Lab 3 Block 1: Kernel Meethods and Neural Networks

Maximilian Pfundstein

2018-12-14

Contents

1 Kernel Methods

Task:

- Implement a kernel method to predict the hourly temperatures for a date and place in Sweden.
- You are asked to provide a temperature forecast for a date and place in Sweden. The forecast should consist of the predicted te mperatures from 4 am to 24 pm in an interval of 2 hours.
- 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.
- Show that your choice for the kernels' width is sensible, i.e. that it gives more weight to closer points. Discuss why your of definition of closeness is reasonable.
- Instead of combining the three kernels into one by summing them up, multiply them. Compare the results obtained in both cases and elaborate on why they may differ.

Note that the file temps50k.csv may contain temperature measurements that are posterior to the day and hour of your forecast. You must filter such measurements out, i.e. they cannot be used to compute the forecast. Feel free to use the template below to solve the assignment.

1.1 Kernels

This exercise uses three kernels which are combined to predict the temperatures on a given location and day. We will have a look at those kernels, how they're implemented and how the smooth parameters have been selected. Before that we will have a look at one helper function which is used by the kernels.

1.1.1 Helper Function

The helper function `min_distance(a, b, ringSize)` calculates the minimal distance between two elements on a discrete Quotient Ring. Let's say we have the two times `03:00` and `23:00`. The closes distance would be `4` hours, not `20:00` hours. We use the function for this example and for calculating the difference between two days as well. The `ringSize` thus ist `24` and `356` (we ignore all teh special cases we haev about years and times, like leap seconds). The function uses two applies to handle the calculcation of a given element `a` with `1:n` elements of `b`. The `pmin()` function is parallelized, so it's somewhat fast (we're still using R in the end).

```
################################################################################
# Kernel Methods
################################################################################

## Helper Functions

min_distance = function(a, b, ringSize) {

  boundOne = sapply(b, FUN = function(x) abs(a - x))
  boundTwo = sapply(b, FUN = function(x) abs(a + ringSize - x))

  return(pmin(boundOne, boundTwo))
}
```

1.1.2 Kernel Implementation

The following code shows the implementation of the three kernels. Each of them takes two elements, where `b` can be of size `1:n` like the `min_distance(a, b, ringSize)` function for faster computation. Each kernel takes as well a `smoothing` parameter and returns the results using the Gaussian Kernel.

- The `kernel_gauss_distance` calculates the distance between two points using the *haversine method*.
- The `kernel_gauss_day` calcualtes the difference between two days. It ignores the year (as we use the cycical behavior and do not count in any long term climate changes) and measures the distance using the `min_distance(a, b, ringSize)` function which results in values between `(0, 183)`.
- The `kernel_gauss_hour` calculates the difference between two times on a day. It ignores the day (again, as we are intersted in teh cyclical behavior) and measures the distance using the `min_distance(a, b, ringSize)` function which results in values between `(0, 12)`.

```
## Kernels
## The definition for the Guassian Kernel is taken from the slides, page 6.

kernel_gauss_distance = function(pointA, pointB, smoothing) {

  # Use distHaversine() as a help
  m = cbind(pointB$longitude, pointB$latitude)
  u = distHaversine(m, c(pointA$longitude, pointA$latitude))
  u = u / smoothing
  return(exp(-(u^2)))
}

kernel_gauss_day = function(dayA, dayB, smoothing) {

  dayA_in_days = as.numeric(strftime(as.Date(dayA$date), '%j'))
  dayB_in_days = as.numeric(strftime(as.Date(dayB$date), '%j'))

  u = min_distance(dayA_in_days, dayB_in_days, 365)
```

