# Lab 3 Block 1: Kernel Meethods and Neural Networks

*Annalena Erhard (anner218), Héctor Plata (hecpl268), Maximilian Pfundstein (maxpf364)*

*2018-12-17*

## 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.

In this case we want to estimate the temperature $y$ given a data point $x$ with a kernel regression which has the following form:

$$\hat{y}\left(X, x^{'}\right) = \frac{k\left(X, x^{'}\right) y}{\sum_{n} k\left(X, x^{'}\right)}$$

In this particular case we are testing two kernels $k\left(X, x^{'}\right)$, one which is the sum of three gaussian kernels and the other one which is the product of three gaussian kernels. The gaussian kernel is defined below:

$$k\left(X, x^{'}\right) = \exp\left(-\frac{\left(x_i - x^{'}\right)^2}{h}\right)$$

Where, $x_i$ is a previous observation, $x^{'}$ is the observation we want to predict and $h$ is a smoothing parameter that is going to help us determine how relevant distant observations are. The kernel could also be rewritten as follows:

$$k\left(X, x^{'}\right) = \frac{1}{e^{\left(\frac{d^2}{h}\right)}}$$

Where $d$ is the distance given by $x_i - x^{'}$. It's easy to see that the bigger the distance $d$ with respect to $h$ the lower the value of the kernel.

### 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 is `24` and `356` (we ignore all the special cases we have 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 = parSapply(cl, b, FUN = function(x) abs(a - x))
  boundTwo = parSapply(cl, b, FUN = function(x) abs(a - ringSize - x))
  boundThree = parSapply(cl, b, FUN = function(x) abs(a + ringSize - x))

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

### 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` calculates the difference between two days. It ignores the year (as we use the cyclical behaviour 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 interested in the cyclical behaviour) and measures the distance using the `min_distance(a, b, ringSize)` function which results in values between `(0, 12)`.

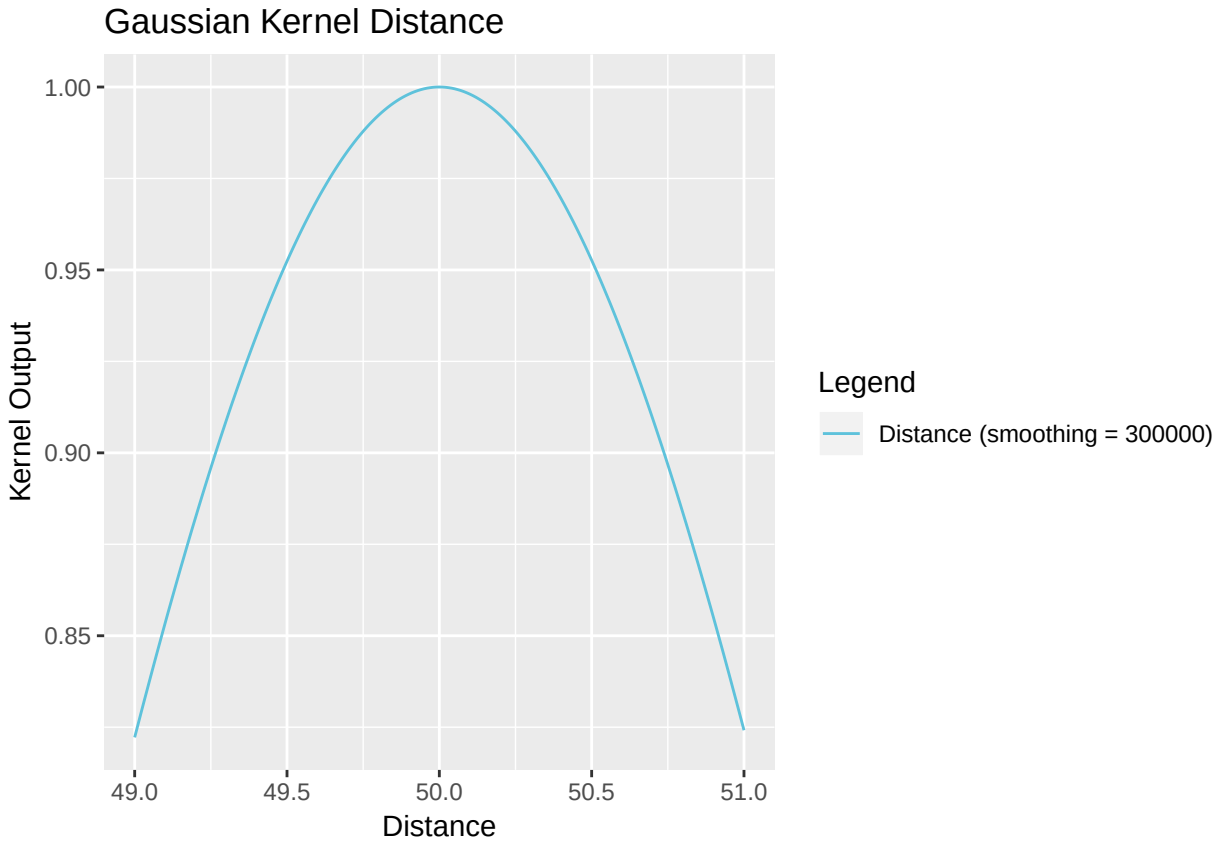
```
## Kernels
## The definition for the Guassian Kernel is taken from the slides, page 6.

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 = parSapply(cl, 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 = parSapply(cl, 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)))
}
```

