Baysian Learning
732A54

Lab 3 BDA

Guilherme Barros
guiba484

Eric Herwin
erihe068

For this task we will implement in Spark (PySpark) a kernel method to predict the hourly temperatures for a date and place in Sweden.

Code:

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

sc = SparkContext(appName="lab_kernel")

def haversine(lon1, lat1, lon2, lat2):
    # 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

#take x_user - x_n
def kernel(diff, h):
    return exp(- diff**2/h**2)

h_distance = 100000 # Up to you
h_date  =  7 #  Up  to  you
h_time = 2 # Up to you

a = 60.2788 # Up to you
b = 12.8538 # Up to you
date = "2013-11-23" # Up to you
time = 6 #time of day
```

```
stations = sc.textFile("path")
temps = sc.textFile("path")

RDD_temp = temps.map(lambda l: l.split(";"))
RDD_stations = stations.map(lambda l: l.split(";"))

#kernel for date and time
diffs_date = RDD_temp.map(lambda x:  (x[0],
                          (((datetime.strptime(date, '%Y-%m-%d') - datetime.strptime(x[1], '%Y-%m-%d'))
                          (float(x[2].split(":")[0]) -  time) , float(x[3])))  )
kern_diff = diffs_date.mapValues(lambda x: (kernel(x[0], h_date), kernel(x[1], h_date), x[2]))

#kernel for position of station
hav_dist = RDD_stations.map(lambda x: (x[0], haversine(a, b, float(x[3]), float(x[4])))  )
kern_hav_diff = hav_dist.mapValues(lambda x: kernel(x, h_distance))
broad_station = sc.broadcast(kern_hav_diff.collectAsMap())
#join_a_b = kern_diff.map(lambda (key, val): (key, broad_station.value.get(key, "-"), val))
```

```
#sum kernels
sum_kernel = join_a_b.map(lambda x: ("1",((x[1] + x[2][0] + x[2][1])*x[2][2],
```

```
                                        x[1] + x[2][0] + x[2][1])) )
sum_kernel = sum_kernel.reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1]))
pred_sum_kernel = sum_kernel.map(lambda x: x[1][0]/x[1][1] )
#prod kernels
prod_kernel = join_a_b.map(lambda x: ("1", ((x[1] * x[2][0] * x[2][1])*x[2][2],
                                          x[1] * x[2][0] * x[2][1])) )
prod_kernel = prod_kernel.reduceByKey(lambda x,y: (x[0]+y[0], x[1]+y[1]))
pred_prod_kernel = prod_kernel.map(lambda x: x[1][0]/x[1][1] )
print(pred_sum_kernel.take(10))
print("--")
print(pred_prod_kernel.take(10))
```

Output:

Sum kernel: [4.44189958894498] – Prod kernel: [-0.6184976430578428]

The station used for the data:

Temperature output: [u'102170', u'2013-11-23', u'06:00:00', u'-12.9', u'G']

Station output: [u'102170', u'swedish letters', u'2.0', u'60.2788', u'12.8538', u'2013-11-01 00:00:00', u'2016-09-30 23:59:59', u'135.0']

As we can see, the kernels when summed tend to get values close to the mean of the dataset, while the product of kernels goes closer to the "real" value.

1

For the value of Hs, we have chosen 100km, 7 days and 2 hours, which seemed appropriate values. It is interesting to notice that if too high numbers are chosen for Hs, the algorithm tends to use all values in the database equally, so the result gets closer to the mean. In the case of value of Hs too high, the algorithm "uses" only a few close points and gives out values based on just these few, which is can be considered overfitting.

2

We can see that summing and multiplying give out different results for the prediction. What is happening is that when summing the algorithm gives equal weights to distances between hours, days and physical distance. This results in points being used from a large variety of the database and the result tends to the mean of the database.

On the other hand, by multiplying the weights, the algorithm only "uses" points that are close for all the three criteria. This solves the problem of all the predictions being close to the mean. However, this also increases the chances of overfitting.