Lab BDA3 - Machine Learning with Spark

Umamaheswarababu Maddela (umama339)

Dinesh Sundaramoorthy (dinsu875)

ASSIGNMENT

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.

Choose an appropriate smoothing coefficient or width for each of the three kernels above. You do not need to use cross-validation.

```
# Pyspark implementation to forecast temperatures using a kernel that is the
sum of three Gaussian kernels

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

# Initialize SparkContext
sc = SparkContext(appName="lab_kernel")

# Haversine function to calculate distance between two points on the earth
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 = lat1 - lat2
    a = sin(dlat / 2)**2 + cos(lat1) * cos(lat2) * sin(dlon / 2)**2
    c = 2 * asin(sqrt(a))
    km = 6367 * c
    return km
```

```python
# kernel widths
h_distance = 500  # for distance
h_date = 20  #  for date
h_time = 4  #  for time

# Coordinates and date for prediction
a = 58.4274  # Latitude
b = 14.826  # Longitude
target_date = "2013-07-04"  # Date of interest

# Define Gaussian kernel
def gaussian_kernel(x, sigma):
    return exp(-(x ** 2) / (2 * sigma ** 2))

# Define kernel for distance
def haversine_distance_kernel(lon1, lat1, lon2, lat2, h_distance):
    dist = haversine(lon1, lat1, lon2, lat2)
    return gaussian_kernel(dist, h_distance)

# Define kernel for date difference
def date_kernel_distance(date1, date2, h_date):
    date_1 = datetime.strptime(date1, '%Y-%m-%d')
    date_2 = datetime.strptime(date2, '%Y-%m-%d')
    date_dist = abs((date_1 - date_2).days)
    return gaussian_kernel(date_dist, h_date)

# Define kernel for time difference
def time_kernel_distance(time1, time2, h_time):
    time_1 = int(time1[:2]) + int(time1[3:5]) / 60
    time_2 = int(time2[:2]) + int(time2[3:5]) / 60
    time_dist = abs(time_1 - time_2)
    return gaussian_kernel(time_dist, h_time)

# Load and process station data
stations = sc.textFile("/content/drive/MyDrive/big_data/stations.csv")
station_lines = stations.map(lambda line: line.split(";"))

# Compute distance kernels
station_kernels = station_lines.map(lambda x: (x[0],
haversine_distance_kernel(a, b, float(x[4]), float(x[3]), h_distance)))
broadcast_stations = sc.broadcast(station_kernels.collectAsMap())

# Load and process temperature data
temps = sc.textFile("/content/drive/MyDrive/big_data/temperature-
readings.csv")
temps_lines = temps.map(lambda line: line.split(";"))
#temps_lines = temps_lines.sample(False, 0.1)

# Filter out the observations posterior to target date
target_datetime = datetime.strptime(target_date, '%Y-%m-%d')
filtered_temps_lines = temps_lines.filter(lambda x: datetime.strptime(x[1],
'%Y-%m-%d') <= target_datetime)
filtered_temps_lines = filtered_temps_lines.cache()  # Cache the filtered
```

```
data

# Function to filter out the observations posterior to target date and time
def filter_temps(record, target_datentime):
    record_datentime = datetime.strptime(record[1] + " " + record[2], '%Y-%m-
%d %H:%M:%S')
    return record_datentime < target_datentime

# Times to predict
times_to_predict = ["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"]

# Initialize a list to collect predictions
predictions = []

# Calculate predictions for each time
for time in times_to_predict:
    if time == "24:00:00":
        # to make "24:00:00" as the start of the next day
        target_datentime = datetime.strptime(target_date, '%Y-%m-%d') +
timedelta(days=1)
        target_datentime = target_datentime.replace(hour=0, minute=0,
second=0)
    else:
        target_datentime = datetime.strptime(target_date + " " + time, '%Y-
%m-%d %H:%M:%S')

    filt_temps = filtered_temps_lines.filter(lambda x: filter_temps(x,
target_datentime))
    mapped_temps = filt_temps.map(lambda x: (x[0], (x[1], x[2], float(x[3]),
broadcast_stations.value.get(x[0], 1e-5))))

    kernel_data = mapped_temps.map(lambda record: (
        record[1][2],  # Temperature
        record[1][3] + date_kernel_distance(record[1][0], target_date,
h_date) + time_kernel_distance(record[1][1], time, h_time)  # Combined kernel
weight (sum of kernels)
    ))

    sum_kernels, sum_wt_temps = kernel_data.map(lambda x: (x[1], x[1] *
x[0])).reduce(lambda x, y: (x[0] + y[0], x[1] + y[1]))

    predicted_temp = sum_wt_temps / sum_kernels

    predictions.append((time, predicted_temp))

# Convert predictions to RDD and save to text file
predictions_rdd = sc.parallelize(predictions)

# Save the predictions as a text file
predictions_rdd.coalesce(1).saveAsTextFile("/content/drive/MyDrive/big_data/B
DA/output")
```

```
sc.stop()
```