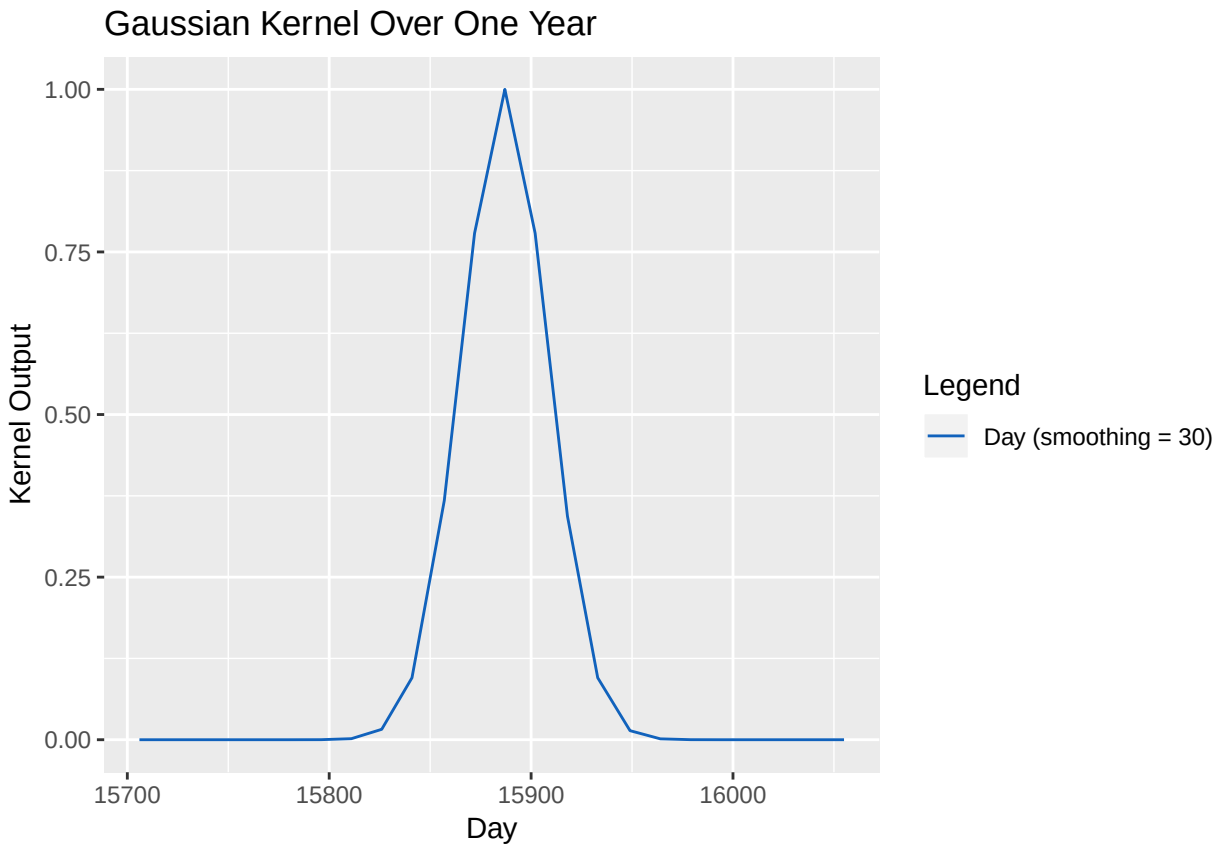
### 1.1.3 Smoothing Parameters

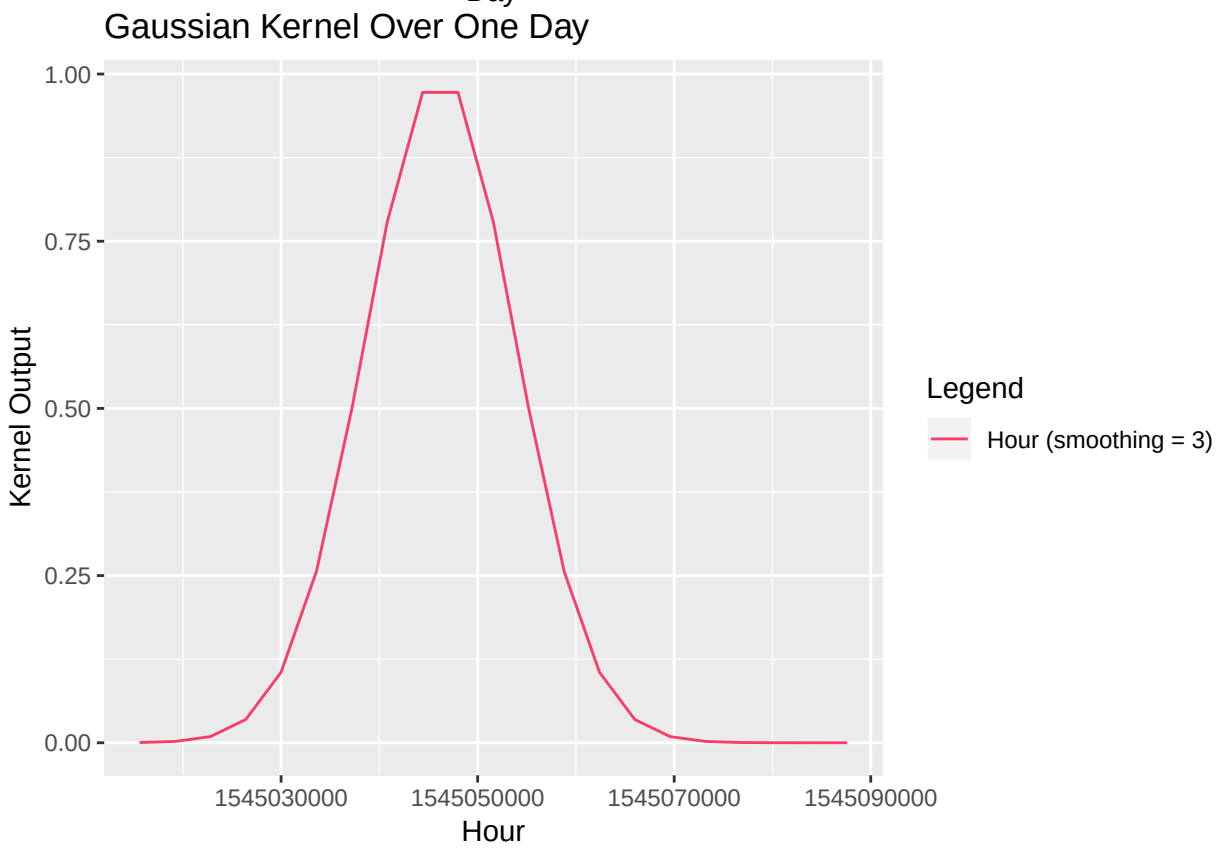
To receive reasonable values for the smoothing parameters we first define some range in which we actually want to consider features. While this is easy for the date and time (one year and one day) it's not that obvious for the distance. We choose 2 for latitude and longitude. Then we adjusted the smoothing 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.