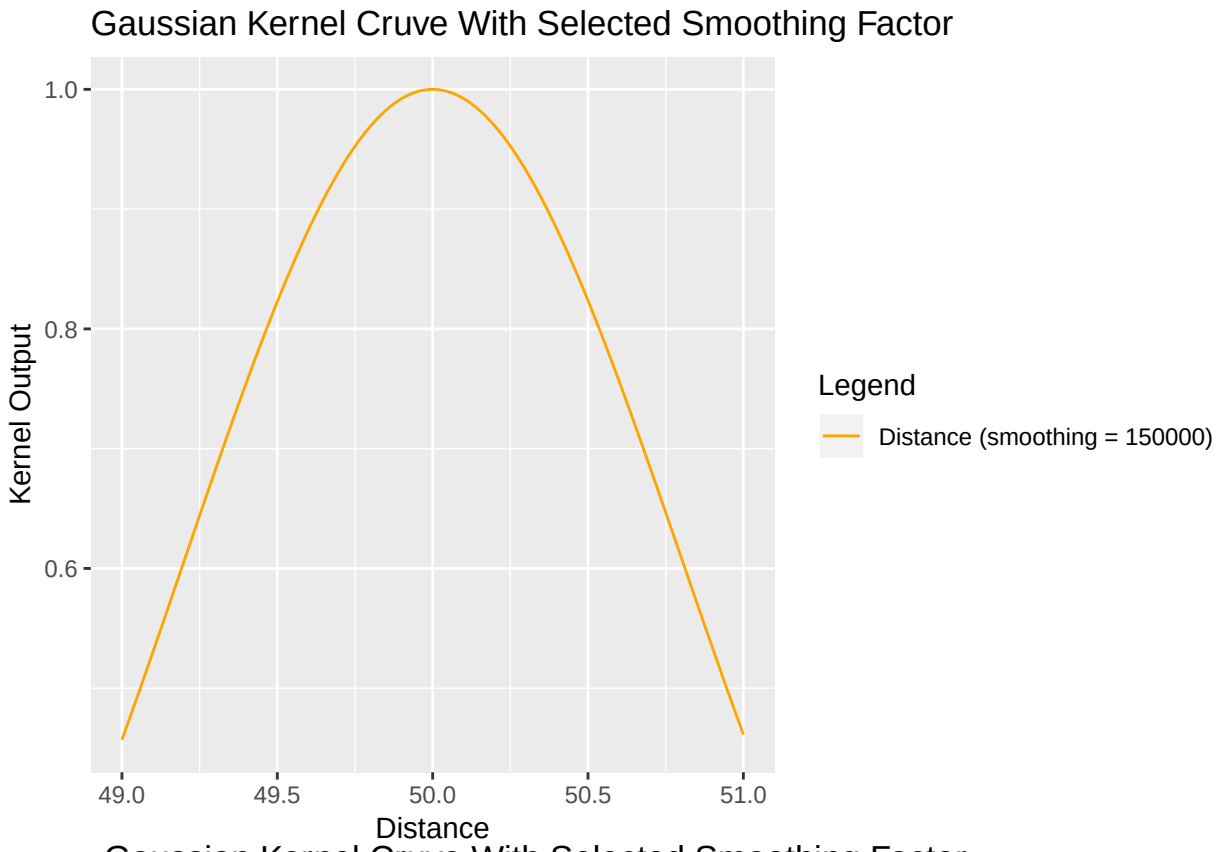
```
  u = u / smoothing
  return(exp(-(u^2)))
}

kernel_gauss_hour = function (hourA, hourB, smoothing) {

  hourA_in_h = sapply(hourA$time, FUN = function(x)
    as.numeric(difftime(strptime(x, format = "%H:%M:%S"),
                        strptime("00:00:00", format = "%H:%M:%S"))))

  hourB_in_h = sapply(hourB$time, FUN = function(x)
    as.numeric(difftime(strptime(x, format = "%H:%M:%S"),
                        strptime("00:00:00", format = "%H:%M:%S"))))

  u = min_distance(hourA_in_h, hourB_in_h, 24)
  u = u / smoothing
  return(exp(-(u^2)))
}
```

