Solar power generation analysis

Mikhail Davydov

7/28/2021

Introduction

This project is the second part of a HarvardX: PH125.9x Capstone. The objective is to show data analysis techniques such as data cleaning, machine learning and data visualization. The primary steps in the project are:

- Data description and visualization
- Cleaning the data
- Training machine learning models
- Results discussion

Data set

The used data set has been taken at https://www.kaggle.com/anikannal/solar-power-generation-data. According to the description provided by the author, this data has been gathered at two solar power plants in India over 34 days. Original files from Kaggle have been merged into one data set. date_time feature has been separated into p_date and p_time. Then we have split the dataset into training (80%) and test set (20%). I have chosen this split because we need lots of data to train a model, and 20% of a data set is enough to evaluate it, given this enormous dataset we have.

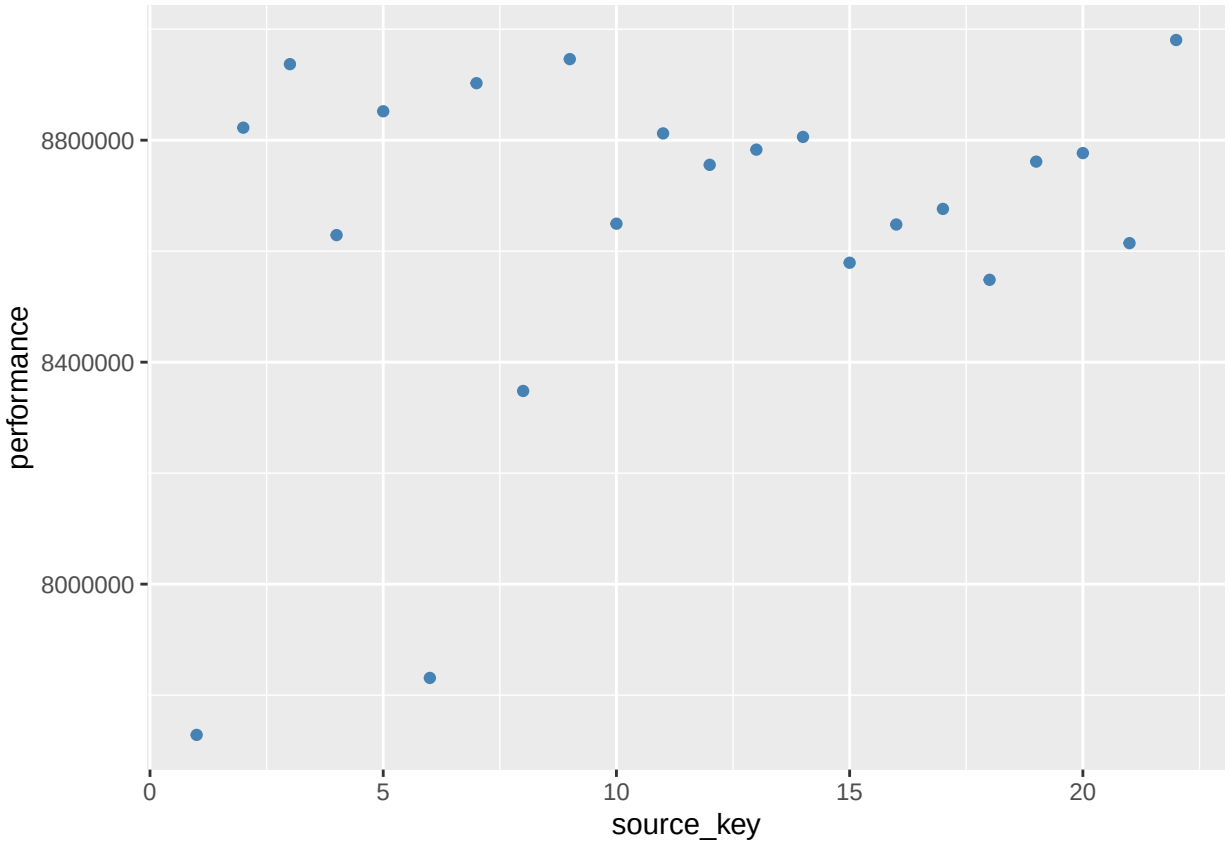
```
##       p_date   p_time plant_id source_key dc_power ac_power daily_yield
## 4 2020-05-15 00:45:00  4135001          1        0        0           0
## 5 2020-05-15 01:00:00  4135001          1        0        0           0
## 6 2020-05-15 01:15:00  4135001          1        0        0           0
## 7 2020-05-15 01:30:00  4135001          1        0        0           0
## 8 2020-05-15 01:45:00  4135001          1        0        0           0
## 9 2020-05-15 02:00:00  4135001          1        0        0           0
##   total_yield ambient_temperature module_temperature irradiation
## 4     6259559            24.84613           22.36085           0
## 5     6259559            24.62153           22.16542           0
## 6     6259559            24.53609           21.96857           0
## 7     6259559            24.63867           22.35293           0
## 8     6259559            24.87302           23.16092           0
## 9     6259559            24.93693           23.02611           0
```

The variables in the dataset are:

- p_date, p_time - date and time observation had been made.
- plant_id - solar power plant id. We have two of them.
- source_key - is the inverter id. Each inverter has a bunch of solar panels. I am going to call it a solar panel for the sake of convenience.
- ac_power, dc_power - show the amount of ac, dc power being produced at 15-minute intervals.
- daily_yield is a cumulative sum of power generated on that day until that point in time.
- total_yield is the total yield for the inverter until that point in time.
- ambient_temperature is the outside temperature at the power plant.
- module_temperature is the temperature of a solar panel.
- irradiation is the amount of solar irradiation for the 15-minute interval. The challenge is to predict at which power plant an observation has been made.

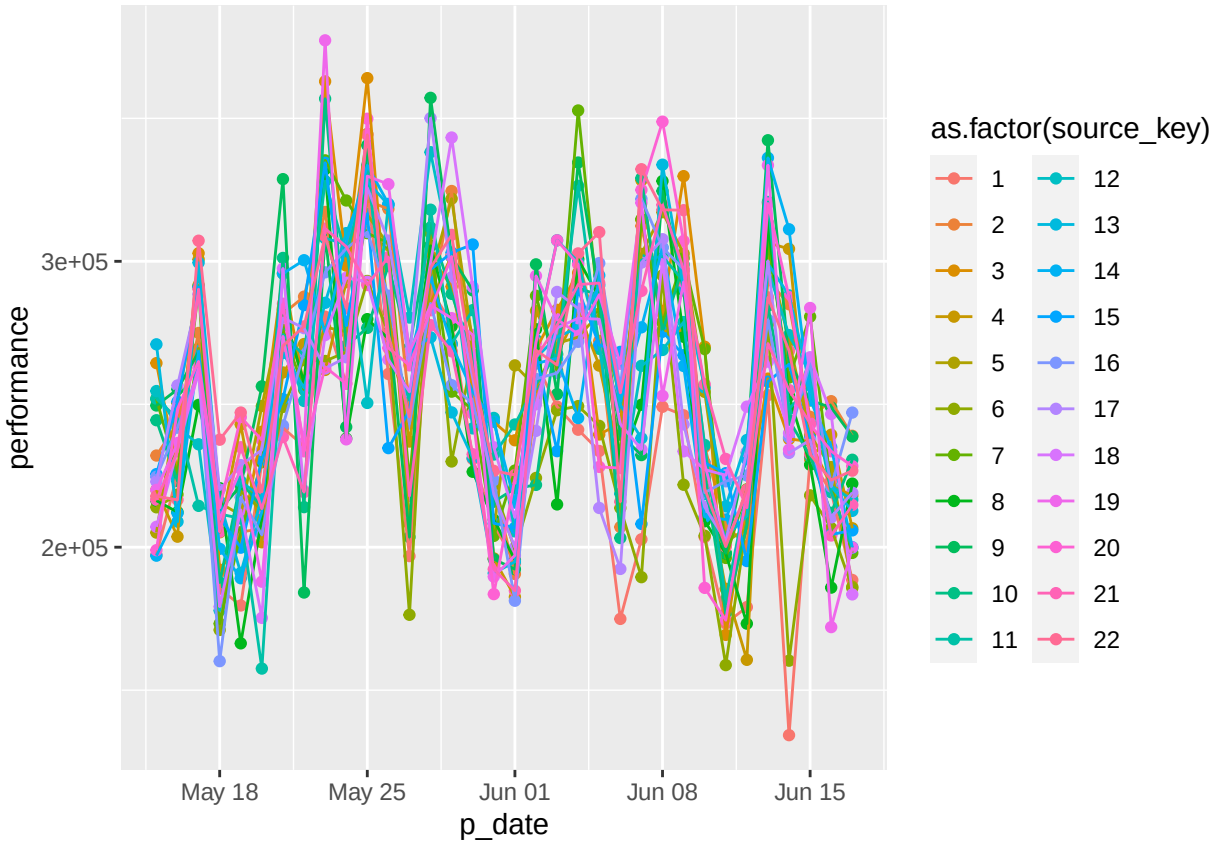
Analysis

Let's take a look at the first plant. Do the solar panels have the same performance?



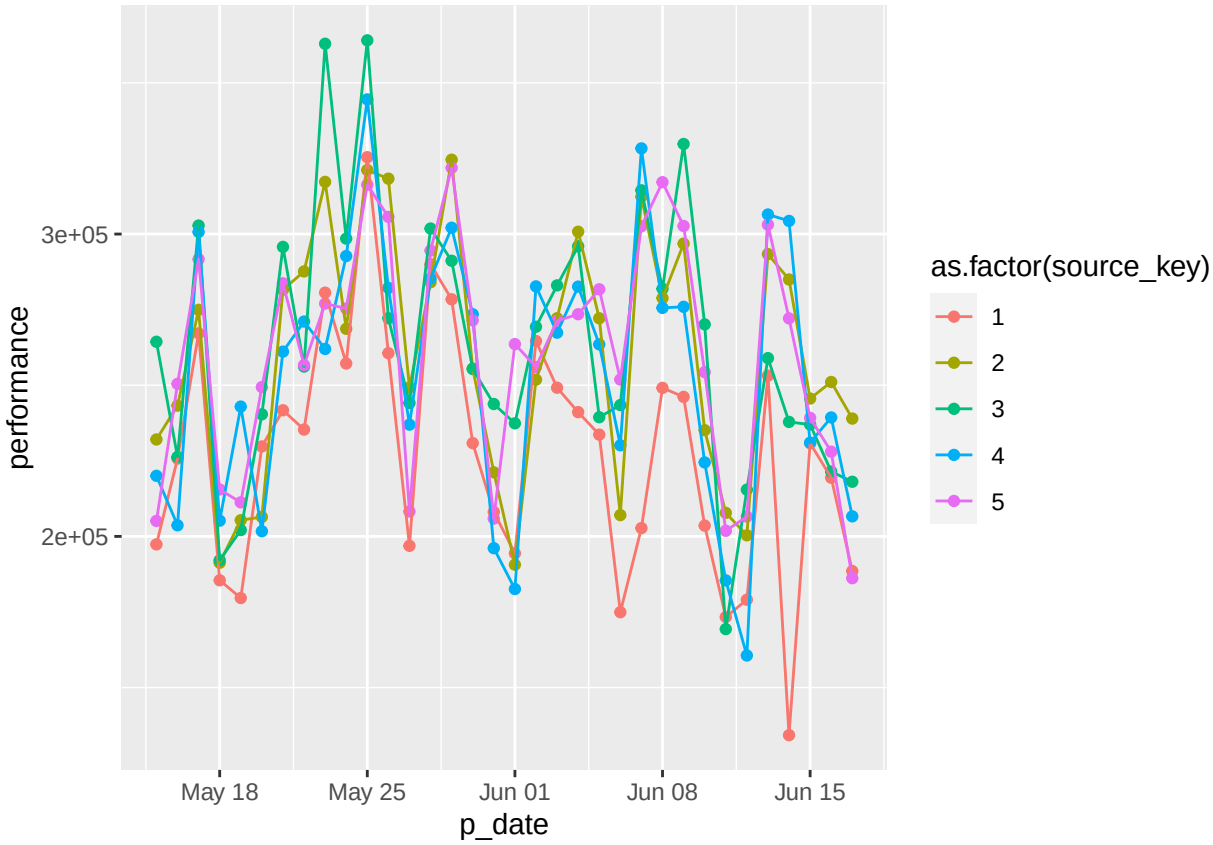
We have two outliers.