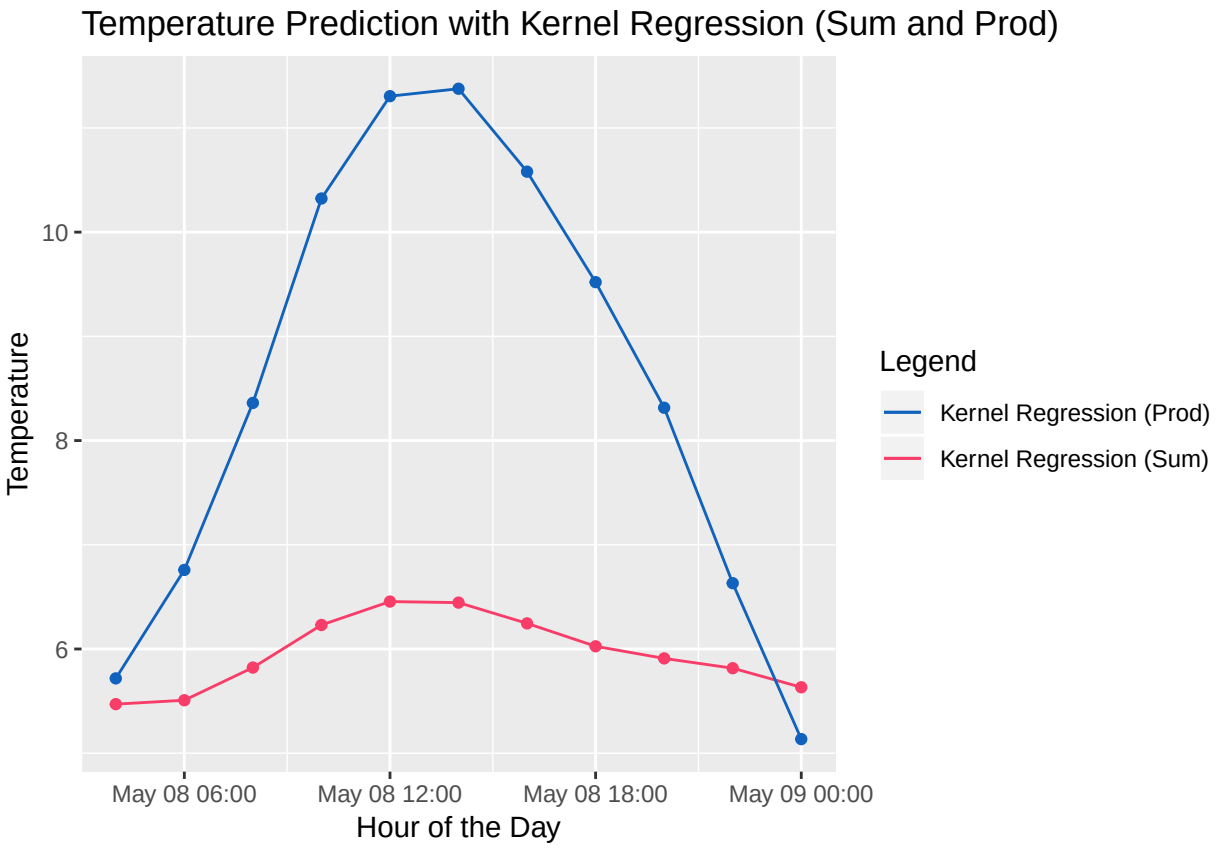

Gaussian Kernel Over One Year
1.00
0.75
0.50
0.25
0.00
Kernel Output
15700
15800
15900
16000
Day
Legend
Day (smoothing = 30)


Gaussian Kernel Over One Day
1.00
0.75
0.50
0.25
0.00
Kernel Output
1545030000
1545050000
1545070000
1545090000
Hour
Legend
Hour (smoothing = 3)

### 1.2 Weather Prediction

For the weather prediction we use Linkoeping (`latitude = 58.4166` and `longitude = 15.6333`) and as a date we use `date = "2000-05-08"` which is basically random.