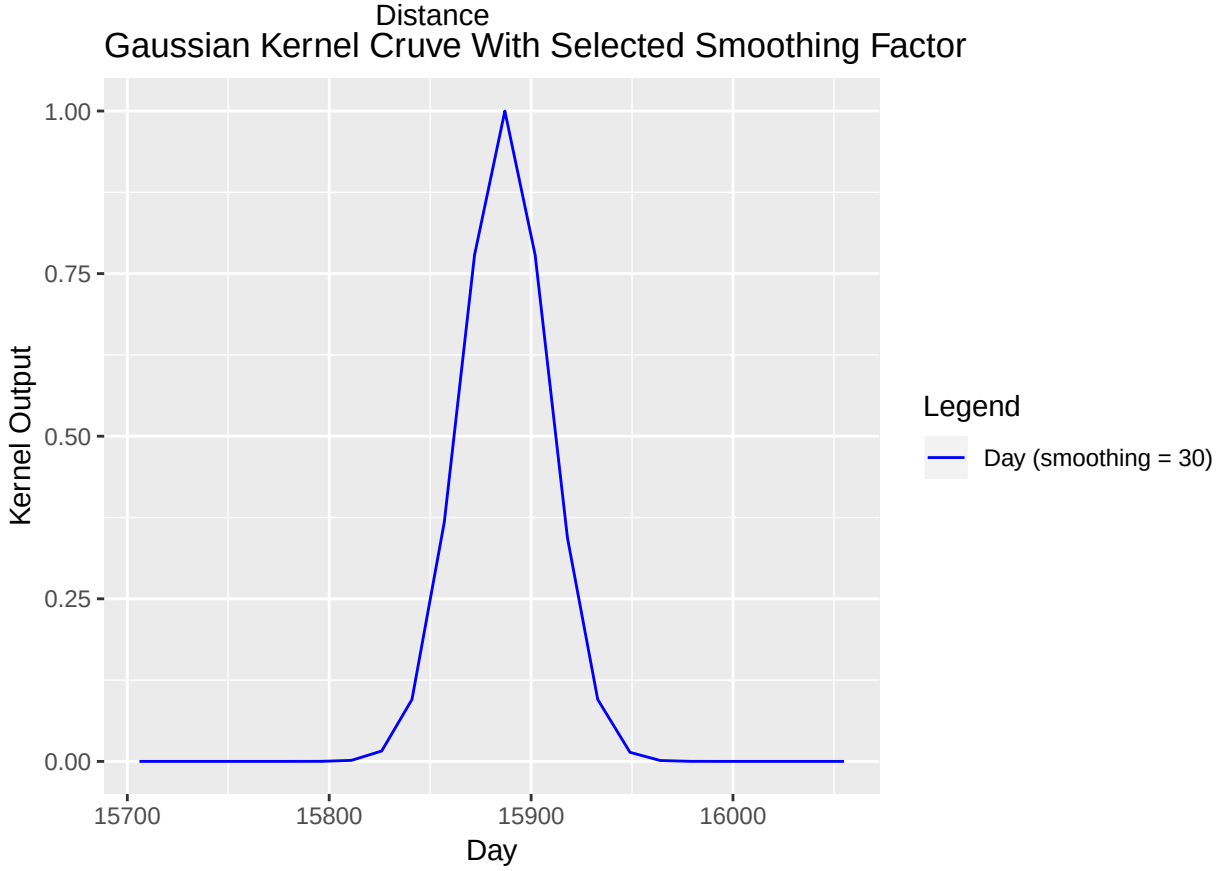
1.1.3 Smoothing Parameters

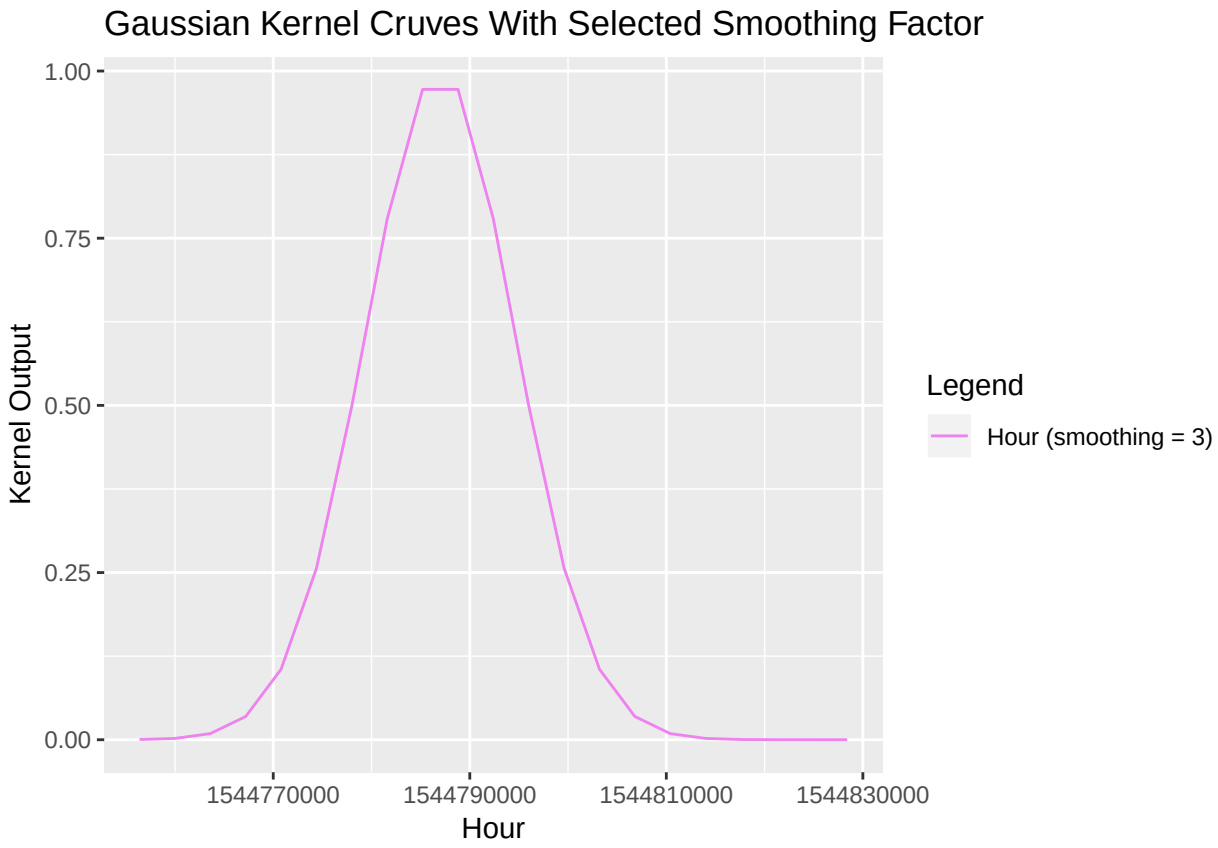
To receive reasonable values for the smoothing parameters we first define some range in which we actually want to consider features. While this is easy for the date and time (one year and one day) it's not that obvious for the distance. We choose `2` for latitude and longitude. Then we adjusted the smoothing parameter that it barely covers the range (so that the features farthest apart barely get a value `> 0`). From that point onwards we decrease/increase the area to look for, thus making smoothing parameter smaller/greater. We end up with some reasonable values that seem to give a good result.

The following plots show the three kernels and their smoothing parameters. They show, which values around the point to predict (always in the center) are taken in to account based on the percentage shown on the y-axis.

We end up with a distance smoothing parameter around `150000` which is basically a radius of 150 kilometers. For the hour/time we have a smoothing parameter of `3`, which is around 3 hours. For the date we have a smoothing parameter of `30` which is close to one month.


Gaussian Kernel Cruve With Selected Smoothing Factor
1.0
0.8
0.6
Kernel Output
49.0
49.5
50.0
50.5
51.0
Distance
Legend
Distance (smoothing = 150000)


Gaussian Kernel Cruve With Selected Smoothing Factor
1.00
0.75
0.50
0.25
0.00
Kernel Output
15700
15800
15900
16000
Day
Legend
Day (smoothing = 30)