Shall we take a look at the performance over time.



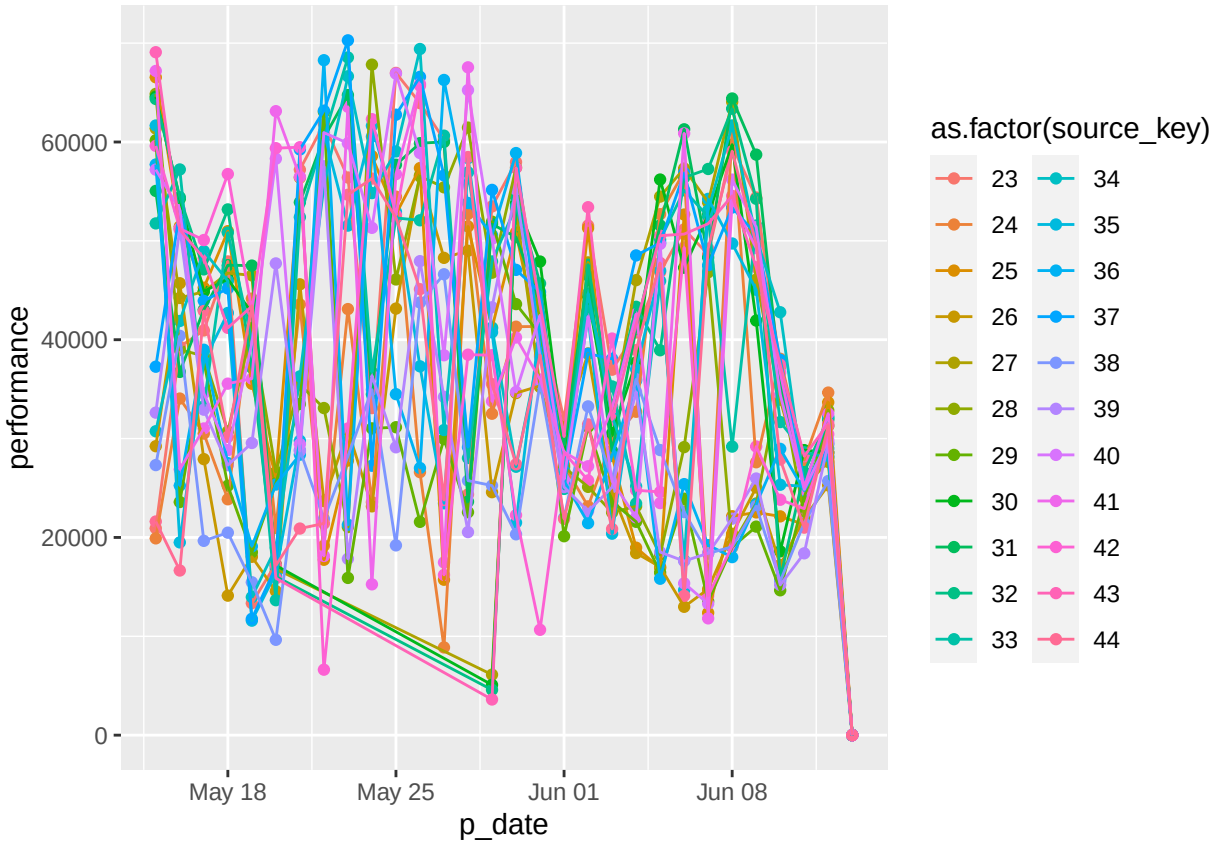
The trend is present, but it is too many panels on the plot.

How about taking a subset by source_key



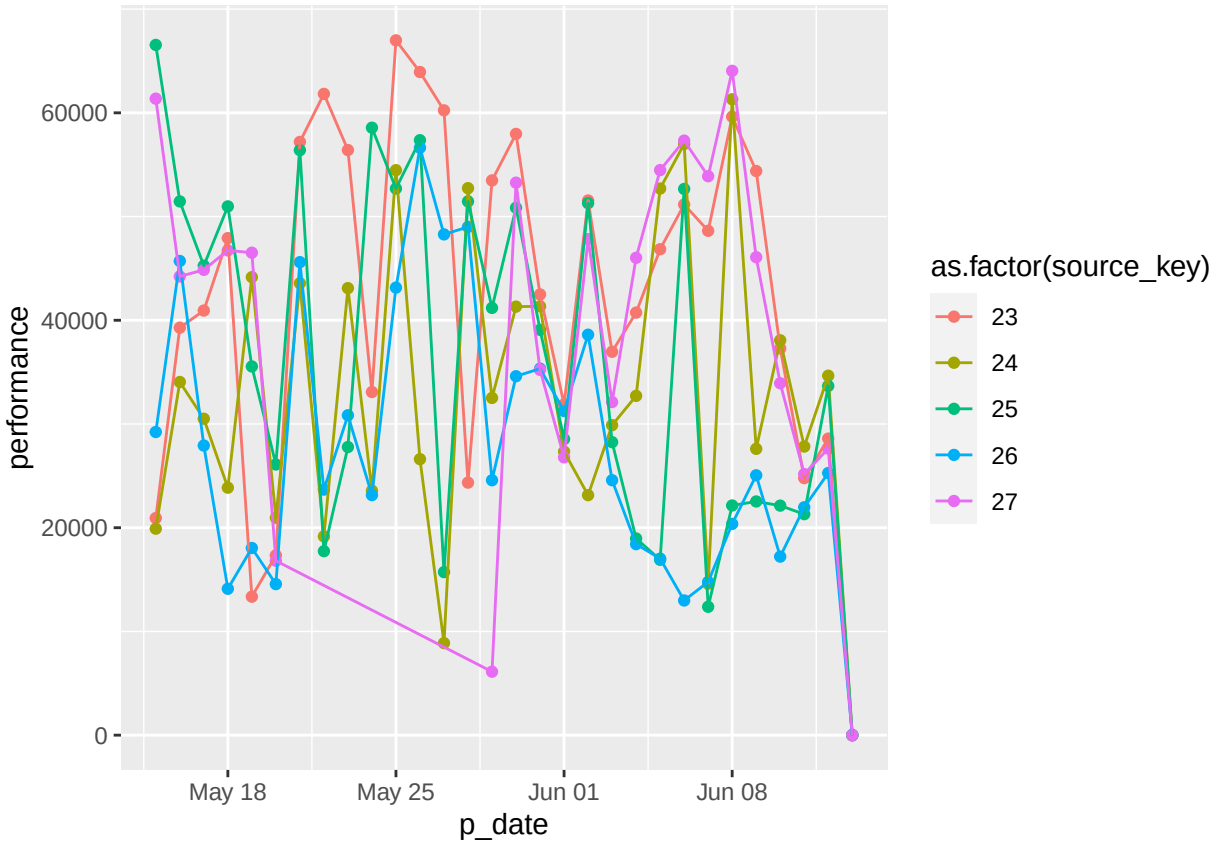
We can see that some panels work better than others. Maybe because solar panels have a different angle to the sun, or variation is the norm regarding solar panel manufacturing. Or maybe new panels work better than old ones.

Would you like to take a look at the second plant.



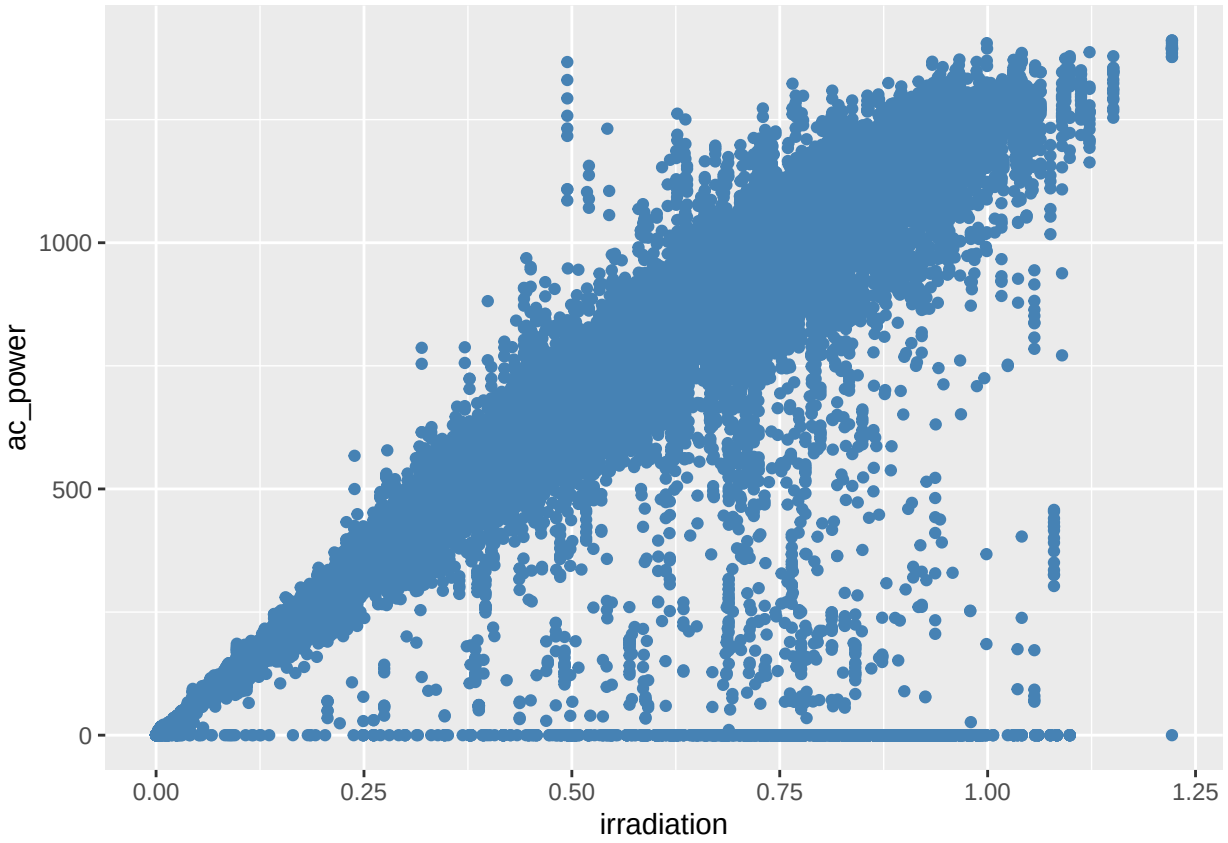
A lot messier than the data from the first plant. Let's dive into it.

