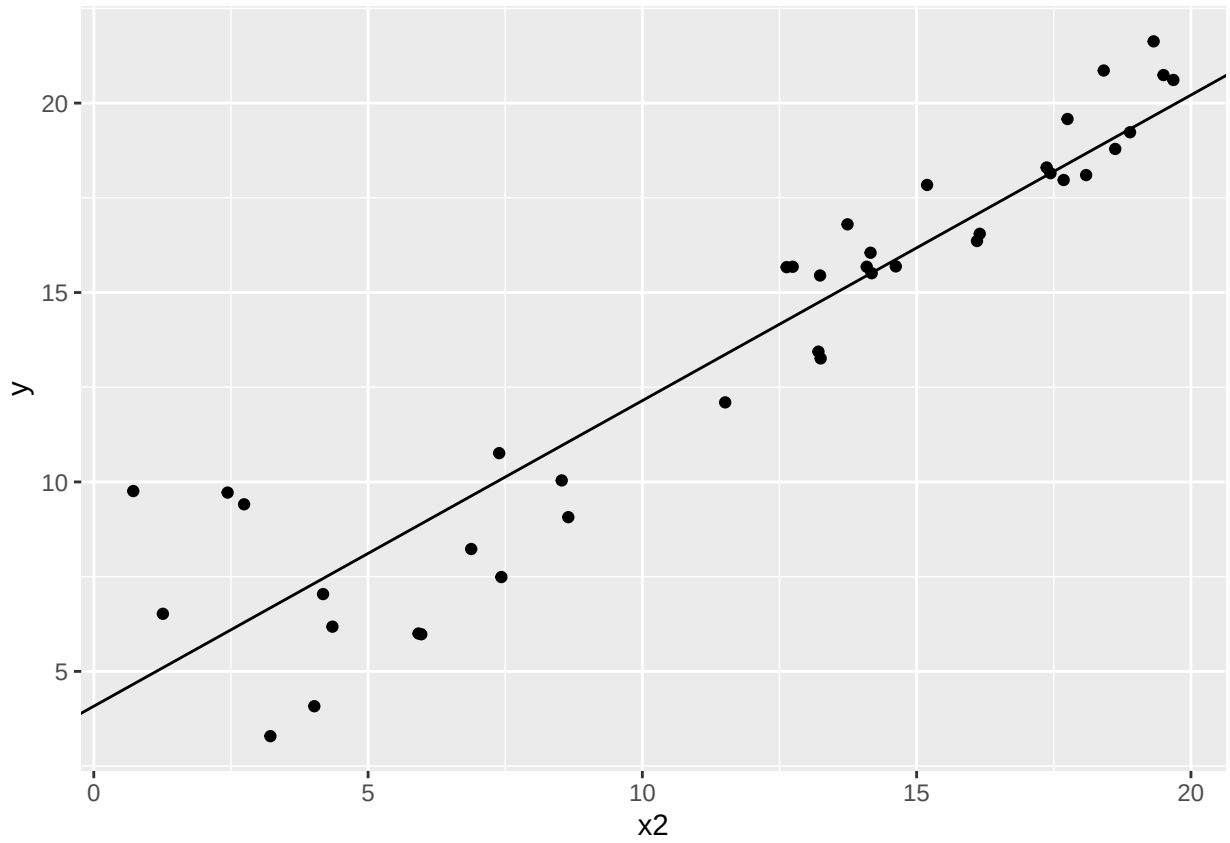
Just based off of visual analysis of the graph above, the model we have generated seems to fit the data very well. However, there is still more analysis we could do to asses the model of this data we have created.

We can plot the regression line generated from stan_glm against the data points for our observed response variable. Even though we have 2 predictors, we can represent this regression line as a 2D graphs by having 2 graphs: one where x1 varies and x2 is set as a constant (its mean) and one where x2 varies and x1 is set as a constant (its mean).‘

```
yc = coef(y_fit)
# Using the average value of x2 for x2 in this plot
x2_avg = mean(pyth40$x2)
sl1 = yc[2]
inter1 = yc[1] + (yc[3] * x2_avg)
ggplot(pyth40, aes(x=x1, y=y)) + geom_point() + geom_abline(intercept = inter1, slope = sl1)
```