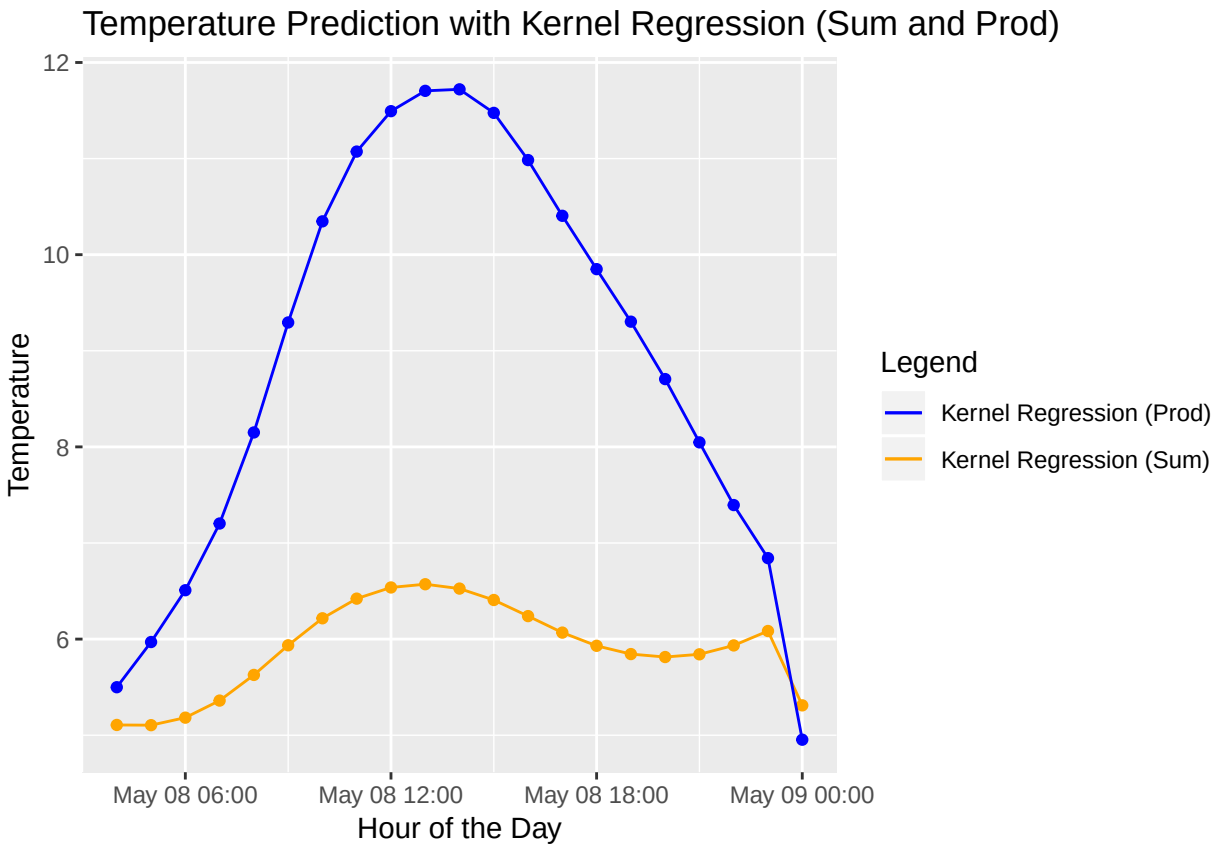


1.2 Weather Prediction

For the weather prediction we use Linkoeping (`latitude = 58.410809` and `longitude = 15.621373`) and as a date we use `date = "2000-05-08"` which is basically random.

Comment: If we select an area or time where we basically have no datapoints, all kernel functions return only `0`, so their sum is zero as well. Dividing by zero is undefined and thus results in NA's. **End Comment**

The following plot shows the weather prediction, we include the predictions for the sum and the product of the three kernels. We observe that the weather prediction seems to work better with the product of the three kernels as the sum struggles in adjusting to the daily differences in temperature. We can as well see that in general the prediction is reasonable (it's colder at nights) and the temperatured for Mai seems reasonable (maybe not that much for the sum, but that's okay as it's not a good model).



2 Support Vector Machines

Task:

Use the function ksvm from the R package kernlab to learn a SVM for classifying the spam dataset that is included with the package. Consider the radial basis function kernel (also known as Gaussian) with a width of 0.05. For the C parameter, consider values 0.5, 1 and 5. This implies that you have to consider three models.

- Perform model selection, i.e. select the most promising of the three models (use any method of your choice except cross-validation or nested cross-validation).
- Estimate the generalization error of the SVM selected above (use any method of your choice except cross-validation or nested cross-validation).
- Produce the SVM that will be returned to the user, i.e. show the code.
- What is the purpose of the parameter C?

