Collection, Modeling, and Visualization of Stock Market Data

Glib Dolotov

June 8, 2018

Here, stock market data will be pulled from the AlphaVantage free API. Code is written to permit for easy collection, tidying, and visualization of data. For more details on the AlphaVantage API seehttps://www.alphavantage.co/documentation/.

We will need the following libraries:

```
library(httr)
library(jsonlite)
library(lubridate)
library(tidyverse)
library(shiny)
library(modelr)
library(moments)
library(splines)
library(gridExtra)
```

Collection

First, let us define a function that will pull JSON data from the API. This can be easily done using the `GET()` function. Constants shared by all API calls are saved as variables.

```
url <- "https://www.alphavantage.co"
path <- "/query"
apikey <- "NLS6EWI7EU6UMENW"

get_content <- function(q){
  data <- GET(url = url,
              path = path,
              query = q)
  raw <- rawToChar(data$content)
  return(fromJSON(raw))
}
```

Sample Query

Since each API method has different arguments, creating queries should be constructed via interactive user interface. A sample query will be a list. Construction will look as follows:

```
query <<- list('function' = "TIME_SERIES_WEEKLY",
               symbol = "NVDA",
               apikey = apikey)
```

The query is then run through the `get_content` function that was previously defined.

```
raw_data <- get_content(query)
```

Tidying the Data

The amount of data is very large, so we will avoid printing it in this document. The analysis done to organize the data into a more usable form will instead be demonstrated with methods that avoid a massive printed output. First, let us look at the names of the components of `raw_data`.

```
names(raw_data)
```

```
## [1] "Meta Data"          "Weekly Time Series"
```

The first item of `raw_data` is fairly small, so we can look at it here:

```
raw_data$`Meta Data`
```

```
## $`1. Information`
## [1] "Weekly Prices (open, high, low, close) and Volumes"
##
## $`2. Symbol`
## [1] "NVDA"
##
## $`3. Last Refreshed`
## [1] "2018-06-08 14:38:03"
##
## $`4. Time Zone`
## [1] "US/Eastern"
```

However, if we were to try to do this with the second item of `raw_data` this PDF file would be very long indeed...

```
length(raw_data$`Weekly Time Series`)
```

```
## [1] 961
```

There are too many data-points to print here. Let's isolate just one:

```
raw_data$`Weekly Time Series`[1]
```

```
## $`2018-06-08`
## $`2018-06-08`$`1. open`
## [1] "259.0000"
##
## $`2018-06-08`$`2. high`
## [1] "266.5900"
##
## $`2018-06-08`$`3. low`
## [1] "257.7000"
```

```
##
## $`2018-06-08`$`4. close`
## [1] "262.5405"
##
## $`2018-06-08`$`5. volume`
## [1] "40498863"
```

Now, let's extract the information we need:

```r
# 1) Symbol
raw_data[[1]][[2]]
```

```
## [1] "NVDA"
```

```r
# 2) Time-Stamps (this is our X-axis data)
as.POSIXct(names(raw_data[[2]])) %>% head(n = 5)
```

```
## [1] "2018-06-08 EDT" "2018-06-01 EDT" "2018-05-25 EDT" "2018-05-18 EDT"
## [5] "2018-05-11 EDT"
```

```r
# 3) Interval
raw_data[[1]][[4]]
```

```
## [1] "US/Eastern"
```

As a side-note, you might notice that in this example, the "Interval" item may seem incorrect. This is due to the output formatting of each API call. Certain API methods will return "Interval" information which we will want to capture. For other methods, there is no "Interval" variable returned. In which case this code will capture another piece of information instead. It is far simpler to create a catch-all and then ignore the false "Interval" than to figure out which API method is being used and create a unique capture method for each. Now, let us begin by, again, looking at what the second argument of `raw_data` looks like

```r
raw_data[[2]] %>% head(n = 1)
```

```
## $`2018-06-08`
## $`2018-06-08`$`1. open`
## [1] "259.0000"
##
## $`2018-06-08`$`2. high`
## [1] "266.5900"
##
## $`2018-06-08`$`3. low`
## [1] "257.7000"
##
## $`2018-06-08`$`4. close`
## [1] "262.5405"
##
## $`2018-06-08`$`5. volume`
## [1] "40498863"
```

