sparkmachinelearning_report

May 21, 2023

[ ]:
```
import sys
from __future__ import division
from math import radians, cos, sin, asin, sqrt, exp
from datetime import datetime
from pyspark import SparkContext, StorageLevel

sc = SparkContext(appName = "Lab")
```

[ ]:
```
# sc.stop()
```

[ ]:
```
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 d_date(date1,date2):
    """
    Calculate the distance between two day
    """
    date_diff=abs((date2 - date1).days)
    return date_diff

def d_time(time1,time2):
    """
    Calculate the distance between two time
```

```
    """
    time_diff=abs((time2 - time1).total_seconds() / 3600)
    return time_diff

def sum_kernel(d_distance,d_date,d_time):
    """
    Calculate the sum of the kernel
    """
    distance_kernel=exp(-(d_distance**2)/(h_distance**2))
    time_kernel=exp(-(d_time**2)/(h_time**2))
    date_kernel=exp(-(d_date**2)/(h_date**2))

    return distance_kernel+time_kernel+date_kernel

def prod_kernel(d_distance,d_date,d_time):
    """
    Calculate the prod of the kernel
    """
    distance_kernel=exp(-(d_distance**2)/(h_distance**2))
    time_kernel=exp(-(d_time**2)/(h_time**2))
    date_kernel=exp(-(d_date**2)/(h_date**2))

    return distance_kernel*time_kernel*date_kernel


h_distance = 480
h_date = 6
h_time = 3
a = 58.4274 # Up to you
b = 14.826 # Up to you
date = "2014-11-04" # Up to you
```

[ ]:
```
DATE_TIMESTAMP=datetime.strptime(date,"%Y-%m-%d")
stations = sc.textFile("s3://bigdatalabs7051/data/stations/stations.csv")
temps = sc.textFile("s3://bigdatalabs7051/data/temperature-readings/
 ↪temperature-readings.csv")
# lines = temps.map(lambda line: line.split(";")).map(lambda x:
 ↪(str(x[0]),datetime.strptime(x[1],"%Y-%m-%d"),datetime.strptime(x[2],"%H:%M:
 ↪%S"),float(x[3])))

# Broadcast
# Station data: station_num, distance
stations_data=stations.map(lambda line: line.split(";")).map(lambda x:
 ↪(str(x[0]),haversine(b,a,float(x[4]),float(x[3]))))
```

```
bc=sc.broadcast(stations_data.collectAsMap())     # Convert the rdd␣
 ↪stations_data to dictionary, then broadcast it.

# Join
# joined_data: station_num, (geo distance, date distance, time, temperature)
joined_data = temps.sample(False, 0.1).map(lambda line: line.split(";"))\
                .map(lambda x:(str(x[0]),(datetime.
 ↪strptime(x[1],"%Y-%m-%d"),datetime.strptime(x[2],"%H:%M:%S"),float(x[3]))))\
                .filter(lambda x: (x[1][0]<DATE_TIMESTAMP))\
                .map(lambda x: (x[0],(bc.
 ↪value[x[0]],d_date(x[1][0],DATE_TIMESTAMP),x[1][1],x[1][2])))

# Persistence
joined_data.cache()

# print(joined_data.collect())
```

```
PythonRDD[5] at RDD at PythonRDD.scala:53
```

