BDA3Report_ver2

June 11, 2023

1 LAB EXERCISE 3: MACHINE LEARNING

1.0.1 Yi-Hung Chen (yihch883) Jonathan Dorairaj (jondo380)

Implement in Spark (PySpark) a kernel model to predict the hourly temperatures for a date and place in Sweden. To do so, you should use the files temperature-readings.csv and stations.csv from previous labs. Specifically, the forecast should consist of the predicted temperatures from 4 am to 24 pm in an interval of 2 hours for a date and place in Sweden. Use a kernel that is the sum of three Gaussian kernels:
The first to account for the distance from a station to the point of interest.
The second to account for the distance between the day a temperature measurement was made and the day of interest.
The third to account for the distance between the hour of the day a temperature measurement was made and the hour of interest.

1.1 Kernel Model

Note: For hour interval we use [0:00:00, 04:00:00, 06:00:00 ...] and not 24:00:00, this is because in temperature-reading.csv, temperature is recorded from 00:00:00, so we keep that for consistancy.

For the changes of improvement, we now has first include all data include target_date, and within the loop we discard the data that is in the target_date but later then target hour

[ ]:
```
from __future__ import division
from math import radians, cos, sin, asin, sqrt, exp
from datetime import datetime
from pyspark import SparkContext
from pyspark.broadcast import Broadcast
import numpy as np

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

def day_difference(day1, day2):
    diff = abs(datetime.strptime(str(day1), "%Y-%m-%d") - datetime.
 ↪strptime(str(day2), "%Y-%m-%d"))
    no_days = diff.days
    return no_days

def hour_diff(time1, time2):
    diff = abs(datetime.strptime(time1, "%H:%M:%S") - datetime.strptime(time2,␣
 ↪"%H:%M:%S"))
    diff = (diff.total_seconds()) / 3600
    return diff

def gaussian_kernel(u, h):
    return np.exp(-u**2 / (2 * h**2))

def sum_kernel(distance_kernel, day_kernel, time_kernel):
    res = distance_kernel + day_kernel + time_kernel
    return res

def product_kernel(distance_kernel, day_kernel, time_kernel):
    res = distance_kernel * day_kernel * time_kernel
    return res

# Value to predict
target_latitude = 58.68681
target_longitude = 15.92183
target_date = '2014-05-17'
hour_list= ["00:00:00", "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"]


# Kernel width
h_dist = 100
h_day = 15
h_time = 5

# Create SparkContext
sc = SparkContext(appName="Lab 3 ML")

temperature_file = sc.textFile("BDA/input/temperature-readings.csv")
station_file = sc.textFile("BDA/input/stations.csv")

temperature_data = temperature_file.map(lambda x: x.split(';'))
stations_data = station_file.map(lambda x: x.split(';'))
```

```
#Filter the previous data until target_date, this can save some computation
target_date_strip = datetime.strptime(target_date, '%Y-%m-%d')
prev_temp = temperature_data.filter(lambda x: datetime.strptime(x[1],␣
 ↪'%Y-%m-%d') <= target_date_strip)

# pre-calculate station_distkernel and broadcast it (to speed up process)
station_distkernel = stations_data.map(lambda x: (x[0],␣
 ↪gaussian_kernel(haversine(target_longitude, target_latitude, float(x[4]),␣
 ↪float(x[3])), h_dist))).collectAsMap()
broadcast_station_distkernel = sc.broadcast(station_distkernel)

#station id, hour, temp, day_kernel,date, cache it to save time
temperature_data_datekernel = prev_temp.map(lambda x: (x[0], x[2], x[3],␣
 ↪gaussian_kernel(day_difference(target_date, x[1]), h_day),x[1])).cache()


predictions = {}
for hour in hour_list:
    target_hour_strip = datetime.strptime(hour, '%H:%M:%S')
    #filter: here, it will keep data from previous date, and current date␣
 ↪before current hour. Using "or" is crucial.
    temp_filtered = temperature_data_datekernel.filter(lambda x: datetime.
 ↪strptime(x[1], '%H:%M:%S') <= target_hour_strip
                                                          or datetime.
 ↪strptime(x[4], '%Y-%m-%d') < target_date_strip)

    #temp, dist_kernel, day_kernel, hour_kernel
    temp_kernels = temp_filtered.map(lambda x: (float(x[2]),
                                                  broadcast_station_distkernel.
 ↪value[x[0]],
                                                  x[3],
                                                  gaussian_kernel(hour_diff(hour,␣
 ↪x[1]), h_time)))
    #temp,sumkernel, prodkernel
    temp_both_kernels = temp_kernels.map(lambda x: (x[0], sum_kernel(x[1],␣
 ↪x[2], x[3]), product_kernel(x[1], x[2], x[3])))

    #sum_kernel, sum_kernel*temp, prod_kernel, prod_kernel*temp
    cal_kerneltemp = temp_both_kernels.map(lambda x :
 ↪(x[1],x[0]*x[1],x[2],x[0]*x[2]))

    #sum all the elements
    sum_kerneltemp = cal_kerneltemp.reduce(lambda x, y: (x[0] + y[0], x[1] +␣
 ↪y[1], x[2] + y[2], x[3] + y[3]))

    #sum_prediction,prod_prediction
    predictions[hour]=(sum_kerneltemp[1]/sum_kerneltemp[0],sum_kerneltemp[3]/
↪sum_kerneltemp[2])

# Convert predictions dictionary to RDD
predictions_rdd = sc.parallelize(list(predictions.items()))
predictions_rdd = predictions_rdd.coalesce(1)
predictions_rdd = predictions_rdd.sortByKey()
# Save the RDD as text files
predictions_rdd.saveAsTextFile("BDA/output/predictions")
```