Unlisting this output makes this portion of the data far more managable:

```r
temp_data <- raw_data[[2]] %>% unlist()
temp_data %>% head(n = 10)
```

```
##   2018-06-08.1. open   2018-06-08.2. high    2018-06-08.3. low
##           "259.0000"           "266.5900"           "257.7000"
##  2018-06-08.4. close 2018-06-08.5. volume   2018-06-01.1. open
##           "262.5405"           "40498863"           "248.5500"
##   2018-06-01.2. high    2018-06-01.3. low  2018-06-01.4. close
##           "257.8700"           "246.7000"           "257.6200"
## 2018-06-01.5. volume
##           "41487904"
```

We will need to separate out the "open", "high", "low", "close", and "volume" data points. To do this, we can use the `grepl()` method:

```
temp_data[grepl("open", names(temp_data))] %>% head(n = 5)
```

```
## 2018-06-08.1. open 2018-06-01.1. open 2018-05-25.1. open
##         "259.0000"         "248.5500"         "249.8800"
## 2018-05-18.1. open 2018-05-11.1. open
##         "256.0700"         "243.2947"
```

All we need to do now is to convert the string output into doubles. With that last step, we have a nice way of collecting the values into vectors.

```
# 4) Open
temp_data[grepl("open", names(temp_data))] %>% as.double() %>% head(n = 5)
```

```
## [1] 259.0000 248.5500 249.8800 256.0700 243.2947
```

```
# 5) High
temp_data[grepl("high", names(temp_data))] %>% as.double() %>% head(n = 5)
```

```
## [1] 266.59 257.87 250.03 258.49 260.50
```

```
# 6) Low
temp_data[grepl("low", names(temp_data))] %>% as.double() %>% head(n = 5)
```

```
## [1] 257.70 246.70 240.25 241.50 242.89
```

```
# 7) Close
temp_data[grepl("[0-9]. close", names(temp_data))] %>% as.double() %>% head(n = 5)
```

```
## [1] 262.5405 257.6200 249.2800 245.9400 254.5300
```

```
# 8) Volume
temp_data[grepl("volume", names(temp_data))] %>% as.double() %>% head(n = 5)
```

```
## [1] 40498863 41487904 58283260 76099940 99988167
```

With this information in-hand, we can easily construct a neat tibble:

```
content_to_tibble <- function(cont){
  cont_2 <- unlist(cont[[2]])
  return(
    tibble(
      symbol = cont[[1]][[2]],
      datetime = as.POSIXct(names(cont[[2]])),
      interval = cont[[1]][[4]],
```

```r
      open = as.double(cont_2[grepl("open",names(cont_2))]),
      high = as.double(cont_2[grepl("high",names(cont_2))]),
      low = as.double(cont_2[grepl("low",names(cont_2))]),
      close = as.double(cont_2[grepl("[0-9]. close",names(cont_2))]),
      volume = as.double(cont_2[grepl("volume",names(cont_2))])
    )
  )
}

tibbled_data <- content_to_tibble(raw_data)
tibbled_data
```

```
## # A tibble: 961 x 8
##    symbol datetime            interval    open  high   low close   volume
##    <chr>  <dttm>              <chr>      <dbl> <dbl> <dbl> <dbl>    <dbl>
##  1 NVDA   2018-06-08 00:00:00 US/Eastern   259   267   258   263 40498863
##  2 NVDA   2018-06-01 00:00:00 US/Eastern   249   258   247   258 41487904
##  3 NVDA   2018-05-25 00:00:00 US/Eastern   250   250   240   249 58283260
##  4 NVDA   2018-05-18 00:00:00 US/Eastern   256   258   242   246 76099940
##  5 NVDA   2018-05-11 00:00:00 US/Eastern   243   260   243   255 99988167
##  6 NVDA   2018-05-04 00:00:00 US/Eastern   227   239   222   239 42342157
##  7 NVDA   2018-04-27 00:00:00 US/Eastern   229   232   210   226 55181779
##  8 NVDA   2018-04-20 00:00:00 US/Eastern   232   239   227   229 51189371
##  9 NVDA   2018-04-13 00:00:00 US/Eastern   217   238   215   232 70536826
## 10 NVDA   2018-04-06 00:00:00 US/Eastern   229   235   213   214 93572928
## # ... with 951 more rows
```

Modeling

Modeling the increase or decrease of share prices in the stock market is extremely difficult. The models presented here are meant as an exercise and demonstration, not for practical usage in trading. With continued study of statistics and corporate finance, I hope to continue applying more advanced and practical modeling techniques.