0.1 Task 1: Use a kernel that is the sum of three Gaussian kernels

[ ]:
```
y_sum=[]

for time in ["0:00:00", "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"]:
    # Your code here

    # sum_kernel_rdd: 1, (sum_kernel, temperature)
    #sum_kernel_rdd=joined_data.map(lambda x:
 ↪(1,(sum_kernel(x[1][0],x[1][1],d_time(time,x[1][2])),x[1][3])))
    #sum_kernel_rdd.cache()
    #denominator=sum_kernel_rdd.groupByKey().mapValues(lambda values: sum([v[0]␣
 ↪for v in values]))
    #numerator=sum_kernel_rdd.groupByKey().mapValues(lambda values:␣
 ↪sum([v[0]*v[1] for v in values]))

    # Avoid using groupByKey and mapValues because it's much slower than␣
 ↪reduceByKey

    # sum_kernel_rdd: 1, (sum_kernel, temperature)
    time=datetime.strptime(time,"%H:%M:%S")
    sum_kernel_rdd=joined_data.map(lambda x:
 ↪(sum_kernel(x[1][0],x[1][1],d_time(time,x[1][2])),x[1][3]))
    sum_kernel_rdd=sum_kernel_rdd.map(lambda x:(x[0],x[0]*x[1])).reduce(lambda␣
 ↪x,y:(x[0]+y[0],x[1]+y[1]))
```

```
    #The result type returned by the reduce operation is consistent with the␣
↪element type in the RDD. What is returned here is a tuple
    y_sum.append(sum_kernel_rdd[1]/sum_kernel_rdd[0])
```

[ ]:
```
time_list=["0:00:00", "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"]
dictionary = {k: v for k, v in zip(time_list, y_sum)}
print("latitude,longitude:",a,",",b)
print("date:",date)
print(dictionary)
print("HyperParams:")
print("h_distance:",h_distance)
print("h_date:",h_date)
print("h_time:",h_time)
```

```
latitude,longitude: 58.4274 , 14.826
date: 2014-11-04
{'0:00:00': 5.576631957058849, '22:00:00': 5.746327086093085, '20:00:00':
5.736053132088273, '18:00:00': 5.869071519404494, '16:00:00': 6.082216782048857,
'14:00:00': 6.252550240405444, '12:00:00': 6.255958093164522, '10:00:00':
6.0550847452374175, '08:00:00': 5.685894263735923, '06:00:00':
5.366830140637157, '04:00:00': 5.275485830982766}
HyperParams:
h_distance: 480
h_date: 6
h_time: 3
```

Q1: 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.

Answer: The parameter of the Gaussian kernel function is the bandwidth, which determines the decay speed and smoothness of the kernel function. The smaller the bandwidth, the narrower the scope of the kernel function, the greater the impact of the current parameter change (in other words: more consideration of the current parameter), and vice versa.

According to common sense, time has the greatest impact on temperature, and date also has a certain impact on temperature. For example, winter is colder and summer is hotter, and the impact of distance is the least obvious (the distance from Oslo to Copenhagen is 563km, but the temperature difference is sometimes less than 1 degree Celsius). . Therefore the choice for time bandwidth should be the smallest and for the distance bandwidth the choice should be the largest.

We choose h_distance=483 because the distance from Uppsala to Tampere is 483km, and the average temperature difference between the two places on May 18 is 5 degrees Celsius.

Choose h_date=6, because Linkoping's average daily temperature increased by 5 degrees Celsius from May 18 to May 24.

Choose h_time=3, because Linkoping May 18 from 7:00 to 10:00, the temperature increased by 5

degrees Celsius

0.2 Task 2: Use a kernel that is the prod of three Gaussian kernels

[ ]:
```
y_prod=[]

for time in ["0:00:00", "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"]:
    # Your code here
    # prod_kernel_rdd: 1, (prod_kernel, temperature)
    time=datetime.strptime(time,"%H:%M:%S")
    prod_kernel_rdd=joined_data.map(lambda x:
 ↪(prod_kernel(x[1][0],x[1][1],d_time(time,x[1][2])),x[1][3]))
    prod_kernel_rdd=prod_kernel_rdd.map(lambda x:(x[0],x[0]*x[1])).
 ↪reduce(lambda x,y:(x[0]+y[0],x[1]+y[1]))
    #The result type returned by the reduce operation is consistent with the␣
 ↪element type in the RDD. What is returned here is a tuple
    y_prod.append(prod_kernel_rdd[1]/prod_kernel_rdd[0])
```