The plot for the first five panels is



Maybe some panels didn't generate power during some periods. That is why the lines on the plot don't move along. Note, panel #27 doesn't have any observations from May 20th through May 29th.

What is the correlation between ac_power and irradiation.



We can see the trend, but at the same time, there are lots of outliers. Why not to take a look at some of them.

```
head(train_set %>%
       filter(irradiation > 0.6 & ac_power > 0 & ac_power < 250))
```

```
##       p_date   p_time plant_id source_key  dc_power ac_power daily_yield
## 1 2020-06-14 11:00:00  4135001          1 2155.3750 210.9000    2393.625
## 2 2020-06-07 12:45:00  4135001         12 1897.5714 185.0143    4520.000
## 3 2020-05-16 10:30:00  4136001         23  238.7133 232.7467    2518.667
## 4 2020-05-19 10:15:00  4136001         23  154.9600 151.4467     282.600
## 5 2020-05-21 10:00:00  4136001         23  228.6267 223.3800     365.600
## 6 2020-05-24 14:15:00  4136001         23  142.9133 139.6867    2481.200
##   total_yield ambient_temperature module_temperature irradiation
## 1   6466056.6            27.08107           47.27521   0.6406613
## 2   7190176.0            29.44681           57.74181   0.9985398
## 3   2435730.7            33.75172           59.49848   0.8944379
## 4    981715.0            28.93707           41.52364   0.7757807
## 5    986430.4            30.53005           52.76717   0.8282022
## 6   2495447.2            37.07135           57.42612   0.7703641
```

You may notice a total_yield drop at 2020-05-19 at 10:15:00. Earlier observations for source_key #23 give you a greater total_yield, which shouldn't occur by design.

Taking a closer look

```
head(train_set %>%
       filter(p_date == ymd("2020-05-19") & p_time > hms(60 * 60 * 9) & source_key == 23), 10)
```

```
##        p_date   p_time plant_id source_key dc_power ac_power daily_yield
## 1  2020-05-19 09:15:00  4136001         23 431.3500 423.4429    1080.000
## 2  2020-05-19 09:30:00  4136001         23 351.1133 345.1533    1177.267
## 3  2020-05-19 09:45:00  4136001         23 344.4600 338.6333    1264.400
## 4  2020-05-19 10:00:00  4136001         23 373.7600 367.3533    1351.333
## 5  2020-05-19 10:15:00  4136001         23 154.9600 151.4467     282.600
## 6  2020-05-19 11:15:00  4136001         23   0.0000   0.0000    1508.000
## 7  2020-05-19 11:30:00  4136001         23   0.0000   0.0000    1508.000
## 8  2020-05-19 11:45:00  4136001         23   0.0000   0.0000    1508.000
## 9  2020-05-19 12:00:00  4136001         23   0.0000   0.0000    1508.000
## 10 2020-05-19 12:15:00  4136001         23   0.0000   0.0000    1508.000
##    total_yield ambient_temperature module_temperature irradiation
## 1      2453952            27.38631           34.54598   0.2753400
## 2      2454049            27.29954           32.59585   0.2372464
## 3      2454136            27.27851           31.88233   0.2245895
## 4      2454223            27.56581           32.14257   0.2635695
## 5       981715            28.93707           41.52364   0.7757807
## 6      2454380            30.96853           50.26547   0.9542262
## 7      2454380            31.45509           51.00280   0.9315328
## 8      2454380            31.76743           50.95130   0.9472872
## 9      2454380            32.06115           51.70768   0.9674635
## 10     2454380            32.56087           52.55826   0.9228362
```

Not only total_yield has dropped but also daily_yield. Remember, these numbers are cumulative. The panel turned off later in the day. This type of total_yield drop is all over the dataset. We need to cut these observations since they represent nothing but noise. To do so, we are going to compare each observation's total_yield with the oldest total_yield for that particular source_key. We can do so because total_yield drops way below the oldest one.

```
# This function cuts troublesome observations out of a data frame. malf_list is a result of a sapply() f
cut_malf <- function(malf_list, data_set){
  # Creating a variable
  malf_vec <- c(0)
  # Transforming a list of vectors in one huge vector
  for (xx in 1:length(malf_list)){
    malf_vec <- c(malf_vec, malf_list[[xx]])
  }
  # We need to delete the first value from the vector, since it had been created to create the vector i
  malf_vec <- malf_vec[2: length(malf_vec)]
  # Deleting broken observations from the data set.
  data_set %>%
    slice(which(!malf_vec))
}

# This function deletes observations with total_yield drop. The name stands for "delete total yield drop
del_ttl_yld_drp <- function(data_set){
  # Creating a list of vectors to tackle total_yield drop. Every vector matches with source_key.
  malf_list1 <- sapply(unique(data_set$source_key), function(xx){
    # Taking the initial total_yield
    init_yield <- head(data_set %>%
            filter(source_key == xx), 1)$total_yield
```

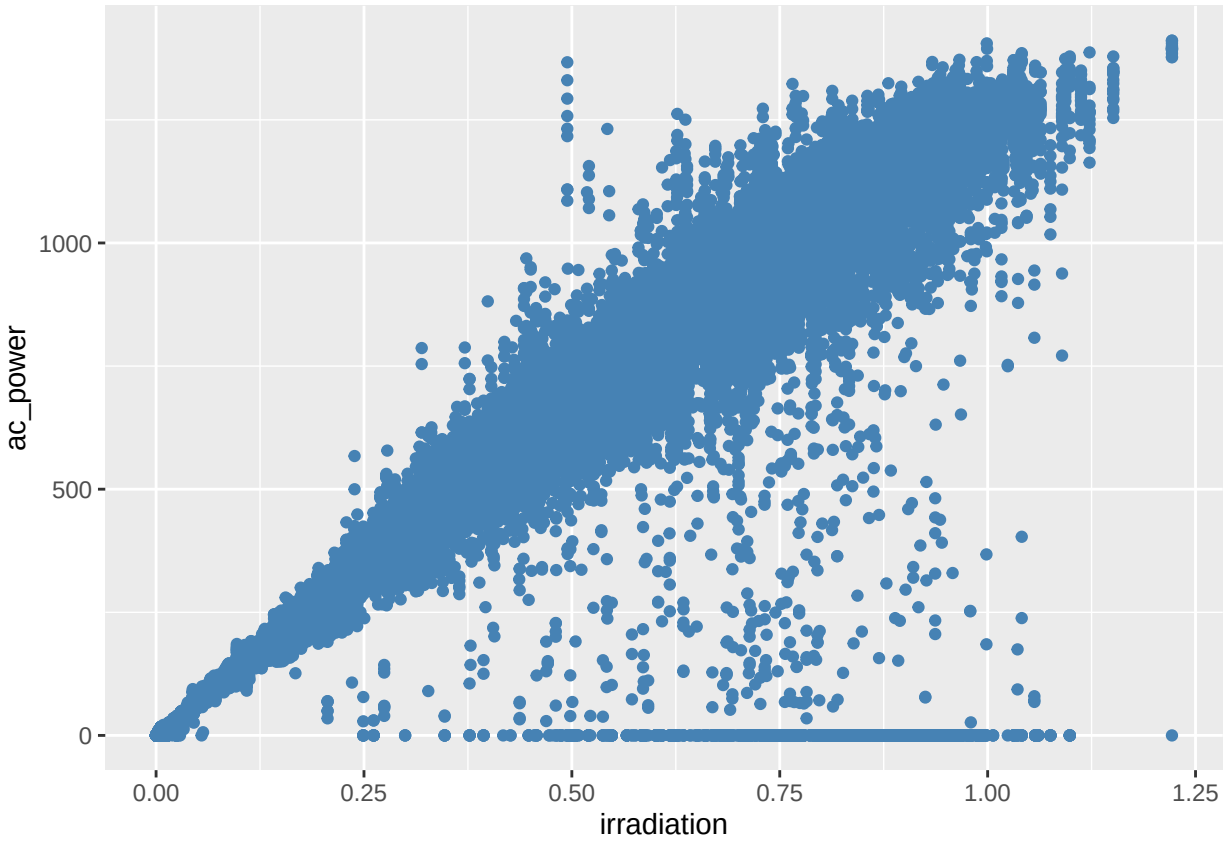
```
    # Comparing total_yield with initial yield we have got earlier
    data_set %>%
      filter(source_key == xx) %>%
      mutate(malf = ifelse(total_yield < init_yield, 1, 0)) %>%
      pull(malf)
  })
  # Cut observations with wrong total_yield. And return clean data set as a result of the function
  cut_malf(malf_list1, data_set)
}
```

We will clean the data set and, at the same time, will see how many observations are damaged.

```
# Number of observations before cleaning
n_before <- nrow(train_set)
# Cutting total yield drops
train_set <- del_ttl_yld_drp(train_set)
# The number of damaged observations is
n_before - nrow(train_set)
```

```
## [1] 1361
```

Enough to make a mess.



The plot looks better, but we still have some issues here.

```
head(train_set %>%
       filter(irradiation > 0.6 & ac_power > 0 & ac_power < 250))
```

```
##       p_date   p_time plant_id source_key  dc_power ac_power daily_yield
## 1 2020-06-14 11:00:00  4135001          1 2155.3750 210.9000    2393.625
## 2 2020-06-07 12:45:00  4135001         12 1897.5714 185.0143    4520.000
## 3 2020-05-16 10:30:00  4136001         23  238.7133 232.7467    2518.667
## 4 2020-05-24 14:15:00  4136001         23  142.9133 139.6867    2481.200
## 5 2020-05-16 14:00:00  4136001         24  134.4333 131.6733    3873.467
## 6 2020-05-16 14:15:00  4136001         24  146.2267 143.1933    3942.400
##   total_yield ambient_temperature module_temperature irradiation
## 1     6466057            27.08107           47.27521   0.6406613
## 2     7190176            29.44681           57.74181   0.9985398
## 3     2435731            33.75172           59.49848   0.8944379
## 4     2495447            37.07135           57.42612   0.7703641
## 5  1215285823            35.17039           48.40738   0.6341718
## 6  1215285892            35.07013           47.87681   0.7138214
```