Creating a Linear Model using `lm()`

For now, let us begin with the simplest family of models: linear models. We are assuming that share prices correlate with a timeline in a linear fashion. R allows us to create a linear model fairly easily:

```r
mod <- lm(close ~ ns(datetime, 20), data = tibbled_data)

mod
```

```
##
## Call:
## lm(formula = close ~ ns(datetime, 20), data = tibbled_data)
##
## Coefficients:
##        (Intercept)   ns(datetime, 20)1   ns(datetime, 20)2
##           83.23375             0.04683           -97.06783
##  ns(datetime, 20)3   ns(datetime, 20)4   ns(datetime, 20)5
##          -44.80461           -80.49351           -40.07458
```

```
##  ns(datetime, 20)6   ns(datetime, 20)7   ns(datetime, 20)8
##          -60.56261           -36.66724           -64.88620
##  ns(datetime, 20)9  ns(datetime, 20)10  ns(datetime, 20)11
##          -78.59446           -65.50070           -68.62548
## ns(datetime, 20)12  ns(datetime, 20)13  ns(datetime, 20)14
##          -66.58681           -72.95130           -66.95163
## ns(datetime, 20)15  ns(datetime, 20)16  ns(datetime, 20)17
##          -63.61847           -61.53035           -44.15959
## ns(datetime, 20)18  ns(datetime, 20)19  ns(datetime, 20)20
##           77.71886           122.50128           191.08522
```

Next, let us create a new data-set, `grid`, that will contain data-points predicted by the model:

```
grid <- tibbled_data %>%
  data_grid(datetime) %>%
  add_predictions(mod)

grid
```

```
## # A tibble: 961 x 2
##    datetime             pred
##    <dttm>              <dbl>
##  1 2000-01-14 00:00:00  83.2
##  2 2000-01-21 00:00:00  82.7
##  3 2000-01-28 00:00:00  82.3
##  4 2000-02-04 00:00:00  81.8
##  5 2000-02-11 00:00:00  81.3
##  6 2000-02-18 00:00:00  80.8
##  7 2000-02-25 00:00:00  80.3
##  8 2000-03-03 00:00:00  79.8
##  9 2000-03-10 00:00:00  79.4
## 10 2000-03-17 00:00:00  78.9
## # ... with 951 more rows
```

Calculating Residuals

Residuals are crucial in understanding how well a model fits existing data. Therefore, we will calculate the residuals of the data and add it to the tibble containing it:

```
tibbled_data <- tibbled_data %>%
  add_residuals(mod)

tibbled_data
```

```
## # A tibble: 961 x 9
##    symbol datetime            interval    open  high   low close   volume
##    <chr>  <dttm>              <chr>      <dbl> <dbl> <dbl> <dbl>    <dbl>
##  1 NVDA   2018-06-08 00:00:00 US/Eastern   259   267   258   263 40498863
##  2 NVDA   2018-06-01 00:00:00 US/Eastern   249   258   247   258 41487904
##  3 NVDA   2018-05-25 00:00:00 US/Eastern   250   250   240   249 58283260
##  4 NVDA   2018-05-18 00:00:00 US/Eastern   256   258   242   246 76099940
##  5 NVDA   2018-05-11 00:00:00 US/Eastern   243   260   243   255 99988167
```

```
## 6 NVDA   2018-05-04 00:00:00 US/Eastern   227   239   222   239 42342157
## 7 NVDA   2018-04-27 00:00:00 US/Eastern   229   232   210   226 55181779
## 8 NVDA   2018-04-20 00:00:00 US/Eastern   232   239   227   229 51189371
## 9 NVDA   2018-04-13 00:00:00 US/Eastern   217   238   215   232 70536826
## 10 NVDA   2018-04-06 00:00:00 US/Eastern   229   235   213   214 93572928
## # ... with 951 more rows, and 1 more variable: resid <dbl>
```

Putting it All Together

It would be useful to have a function that does all of this for us in one step. The below method takes a tibble of data and an input containing bounding limits on the x-values (the datetime) and the y-values (close). It will first filter data-points that do not fall within the specified ranges. This allows us to easily analyze subsets of the data instead of being forced to analyze it in entirety. This becomes useful if we wish to eliminate outliers.

```r
model <- function(pulled_data, input){
  active_data <- pulled_data %>%
    filter(datetime >= input$x_range[1] & datetime <= input$x_range[2]) %>%
    filter(close >= input$y_range[1] & close <= input$y_range[2])

  mod <- lm(close ~ ns(datetime, input$spline_count), data = active_data)

  grid <- active_data %>%
    data_grid(datetime) %>%
    add_predictions(mod)

  active_data <- active_data %>%
    add_residuals(mod)

  out <- list(active_data, grid)
  return(out)
}
```

