LAB-3

Siddhesh Sreedar (sidsr770)
Hugo Morvan (hugmo418)

CODE

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

sc = SparkContext(appName="lab_kernel")

#Linkoping
pred_lat = 58.394241 #latitude
pred_long = 15.583155 #longitude

pred_date_str = "2014-06-07" # Up to you
pred_date = date(2014, 6, 7)

#--- Helper Functions ----

def haversine(lon1, lat1, lon2, lat2):
    """
    Calculate the great circle distance between two points
    on the earth (specified in decimal degrees)
    """
    # convert decimal degrees to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])
    # 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
    return km


def gaussian_kernel(x,h):
    #function to calculate the gaussian kernel for all three kernels
    return exp(-(x/h)**2)

#print("@valuekern")
#print(gaussian_kernel(4,7))


def get_k_dist(long1, lat1, long2, lat2,h):
    #returns the kernel function for the difference in distance
    dist = haversine(long1, lat1,long2,lat2)
    return gaussian_kernel(dist, h)

#print("@valuedist")
#print(get_k_dist(20, 20, 1, 1, 1))


def get_k_days(day, pred_day,h):
    #Return the kernel function for the difference in days
    value = (pred_day - day).days
    return  gaussian_kernel(value,h)

#print("@valuedays")
#print(get_k_days(datetime.strptime(pred_date_str, "%Y-%m-
%d").date(),datetime.strptime("2013-07-03", "%Y-%m-%d").date(),1))
#print(datetime.strptime(pred_date_str, "%Y-%m-%d"))


def get_k_hour(timeA,timeB,h):
    #Return the kernel function for the difference in hours
    timeA= int(timeA[0:2])
    timeB = int(timeB[0:2])
    value =  abs((timeB - timeA))
    return  gaussian_kernel(value,h)

#print("@valuehour")
#print(get_k_hour("24:00:00","22:00:00",1))


# --- H Parameters ---

#Derived from previous work in Machine Learning
h_dist = 200
h_days = 30
h_time = 6


# --- Reading in the data and mapping---

stations = sc.textFile("BDA/input/stations.csv")
stations = stations.map(lambda line: line.split(";"))
stations = stations.map(lambda x: (x[0],(float(x[3]),float(x[4]))))
#(stations, (lat,long))

#temps = sc.textFile("BDA/input/temperature-readings-small.csv")
#for testing
temps = sc.textFile("BDA/input/temperature-readings.csv")
temps = temps.map(lambda line: line.split(";"))
temps = temps.map(lambda x: (x[0], (datetime.strptime(x[1], "%Y-%m-
%d").date(), x[2], float(x[3])))) #(stations, (date, time, temp) )
#('102170', (datetime.date(2014, 12, 31), '06:00:00', -5.4))

# --- Filter out anything after 2013-07-03 (1 day prior to desired
date) ---
temps_filtered = temps.filter(lambda x: (x[1][0]<date(2014, 6, 7)) )

#stations = sc.parallelize(stations.collectAsMap())
#stations.cache()

# --- Joining the stations data to the temperatures using broadcast
:
stations = stations.collectAsMap()
bc = sc.broadcast(stations)

#(station,(date,time,temp,lat,long))
joined = temps_filtered.map(lambda x:
(x[0],(x[1][0],x[1][1],x[1][2],bc.value.get(x[0]))))

#bc.unpersist()

# --- Defining the distance kernel, cached because it is called
several times in the for loop
partial_sum_rdd = joined.map(lambda x:
(get_k_dist(x[1][3][1],x[1][3][0],pred_long,pred_lat,h_dist)+get_k_d
ays(x[1][0], pred_date,h_days), x[1][1], x[1][2])).cache() #
(partial_sum, time, temp)

# --- Defining the date kernel, cached because it is called several
times in the for loop
partial_prod_rdd = joined.map(lambda x:
(get_k_dist(x[1][3][1],x[1][3][0],pred_long,pred_lat,h_dist)*get_k_d
ays(x[1][0], pred_date,h_days), x[1][1], x[1][2])).cache() #
(partial_prod, time, temp)

#Initialising the predictions array
pred_all_sum = []
pred_all_mup = []
```

```
for time in ["24:00:00", "22:00:00", "20:00:00", "18:00:00",
"16:00:00", "14:00:00",
"12:00:00", "10:00:00", "08:00:00", "06:00:00", "04:00:00"]:

    # Defining the hour kernel for the loop hour
    #k_hour = joined.map(lambda x:( exp(-
(hours_to_desired_pred(x[1][1], time))**2)/(2*h_date**2),x[1][2]))

    #combined_kernel = joined.map(lambda x:
(x[1][2],(dist_kernel,days_kernel,get_k_hour(x[1][1],
time,h_time))))

