BDA3 - Machine Learning with Spark - Exercises

Naveen Gabriel (navga709), Lennart Schilling (lensc874)

2019-06-08

Contents

1. Identifying appropriate smoothing coefficients for the kernels.

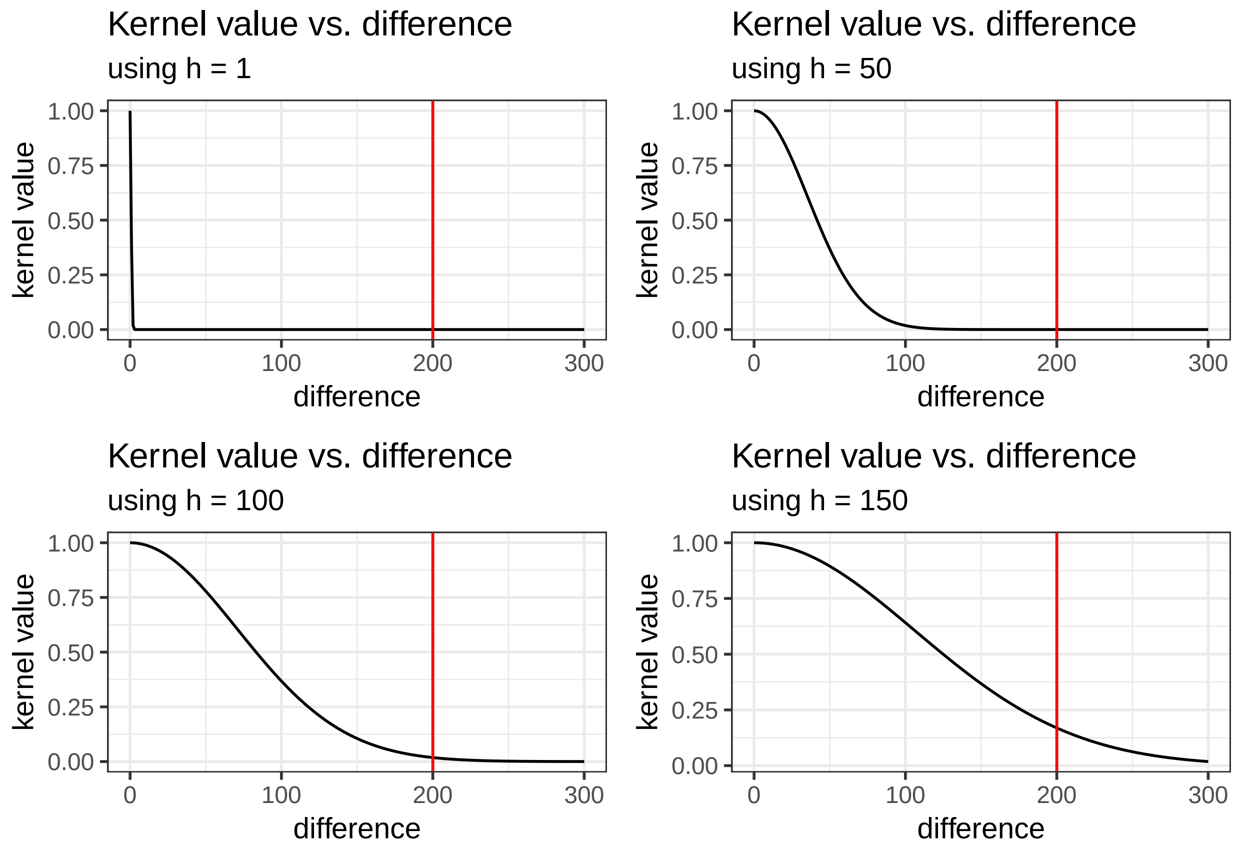
Before starting to implement the kernel method itself, we need to find out which smoothing coefficients are reasonable for each of the three gaussian kernels. To do so, we first implement the Gaussian kernel function.

```r
# Implementing gaussian kernel function.
gaussian_kernel = function(diff, h) {
  return(exp(-(diff/h)^2))
}
```