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 temperature seems reasonable as well (maybe not that much for the sum, but that's okay as it's not a good model). The only "akward" thing to obersve is the "bumby" temperate drop at the end of the day.




## 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?

**Answer:** For this problem the data is split between a train, validation and test set (70/20/10). The train set is used to train all of the three models and each model is being evaluated on the validation set. Once we select the most promising model (from the results on the validation set). The model is re-trained with the train and validation data and then the generalization error (Recall, Precision, Missclasification, Accuracy and FPR) is calculated over the test set.

Below on the tables 1, 2 and 3 we can see the performance metrics of each of the three models. All of the models have a relative similar performance, so in this case, the best model is selected by the lowest false positive rate (FPR), since it would be ideal to minimize the amount of spam mails (negative class) being selected as a normal email, since spam mails could be hazardous for the user. With this in mind, the model where $c = 0.5$ is selected.

Table 1: Train and Validation metrics for c = 0.5

| | Train | Validation |
|---|---|---|
| Recall | 0.9147593 | 0.8591954 |
| Precision | 0.9594371 | 0.9373041 |
| Missclasification | 0.0487578 | 0.0750000 |
| Accuracy | 0.9512422 | 0.9250000 |
| FPR | 0.0250896 | 0.0349650 |

Table 2: Train and Validation metrics for c = 1

| | Train | Validation |
|---|---|---|
| Recall | 0.9313339 | 0.8821839 |
| Precision | 0.9656301 | 0.9246988 |
| Missclasification | 0.0400621 | 0.0717391 |
| Accuracy | 0.9599379 | 0.9282609 |
| FPR | 0.0215054 | 0.0437063 |

Table 3: Train and Validation metrics for c = 5

| | Train | Validation |
|---|---|---|
| Recall | 0.9613260 | 0.8764368 |
| Precision | 0.9854369 | 0.9214502 |
| Missclasification | 0.0208075 | 0.0750000 |
| Accuracy | 0.9791925 | 0.9250000 |
| FPR | 0.0092166 | 0.0454545 |

We can see the generalization error (test) on different metrics of the best model ($c = 0.5$) on the table 4 below.

Table 4: Train and test metrics of the best model

| | Train | Test |
|---|---|---|
| Recall | 0.9232198 | 0.8434343 |
| Precision | 0.9576108 | 0.9435028 |
| Missclasification | 0.0458937 | 0.0889371 |
| Accuracy | 0.9541063 | 0.9110629 |
| FPR | 0.0261386 | 0.0380228 |

Now that we have an estimation of the generalization error, the model that is going to be returned to the user is the one who is going to be trained on the whole dataset since the more data we feed to our algorithm the better, since it will give a better performance on unseen data and will help us avoid overfitting.

```
# Training the classifier.
clf = ksvm(type ~ .,
           df,
           kernel='rbfdot',
           kpar=list(sigma=0.05),
           C=0.5)

print(clf)
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 0.5
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  0.05
##
## Number of Support Vectors : 1785
##
## Objective Function Value : -528.9119
## Training error : 0.046294
```

Finally to conclude with the assignment, the purpose of the parameter $\mathcal{C}$ is to penalize for the violation of the constrain on the Lagrange formulation. This means that the bigger $\mathcal{C}$ is, the more strict the boundary must be since it will penalize more the variables that are missclasified by the margin boundary. This can be seen on the optimization problem that defines a SVM:

$$\underset{w,b}{\operatorname{argmin}} \frac{1}{2}\|w\|^2 + \mathcal{C}\sum_n \xi_n$$

Where the slack variable $\xi_n$ is define for each data point $n$ that it's outside of the margin and it's defined as $\xi_n = |t_n - y(x_n)|$.