I pick 3rd observation to explore.

```
head(train_set %>%
       filter(p_date == ymd("2020-05-16") & p_time > hms(60 * 60 * 9) & source_key == 23), 10)
```

```
##        p_date   p_time plant_id source_key  dc_power  ac_power daily_yield
## 1  2020-05-16 09:15:00  4136001         23  946.6733  925.9067    1373.400
## 2  2020-05-16 09:30:00  4136001         23  996.3857  974.2714    1602.500
## 3  2020-05-16 09:45:00  4136001         23 1048.2200 1024.6733    1849.600
## 4  2020-05-16 10:00:00  4136001         23  865.7733  846.1667    2079.600
## 5  2020-05-16 10:15:00  4136001         23 1138.4067 1111.4133    2323.600
## 6  2020-05-16 10:30:00  4136001         23  238.7133  232.7467    2518.667
## 7  2020-05-16 11:00:00  4136001         23    0.0000    0.0000    2525.000
## 8  2020-05-16 11:15:00  4136001         23    0.0000    0.0000    2525.000
## 9  2020-05-16 11:30:00  4136001         23    0.0000    0.0000    2525.000
## 10 2020-05-16 11:45:00  4136001         23    0.0000    0.0000    2525.000
##    total_yield ambient_temperature module_temperature irradiation
## 1      2434585            31.13130           51.06770   0.7055704
## 2      2434814            31.73716           54.30058   0.7526609
## 3      2435062            32.11526           56.26471   0.7908360
## 4      2435292            32.42292           56.88568   0.8210549
## 5      2435536            32.97431           58.69473   0.8508264
## 6      2435731            33.75172           59.49848   0.8944379
## 7      2435737            34.15109           61.63474   0.9198568
## 8      2435737            34.83831           62.84589   0.9303221
## 9      2435737            34.83091           63.09210   0.9810387
## 10     2435737            35.47821           63.82718   0.9896192
```

This observation had been made right before downtime. It turns out measurements right before or after pause often have low productivity. My guess is - a panel didn't work the whole 15-minute interval, so the observation is not representative. We are going to eliminate peripherals, but we are going to do it smartly. We will cut only showing an anomaly in the correlation between irradiation and ac_power because lots of peripheral observations have been made at the beginning or the end of the day. First, we mark peripherals with "1" in the corresponding variable. The rest of the observations are "0".

```
# This function creates a variable showing a switch from zero to some number or vice versa in the ac_po
add_swtch_vr_ac <- function(data_set){
  # Creating a feature representing switches from zero to non-zero number in ac_power. First non-zero o
```

```
  data_set <- data_set %>%
    # Creating a feature showing whether ac_power has zero or non-zero value.
    mutate(swtch_f_z_ac = ifelse(ac_power == 0, 1, 2)) %>%
    # Comparing each observation with its previous one. If the two observations are the same - the sum
    mutate(swtch_f_z_ac = (swtch_f_z_ac + c(1, swtch_f_z_ac[1:(n()-1)])) %% 2) %>%
    # If current ac_power == 0 and in the previous observation ac_power has a non-zero value, the line
    mutate(swtch_f_z_ac = ifelse(ac_power == 0, 0, swtch_f_z_ac)) %>%
    relocate(swtch_f_z_ac, .after = ac_power)
  # Creating a feature representing switches from a non-zero value to zero in ac_power. Last non-zero o
  data_set <- data_set %>%
    # Creating a feature showing weather ac_power has zero or non-zero value.
    mutate(swtch_f_n_ac = ifelse(ac_power == 0, 1, 2)) %>%
    # Comparing each observation with its next one. If the two observations are the same - the sum will
    mutate(swtch_f_n_ac = (swtch_f_n_ac + c(swtch_f_n_ac[2:(n())], 0)) %% 2) %>%
    # If current ac_power == 0 and in the next observation ac_power has a non-zero value, the line abov
    mutate(swtch_f_n_ac = ifelse(ac_power == 0, 0, swtch_f_n_ac)) %>%
    relocate(swtch_f_n_ac, .after = ac_power)
  # Combine switches from zero to non-zero with switches from non-zero to zero.
  data_set %>%
    mutate(swtch_ac = swtch_f_n_ac + swtch_f_z_ac) %>%
    relocate(swtch_ac, .after = ac_power) %>%
    select(-swtch_f_n_ac, -swtch_f_z_ac)
}
# Marking peripheral observations
train_set <- add_swtch_vr_ac(train_set)
```

The next thing we are going to do is trying to estimate if ac_power is reasonable for irradiation a solar panel gets. To do that, we will bin irradiation and compute the mode value of ac_power for each bin. But first, we need to define the mode function since R doesn't have one.

```
# Finding mode. https://www.tutorialspoint.com/r/r_mean_median_mode.htm .According to https://courses.e
getmode <- function(v) {
  # Getting unique values from the vector v
  uniqv <- unique(v)
  # Getting the most common value
  uniqv[which.max(tabulate(match(v, uniqv)))]
}
# This function bins irradiation feature and computes mode of ac_power and dc_power for each bin
get_pwr_md_irr <- function(data_set, n_bins){
  # Binning irradiation feature, so we can find the mode of ac_power and dc_power for each bin
  data_set <- data_set %>%
    mutate(irrad_bin = round(irradiation * n_bins))

  # Finding the mode of ac_power and dc_power
  data_set %>%
    # Deleting zeros, so they don't spoil the mode. We don't need them anyway
    slice(which(ac_power != 0 & dc_power != 0)) %>%
    group_by(irrad_bin) %>%
    summarise(irrad_mode_ac = getmode(ac_power), irrad_mode_dc = getmode(dc_power))
}
```

We are ready to define a function that cuts peripheral observations with low productivity.

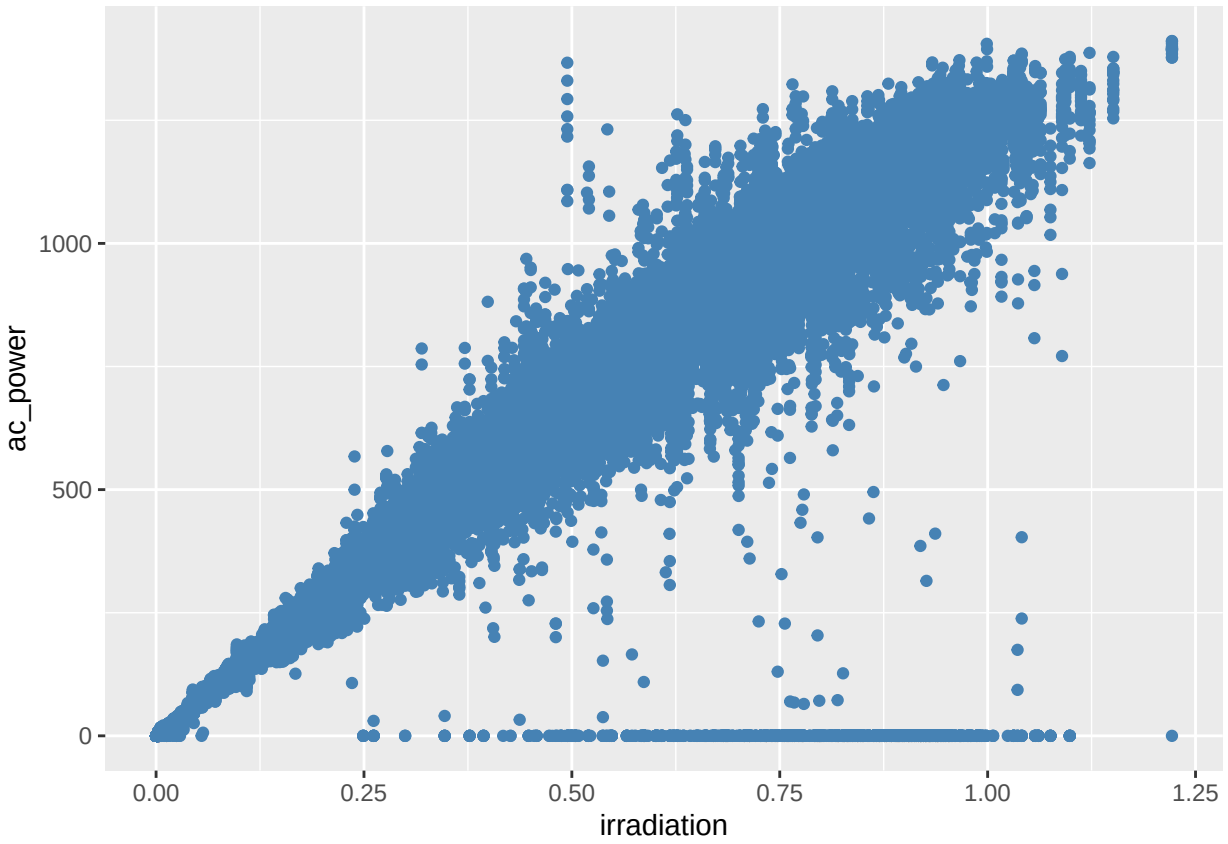
```
cut_swtch_malf <- function(data_set, irr_bin, num_irr_bin){
    # Binning irradiation feature, so we can join the data set with irr_bin
```

```
  data_set <- data_set %>%
    mutate(irrad_bin = round(irradiation * num_irr_bin))
  # Adding mode of ac_power and dc_power to the data set and find the distance between ac_power and the
  data_set <- left_join(data_set, irr_bin, by = "irrad_bin") %>%
    mutate(mode_dist_ac = irrad_mode_ac - ac_power,
           mode_dist_dc = irrad_mode_dc - dc_power)



  # Cutting off observations where c_power significantly differs from its mode given irradiation and th
  data_set %>%
    filter(irradiation > 0) %>%
    # We measure the significance of the difference as sqrt() function because when irradiation is low,
    slice(which(!(abs(mode_dist_ac) > (sqrt(irradiation) * 400) & swtch_ac != 0))) %>%
    # Deleting transitory features.
    select(-mode_dist_ac, -mode_dist_dc, -irrad_mode_ac, -irrad_mode_dc)
}
```

Let's see what we have done.