1.2 Time(Hour),Prediction using summation kernel, Prediction using Prodoct kernel

```
('00:00:00', (4.2202706240460373, 5.3158347309906233))
('04:00:00', (4.3553966731824358, 6.4255932055837439))
('06:00:00', (4.6247919485850693, 7.1948198019356946))
('08:00:00', (4.9631010258454413, 8.0067912454985475))
('10:00:00', (5.2943558025072575, 8.7488327291660362))
('12:00:00', (5.5389561819903044, 9.2817195697529904))
('14:00:00', (5.6468414622366607, 9.4996309673011758))
('16:00:00', (5.6199946175495556, 9.4069418318349722))
('18:00:00', (5.504789343885677, 9.103157421362587))
('20:00:00', (5.3653601048579729, 8.6470786196598564))
('22:00:00', (5.2628633694851175, 8.1518370736988146))
```

1.3 MLlib

NOTE: For improvement of Mllib assignment, we are asked to re-train the model everytime for differnt hour. Here, like in kernel model, we decide to train on all availabe data from previous day and also the data from the target_date before target_hour. The reason here is we think it is more reasonable to train on all availabel data for comparison, and also using filter for hour before 00:00:00 will result in empty RDD, and it could not be trained.

```
[ ]: from __future__ import division
from pyspark import SparkContext
from pyspark.mllib.linalg import Vectors
from pyspark.mllib.feature import StandardScaler
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.tree import DecisionTree
from datetime import datetime
from math import radians, cos, sin, asin, sqrt, exp
from pyspark.broadcast import Broadcast
from pyspark.mllib.tree import RandomForest

# For data handling, the plan is as below
# Use the distance difference from the geographical center of sweden
# Use how many dates have passed since 1950/01/01
# Use the hour difference from 00:00:00
# Which means the features are distance,day_diff, hour_diff

# Function to calculate haversine distance
def haversine(lon1, lat1, lon2=16.321998712, lat2=62.38583179): #geographical␣
↪center of Sweden
    # 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 day_diff(day1, day2="1950-01-01"):
    diff = abs(datetime.strptime(str(day1), "%Y-%m-%d") - datetime.
↪strptime(str(day2), "%Y-%m-%d"))
    no_days = diff.days
    return no_days

def hour_diff(time1, time2="00:00:00"):
    diff = abs(datetime.strptime(time1, "%H:%M:%S") - datetime.strptime(time2,␣
↪"%H:%M:%S"))
    diff = (diff.total_seconds()) / 3600
    return diff


sc = SparkContext(appName="Lab 3 ML")
target_date = '2014-5-17'
target_latitude = 58.68681
target_longitude = 15.92183

target_distance = haversine(lon1=target_longitude, lat1=target_latitude)
target_date_diff = day_diff(day1=target_date)

temperature_file = sc.textFile("BDA/input/temperature-readings.csv")
temperature_data = temperature_file.map(lambda x: x.split(';'))

#filter out data
target_date_strip = datetime.strptime(target_date, '%Y-%m-%d')
prev_temp = temperature_data.filter(lambda x: datetime.strptime(x[1],␣
↪'%Y-%m-%d') <= target_date_strip)


station_file = sc.textFile("BDA/input/stations.csv")
stations_data = station_file.map(lambda x: x.split(';'))
```

```
#Same for kernel model, broadcast distance for faster access
broadcast_stations_distance = sc.broadcast(stations_data.map(lambda x: (x[0],␣
 ↪haversine(lat1=float(x[3]),
                                                                                    ␣
 ↪        lon1=float(x[4])))).collectAsMap())


#temp,day_difference, hour_difference,station_difference
training_temp = prev_temp.map(lambda x: (
    float(x[3]), day_diff(x[1]), hour_diff(x[2]), broadcast_stations_distance.
 ↪value[x[0]]))

#standardized
features = training_temp.map(lambda x: x[1:])
standardizer = StandardScaler()
model = standardizer.fit(features)
features_transform = model.transform(features)
label = training_temp.map(lambda x: x[0])
standardized_data = label.zip(features_transform)
#create training data with standardized features
train_data = standardized_data.map(lambda x: LabeledPoint(x[0], [x[1]])).cache()
#has the form [LabeledPoint(6.8, [407.396549514,0.998736575617,0.0])]


# Create 2 hours interval
hour_list = ["00:00:00", "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"]

prediction = {}
for hour in hour_list:
    target_hour = hour_diff(time1=hour)
    target_feature = Vectors.dense([float(target_date_diff), target_hour,␣
 ↪target_distance])
    target_features_rdd = sc.parallelize([target_feature])
    standardized_target_features = model.transform(target_features_rdd)

    #calculate the threshold to filter out the data, since the train_data is␣
 ↪already normalized
    #so here the threshold is also normalized
    hour_threshold = standardized_target_features.first()[1]
    date_threshold = standardized_target_features.first()[0]

    ##Create models##
    #Like in Kernel model, using filter to keep data that is from previous day␣
↪plus the data from target date before current hour
    current_train_data = train_data.filter(lambda x: x.features[1] <␣
↪hour_threshold or x.features[0] < date_threshold)

    dt_model = DecisionTree.trainRegressor(current_train_data,␣
↪categoricalFeaturesInfo={}, maxDepth= 2)
    rf_model = RandomForest.trainRegressor(current_train_data,␣
↪categoricalFeaturesInfo={}, numTrees=2, maxDepth = 2, maxBins= 4)

    dt_predictions = dt_model.predict(standardized_target_features).collect()[0]
    rf_predictions = rf_model.predict(standardized_target_features).collect()[0]


    prediction[hour]=(rf_predictions,dt_predictions)


sc.parallelize(prediction.items()).coalesce(1).sortByKey().saveAsTextFile("BDA/
↪output/prediction")
```