    # SUM OF THE KERNELS
    #k_sum = combined_kernel.map(lambda x: (1,
((x[1][0]+x[1][1]+x[1][2])*x[0], x[1][0]+x[1][1]+ x[1][2]) ) )
    k_sum = partial_sum_rdd.map(lambda x: (1,
((x[0]+get_k_hour(time, x[1], h_time))*x[2],
                                               x[0]+get_k_hour(time,
x[1], h_time)) )) #(1, (numerator, denominator))

    k_sum = k_sum.reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1])) #
Adds the numerators and the denominators
    pred_sum = k_sum.map(lambda x: (x[1][0]/x[1][1])).collect() #
numerator/denominator

    #PRODUCT OF THE KERNELS
    #k_mup = combined_kernel.map(lambda x: (1,
((x[1][0]*x[1][1]*x[1][2])*x[0], x[1][0]*x[1][1]*x[1][2]) ) )
    k_prod = partial_prod_rdd.map(lambda x: (1,
((x[0]*get_k_hour(time, x[1], h_time))*x[2],

x[0]*get_k_hour(time, x[1], h_time)) )) #(1, (numerator,
denominator))

    k_prod = k_prod.reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1]))
    pred_mup = k_prod.map(lambda x: (x[1][0]/x[1][1])).collect()

    #save in the output array
    pred_all_sum.append(pred_sum)
    pred_all_mup.append(pred_mup)


print("@output")
print("___ Sum Kernel___")
print(pred_all_sum)
print("___ Mult Kernel___")
print(pred_all_mup)
```

```
#pred.saveAsTextFile("BDA/output")
```

OUTPUT

Reminder of the times:

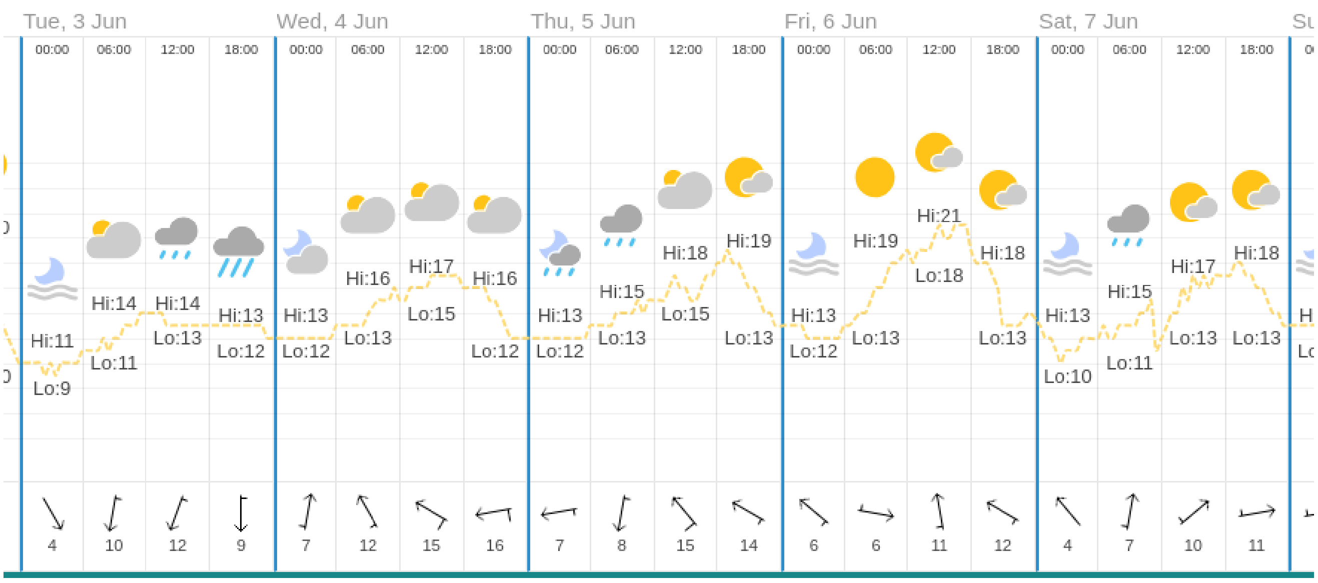
```
["24:00:00","22:00:00","20:00:00","18:00:00","16:00:00","14:00:00","
12:00:00","10:00:00","08:00:00","06:00:00","04:00:00"]
```

@output

```
___ Sum Kernel___
[[5.610826717906215], [5.528487392142649], [5.5831617815418815],
[5.728357761889621], [5.887377961980705], [5.9652105988354736], [5.88183079326927],
[5.62288117440531], [5.259965593868862], [4.911462396230766], [4.681348868378 39]]