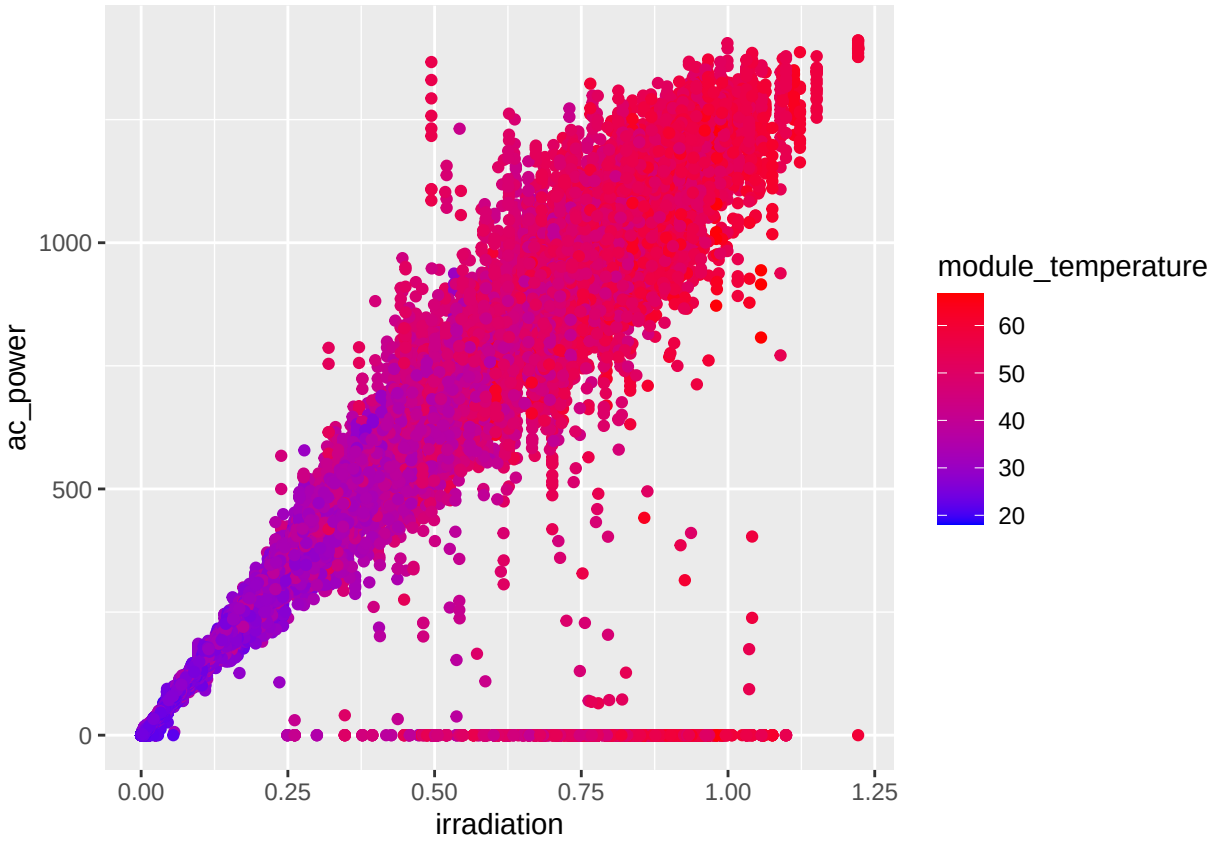
```
# Number of observations before cleaning
n_before <- nrow(train_set)
# Number of observations where irradiation == 0
n_zeros <- length(which(train_set$irradiation == 0))

# This number defines the number(not exact) of irradiation bins.
num_irr_bin <- 18
# Getting ac_power mode for each irradiation bin
irr_bin <- get_pwr_md_irr(train_set, num_irr_bin)
# Cutting peripheral observations with low productivity
train_set <- cut_swtch_malf(train_set, irr_bin, num_irr_bin)

# The number of damaged observations is
n_before - n_zeros - nrow(train_set)
```

```
## [1] 285
```



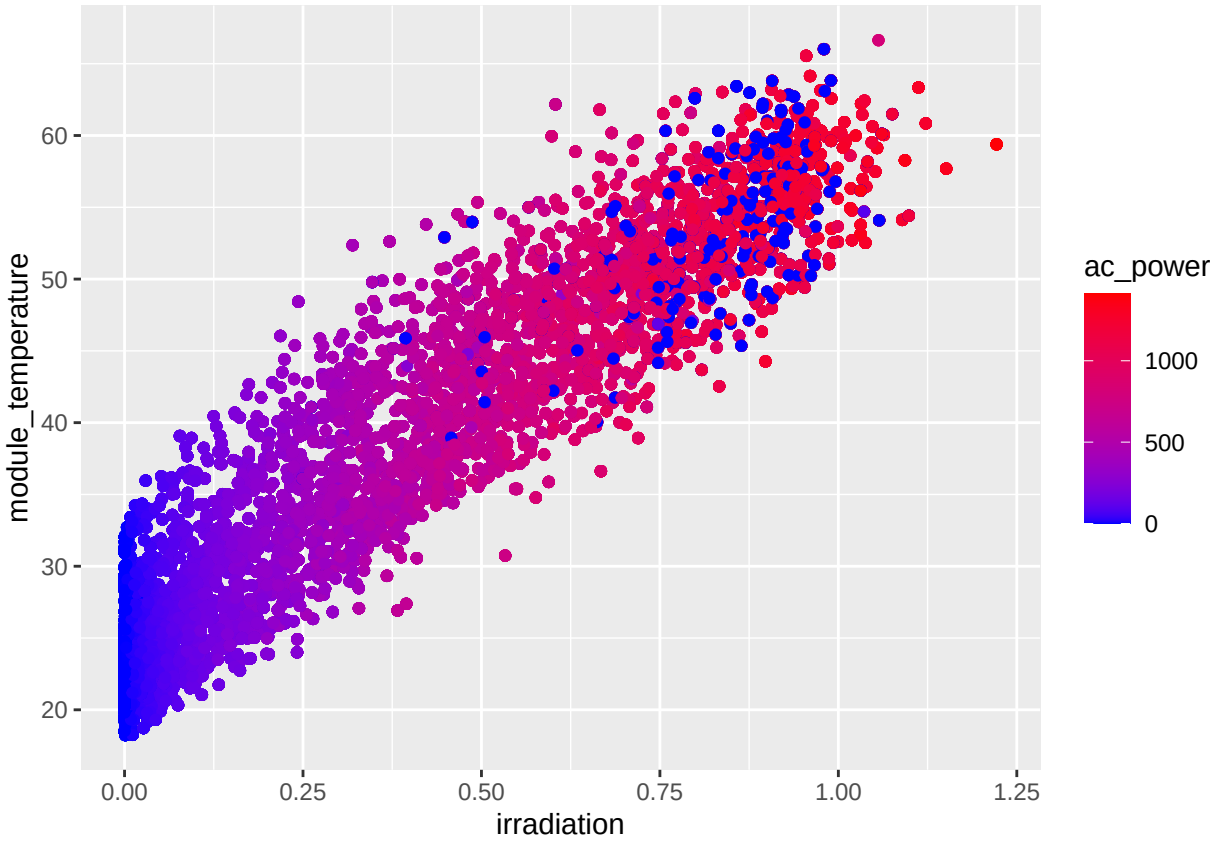
Lot better. Given that we cut so few values.

What else can we find? Let's throw some colours on the plot above.



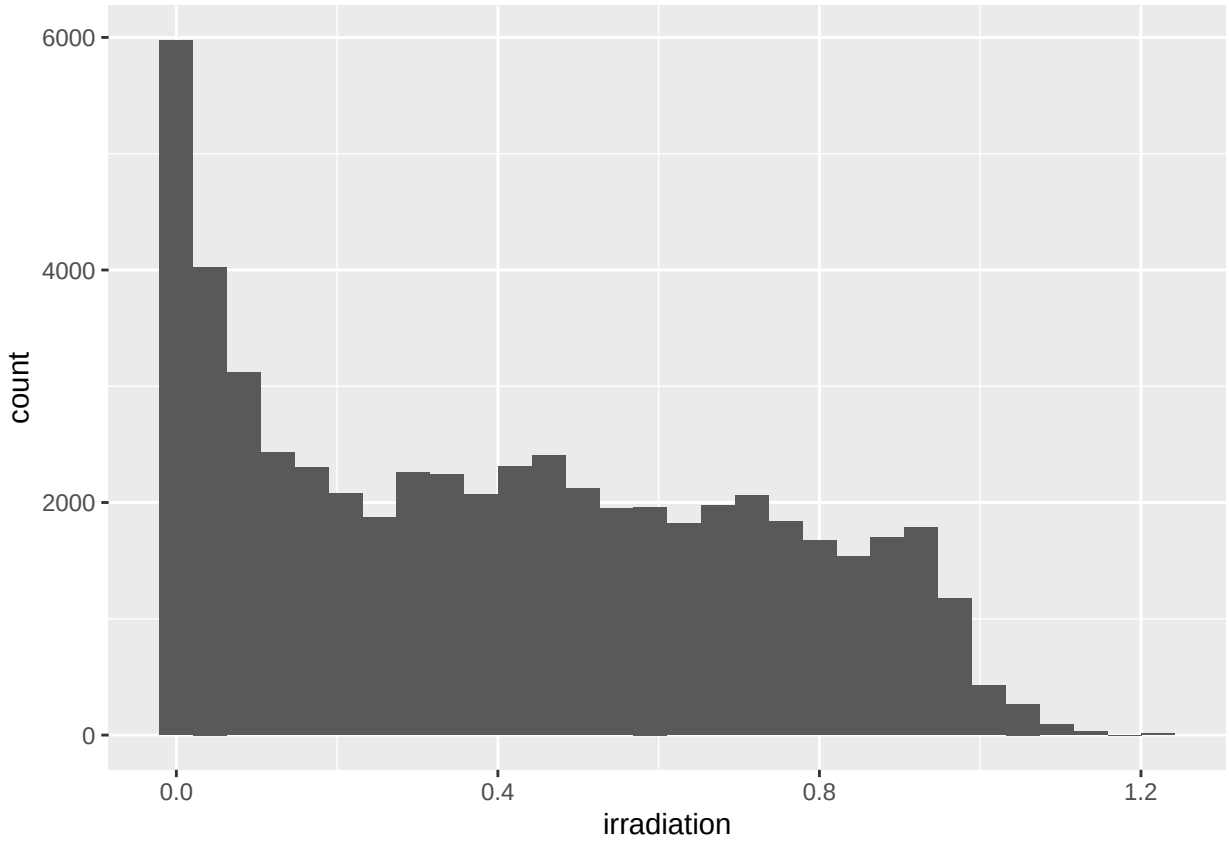
Outliers tend to have high module_temperature.

At first, I thought panels are overheating and stop working. It turns out not to be true. Watch it from another angle.

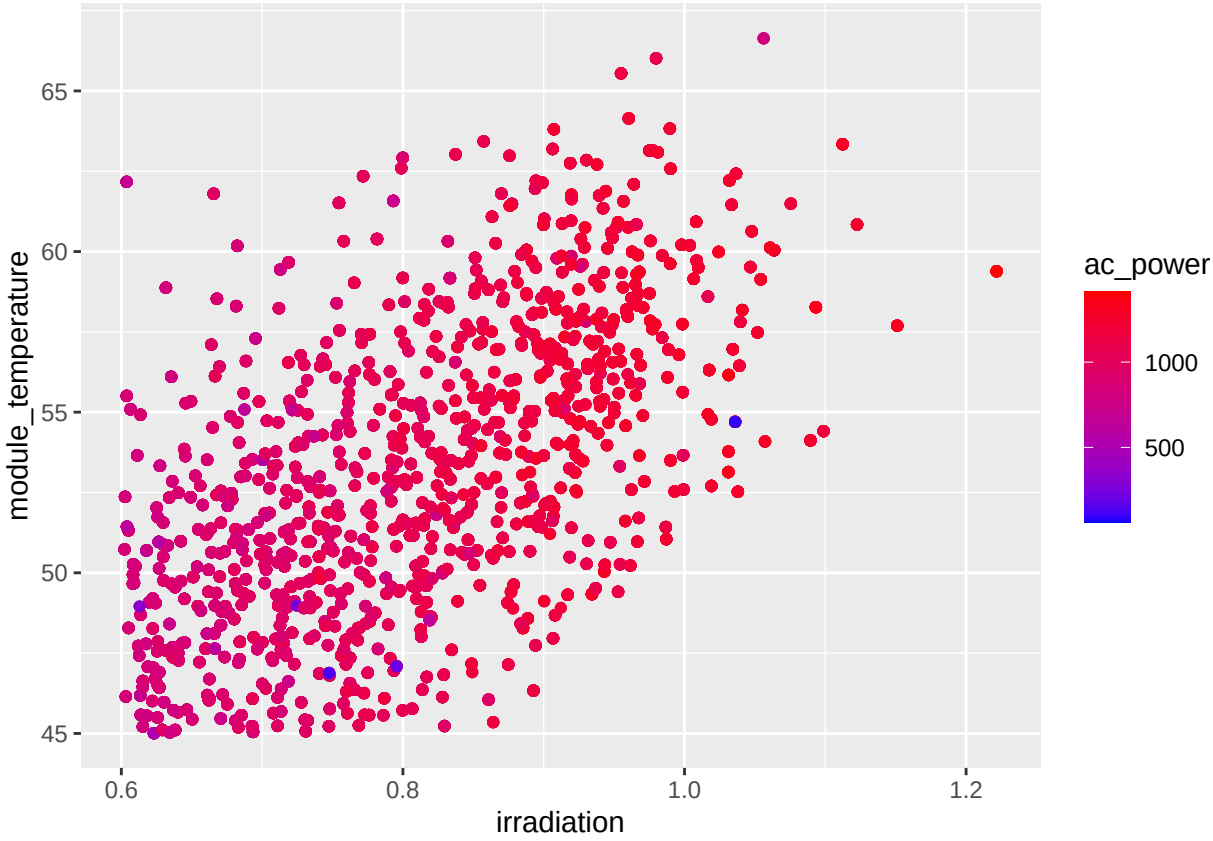


If my assumption about overheating was true, then we would have an "overheating" line with the majority of the dots being blue above it - which is not the case. But for some reason, ac_power tends to drop at high irradiation values.