# 3 Appendix: Source Code

```
knitr::opts_chunk$set(echo = FALSE, cache = TRUE)
library(geosphere)
library(ggplot2)
library(kernlab)
library(knitr)
library(parallel)
cl = makeCluster(detectCores())


################################################################################
# Kernel Methods
################################################################################

## Helper Functions
```

```r
min_distance = function(a, b, ringSize) {

  boundOne = parSapply(cl, b, FUN = function(x) abs(a - x))
  boundTwo = parSapply(cl, b, FUN = function(x) abs(a - ringSize - x))
  boundThree = parSapply(cl, b, FUN = function(x) abs(a + ringSize - x))

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


## 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 = parSapply(cl, 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 = parSapply(cl, 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 = 300000
h_date = 30
h_time = 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 = 300000)")) +
  labs(title = "Gaussian Kernel Distance",
       y = "Kernel Output",
       x = "Distance", color = "Legend") +
  scale_color_manual(values = c("#5ec2db"))

ggplot(plot_data_frame_day) +
  geom_line(aes(x = plot_values_day,
                y = days, colour = "Day (smoothing = 30)")) +
  labs(title = "Gaussian Kernel Over One Year",
       y = "Kernel Output",
       x = "Day", color = "Legend") +
```

```
  scale_color_manual(values = c("#1162bc"))

ggplot(plot_data_frame_hour) +
  geom_line(aes(x = plot_values_hour, y = hours,
                colour = "Hour (smoothing = 3)")) +
  labs(title = "Gaussian Kernel Over One Day",
       y = "Kernel Output",
       x = "Hour", color = "Legend") +
  scale_color_manual(values = c("#f83d69"))


predict_weather = function(u_latitude, u_longitude, u_date) {

  # 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 = c("04:00:00", "06:00:00", "08:00:00",  "10:00:00",
            "12:00:00",  "14:00:00","16:00:00",  "18:00:00",
            "20:00:00",  "22:00:00", "24:00:00")

  # Temperatures to predict
  temp = data.frame()

  # Distance Kernel
  kernel_distance = kernel_gauss_distance(list(
      latitude = u_latitude, longitude = u_longitude), st, h_distance)

  # Day Kernel
  kernel_day = kernel_gauss_day(list(date = u_date), st, h_date)

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

    # Hour Kernel
    kernel_hour = kernel_gauss_hour(list(time = times[i]), st, h_time)

    kernel_sum = kernel_distance + kernel_day + kernel_hour
    kernel_prod = kernel_distance * kernel_day * kernel_hour

    # 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_sum = kernel_sum %*% st$air_temperature/sum(kernel_sum)
    y_prod = kernel_prod %*% st$air_temperature/sum(kernel_prod)

    # Let's save this predicted temperature
    temp = rbind(temp, list(kernel_sum = y_sum, kernel_prod = y_prod))
  }
  return(cbind(times, temp))
}

set.seed(1234567890)

latitude = 58.4166
longitude = 15.6333
date = "2000-05-08"


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

plot_data_frame =
  data.frame(as.POSIXct(paste(date ,pred_temp_sum$times),
                        format = "%Y-%m-%d %H:%M"), pred_temp_sum[2:3])
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 = "#f83d69") +
  geom_line(aes(x = hour, y = predWithProd, group = 1,
                colour = "Kernel Regression (Prod)")) +
  geom_point(aes(x = hour, y = predWithProd), colour = "#1162bc") +
  labs(title = "Temperature Prediction with Kernel Regression (Sum and Prod)",
       y = "Temperature", x = "Hour of the Day", color = "Legend") +
  scale_colour_manual(values=c("#1162bc", "#f83d69"))

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