Visualization

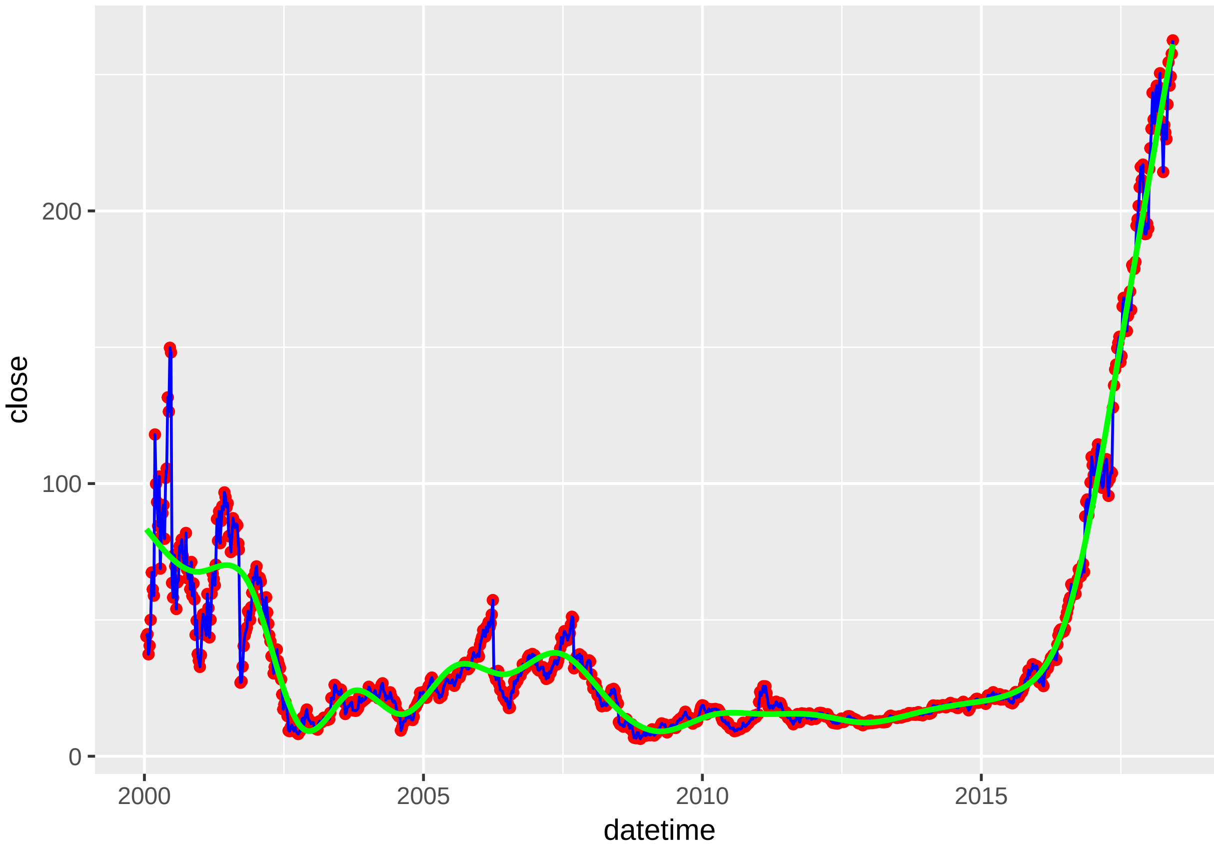
Now that we have the data in a tidy format, a linear model fitted to the data, and residual data comparing the dataset to the model we can begin to create useful visualizations of all three.

Time-Series and Model

Our first graph will include the time-series data and the values predicted by the linear model:

```r
# Create the ggplot using our tibbled_data
ggplot(tibbled_data, aes(datetime)) +
  # Next, plot the data-points.
  geom_point(aes(y = close), color = "red") +
  # For ggplot to work nicely with date-time values, we must scale the axis
  # accordingly
  scale_x_datetime() +
```