One may think we don't have many observations where irradiation is in the segment from 0.15 to 0.45 - that is why we don't have many outliers there. It is not true either.

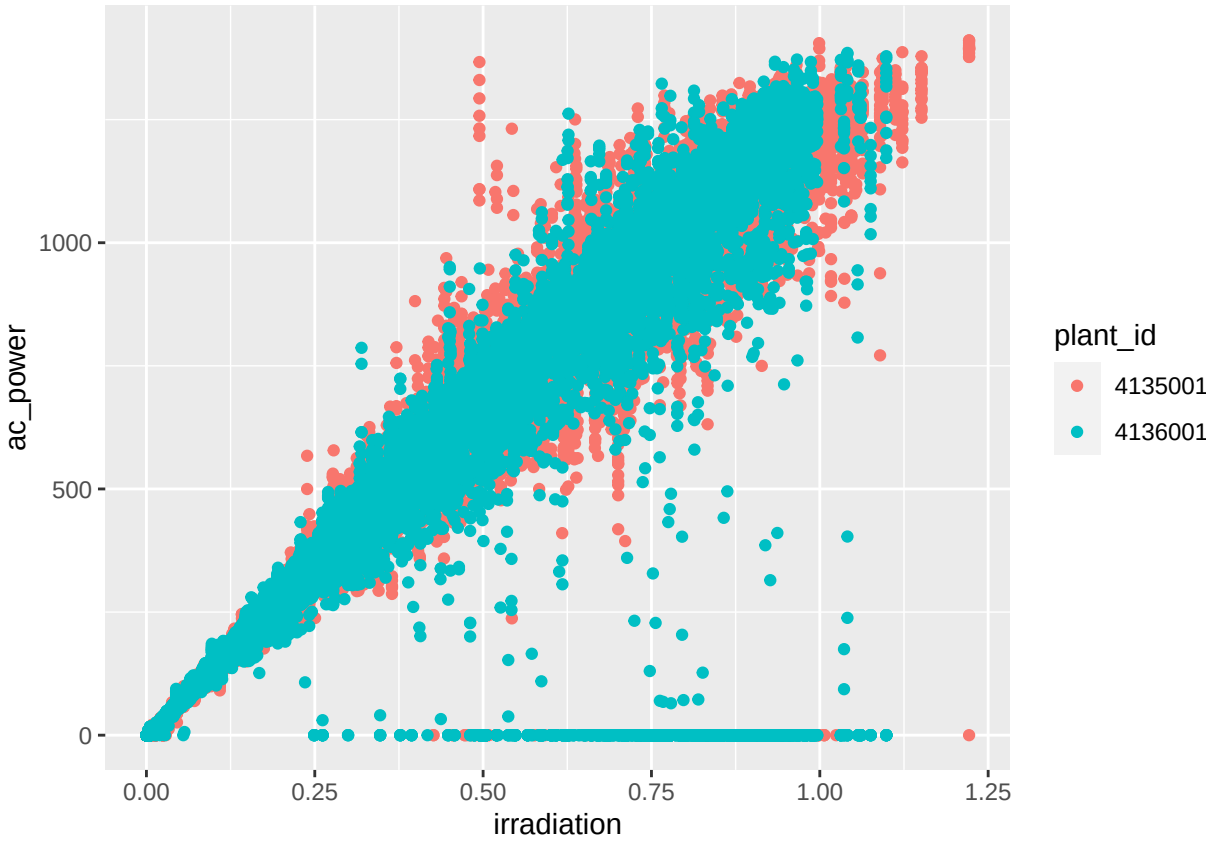


The same plot with a scope on high irradiation and non-zero ac_power.



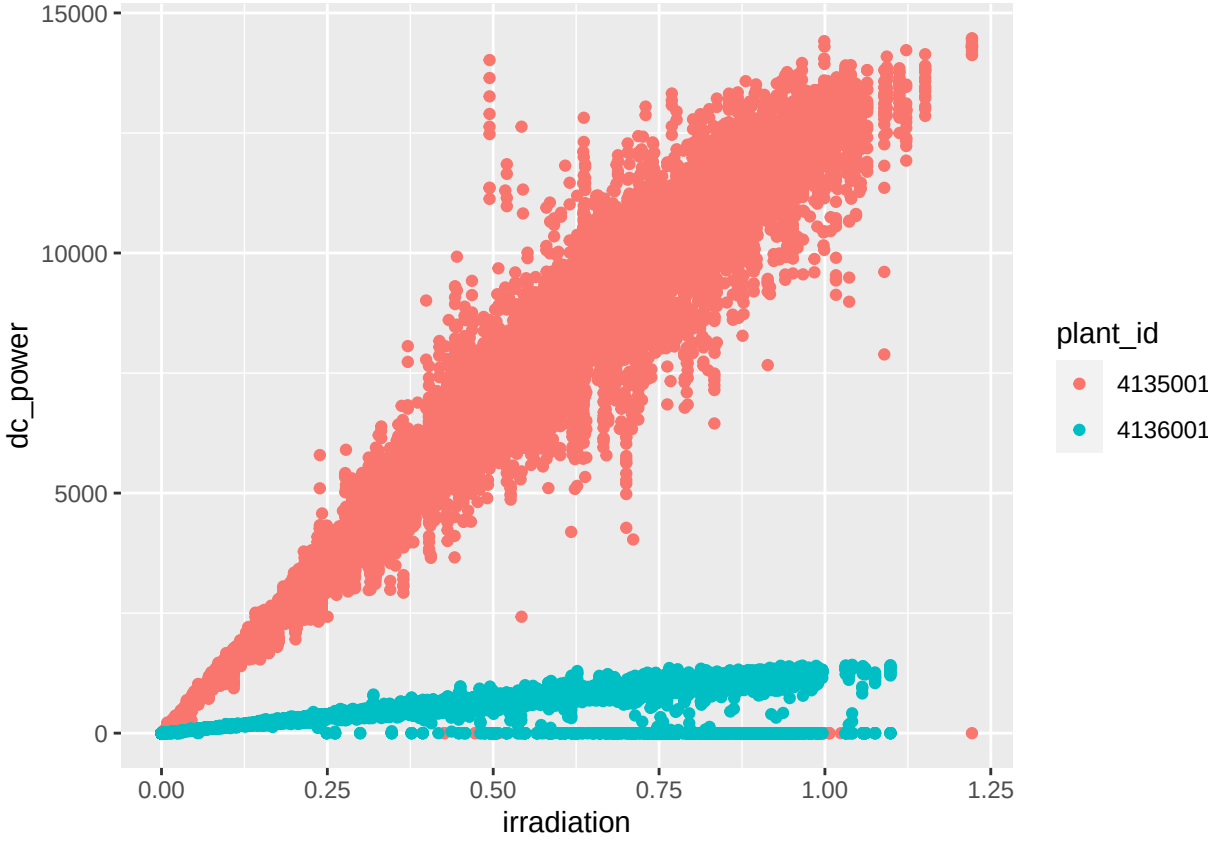
Non-zero ac_power values are present. I don't see any sound reason to delete the rest of the outliers. If you disagree, you can cut them with functions I have left for you. You will find them commented in the code.R file. I am not going to explain how they work since the way they work is pretty much the same as irr_bin() and cut_swtch_malf().

By the way, notice most of the outliers are from the second power plant.



At this point, I see the data as clean.

Recall, we are trying to predict at which power plant an observation has been made. Look at the plot.



Power plants produce different amounts of dc_power given the same irradiation. So we can use these two variables to make a prediction. I see two possible ways to solve this classification problem. We can set a boundary as a line, or we can predict based on the values neighbours have. We will implement three algorithms: lda, knn and gemLoess

First, we clean the test set the way we have done the training set.

```
test_set <- add_swtch_vr_ac(test_set)
test_set <- del_ttl_yld_drp(test_set)
test_set <- cut_swtch_malf(test_set, irr_bin, num_irr_bin)
```

Then we cut features we don't need.

```
train_set <- train_set %>%
  select(plant_id, dc_power, irradiation)

test_set <- test_set %>%
  select(plant_id, dc_power, irradiation)
```