# Helper function to calculate the
# classification metrics given a
# confusion matrix.
# Rows (0, 1) must be from the classifier
# Columns (0, 1) must be true values.
analyze_cm = function(cm)
    {
        cm_df = as.data.frame(cm)
        recall = cm_df[4, "Freq"] / (cm_df[4, "Freq"] + cm_df[2, "Freq"])
        precision = cm_df[4, "Freq"] / (cm_df[4, "Freq"] + cm_df[3, "Freq"])
        accuracy = (cm_df[1, "Freq"] + cm_df[4, "Freq"]) / sum(cm_df[, "Freq"])
        mcr = 1 - accuracy
        fpr = cm_df[3, "Freq"] / (cm_df[1, "Freq"] + cm_df[3, "Freq"])
```

```
        return(list(Recall=recall,
                    Precision=precision,
                    Missclasification=mcr,
                    Accuracy=accuracy,
                    FPR=fpr))
}

# Helper function to get metrics from a confusion matrix.
get_metrics_cm = function(cm_train, cm_test)
{
  metrics = data.frame(analyze_cm(cm_train))
  metrics = rbind(metrics, analyze_cm(cm_test))
  rownames(metrics) = c("Train", "Validation")

  return(t(metrics))
}

# Transforms a confusion matrix
# to a data frame. It assumes
# a 2 by 2 binary confusion matrix.
prettyfy_cm = function(cm_table)
{
  pretty_table = data.frame(list(titles=c("Non-Spam", "Spam"),
                                 Bad=c(cm_table[1, 1], cm_table[2, 1]),
                                 Good=c(cm_table[1, 2], cm_table[2, 2])))

  colnames(pretty_table) = c("True / Predicted", "Non-Spam", "Spam")

  return(pretty_table)
}

# Loading the data.
data(spam)
df = spam

# Train / Validation / Test split.
# 70 / 20 / 10.

# Train
n = dim(df)[1]
set.seed(12345)
id = sample(1:n, floor(n * 0.7))
train = df[id,]

# Validation.
id1 = setdiff(1:n, id)
set.seed(12345)
id2 = sample(id1, floor(n * 0.20))
val = df[id2,]

# Test.
id3 = setdiff(id1,id2)
test = df[id3,]
```

```
# Grid search.
c_grid = c(0.5, 1, 5)

# Variable that is going to
# hold the results.
results = NULL

# Doing hyperparameter optimization
# using a grid search.
for (i in 1:3)
{
  # Getting the hyperparameter to test.
  c = c_grid[i]

  # Training the classifier.
  clf = ksvm(type ~ .,
             train,
             kernel='rbfdot',
             kpar=list(sigma=0.05),
             C=c)

  # Predicting.
  yhat_train = predict(clf, newdata=train)
  yhat_val= predict(clf, newdata=val)

  # Creating the confusion matrixes.
  cm_train = table(train$type, yhat_train)
  cm_val = table(val$type, yhat_val)

  # Getting some metrics out of them.
  metrics = get_metrics_cm(cm_train, cm_val)

  # Appending them to the results.
  results = rbind(results, metrics)
}

kable(results[1:5, ],
      caption='Train and Validation metrics for c = 0.5')
kable(results[6:10, ],
      caption='Train and Validation metrics for c = 1')
kable(results[11:15, ],
      caption='Train and Validation metrics for c = 5')


# Helper function to get metrics from a confusion matrix.
get_metrics_cm = function(cm_train, cm_test)
{
  metrics = data.frame(analyze_cm(cm_train))
  metrics = rbind(metrics, analyze_cm(cm_test))
  rownames(metrics) = c("Train", "Test")

  return(t(metrics))
}
```

```r
# Mergin the train and validation data.
new_train = rbind(train, val)

# Training the classifier.
clf = ksvm(type ~ .,
           new_train,
           kernel='rbfdot',
           kpar=list(sigma=0.05),
           C=0.5)

# Predicting.
yhat_train = predict(clf, newdata=new_train)
yhat_test= predict(clf, newdata=test)

# Creating the confusion matrixes.
cm_train = table(new_train$type, yhat_train)
cm_test = table(test$type, yhat_test)

# Getting some metrics out of them.
metrics = get_metrics_cm(cm_train, cm_test)
kable(metrics, caption='Train and test metrics of the best model')

# Training the classifier.
clf = ksvm(type ~ .,
           df,
           kernel='rbfdot',
           kpar=list(sigma=0.05),
           C=0.5)

print(clf)
```