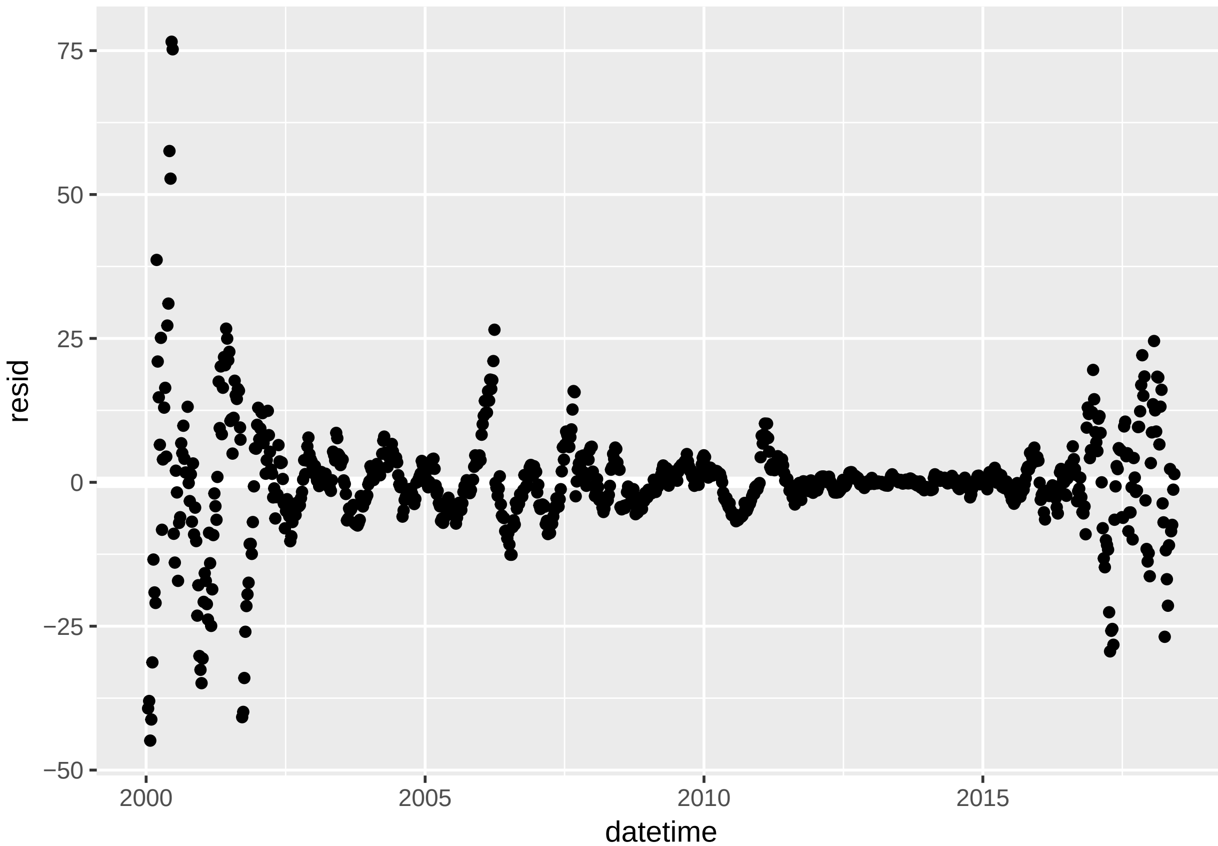
```r
  # We connect the data-points with a blue line.
  geom_line(aes(y = close), color = "blue") +
  # We plot the linear model in the same graph.
  geom_line(aes(y = pred), data = grid,
            color = "green", size = 1)
```



Residuals

While this graph is nice and it *looks* as though it fits nicely, we should take a look at the residuals before jumping to any conclusions. To plot them:

```r
ggplot(tibbled_data, aes(datetime, resid)) +
    geom_ref_line(h = 0) +
    geom_point()
```