2.1 Model Selection by Generalization Error

The confusion matrices look like this (C = 0.5, C = 1, C = 5):

```
##          model_svm_05_prediction
##           nonspam spam
##   nonspam     819   27
##   spam        104  430
```

```
##          model_svm_1_prediction
##           nonspam spam
```

```
##   nonspam     816   30
##   spam         88  446
```

```
##          model_svm_5_prediction
##           nonspam spam
##   nonspam     814   32
##   spam         79  455
```

The error rates are (`C = 0.5`, `C = 1`, `C = 5`):

```
## [1] 0.09492754
```

```
## [1] 0.08550725
```

```
## [1] 0.08043478
```

Therefore we select the model where `C = 5`. It doesn't really make sense to select a model just based on the training data, that why the two steps were combined and the generalization error was considered using the validation data set. We use the test data set to actually get a good esimation of our model (holdout).

2.2 Produce the SVM

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 5
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  0.05
##
## Number of Support Vectors : 831
##
## Objective Function Value : -891.6116
## Training error : 0.018478
```

```
## [1] "Error rate: 0.0839971035481535"
```

2.3 C Parameter

C defines the cost of the constraint violation penalty. The default value is set to 1 and it is an factor added to the error function. This means, for `C = 0.5` the errors are halved and for `C = 5` they are five times as large. Keep in mind that we also have ϵ which defines the threshold, when an error is applied. C must always be greater than 0, otherwise there would never be an error. Normally C and ϵ are selected by Cross-Validation.

3 Appendix: Source Code

```
knitr::opts_chunk$set(echo = TRUE)
library(geosphere)
library(ggplot2)
library(kernlab)


################################################################################
# Kernel Methods
```

```r
################################################################################

## Helper Functions

min_distance = function(a, b, ringSize) {

  boundOne = sapply(b, FUN = function(x) abs(a - x))
  boundTwo = sapply(b, FUN = function(x) abs(a + ringSize - x))

  return(pmin(boundOne, boundTwo))
}


## Kernels
## The definition for the Guassian Kernel is taken from the slides, page 6.

kernel_gauss_distance = function(pointA, pointB, smoothing) {

  # Use distHaversine() as a help
  m = cbind(pointB$longitude, pointB$latitude)
  u = distHaversine(m, c(pointA$longitude, pointA$latitude))
  u = u / smoothing
  return(exp(-(u^2)))
}

kernel_gauss_day = function(dayA, dayB, smoothing) {

  dayA_in_days = as.numeric(strftime(as.Date(dayA$date), '%j'))
  dayB_in_days = as.numeric(strftime(as.Date(dayB$date), '%j'))

  u = min_distance(dayA_in_days, dayB_in_days, 365)
  u = u / smoothing
  return(exp(-(u^2)))
}

kernel_gauss_hour = function (hourA, hourB, smoothing) {

  hourA_in_h = sapply(hourA$time, FUN = function(x)
    as.numeric(difftime(strptime(x, format = "%H:%M:%S"),
                        strptime("00:00:00", format = "%H:%M:%S"))))

  hourB_in_h = sapply(hourB$time, FUN = function(x)
    as.numeric(difftime(strptime(x, format = "%H:%M:%S"),
                        strptime("00:00:00", format = "%H:%M:%S"))))

  u = min_distance(hourA_in_h, hourB_in_h, 24)
  u = u / smoothing
  return(exp(-(u^2)))
}


plot_values_distance = list(latitude = seq(from = 49, to = 51, by = 0.01),
                            longitude = seq(from = 49, to = 51, by = 0.01))
```

```
plot_values_day =
  list(date = c("2013-01-01", "2013-01-16", "2013-02-01", "2013-02-16",
                "2013-03-01", "2013-03-16", "2013-04-01", "2013-04-16",
                "2013-05-01", "2013-05-16", "2013-06-01", "2013-06-16",
                "2013-07-01", "2013-07-16", "2013-08-01", "2013-08-16",
                "2013-09-01", "2013-09-16", "2013-10-01", "2013-10-16",
                "2013-11-01", "2013-11-16", "2013-12-01", "2013-12-16"))

plot_values_hour =
  list(time = c("04:00:00", "05:00:00", "06:00:00", "07:00:00",
                "08:00:00", "09:00:00", "10:00:00", "11:00:00",
                "12:00:00", "13:00:00", "14:00:00", "15:00:00",
                "16:00:00", "17:00:00", "18:00:00", "19:00:00",
                "20:00:00", "21:00:00", "22:00:00", "23:00:00",
                "24:00:00"))

compare_value_distance = list(latitude = 50, longitude = 50)
compare_value_day = list(date = "2018-07-01")
compare_value_hour = list(time = "12:30:00")