1.1 Smoothing coefficient for the location distance kernel

Our goal is to get a kernel function which leads to kernel values close to zero for distances >= 200 km. We think that it is a reasonable choice that temperatures of locations with a distance >= 200 km should not influence our prediction for the chosen location. Also, the kernel value should decrease with an increasing distance. Using R, we plot the kernel values against the distance for different choices of the smoothing coefficient `h`. The distance of 200 km is integrated into the plots which helps us to identify a reasonable choice of `h`. Since we repeat the same process for different values of `h` and `diff`, we do this by implementing a function which we can use for the smoothing coefficients for the other kernels as well.

```r
library(ggplot2)
library(gridExtra)

plot_kernels_vs_h = function(h_vector, diff_vector, diff_vline) {
  # Initializing list for storing plots.
  plot_list = list()
  # Collecting results for different values of h
  for (h in h_vector) {
    # Initializing vectors to store results.
    diff_collection = c()
    kernel_collection = c()
    # Collecting kernel values for values in diff_vector.
    for (diff in diff_vector) {
      # Storing distance value.
      diff_collection = c(diff_collection, diff)
      # Storing kernel value.
      kernel_collection = c(kernel_collection, gaussian_kernel(diff, h))
    }
    # Storing results as a plot (kernel values vs. diff) in list.
    plot_list[[as.character(h)]] = ggplot(data = data.frame(diff = diff_collection,
                                                            kernel = kernel_collection)) +
      geom_line(aes(x = diff,
                    y = kernel)) +
      geom_vline(xintercept = diff_vline,
                 color = "red") +
      theme_bw() +
      labs(title = "Kernel value vs. difference",
           subtitle = paste0("using h = ", h),
           x = "difference",
           y = "kernel value")
  }
  # Plotting all results.
  do.call(grid.arrange, plot_list)
}
plot_kernels_vs_h(h_vector = c(1, 50, 100, 150),
                  diff_vector = seq(from = 0, to = 300, by  = 1),
                  diff_vline = 200)
```