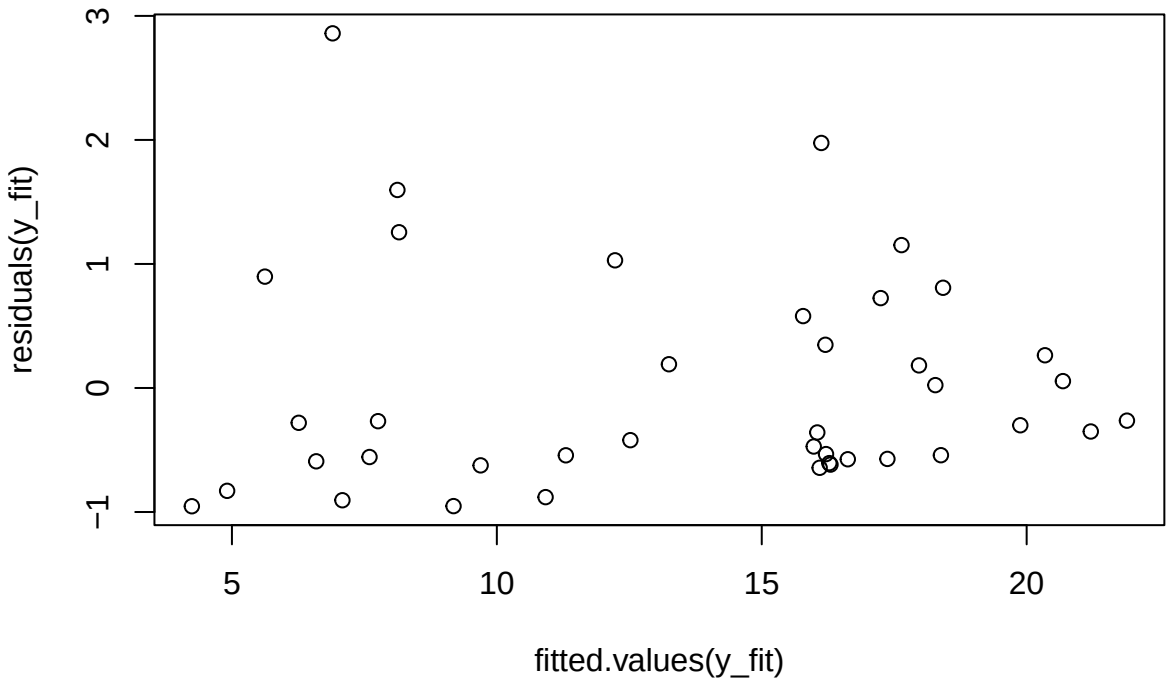



```
# Using the average value of x1 for x1 in this plot
x1_avg = mean(pyth40$x1)
sl2 = yc[3]
inter2 = yc[1] + (yc[2] * x1_avg)
ggplot(pyth40, aes(x=x2, y=y)) + geom_point() + geom_abline(intercept = inter2, slope = sl2)
```



Although graphing our model visually generally seems to show a pretty good fit to the data, we can further check our regression analysis by generating a residual plot to check to see if the assumptions inherit in regression analysis hold true. The following R code generates a residual plot which we can validate visually to see if any assumptions are obviously violated.

```
plot(residuals(y_fit)~fitted.values(y_fit), data=pyth40)
```



Although this residual plot seems to show positive residuals with a greater magnitude than the negative residuals, there still does not seem to be any strong patterns that would suggest a violation of the additivity, linearity, or equal variance of error assumptions necessary for regression analysis. Therefore, we can keep our current model without making any transformations on the data to preserve any of our assumptions.

Now that we have a solid model worked out that fits our data, we can predict the response value of the last 20 data points that have values for the predictors, but not for y. We can do this with the R code below:

```
pyth20 = pyth[41:60,2:3]
pred20 <- predict(y_fit,newdata=pyth20)
print(pred20)
```

```
##        41        42        43        44        45        46        47        48
## 14.808069 19.137489  5.914640 10.528935 19.007225 13.395839  4.828368  9.142806
##        49        50        51        52        53        54        55        56
##  5.890817 12.336840 18.904483 16.060926  8.960678 14.969952  5.858875  7.373887
##        57        58        59        60
##  4.534749 15.129478  9.099406 16.080262
```

Based on the visual comparison we did of the simulated predicted responses against the observed response values and the checking of regression assumptions with our residual plot, I am fairly confident in the ability to make a linear model with this data and of the predictive ability of this particular model we have generated. Because of this, I am confident in the response predictions we made for the last 20 datapoints—even if there are no observed values to compare them against.