# Smoothing factors
h_distance = 150000 #150000
h_date = 30 #30
h_time = 3 #3

distances =kernel_gauss_distance(compare_value_distance,
                                 plot_values_distance, h_distance)
days = kernel_gauss_day(compare_value_day, plot_values_day, h_date)
hours = kernel_gauss_hour(compare_value_hour, plot_values_hour, h_time)

# Need to be continuous for plotting
plot_values_day = as.numeric(as.Date(plot_values_day$date))
plot_values_hour =
  as.numeric(strptime(plot_values_hour$time, format = "%H:%M:%S"))

# Dataframes for plotting
plot_data_frame_distance = data.frame(plot_values_distance$latitude, distances)
plot_data_frame_day = data.frame(plot_values_day, days)
plot_data_frame_hour = data.frame(plot_values_hour, hours)


ggplot(plot_data_frame_distance) +
  geom_line(aes(x = plot_values_distance$latitude, y = distances,
                colour = "Distance (smoothing = 150000)")) +
  #geom_point(aes(x = plot_values_distance$latitude, y = distances),
  #           colour = "orange") +
  labs(title = "Gaussian Kernel Cruve With Selected Smoothing Factor",
       y = "Kernel Output",
       x = "Distance", color = "Legend") +
  scale_color_manual(values = c("orange"))

ggplot(plot_data_frame_day) +
  geom_line(aes(x = plot_values_day,
```

```
                y = days, colour = "Day (smoothing = 30)")) +
  #geom_point(aes(x = plot_values_day, y = days), colour = "blue") #+
  labs(title = "Gaussian Kernel Cruve With Selected Smoothing Factor",
       y = "Kernel Output",
       x = "Day", color = "Legend") +
  scale_color_manual(values = c("blue"))

ggplot(plot_data_frame_hour) +
  geom_line(aes(x = plot_values_hour, y = hours,
                colour = "Hour (smoothing = 3)")) +
  #geom_point(aes(x = plot_values_hour, y = hours), colour = "green")
  #
  labs(title = "Gaussian Kernel Cruves With Selected Smoothing Factor",
       y = "Kernel Output",
       x = "Hour", color = "Legend") +
  scale_color_manual(values = c("violet"))


kernel_sum = function(A, B, h_distance, h_date, h_time) {
  return(
    kernel_gauss_distance(A, B, h_distance) +
    kernel_gauss_day(A, B, h_date) +
    kernel_gauss_hour(A, B, h_time)
  )
}

kernel_product = function(A, B, h_distance, h_date, h_time) {
  return(
    kernel_gauss_distance(A, B, h_distance) *
    kernel_gauss_day(A, B, h_date) *
    kernel_gauss_hour(A, B, h_time)
  )
}

predict_weather = function(u_latitude, u_longitude, u_date, u_type) {

  # Data
  stations = read.csv("stations.csv", encoding = "UTF-8")
  temps = read.csv("temps50k.csv", encoding = "UTF-8")
  st = merge(stations, temps, by="station_number")

  # Filter all date posterior to the given date (doesn't make sense to predict
  # something that we already now)
  st = st[as.Date(st$date) < as.Date(u_date),]

  # Given Data
  # Each user input should be a list of:
  # latitude, longitude, date, time
  # time is created with the loop at we predict for every time for the given day

  # Times to predict
  times = c("04:00:00", "05:00:00", "06:00:00", "07:00:00",
            "08:00:00", "09:00:00", "10:00:00", "11:00:00",
```

```
            "12:00:00", "13:00:00", "14:00:00", "15:00:00",
            "16:00:00", "17:00:00", "18:00:00", "19:00:00",
            "20:00:00", "21:00:00", "22:00:00", "23:00:00",
            "24:00:00")