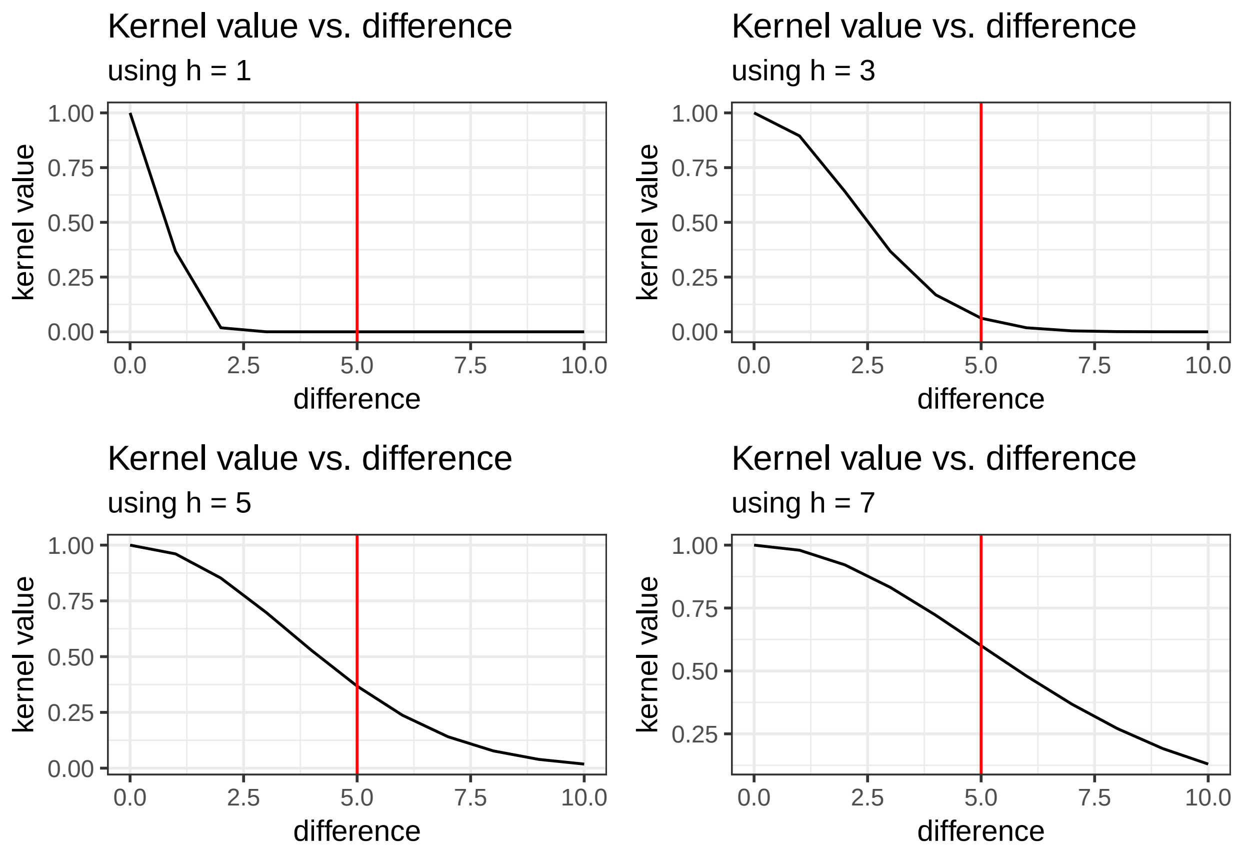


As a result, choosing a smoothing coefficient `h` = 100 seems to be a reasonable choice for the distance kernel.

1.2 Smoothing coefficient for the date distance kernel

The second kernel accounts for the distance between the day a temperature measurement was made and the day of interest. We think it is reasonable to assume that the previous five days should be weighted high, but of course decreasing. Measurements from a time further away than five days should have a low influence on the prediction. That is why we set `diff_vline` to 5.

```
plot_kernels_vs_h(h_vector = c(1, 3, 5, 7),
                  diff_vector = seq(from = 0, to = 10, by  = 1),
                  diff_vline = 5)
```

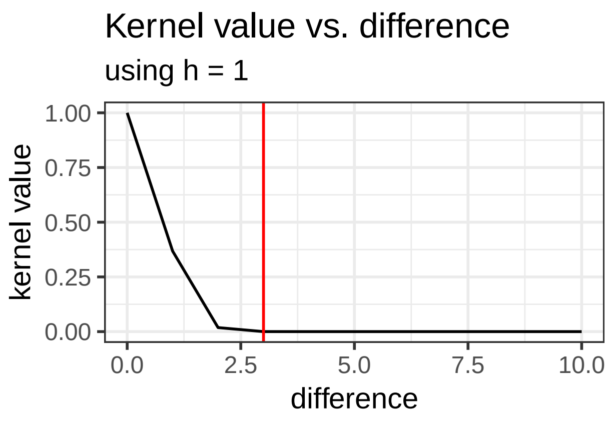


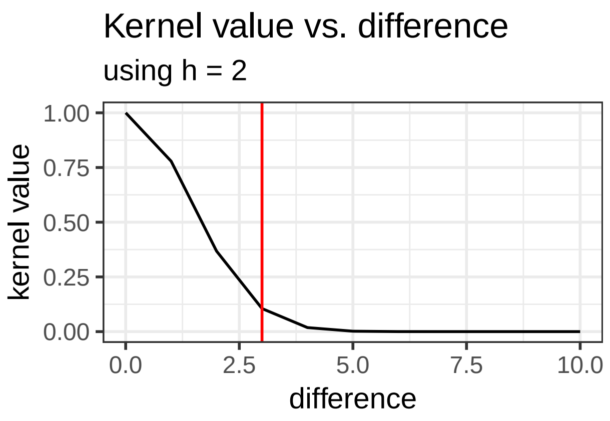
Choosing `h` $= 3$ seems to be a promising solution, because differences in days > 5 have a very small kernel value (close to zero). Differences < 5 are higher and decreasing which means that the closest day has the highest weight and that the kernel value is permanently decreasing.