11.6: Fitting a wrong model

We can simulate the data this problem is asking for with the R code below.

```
library(metRology)  # Need this library to generate random, scaled t-distriubution values
# Simulate our predictors and error
x1 <- 1:100
x2 <- rbinom(100, 1, 0.5)
error <- rt.scaled(100, 4,0,5)
# Simulate the response variable y
y = 3 + (0.1 * x1) + (0.5 * x2) + error
# Put everything together in a data frame
sim_data = data.frame(x1,x2,y)
```

We can then use a Bayesian approach to fit a regression line to the simulated data using the stan_glm function.

```
sim_fit = stan_glm(y~x1 + x2, data=sim_data, refresh=0)
print(sim_fit)
```

```
## stan_glm
##  family:       gaussian [identity]
##  formula:      y ~ x1 + x2
##  observations: 100
##  predictors:   3
## ------
##             Median MAD_SD
## (Intercept) 2.7    1.8
## x1          0.1    0.0
## x2          2.4    1.6
##
## Auxiliary parameter(s):
##       Median MAD_SD
## sigma 7.8    0.6
##
## ------
## * For help interpreting the printed output see ?print.stanreg
## * For info on the priors used see ?prior_summary.stanreg
```

We can then take our fitted regression and create 68% uncertainty intervals around the estimate for each coefficient by adding 1 standard error to each side of the estimated coefficient. Because we simulated the data and know the real coefficients, we can see if this 68% interval has the real coefficient value included within it.

```
sf_coef = coef(sim_fit); sf_se = se(sim_fit)
inter_int = 1:2; x1_int = 1:2; x2_int = 1:2
# Creating our intervals in the form of lists with lower bound
# first and upperbound second
inter_int[1] = sf_coef[1] - sf_se[1]; inter_int[2] = sf_coef[1] + sf_se[1]
x1_int[1] = sf_coef[2] - sf_se[2]; x1_int[2] = sf_coef[2] + sf_se[2]
x2_int[1] = sf_coef[3] - sf_se[3]; x2_int[2] = sf_coef[3] + sf_se[3]
```

```r
# Printing out the test results for the intercept
if ((inter_int[1] < 3) & (3 < inter_int[2])) {
  print("The true intercept value of 3 is in between the 68 percent uncertainty")
  sprintf("interval of (%f, %f)", inter_int[1], inter_int[2])
} else {
  print("The true intercept value of 3 is NOT in between the 68 percent uncertainty")
  sprintf("interval of (%f, %f)", inter_int[1], inter_int[2])
}
```

```
## [1] "The true intercept value of 3 is in between the 68 percent uncertainty"
```

```
## [1] "interval of (0.899113, 4.419101)"
```

```r
# Printing out the test results for x1
if ((x1_int[1] < 0.1) & (0.1 < x1_int[2])) {
  print("The true x1 coefficient of 0.1 is in between the 68 percent uncertainty")
  sprintf("interval of (%f, %f)", x1_int[1], x1_int[2])
} else {
  print("The true x1 coefficient of 0.1 is NOT in between the 68 percent uncertainty")
  sprintf("interval of (%f, %f)", x1_int[1], x1_int[2])
}
```

```
## [1] "The true x1 coefficient of 0.1 is in between the 68 percent uncertainty"
```

```
## [1] "interval of (0.067984, 0.121866)"
```

```r
# Printing out the test results for x2
if ((x2_int[1] < 0.5) & (0.5 < x2_int[2])) {
  print("The true x2 coefficient of 0.5 is in between the 68 percent uncertainty")
  sprintf("interval of (%f, %f)", x2_int[1], x2_int[2])
} else {
  print("The true x2 coefficient of 0.5 is in between the 68 percent uncertainty")
  sprintf("interval of (%f, %f)", x2_int[1], x2_int[2])
}
```

```
## [1] "The true x2 coefficient of 0.5 is in between the 68 percent uncertainty"
```

```
## [1] "interval of (0.752109, 3.992928)"
```