  # Temperatures to predict
  temp = vector(length=length(times))

  # Prediction for each temp to predict
  for (i in 1:length(times)) {

    # User data point for this iteration
    user_data_point = list(
      latitude = u_latitude,
      longitude = u_longitude,
      date = u_date,
      time = times[i])

    if (u_type == "sum") {
      # Parameters: A, B, h_distance, h_date, h_time
      kernel = kernel_sum(user_data_point, st, h_distance, h_date, h_time)
    }
    else if (u_type == "prod") {
      # Parameters: A, B, h_distance, h_date, h_time
      kernel =
        kernel_product(user_data_point, st, h_distance, h_date, h_time)
    }
    else {
      stop("Du Nasenbaer!")
    }

    # Now that we have the kernel value, we can calcuate the actual prediction
    # Formula taken from slide 8 from
    # "Histogram, Moving Window, and Kernel Regression"
    y = kernel %*% st$air_temperature/sum(kernel)

    # Let's save this predicted temperature
    temp[i] = y
  }
  return(data.frame(times, temp))
}

set.seed(1234567890)

# Linkoeping
latitude = 58.4166
longitude = 15.6333
date = "2000-05-08"
#date = "2018-12-14"


# Students' code here
pred_temp_sum = predict_weather(latitude, longitude, date, "sum")
```

```r
pred_temp_prod = predict_weather(latitude, longitude, date, "prod")

plot_data_frame =
  data.frame(as.POSIXct(paste(date ,pred_temp_sum$times),
                        format = "%Y-%m-%d %H:%M"),
             pred_temp_sum$temp, pred_temp_prod$temp)
colnames(plot_data_frame) = c("hour", "predWithSum", "predWithProd")

ggplot(plot_data_frame) +
  geom_line(aes(x = hour, y = predWithSum, group = 1,
                colour = "Kernel Regression (Sum)")) +
  geom_point(aes(x = hour, y = predWithSum), colour = "orange") +
  geom_line(aes(x = hour, y = predWithProd, group = 1,
                colour = "Kernel Regression (Prod)")) +
  geom_point(aes(x = hour, y = predWithProd), colour = "blue") +
  labs(title = "Temperature Prediction with Kernel Regression (Sum and Prod)",
       y = "Temperature", x = "Hour of the Day", color = "Legend") +
  scale_color_manual(values = c("blue", "orange"))


################################################################################
# Support Vector Machines
################################################################################

data(spam)
n = dim(spam)[1]
set.seed(12345)
id = sample(1:n, floor(n*0.4))
train_spam = spam[id,]
id1 = setdiff(1:n, id)
set.seed(12345)
id2 = sample(id1, floor(n*0.3))
val_spam = spam[id2,]
id3 = setdiff(id1,id2)
test_spam = spam[id3,]

# Our three models depending on
model_svm_05 =
  ksvm(type ~ ., train_spam, kernel = "rbfdot", kpar = list(sigma = 0.05), C = 0.5)
model_svm_1 =
  ksvm(type ~ ., train_spam, kernel = "rbfdot", kpar = list(sigma = 0.05), C = 1)
model_svm_5 =
  ksvm(type ~ ., train_spam, kernel = "rbfdot", kpar = list(sigma = 0.05), C = 5)

model_svm_05_prediction = predict(model_svm_05, newdata = val_spam)
model_svm_1_prediction = predict(model_svm_1, newdata = val_spam)
model_svm_5_prediction = predict(model_svm_5, newdata = val_spam)

cm_svm_05 = table(val_spam$type, model_svm_05_prediction)
cm_svm_1 = table(val_spam$type, model_svm_1_prediction)
cm_svm_5 = table(val_spam$type, model_svm_5_prediction)

error_svm_05 = 1 - sum(diag(cm_svm_05)/sum(cm_svm_05))
```

```r
error_svm_1 = 1 - sum(diag(cm_svm_1)/sum(cm_svm_1))
error_svm_5 = 1 - sum(diag(cm_svm_5)/sum(cm_svm_5))


print(cm_svm_05)
print(cm_svm_1)
print(cm_svm_5)


print(error_svm_05)
print(error_svm_1)
print(error_svm_5)


selected_model_prediction = predict(model_svm_5, newdata = test_spam)

cm_selected_model = table(test_spam$type, selected_model_prediction)

error_selected_model = 1 - sum(diag(cm_selected_model)/sum(cm_selected_model))

print(model_svm_5)
print(paste("Error rate:", error_selected_model))
```