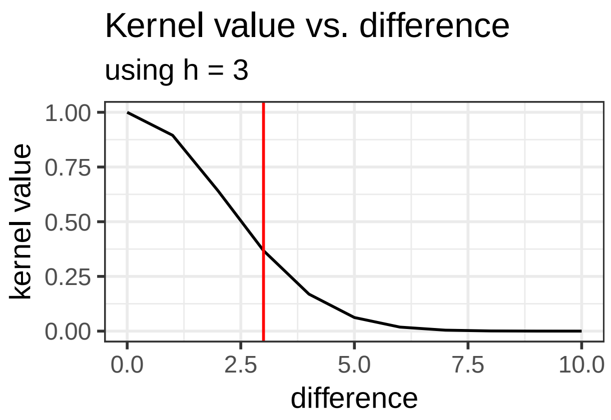
1.3 Smoothing coefficient for the hour distance kernel

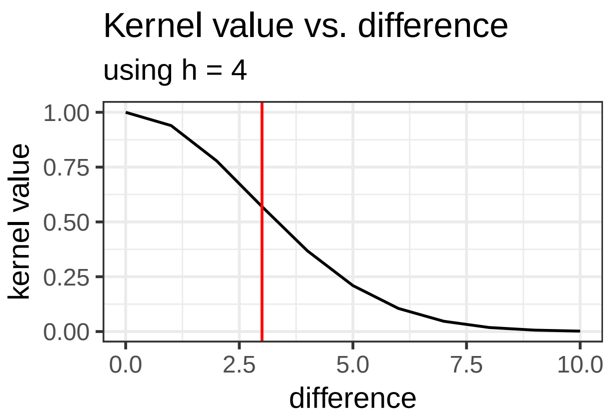
The third kernel accounts for the distance between the hour of the day a temperature measurement was made and the hour of interest. We think it is reasonable to assume that the previous three house should be weighted high, but of course decreasing. Measurements from a time further away than three hours should have a low influence on the prediction. That is why we set `diff_vline` to 3.

```r
plot_kernels_vs_h(h_vector = c(1, 2, 3, 4),
                  diff_vector = seq(from = 0, to = 10, by  = 1),
                  diff_vline = 3)
```









Choosing `h` $= 2$ seems to be a promising solution, because differences in hours > 3 have a very small kernel value (close to zero). Differences < 3 are higher and decreasing which means that the closest day has the highest weight and that the kernel value is permanently decreasing.