1.4 Output of RandomForeset, DecisionTree

```
('00:00:00', (3.0985588718032466, 4.113541626536199))
('04:00:00', (3.4251056247749587, 2.943337272515525))
('06:00:00', (0.6748955899868561, 3.6775034998326688))
('08:00:00', (5.781932868414285, 5.141295977786772))
('10:00:00', (6.104406643287962, 7.721845048955009))
('12:00:00', (5.827991294663879, 7.755180135712269))
('14:00:00', (5.879698886679979, 7.728443930850527))
('16:00:00', (6.387115848117571, 6.625958810394981))
('18:00:00', (5.243456398157547, 5.2804620059489435))
('20:00:00', (4.860529197709121, 4.791182468908703))
('22:00:00', (3.9453976616173683, 4.797614388221039))
```

1.4.1 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.

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.

Repeat the exercise using at least two MLlib library models to predict the hourly temperatures for a date and place in Sweden. Compare the results with two Gaussian kernels. If they differ, explain why.

1.4.2 Answer

Ans1 and Ans2 do not change after the updated code.

Ans1. We choose below, kernels' width
h_dist = 100 km h_day = 15 days h_time = 5 hours Since the weather changes greatly across large distances. So we set the kernel width for distance as 100 km, which give larger weight to nearby station that is(< 100km).

For date kernel, since the temperature will varies from different seasons, and we will like to keep the larger weight to the observation closer to the target date. However, it is important to note that, this will also decrease the weight of the same date from previous year, but for simplicity, we keep it as 15

For time(hour) kernel, since we set the target location in a random place in Östergötlands län, and we experient large fluctuations of temperature from day and night recently. Hence, we choose 5 hour as our kernel's width to put more emphasize to closer hour.

Ans2. By observing our prediction, we can see that the prediction using summation model is lower and also does not vary much compare to multiplication model. This is due to the difference of combanation For example, if there is a observation that is at the same hour, one day before target date, but 1000km away. Using summation model it will still result in large weight because of hour and date. On the other hand, using multiplication the small weight of distance will keep the overall weight down. We will say that the multiplication model is more ideal because we need to consider all three kernels when doing prediction.

Ans3. For the updated code, we use RandomForest and Decision Tree within the loop to train for each hour.

The first model we use RandomForeset(RF), since last time when we use LinearRegression(LR), the result of LR was not ideal and requires higer run time in order to use better parameters to avoid large weight and wrong prediction result.

The Second model we use is Decision Tree regression(DT) since it is usable to capture non-linear relationship between variables (same for RF).

In terms of result, the RF prediction has higer temperature druing noon and decresase over night, which is as expected. However, since the time constraint we need to tune the parameter to lower tree count, shallow depth, smallerBins. These parameters setting makes the result of 06:00:00 being very low. Using better parameters can improve this, but will take much longer to run.

For Decision Tree(DT), the time of higher and lower temperature are also as expected. We also use shallow depth here to improve speed.

Overall, we will say that both RF and DT prediction are similar but not perfect. Also, the way we handle data (by giving reference starting point for location and date) have make our MLlib model result different to Gaussian kernels.

[ ]:
```

```