[ ]:
```
time_list=["0:00:00", "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"]
dictionary = {k: v for k, v in zip(time_list, y_prod)}
print("latitude,longitude:",a,",",b)
print("date:",date)
print(dictionary)
print("HyperParams:")
print("h_distance:",h_distance)
print("h_date:",h_date)
print("h_time:",h_time)
```

```
latitude,longitude: 58.4274 , 14.826
date: 2014-11-04
{'0:00:00': 6.925568411088358, '22:00:00': 7.7684882157681265, '20:00:00':
7.920901094219781, '18:00:00': 8.067762477225351, '16:00:00': 8.378497070124379,
'14:00:00': 8.823294422587335, '12:00:00': 8.914151426197472, '10:00:00':
8.446565636573004, '08:00:00': 7.774179683273953, '06:00:00': 7.265615572246946,
'04:00:00': 6.994351544760919}
HyperParams:
h_distance: 480
h_date: 6
h_time: 3
```

Q2: 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.

Answer: Using the product of the three Gaussian kernels, overall, the predicted air temperature is higher than using the kernel that is the sum of three Gaussian kernels.

The product of Gaussian kernel functions can enhance the similarity between samples, that is, when the values of the three Gaussian kernel functions are larger at the same time, the similarity is higher. Conversely, if one of the Gaussian kernels has a lower value, then the kernel value will also be lower. Therefore, it pays more attention to the common characteristics of the three samples in the feature space. The sum of Gaussian kernel functions can comprehensively consider the similarity between samples. It pays more attention to the respective characteristics of the three samples in the feature space, and considers them comprehensively by adding.

0.3 Task 3: Using at least two MLlib library models to predict the hourly temperatures for a date and place in Sweden.

0.3.1 Linear Regression

[ ]:
```
from datetime import date

# Transform time to float,range from 0-100
def time_to_float(time):
    hour, minute, sec= map(int, time.split(":"))
    total_minutes = hour * 60 + minute
    relative_minutes = total_minutes / (24 * 60)
    return relative_minutes * 100

# Transform date to float,range from 0-100
def date_to_float(date_str):
    year, month, day = map(int, date_str.split("-"))
    date_obj = date(year, month, day)
    year_start = date(year, 1, 1)
    year_end = date(year, 12, 31)
    total_days = (date_obj - year_start).days
    relative_days = total_days / (year_end - year_start).days
    return relative_days * 100
```

[ ]:
```
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.regression import LinearRegressionWithSGD

# Broadcast
# station_num, longitude, latitude
stations_data=stations.map(lambda line: line.split(";")).map(lambda x:
 ↪(str(x[0]),(float(x[4]),float(x[3]))))
bc=sc.broadcast(stations_data.collectAsMap())    # Convert the rdd␣
 ↪stations_data to dictionary, then broadcast it.

# LabeledPoint(labeled_data) : temperature, longitude, latitude, date, time
labeled_data = temps.sample(False, 0.1).map(lambda line: line.split(";"))\
```

```
                .map(lambda x:(bc.
 ↪value[str(x[0])],date_to_float(x[1]),time_to_float(x[2]),float(x[3])))\
                .map(lambda x: LabeledPoint(x[3], [x[0][0], x[0][1], x[1],␣
 ↪x[2]]))

train_data, test_data = labeled_data.randomSplit([0.7, 0.3])

# Train the linear regression model
model = LinearRegressionWithSGD.train(train_data, iterations=100, step=0.001)

print("Training complete!")
```

```
AttributeError: 'PipelinedRDD' object has no attribute '_jdf'
```