2. Implementing kernel method to predict the hourly temperatures for a date and place

2.1 Code

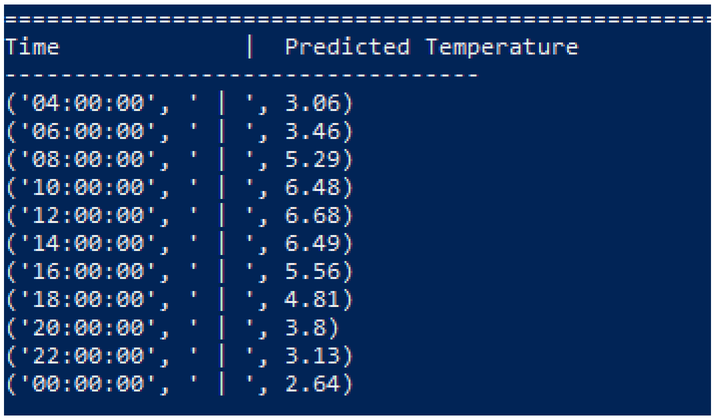
Using the identified appropriate values for each smoothing coefficient `h_distance = 100`, `h_date = 3`, `h_time = 2`, our kernel method will be implemented for the date **2015-07-04** and location **latitude = 58.4274, longitutde = 14.826** :

```
from __future__ import division
from math import radians, cos, sin, asin, sqrt, exp
from datetime import datetime, timedelta
from pyspark import SparkContext
from collections import OrderedDict

# Defining haversine-function to calculate great circle distance between two points on earth.
def haversine(lon1, lat1, lon2, lat2):
    # Converting decimal degrees to radians.
    lon1, lati1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])
    # Calculating distance in km using haversine formula.
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    km = 6367 * c
    # Returning distance in km.
    return km

# Setting up Spark's execution environment.
sc = SparkContext(appName="BDA3")

# Setting up input parameters.
h_distance = 100
h_date = 3
h_time = 2
lat_given = 58.4274
lon_given = 14.826
given_date = "2015-07-04"
start =  datetime.strptime("04:00:00","%H:%M:%S")
given_time = [start + timedelta(hours=2*x) for x in range(0, 11)]

# Reading stations data and broadcasting it.

## Reading data.
stations = sc.textFile("Data/stations.csv")
stations = stations.map(lambda line: line.split(";"))
## Extracting station number (key) and longitude, latitude (value).
stations = stations.map(lambda x: (x[0], (x[3], x[4])))
## Broadcasting stations data to each node (accessable over attribute 'value').
stations = stations.collectAsMap()
stations = sc.broadcast(stations)

# Reading temperature data.
```

```
## Reading data.
temps = sc.textFile("Data/temperature-readings.csv")
temps = temps.map(lambda line: line.split(";"))
## Extracting station number (key) and date, time, temperature (value).
temps = temps.map(lambda x: (x[0], (x[1], x[2], float(x[3]))))

# Filtering temperature data related to given date
# (keep onlydata prior to date. Posterior data should not be considererd for the prediction.)

## Filtering data with date <= given date.
temps = temps.filter(lambda x: x[1][0] <= given_date)
## Filtering time < 4:00 for given_date.
temps.filter(lambda x: x[1][1][0:3] < "04")

# Adding longitute, latitude of stations to temperature data.
temps = temps.map(lambda x: (x[0], (stations.value.get(x[0], "-"), x[1])))
temps = temps.map(lambda x: (x[0], (x[1][0][0], x[1][0][1], x[1][1][0],
                                    x[1][1][1], x[1][1][2])))

# Implementing kernel function.
def gaussian_kernel(diff, h):
    return exp(-(diff / h) ** 2)

# Creating an emtpy dictionary to store prediction.
pred = OrderedDict()

for i in range(len(given_time)):
    # Remapping temperature data including distances for location, day and hour to
    # given location, given date and to each given time.
    # New format: station_nr, distance in km, distance in days, distance in hours, temperature.
    temp = temps.map(lambda x: (x[0], (haversine(lon_given,
                                       lat_given,float(x[1][1]), float(x[1][0])),
                                      (datetime.strptime(x[1][2], '%Y-%m-%d') -
                                       datetime.strptime(given_date,'%Y-%m-%d')).days,
                                      (datetime.strptime(x[1][3], '%H:%M:%S') -
                                       given_time[i]).seconds / 3600,
                                       x[1][4])))

    # Remapping using kernel function.
    # Key will be set to 0 for later aggregation.
    # New format: 0, kernel value for distance in km, kernel value for distance in days,
    # kernel value for distance in hours, temperature.
    # Storing new value in temp instead of temps because we need to use temps again.
    temp = temp.map(lambda x: (0, (gaussian_kernel(x[1][0],
                      h_distance), gaussian_kernel(x[1][1], h_date),
                      gaussian_kernel(x[1][2], h_time), x[1][3])))

    # Remapping.
    # New format:
    # (0, (distance kernel * temp + day kernel * temp + hour kernel * temp, sum of kernels))
    temp = temp.map(lambda x: (x[0], (x[1][0] * x[1][3] + x[1][1]
                      * x[1][3] + x[1][2] * x[1][3], x[1][0] + x[1][1]
                      + x[1][2])))
    # Aggregating by key (we implemented same key for every observation,
    # so aggreagating over all observations.
    # New format: (0, (sum of kernel values * temps, sum of kernel values))
    temp = temp.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))

    # Remapping to extract result.
    # Prediction = (sum of kernel values * temps) / sum of kernel values
    temp = temp.map(lambda x: x[1][0] / x[1][1])

    # Storing predicted value.
    pred[given_time[i].strftime("%H:%M:%S")] = temp.take(1)

# Printing results.
print("Time              |  Predicted Temperature")
print("-----------------------------------")
for key,value in pred.items() :
    print(key,' | ',round(value[0],2))

# Closing spark enviorenment.
sc.stop()
```