Output:

```
('24:00:00', 4.300139230711549)
('22:00:00', 4.554811229350367)
('20:00:00', 4.883683985351379)
('18:00:00', 5.248094848472269)
('16:00:00', 5.575474896309902)
('14:00:00', 5.7431173691897)
('12:00:00', 5.62649700793191)
('10:00:00', 5.1995338592711065)
('08:00:00', 4.58002126569909)
('06:00:00', 3.9509991054038283)
('04:00:00', 3.4428266352659977)
```

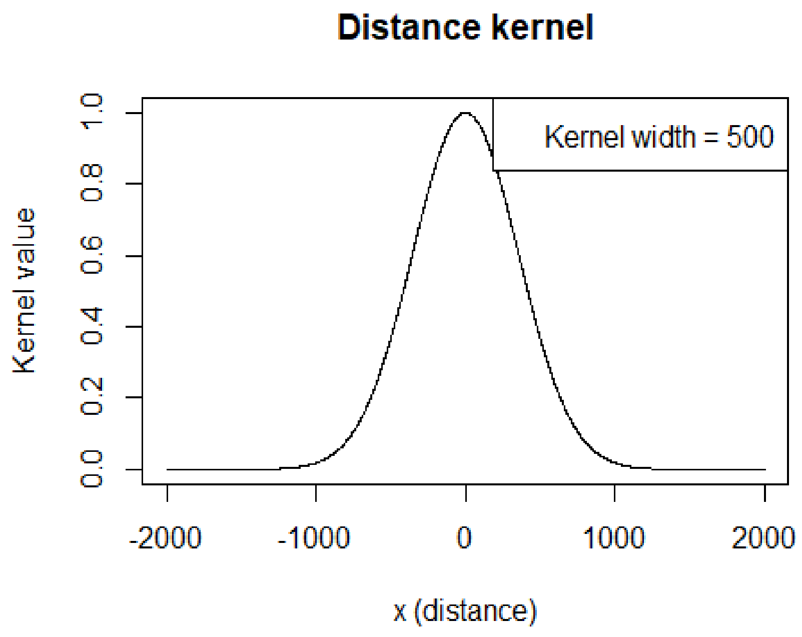
Question 1:

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:

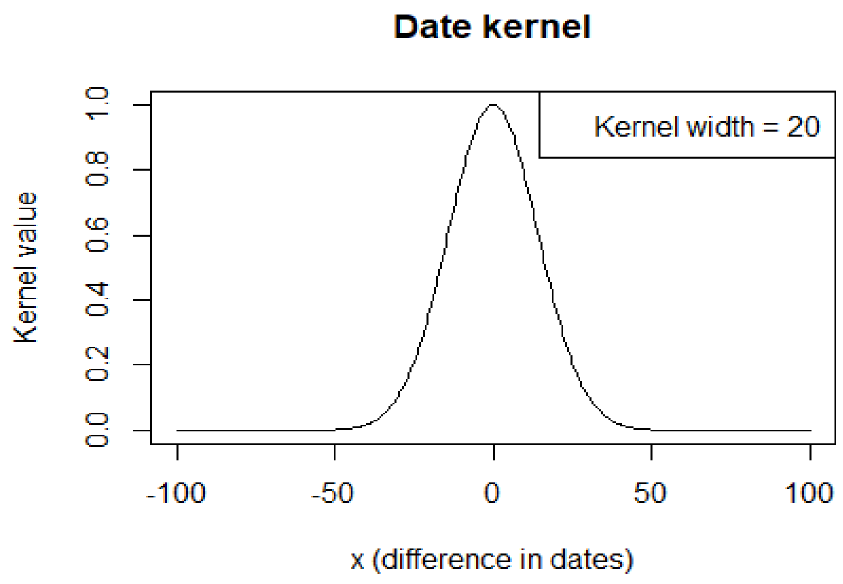
The three kernel widths are chosen as `h_distance = 500`, `h_date = 20`, and `h_time = 4`. The plots below demonstrate how the Gaussian kernel value decreases with distance, date difference, and time difference.