This is one instance of testing to see if our 68% uncertainty intervals include the true coefficients, but we can test the effectiveness of these intervals more rigorously by doing this experiment 1000 times and seeing how often the true parameters are included in the intervals. Note that in the tests below, I use the lm function instead of stan_glm–I do this not for a preference of least-squares regression over Bayesian regression, but rather because the computation time for lm is so much less than stan_glm and that makes a significant difference over the 1000 iterations.

```r
# Here is a function that simulates the data we'll use for fitting our line
# the function returns a data frame populated with the simulated values of both
# predictors and response
simulate_data <- function() {
  x1 <- 1:100
```

```
  x2 <- rbinom(100, 1, 0.5)
  error <- rt.scaled(100, 4,0,5)
  y = 3 + (0.1 * x1) + (0.5 * x2) + error
  fake_data = data.frame(x1,x2,y)
  return(fake_data)
}

# Here is a function that fits a least-squares regression model to the simulated
# data and then returns a summary of the fit in the form of a vector
fit_reg <- function(data) {
  sim_fit = lm(y~x1 + x2, data = data)
  sim_summ = summary(sim_fit)
  fit_vals = coef(sim_summ)
  return(fit_vals)
}

# This function generates 68% uncertainty intervals based on the fit values passed in
# and then tests to see if the true coefficient values from the simulation are in
# those intervals. The function returns a vector with binary indicators telling
# if the interval contained the coefficient in question
test_intervals <- function(fit_vals) {
  test_results = 1:3
  # Testing to see if the true intercept is in the 68% uncertainty interval
  if (((fit_vals[1] - fit_vals[4]) < 3) & (3 < (fit_vals[1] + fit_vals[4]))) {
    test_results[1] = 1
  } else {
    test_results[1] = 0
  }
  # Testing to see if the true x1 coefficient is in the 68% uncertainty interval
  if (((fit_vals[2] - fit_vals[5]) < 0.1) & (0.1 < (fit_vals[2] + fit_vals[5]))) {
    test_results[2] = 1
  } else {
    test_results[2] = 0
  }
  # Testing to see if the true x2 coefficient is in the 68% uncertainty interval
  if (((fit_vals[3] - fit_vals[6]) < 0.5) & (0.1 < (fit_vals[3] + fit_vals[6]))) {
    test_results[3] = 1
  } else {
    test_results[3] = 0
  }
  return(test_results)
}

# Here is our for-loop that does the above interval testing 1000 times
num_trials = 1000
inter_good = 1:num_trials; x1_good = 1:num_trials; x2_good = 1:num_trials
for (i in 1:num_trials) {
  sims = simulate_data()
  sims_fit = fit_reg(sims)
  test_results = test_intervals(sims_fit)
  inter_good[i] = test_results[1]; x1_good[i] = test_results[2]; x2_good[i] = test_results[3]
}
```

```
sprintf("The proportion of 'good' intervals for the intercept is: %f ", mean(inter_good))
```

```
## [1] "The proportion of 'good' intervals for the intercept is: 0.709000 "
```

```
sprintf("The proportion of 'good' intervals for the x1 coefficient is: %f ", mean(x1_good))
```

```
## [1] "The proportion of 'good' intervals for the x1 coefficient is: 0.696000 "
```

```
sprintf("The proportion of 'good' intervals for the x2 coefficient is: %f ", mean(x2_good))
```

```
## [1] "The proportion of 'good' intervals for the x2 coefficient is: 0.738000 "
```

The results of the above 1000 trials will be different every time as the simulated data is generated using pseudo-random numbers. However, after running this code chunk several times, I have noticed that the proportion of intervals containing the true parameter ("good intervals") for the intercept is usually slightly above 68%; the proportion of "good" intervals for the x1 coefficient is usually slightly below 68%; and the proportion of "good" intervals for the x2 coefficient is usually well above 68%.

These findings suggest that having non-normal error calls for different approaches to estimating coefficients in a model and calculating the error around these estimates. However, none of the proportions are ever extremely far from the expected 68% under normal errors. so this experiment also tells us that while flawed, models made with the assumption of normal error can still be very useful when errors actually follow a scaled t-distribution.