2.2 Result

Running our code for the date **2015-07-04** and location **latitude = 58.4274, longitutde = 14.826** within the Heffa cluster leads to the following output:



```
=====================================================
Time                |  Predicted Temperature
----------------------------------
('04:00:00', ' | ', 3.06)
('06:00:00', ' | ', 3.46)
('08:00:00', ' | ', 5.29)
('10:00:00', ' | ', 6.48)
('12:00:00', ' | ', 6.68)
('14:00:00', ' | ', 6.49)
('16:00:00', ' | ', 5.56)
('18:00:00', ' | ', 4.81)
('20:00:00', ' | ', 3.8)
('22:00:00', ' | ', 3.13)
('00:00:00', ' | ', 2.64)
```

The predicted temperature does rise and fall during the day as expected but given the temperature we are trying to predict in July the degree of temperature is less. We account this behaviour due to independent assumption of kernel. One of the kernel (probably date), we assume, contributes to the prediction but since it is assumed independent the prediction is not full fledged. Had the resultant kernels been multiplicative then we could have got better prediction of temperatures.

2.3 Improvement

The given instructions to improv our code have been imlemented.

```
from __future__ import division
from math import radians, cos, sin, asin, sqrt, exp
from datetime import datetime, timedelta
from pyspark import SparkContext
from collections import OrderedDict

# Defining haversine-function to calculate great circle distance between two points on earth.
def haversine(lon1, lat1, lon2, lat2):
    # Converting decimal degrees to radians.
    lon1, lati1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])
    # Calculating distance in km using haversine formula.
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    km = 6367 * c
    # Returning distance in km.
    return km

# Setting up Spark's execution environment.
sc = SparkContext(appName="BDA3")

# Setting up input parameters.
h_distance = 100
h_date = 3
h_time = 2
lat_given = 58.4274
lon_given = 14.826
given_date = "2015-07-04"
start = datetime.strptime("04:00:00","%H:%M:%S")
given_time = [start + timedelta(hours=2*x) for x in range(0, 11)]

# Reading stations data and broadcasting it.
## Reading data.
stations = sc.textFile("/user/x_lensc/data/stations.csv")
stations = stations.map(lambda line: line.split(";"))
## Extracting station number (key) and longitude, latitude (value).
stations = stations.map(lambda x: (x[0], (x[3], x[4])))
## Broadcasting stations data to each node (accessable over attribute 'value').
stations = stations.collectAsMap()
stations = sc.broadcast(stations)

# Reading temperature data.
## Reading data.
temps = sc.textFile("/user/x_lensc/data/temperature-readings.csv")
temps = temps.map(lambda line: line.split(";"))
## Extracting station number (key) and date, time, temperature (value).
temps = temps.map(lambda x: (x[0], (x[1], x[2], float(x[3]))))

# Filtering temperature data related to given date
# (keep onlydata prior to date. Posterior data should not be considererd for the prediction.)
## Filtering data with date <= given date.
temps = temps.filter(lambda x: x[1][0] <= given_date)

# Adding longitute, latitude of stations to temperature data.
temps = temps.map(lambda x: (x[0], (stations.value.get(x[0], "-"), x[1])))
temps = temps.map(lambda x: (x[0], (x[1][0][0], x[1][0][1], x[1][1][0],
x[1][1][1], x[1][1][2])))

# Caching data.
temps = temps.cache()

# Implementing kernel function.
def gaussian_kernel(diff, h):
    return exp(-(diff / h) ** 2)
# Creating an emtpy dictionary

# Creating empty dictionaries to store predictions.
# Results for addition and multiplication of kernels will be stored.
predAddition = OrderedDict()
predMultiplication = OrderedDict()

for i in range(len(given_time)):

    # Remapping temperature data including distances for location, day and hour to