1. **Distance Kernel (`h_distance = 500`)**:



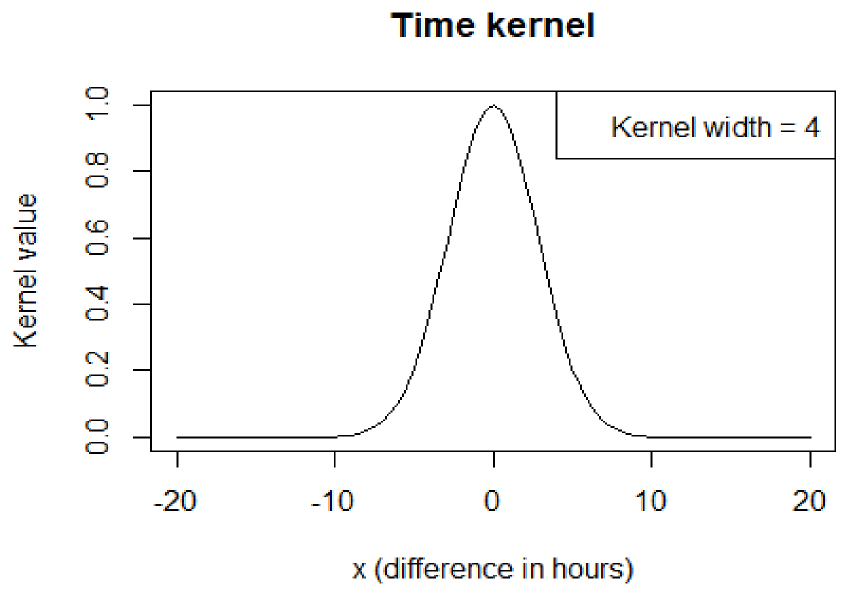
The plot shows that for distances greater than approximately 1000 km, the kernel value is very low, indicating that these points have less influence on the prediction. This demonstrates that closer points (within 500 km) are given higher weight, making the kernel width sensible.

2. **Date Kernel (`h_date = 20`)**:



The plot indicates that the kernel value drops significantly for date differences greater than approximately 50 days. This suggests that recent temperature readings (within 20 days) have more influence on the prediction, while older readings are less relevant.

3. **Time Kernel (`h_time = 4`)**:



The plot shows that for time differences greater than approximately 10 hours, the kernel value is very low, indicating that observations far apart in time are less relevant. This gives higher weight to observations close in time (within 4 hours).

Question 2:

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.

```
# Pyspark implementation to forecast temperatures using a kernel that is the
product of three Gaussian kernels

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

# Initialize SparkContext
sc = SparkContext(appName="lab_kernel")

# Haversine function to calculate distance between two points on the earth
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 = lat1 - lat2
    a = sin(dlat / 2)**2 + cos(lat1) * cos(lat2) * sin(dlon / 2)**2
    c = 2 * asin(sqrt(a))
    km = 6367 * c
    return km

# kernel widths
h_distance = 500  # for distance
h_date = 20  #  for date
h_time = 4  #  for time

# Coordinates and date for prediction
a = 58.4274  # Latitude
b = 14.826  # Longitude
target_date = "2013-07-04"  # Date of interest

# Define Gaussian kernel
def gaussian_kernel(x, sigma):
    return exp(-(x ** 2) / (2 * sigma ** 2))

# Define kernel for distance
def haversine_distance_kernel(lon1, lat1, lon2, lat2, h_distance):
    dist = haversine(lon1, lat1, lon2, lat2)
    return gaussian_kernel(dist, h_distance)

# Define kernel for date difference
def date_kernel_distance(date1, date2, h_date):
    date_1 = datetime.strptime(date1, '%Y-%m-%d')
    date_2 = datetime.strptime(date2, '%Y-%m-%d')
    date_dist = abs((date_1 - date_2).days)
    return gaussian_kernel(date_dist, h_date)

# Define kernel for time difference
def time_kernel_distance(time1, time2, h_time):
    time_1 = int(time1[:2]) + int(time1[3:5]) / 60
    time_2 = int(time2[:2]) + int(time2[3:5]) / 60
    time_dist = abs(time_1 - time_2)
    return gaussian_kernel(time_dist, h_time)

# Load and process station data
stations = sc.textFile("/content/drive/MyDrive/big_data/stations.csv")
station_lines = stations.map(lambda line: line.split(";"))

# Compute distance kernels
station_kernels = station_lines.map(lambda x: (x[0],
haversine_distance_kernel(a, b, float(x[4]), float(x[3]), h_distance)))
broadcast_stations = sc.broadcast(station_kernels.collectAsMap())

# Load and process temperature data
temps = sc.textFile("/content/drive/MyDrive/big_data/temperature-
readings.csv")
temps_lines = temps.map(lambda line: line.split(";"))
#temps_lines = temps_lines.sample(False, 0.1)

# Filter out the observations posterior to target date
target_datetime = datetime.strptime(target_date, '%Y-%m-%d')
filtered_temps_lines = temps_lines.filter(lambda x: datetime.strptime(x[1],
'%Y-%m-%d') <= target_datetime)
filtered_temps_lines = filtered_temps_lines.cache()  # Cache the filtered
data

# Function to filter out the observations posterior to target date and time
def filter_temps(record, target_datentime):
    record_datentime = datetime.strptime(record[1] + " " + record[2], '%Y-%m-
%d %H:%M:%S')
    return record_datentime < target_datentime

# Times to predict
times_to_predict = ["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"]

# Initialize a list to collect predictions
predictions = []

# Calculate predictions for each time
for time in times_to_predict:
    if time == "24:00:00":
        # to make "24:00:00" as the start of the next day
        target_datentime = datetime.strptime(target_date, '%Y-%m-%d') +
timedelta(days=1)
        target_datentime = target_datentime.replace(hour=0, minute=0,
second=0)
    else:
        target_datentime = datetime.strptime(target_date + " " + time, '%Y-
%m-%d %H:%M:%S')

    filt_temps = filtered_temps_lines.filter(lambda x: filter_temps(x,
target_datentime))
    mapped_temps = filt_temps.map(lambda x: (x[0], (x[1], x[2], float(x[3]),
broadcast_stations.value.get(x[0], 1e-5))))

    kernel_data = mapped_temps.map(lambda record: (
        record[1][2],  # Temperature
        record[1][3] * date_kernel_distance(record[1][0], target_date,
h_date) * time_kernel_distance(record[1][1], time, h_time)  # Combined kernel
weight (product of kernels)
    ))

    sum_kernels, sum_wt_temps = kernel_data.map(lambda x: (x[1], x[1] *
x[0])).reduce(lambda x, y: (x[0] + y[0], x[1] + y[1]))

    predicted_temp = sum_wt_temps / sum_kernels

    predictions.append((time, predicted_temp))

# Convert predictions to RDD and save to text file
predictions_rdd = sc.parallelize(predictions)

# Save the predictions as a text file
predictions_rdd.coalesce(1).saveAsTextFile("/content/drive/MyDrive/big_data/B
DA/output")

sc.stop()
```

Output:

```
('24:00:00', 14.152919451049613)
('22:00:00', 14.505265759680233)
('20:00:00', 14.974178336798925)
('18:00:00', 15.500739022141314)
('16:00:00', 15.996286592638887)
('14:00:00', 16.32004610128123)
('12:00:00', 16.349111878155043)
('10:00:00', 16.05219855289566)
('08:00:00', 15.492834110518537)
('06:00:00', 14.814234310578946)
('04:00:00', 14.166780624946137)
```

Answer:

The results differ significantly between the sum of kernels and the product of kernels. When using the sum, the predicted temperatures are more moderate because it averages the influences from nearby points. With the product of kernels, the predictions are much higher because multiplying the kernel weights amplifies the influence of points that are very close in space and time. This amplification can lead to overestimation of the temperatures, which is why the results are higher with the product method.