[ ]:
```
# Predict!
a = 58.4274 # latitude
b = 14.826 # longitude
date1 = "2014-11-04" # Up to you

y_linreg=[]
predict_data=[]
for time1 in ["0:00:00", "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"]:
    # Your code here
    predict_data.append([b,a,date_to_float(date1),time_to_float(time1)])

rdd1 = sc.parallelize(predict_data)
print("Training data:")
print(rdd1.collect())
predictions = model.predict(rdd1)
y_linreg=predictions.collect()

print("Predict result:")
print(y_linreg)
```

```
Training data:
[[14.826, 58.4274, 84.34065934065934, 0.0], [14.826, 58.4274, 84.34065934065934,
91.66666666666666], [14.826, 58.4274, 84.34065934065934, 83.33333333333334], [14.826, 58.4274,
84.34065934065934, 75.0], [14.826, 58.4274, 84.34065934065934, 66.66666666666666], [14.826,
58.4274, 84.34065934065934, 58.333333333333336], [14.826, 58.4274, 84.34065934065934,
50.0], [14.826, 58.4274, 84.34065934065934, 41.66666666666667], [14.826, 58.4274,
84.34065934065934, 33.33333333333333], [14.826, 58.4274, 84.34065934065934, 25.0], [14.826,
58.4274, 84.34065934065934, 16.666666666666664]]
Predict result:
[5.523545240468239, 8.100466489788737, 7.866200921668692, 7.631935353548646,
7.397669785428601, 7.163404217308555, 6.92913864918851, 6.694873081068465,
6.460607512948419, 6.226341944828374, 5.992076376708329]
```

0.3.2 ridge regression

[ ]:
```
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.regression import RidgeRegressionWithSGD

# Broadcast
# station_num, longitude, latitude
stations_data=stations.map(lambda line: line.split(";")).map(lambda x:
 ↪(str(x[0]),(float(x[4]),float(x[3]))))
bc=sc.broadcast(stations_data.collectAsMap())    # Convert the rdd␣
 ↪stations_data to dictionary, then broadcast it.

# LabeledPoint(labeled_data) : temperature, longitude, latitude, date, time
labeled_data = temps.sample(False, 0.1).map(lambda line: line.split(";"))\
                .map(lambda x:(bc.
 ↪value[str(x[0])],date_to_float(x[1]),time_to_float(x[2]),float(x[3])))\
                .map(lambda x: LabeledPoint(x[3], [x[0][0], x[0][1], x[1],␣
 ↪x[2]]))

train_data, test_data = labeled_data.randomSplit([0.7, 0.3])

# Train the linear regression model
model2 = RidgeRegressionWithSGD.train(train_data, iterations=100, step=0.0005)
```

[ ]:
```
# Predict!
a = 58.4274 # latitude
b = 14.826 # longitude
date1 = "2014-11-04" # Up to you

y_linreg=[]
predict_data=[]
for time1 in ["0:00:00", "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"]:
    # Your code here
    predict_data.append([b,a,date_to_float(date1),time_to_float(time1)])

rdd1 = sc.parallelize(predict_data)
print("Training data:")
print(rdd1.collect())
predictions = model2.predict(rdd1)
y_linreg=predictions.collect()

print("Predict result:")
print(y_linreg)
```

Training data:
[[14.826, 58.4274, 84.34065934065934, 0.0], [14.826, 58.4274, 84.34065934065934,

91.66666666666666], [14.826, 58.4274, 84.34065934065934, 83.33333333333334], [14.826, 58.4274, 84.34065934065934, 75.0], [14.826, 58.4274, 84.34065934065934, 66.66666666666666], [14.826, 58.4274, 84.34065934065934, 58.333333333333336], [14.826, 58.4274, 84.34065934065934, 50.0], [14.826, 58.4274, 84.34065934065934, 41.66666666666667], [14.826, 58.4274, 84.34065934065934, 33.33333333333333], [14.826, 58.4274, 84.34065934065934, 25.0], [14.826, 58.4274, 84.34065934065934, 16.666666666666664]]
Predict result:
[5.543807177181344, 8.101139486463167, 7.868654731073911, 7.636169975684654, 7.403685220295397, 7.171200464906141, 6.9387157095168845, 6.706230954127628, 6.473746198738371, 6.2412614433491145, 6.008776687959857]

Q3: 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

Answer: we can see that the predicted result from LinearRegressionWithSGD and RidgeRegressionWithSGD are all similar to each other(there isn't too much features and the regularizer parameter is set as 0.01 by default which didn't make too significant impact). And the regression results are similar to kernel results especially the prod kernel