    # given location, given date and to each given time.
    # New format: station_nr, distance in km, distance in days, distance in hours, temperature.
    temp = temps.map(lambda x: (x[0], (haversine(lon_given,
    lat_given,float(x[1][1]), float(x[1][0])),
    (datetime.strptime(x[1][2], '%Y-%m-%d') -
    datetime.strptime(given_date,'%Y-%m-%d')).days,
    (datetime.strptime(x[1][3], '%H:%M:%S') -
    given_time[i]).seconds / 3600,
    x[1][4])))

    # Remapping using kernel function.
    # Key will be set to 0 for later aggregation.
    # New format: 0, kernel value for distance in km, kernel value for distance in days,
    # kernel value for distance in hours, temperature.
    # Storing new value in temp instead of temps because we need to use temps again.
    temp = temp.map(lambda x: (0, (gaussian_kernel(x[1][0],
    h_distance), gaussian_kernel(x[1][1], h_date),
    gaussian_kernel(x[1][2], h_time), x[1][3])))

    # Addition of kernels.

    ## Remapping.
    ## New format:
    ## (0, (distance kernel * temp + day kernel * temp + hour kernel * temp, sum of kernels))
    tempAdd = temp.map(lambda x: (x[0], (x[1][0] * x[1][3] + x[1][1]
    * x[1][3] + x[1][2] * x[1][3],
    x[1][0] + x[1][1] + x[1][2])))

    ## Aggregating by key (we implemented same key for every observation,
    ## so aggreagating over all observations.
    ## New format: (0, (sum of kernel values * temps, sum of kernel values))
    tempAdd = tempAdd.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))

    ## Remapping to extract result:
    ## Prediction = (sum of kernel values * temps) / sum of kernel values
    tempAdd = tempAdd.map(lambda x: x[1][0] / x[1][1])

    ## Storing predicted values.
    predAddition[given_time[i].strftime("%H:%M:%S")] = tempAdd.take(1)

    # Multiplication of kernels.

    ## Remapping.
    ## New format:
    ## (0, (distance kernel * day kernel * hour kernel * temp, product of kernels))
    tempMult = temp.map(lambda x: (x[0], (x[1][0] * x[1][1] * x[1][2] * x[1][3],
    x[1][0] * x[1][1] * x[1][2])))

    ## Aggregating by key (we implemented same key for every observation,
    ## so aggreagating over all observations.
    ## New format: (0, (sum of kernel values * temps, sum of kernel values))
    tempMult = tempMult.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))

    ## Remapping to extract result:
    ## Prediction = (sum of kernel values * temps) / sum of kernel values
    tempMult = tempMult.map(lambda x: x[1][0] / x[1][1])

    ## Storing predicted values.
    predMultiplication[given_time[i].strftime("%H:%M:%S")] = tempMult.take(1)


# Printing results.
print("Results for addition of kernels:\n")
print("Time | Predicted Temperature")
print("-----------------------------------")
for key,value in predAddition.items() :
    print(key,' | ',round(value[0],2))

print("Results for multiplication of kernels:\n")
print("Time | Predicted Temperature")
print("-----------------------------------")
for key,value in predMultiplication.items() :
    print(key,' | ',round(value[0],2))

# Closing spark enviorenment.
sc.stop()
```

The unnecessary filtering has been removed. Furthermore, the data is cached before we iterate over multiple choices of given_time. Also, we implemented the structure to generate results for the multiplication of kernels. Unfortunately, the result can not be printed. The reason is that at step *tempMult = tempMult.map(lambda x: x[1][0] / x[1][1])*, a ZeroDivisionError is returned. The reason is that when calculating the product of the three kernels for every observation, there is no observation within the data where all kernels are > 0. It follows that for every observation the product of the three kernels equals zero so that the sum over all products also equals zero. Unfortunately, we do not see any possible solution for this problem.