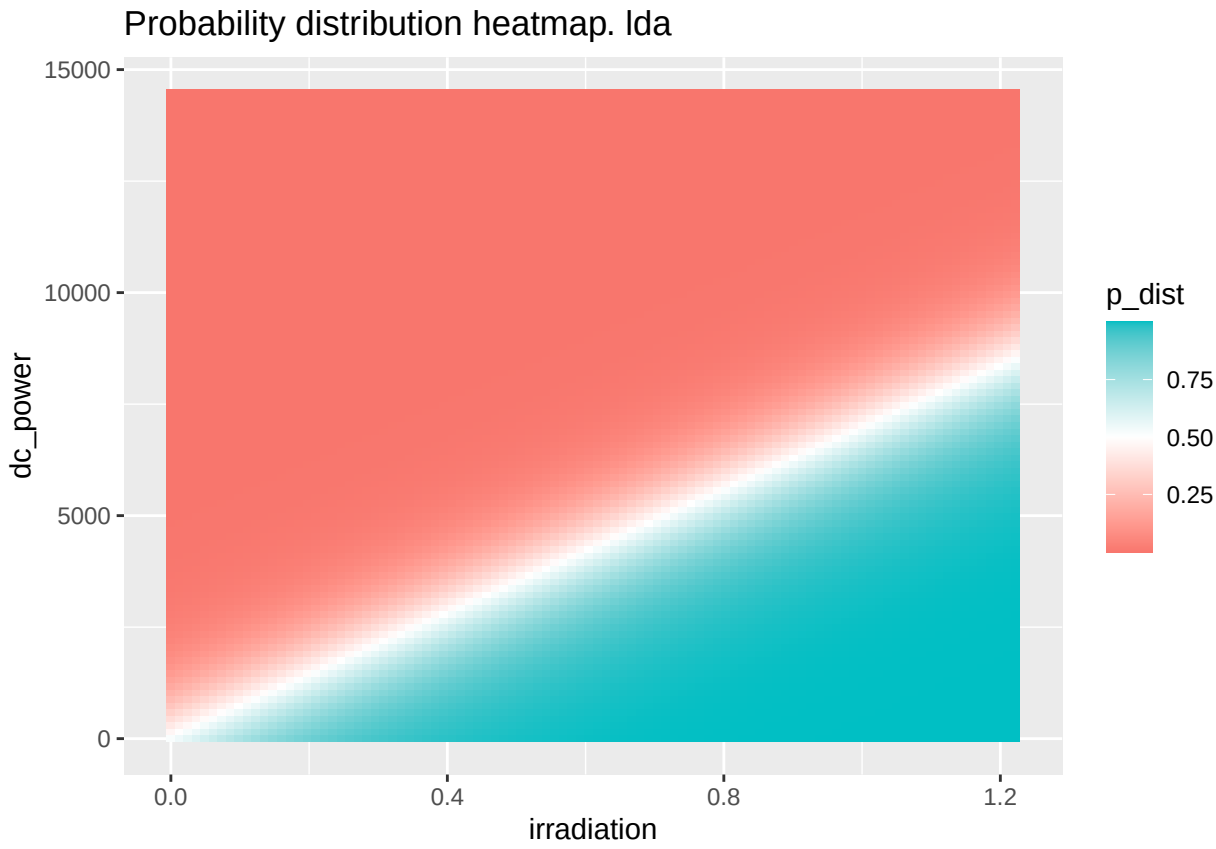
Results

Training Linear discriminant analysis

```
# lda
model_lda <- train(plant_id ~ ., data = train_set, method = "lda")
```

We can make a probability distribution heatmap to see what the algorithm predicts.

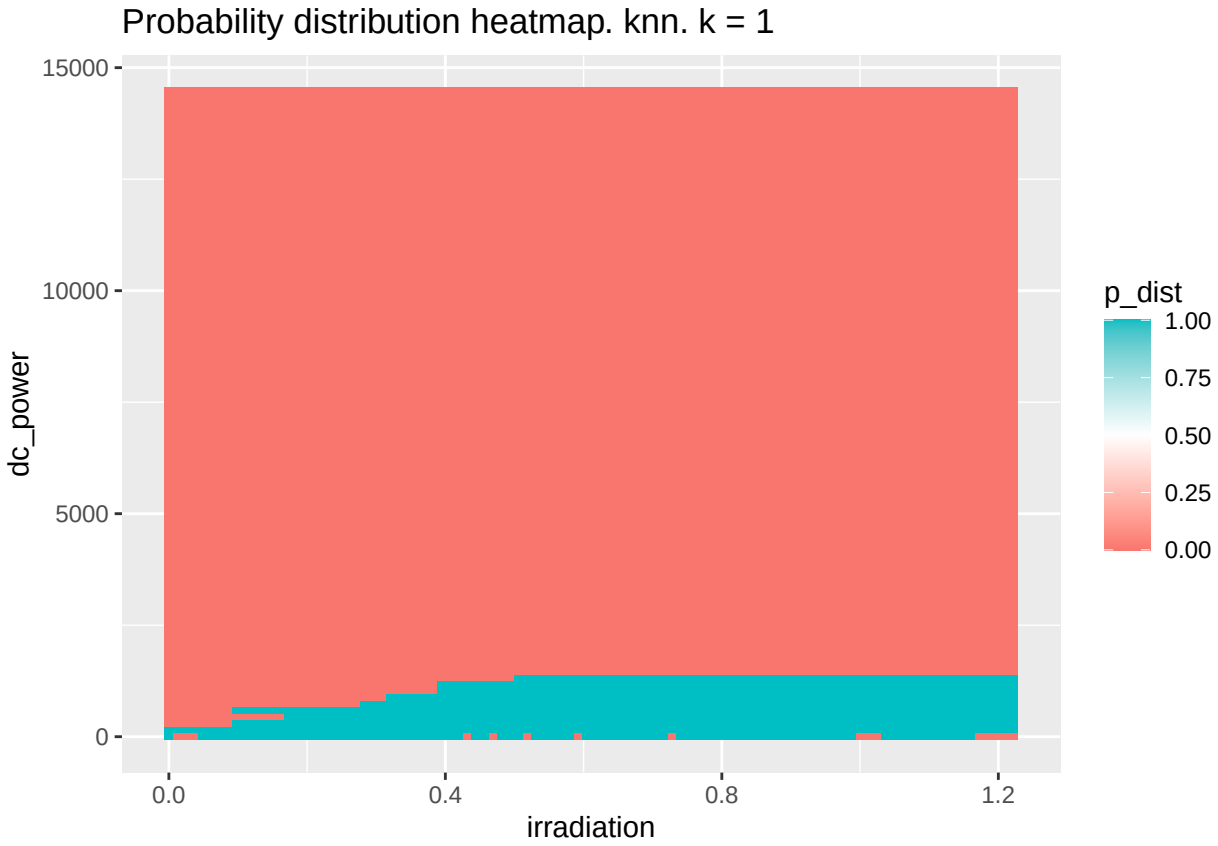
```
# Constructing the values' grid
pow_grid <- rep(seq(0, max(train_set$dc_power), length.out = 100), 100)
irr_grid <- rep(seq(0, max(train_set$irradiation), length.out = 100), each = 100)
# Combining it into a data frame to predict from
prob_dist <- data.frame(irradiation = irr_grid, dc_power = pow_grid)
# Computing probabilities of getting the second power plant as a result
lda_prob_dist <- predict(model_lda, prob_dist, type = "prob")[,2]
# Drawing the probabilities
prob_dist %>%
  mutate(p_dist = lda_prob_dist) %>%
  ggplot(aes(irradiation, dc_power, p_dist, fill = p_dist)) + geom_raster() +
  scale_fill_gradientn(colors=c("#F8766D","white","#00BFC4")) +
  labs(title = "Probability distribution heatmap. lda")
```



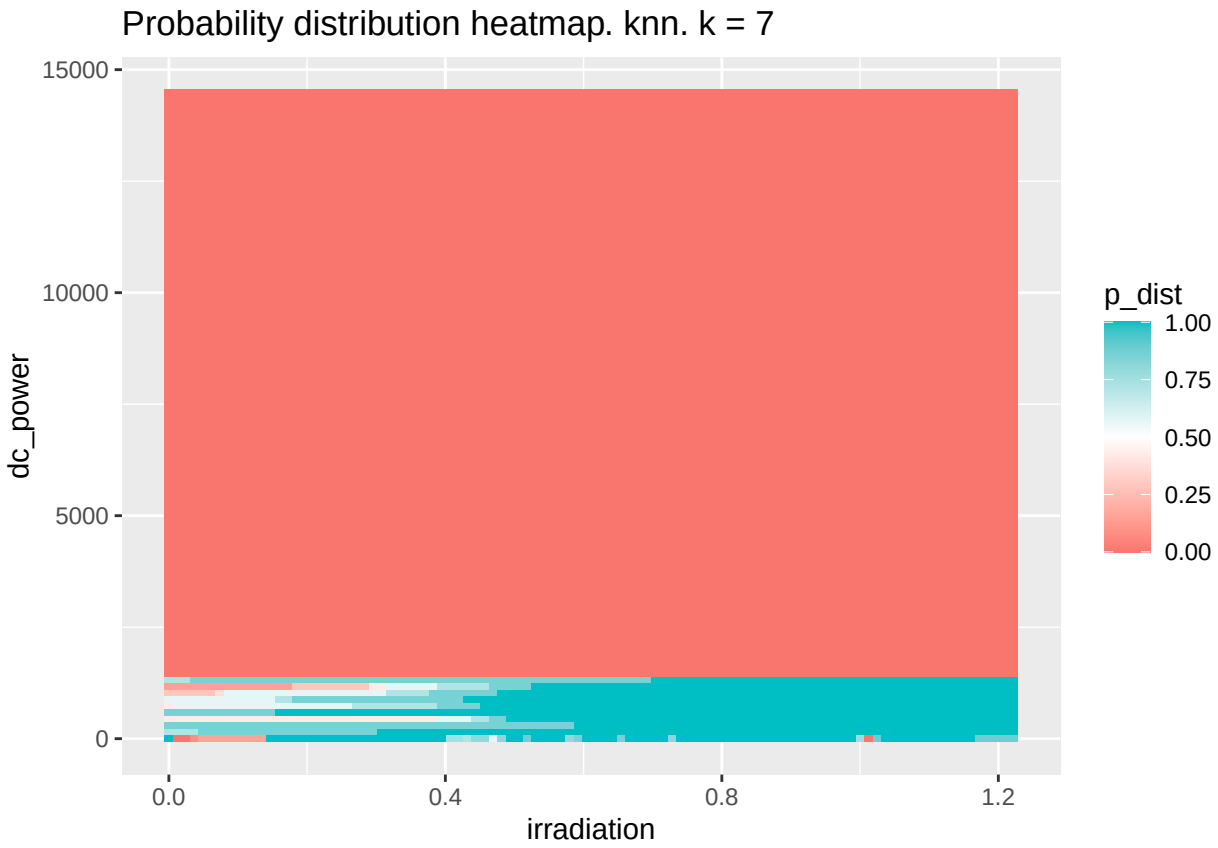
Training knn

I have tuned the algorithm in advance. With k = 1, we have the best accuracy. Generally, k = 1 may result in overfitting.