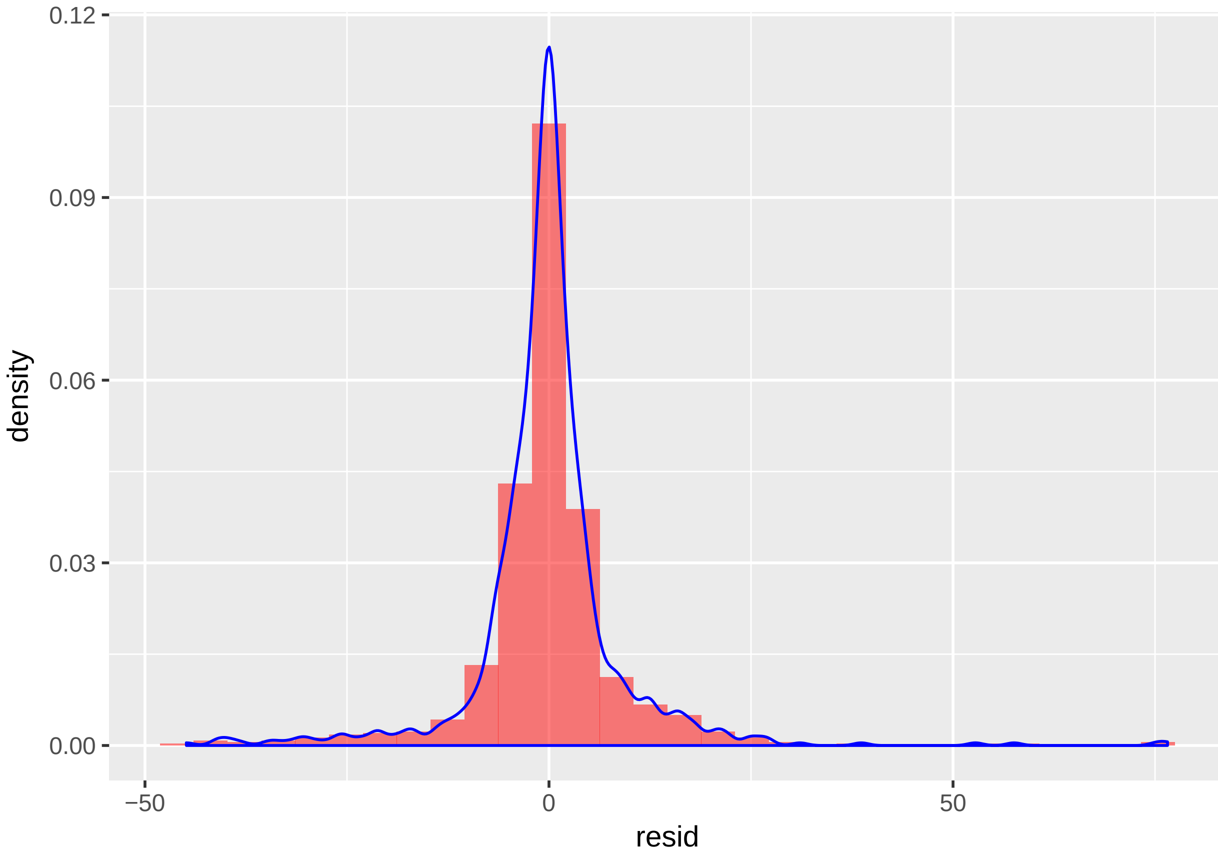


Unfortunately, the residuals of this graph are organized into patterns. More specifically, we can see a periodicity in the residuals. This implies that the model has room for improvement. But we knew that going in: applying a linear model to stock price data is not useful since stock prices do not behave linearly.

Residual Distribution

Ideally, the residuals would be distributed randomly. However, it is difficult for us to spot "randomness" in a graph such as the one above. So, a third graph showing the density distribution of the residuals. Such a graph would be useful in determining whether the residuals truly fall randomly or if they simply "look" random.

```
ggplot(tibbled_data, aes(resid)) +
    geom_histogram(aes(y = ..density..), fill = 'red', alpha = 0.5) +
    geom_density(color = 'blue')
```



Needless to say, visualizing this data is useful but it is no substitute for quantitative analysis. However deeper quantitative analysis will be saved for a later date.

Putting it All Together

The below method accepts tibbled data (as well as bounding data collected from a user interface) to create plots similar to the ones above. A user can specify a datetime range, a price range, and the degree polynomial to which a model will be fitted. All three graphs are rendered and displayed back to the user.

```
visualize <- function(pulled_data, input){
  model_output <- model(pulled_data, input)

  active_data <- model_output[[1]]
  grid <- model_output[[2]]

  p1 <- ggplot(active_data,
               aes(datetime)) +
    geom_point(aes(y = close), color = "red") +
    scale_x_datetime() +
    geom_line(aes(y = close), color = "blue") +
    geom_line(aes(y = pred), data = grid,
              color = "green", size = 1)

  p2 <- ggplot(active_data, aes(datetime, resid)) +
    geom_ref_line(h = 0) +
```

```
    geom_point()

  p3 <- ggplot(active_data, aes(resid)) +
    geom_histogram(aes(y = ..density..), fill = 'red', alpha = 0.5) +
    geom_density(color = 'blue')

  grid.arrange(p1,p2,p3, ncol=1)
}
```