___ Mult Kernel___
[[11.233477512858554], [11.817902749644796], [12.486808268781376],
[13.172145817484779], [13.746405004824018], [14.037697206616999],
[13.89976634268463], [13.309808957047963], [12.401742366509529],
[11.390318094425576], [10.461511747440966]]
```

Comparing with the actual values:

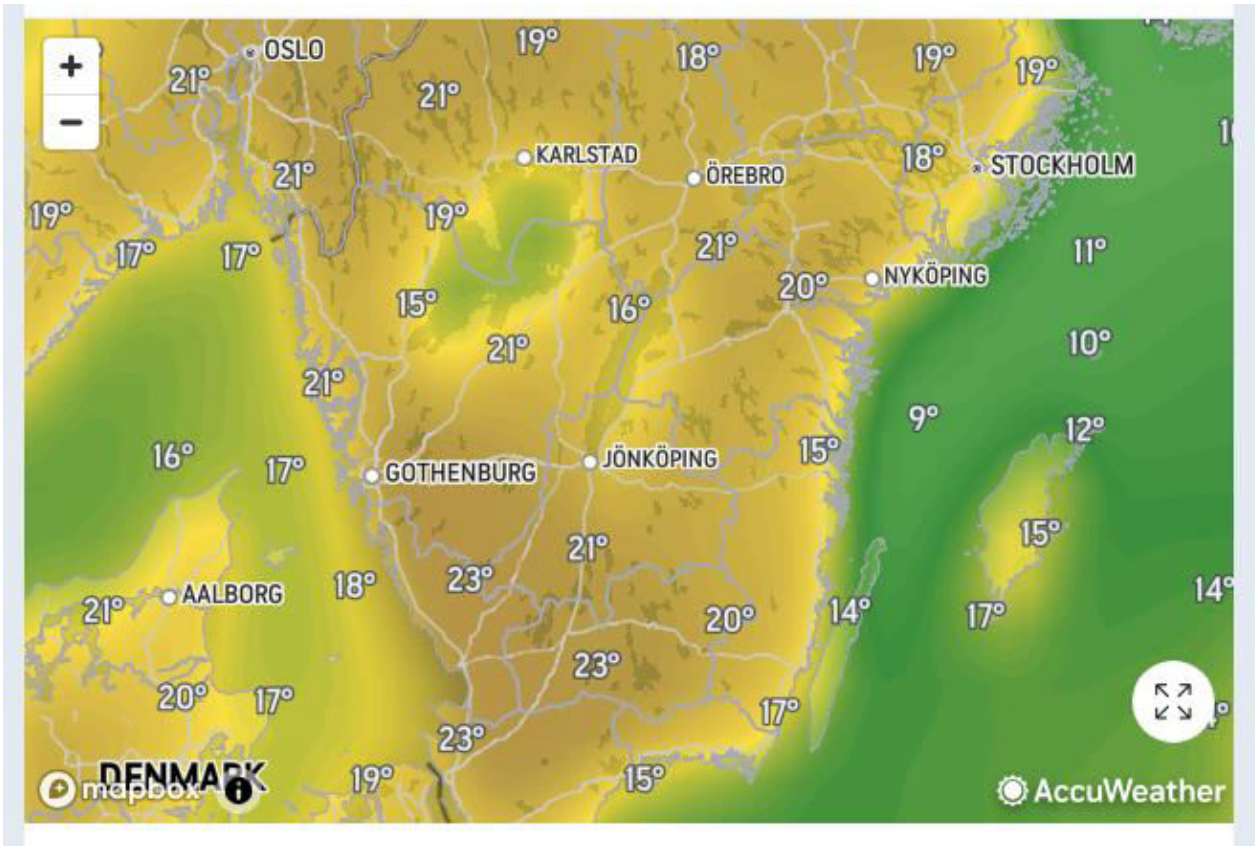


The results seem to be quite close for the product kernel, and a bit more off for the additive kernel.

QUESTIONS

● Show that your choice for the kernels' width is sensible, i.e. it gives more weight to closer points. Discuss why your definition of closeness is reasonable.

For our choices of distance kernels we chose `h_dist = 200 (km), h_days = 30 (days) and h_time = 6 (hours), i.e.`



For the Distance kernel :
Karlstad is about 165km from Linköping, Stockholm is 175km, Gothenburg 225km, Jönköping is 110km, Örebro is 100km away, hence a kernel width of 200km centered around Linköping is reasonable is the temperatures (on land) within a 200km radius from Linköping seems to be consistent, and we assumed that all the stations were on land.

For the days kernel:
We chose a kernel distance of 30 days / 1 month as it seemed reasonable for us that temperatures are somewhat consistent within a 1 month window, especially in June.

For the Hours Kernel, we chose 6 hours or a quarter of a day, as 6 hours seems like a reasonable time frame where temperatures could be around the same. It is also how the day is divided on the weather prediction website shown above.

● Repeat the exercise using a kernel that is the product of the three Gaussian kernels above. Compare the results with those obtained for the additive kernel. If they differ, explain why

The product kernel seems to work much better than the additive kernel. This could be because in a product kernel, each kernel influences/regulates the outcome of the other two. For example, in a situation where a temperature measurement was made at the same hour and same day but very far apart, the combined kernel would be very high for the additive kernel (which could be problematic if one station is very far north and the prediction is very far south); and very low for the product kernel as the distance kernel would multiply the other two kernels by a value close to 0, hence canceling the influence of that temperature measurement on the final prediction.