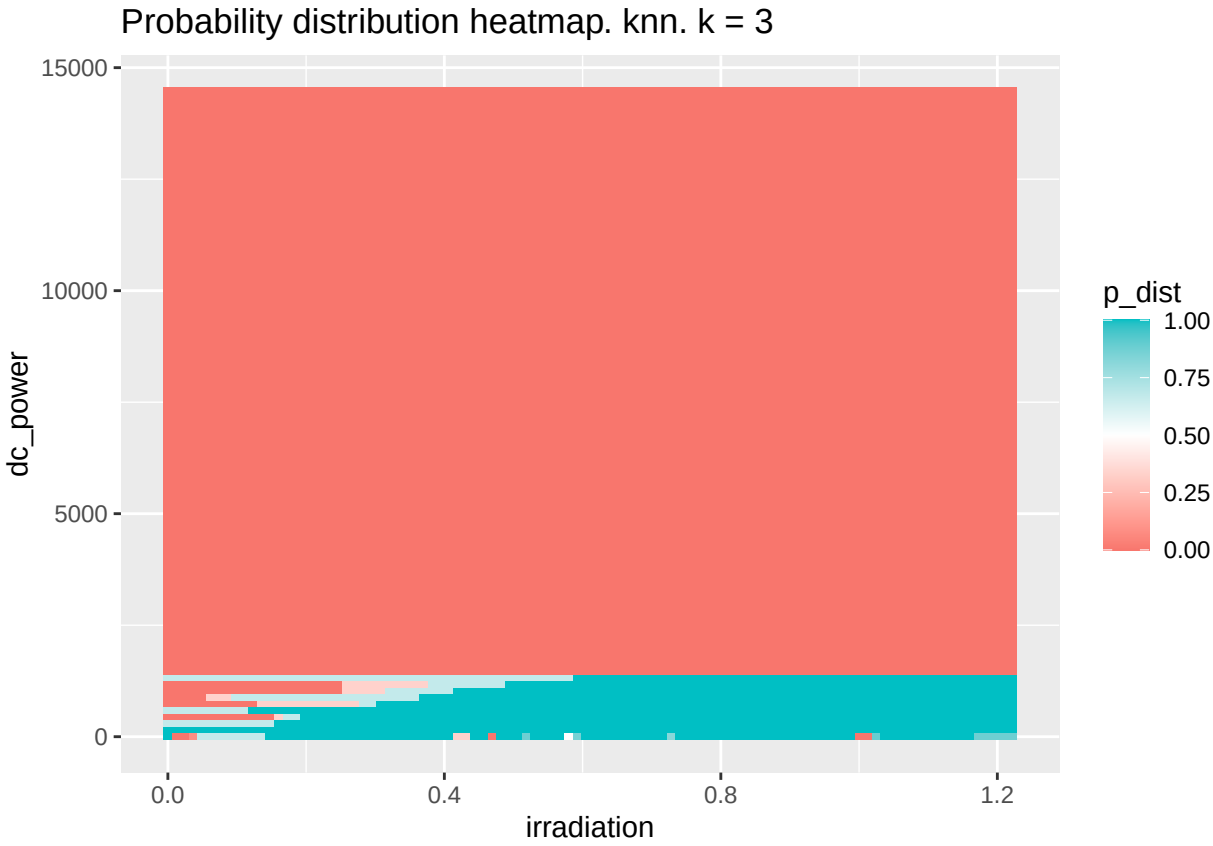
```
# knn
model_knn <- train(plant_id ~ ., data = train_set, method = "knn",
                   tuneGrid = data.frame(k = 1))
```



Every cell in the grid either has a probability of 0 or 1. There is Nothing in between. As far as I can tell, it looks like overfitting. Let's try k = 7



Too much smoothing. Let's try k = 3.

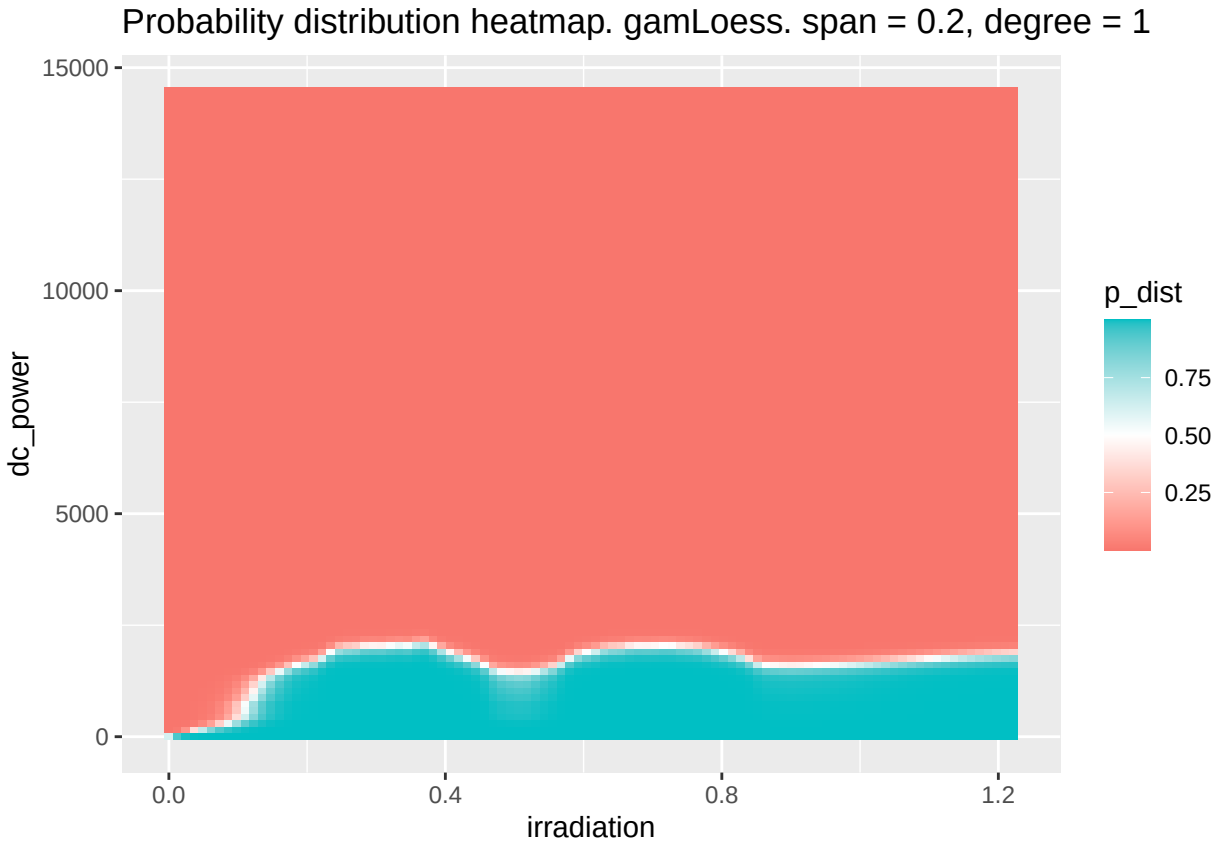


This tune is the most reasonable one, in my opinion.

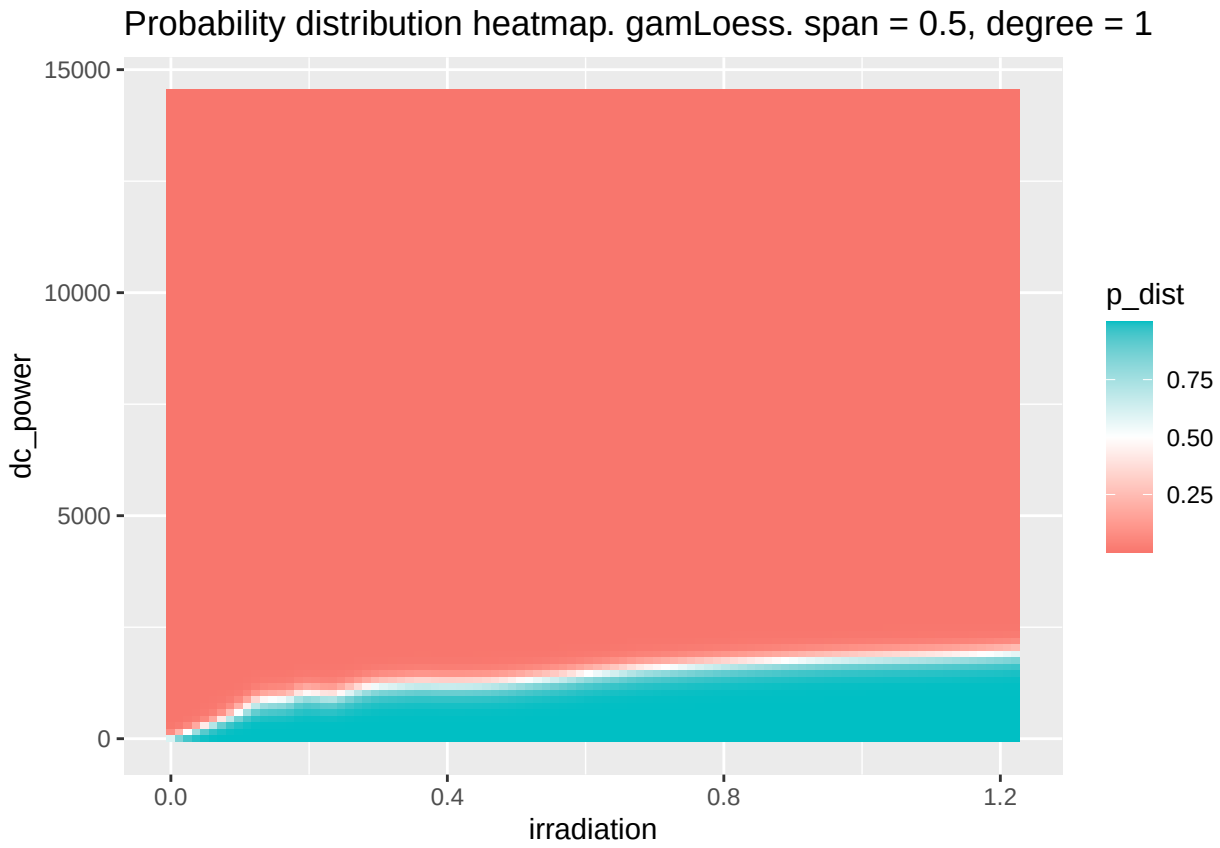
Training gamLoess

This algorithm is computationally expensive. We will take 5% of the training set to train it. I have already tuned parameters, so we are going to train the algorithm using the best tune. Tuning code you can find in code.R

```
# Creating a subset of data, since the algorithm is computationally expensive
g_train_index <- createDataPartition(y = train_set$plant_id, times = 1, p = 0.05,
                                     list = FALSE)
g_train_set <- train_set[g_train_index,]


# gamLoess
grid <- expand.grid(span = 0.2, degree = 1)
model_gam <- train(plant_id ~ ., data = g_train_set, method = "gamLoess",
                   tuneGrid = grid)
```



It looks like overfitting. No wonder. We have used less than 3000 observations to train the algorithm. I am going to change the tuning parameter.



I like this one. How about compare these algorithms in terms of accuracy.

```
y_hat_lda <- predict(model_lda, test_set, type = "raw")
y_hat_knn <- predict(model_knn, test_set, type = "raw")
y_hat_gam <- predict(model_gam, test_set, type = "raw")

tibble(lda = confusionMatrix(y_hat_lda, test_set$plant_id)$overall["Accuracy"],
       knn = confusionMatrix(y_hat_knn, test_set$plant_id)$overall["Accuracy"],
       gam = confusionMatrix(y_hat_gam, test_set$plant_id)$overall["Accuracy"])
```

```
## # A tibble: 1 x 3
##     lda   knn   gam
##   <dbl> <dbl> <dbl>
## 1 0.965 0.962 0.975
```

There is no significant difference in algorithms' accuracy. My choice for this task is lda. It is fast, accurate and uses all the data available. While in reality, lda is not as prevalent because it is almost impossible to draw a straight line to get something beneficial in complicated cases.

Conclusion

We have cleaned the data in a responsible way - which means every change we made is reasonable. We predicted the outcomes with three different algorithms and discussed the results. We have seen two ways to classify - classification by neighbours, which manifested in knn and its smoothed version - gemLoess and drawing a line - lda in our case. We have seen flaws and advantages of each algorithm in performance, computational complexity and ability to capture the trend. It is amusing the most basic and fastest model of all is the optimal one. The next step in the pipeline is to start using all CPU cores, which may open the door for training gamLoess on all the data we have. Also, it is crucial to investigate turn-offs of the panels when

irradiation is high. If we solve this issue, then the electricity going to be more accessible.