Computer lab 3

Andreas Arrestam
Emma Bertmar
Manfred Clase

LINKÖPINGS UNIVERSITET

Contents

Statement of Contribution

During Lab 3, each team member was responsible for one assignment, which included both the code and the corresponding part of the report. Manfred Clase was responsible for Assignment 2, Emma Bertmar for Assignment 3, and Andreas Arrestam for Assignment 4. We sat together to answer the first assignment. However, all team members discussed the tasks together and supported each other in completing the code and writing the report sections. Once all parts were finished, the members sat together to review the problems and solutions. This ensured that every group member understood the code, the results, and the conclusions.

1. Assignment 1. Theory

This chapter answers some questions using the course book *Machine Learning A First Course for Engineers and Scientists* (2022), written by Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten and Thomas B. Schön.

1.1 What is the kernel trick?

The kernel trick is the idea that, if a learning algorithm depends on the input data only through inner products of feature mappings $\phi(x)^\top \phi(x')$, then these inner products can be replaced by a kernel function $\kappa(x, x')$ computed directly in the original input space. This avoids explicitly constructing or working in a potentially high- or infinite-dimensional feature space, while still allowing the model to learn nonlinear relationships.

1.2 In the literature, it is common to see a formulation of SVMs that makes use of a hyperparameter C. What is the purpose of this hyperparameter?

The hyperparameter C is used as a regularization parameter in Support Vector Machines. It can be expressed as $C = \frac{1}{2\lambda}$ or $C = \frac{1}{2n\lambda}$, where λ represents the regularization parameter. The value of C therefore determines the strength of the regularization, with larger values of C corresponding to weaker regularization and smaller values of C corresponding to stronger regularization (pages 211-212).

1.3 In neural networks, what do we mean by mini-batch and epoch?

According to the book, for large datasets in neural networks when computing the gradient descent it can become very computationally heavy. Using stochastic gradient descent instead, it's possible to compute the gradient for the new parameters based on the second half of the training data. In other words only a subsample of the training data is used to compute the gradient. This small subsample of data is called a *mini-batch.* This typically contains 10, 100 or 1000 data points. One complete pass trough the training data is called an *epoch* (pages 124-125).

2. Assignment 2. Kernel Methods

The purpose of this assignment was to explore kernel methods by implementing a kernel method for predicting the outdoor temperature on a given date and place from the hours 04:00AM to 12:00PM by two hour intervals. For this assignment the datasets **stations.csv** and **temp50k.csv** was provided together containing a number of SMHI weather observation stations along with measurements at different points of time. To calculate the geospacial distance the R-library **geosphere** was used.

2.1 Data pre-processing

The datasets **stations.csv** and **temps50k.csv** were first read and merged using the station identifier **station_number**, resulting in a combined dataset containing both geographical information and temperature measurements.

Date and time variables were preprocessed to allow temporal distance calculations. The date variable was converted to Date format, while the time variable was truncated to hours and minutes. An additional numeric hour variable was extracted to represent the hour of the day.

Geographical coordinates (longitude and latitude) of the stations were selected and used to compute spatial distances between each station and the prediction location. These distances were calculated using the Haversine formula via the **distHaversine** function from the **geosphere** package.

Listing 2.1: Merging datasets

```
stations <- read.csv("stations.csv", fileEncoding = "latin1")
temperatures <- read.csv("temps50k.csv")
st <- merge(stations, temperatures, by="station_number")


#date and time pre-processing
st$date <- as.Date(st$date)
st$time <- substr(st$time, 1, 5)
st$hour <- as.numeric(substr(st$time, 1, 2))
```

2.2 Kernel Methods

In the assignment it was specified that three Gaussian kernels should be created: one spacial distance kernel, the second kernel to measure the distance in date and finally one kernel to measure the distance in hours. The gaussian kernel methods was abstracted to the `gaussian_kernel` and was implemented as follows:

$$K(u) = \exp(-u^2)$$

2.3 Smoothing Coefficient

Three smoothing coefficients (bandwidths) were manually chosen to control the influence of spatial, temporal, and hourly distances in the Gaussian kernels. The spatial bandwidth was set to $h_{\text{space}} = 80{,}000$ meters, assigning high weight to stations within approximately 80 km of the prediction location. The date bandwidth was set to $h_{\text{date}} = 20$ days, allowing observations from nearby days to contribute substantially to the prediction. Finally, the time-of-day bandwidth was set to $h_{\text{time}} = 2$ hours, emphasizing measurements close to the target hour.

These values were selected to ensure that nearby observations receive large kernel weights while distant observations are strongly down-weighted, without relying on cross-validation.

Listing 2.2: Assigning Smoothing Coefficients

```
h_space <- 80000
h_date <- 20
h_time <- 2
```

2.4 Making the predictions

The temperature forecasts were generated using two kernel-based predictors. The first predictor computes weights as the sum of the spatial, date, and time Gaussian kernel values, while the second predictor computes weights as the product of these kernels. In both cases, the predicted temperature is obtained as a weighted average of past temperature observations, where observations closer in space and time receive higher weights.

Listing 2.3: Assigning Smoothing Coefficients

```
times <- as.numeric(substr(times, 1, 2))

temp_sum <- vector(length = length(times))     # sum of kernels
temp_prod <- vector(length = length(times))    # product of kernels


#scaled_distance <- function(data, prediction, smoothing) (data-prediction)/h
gaussian_kernel <- function(u) exp(-u^2)


for(i in seq_along(times)){
  pred_time <- times[i]

  st_filtered <- st %>%
    filter(date < date | (date == date & hour <= pred_time))

  #calculate the distances in each domain
  d_space <- distHaversine(cbind(st_filtered$longitude, st_filtered$latitude),
       c(b,a))
  d_date <- abs(as.numeric(st_filtered$date - date))
  d_time <- abs(st_filtered$hour - pred_time)
```

```r
  #scale the distances
  u_space <- d_space/h_space
  u_date <- d_date/h_date
  u_time <- d_time/h_time

  #calculate the kernel values
  k_space <- gaussian_kernel(u_space)
  k_date <- gaussian_kernel(u_date)
  k_time <- gaussian_kernel(u_time)

  w_sum <- k_space + k_date + k_time
  w_prod <- k_space * k_date * k_time #product of kernels

  temp_sum[i] <- sum(w_sum * st_filtered$air_temperature) / sum(w_sum)
  temp_prod[i] <- sum(w_prod * st_filtered$air_temperature) / sum(w_prod)
}
```

2.5 Results

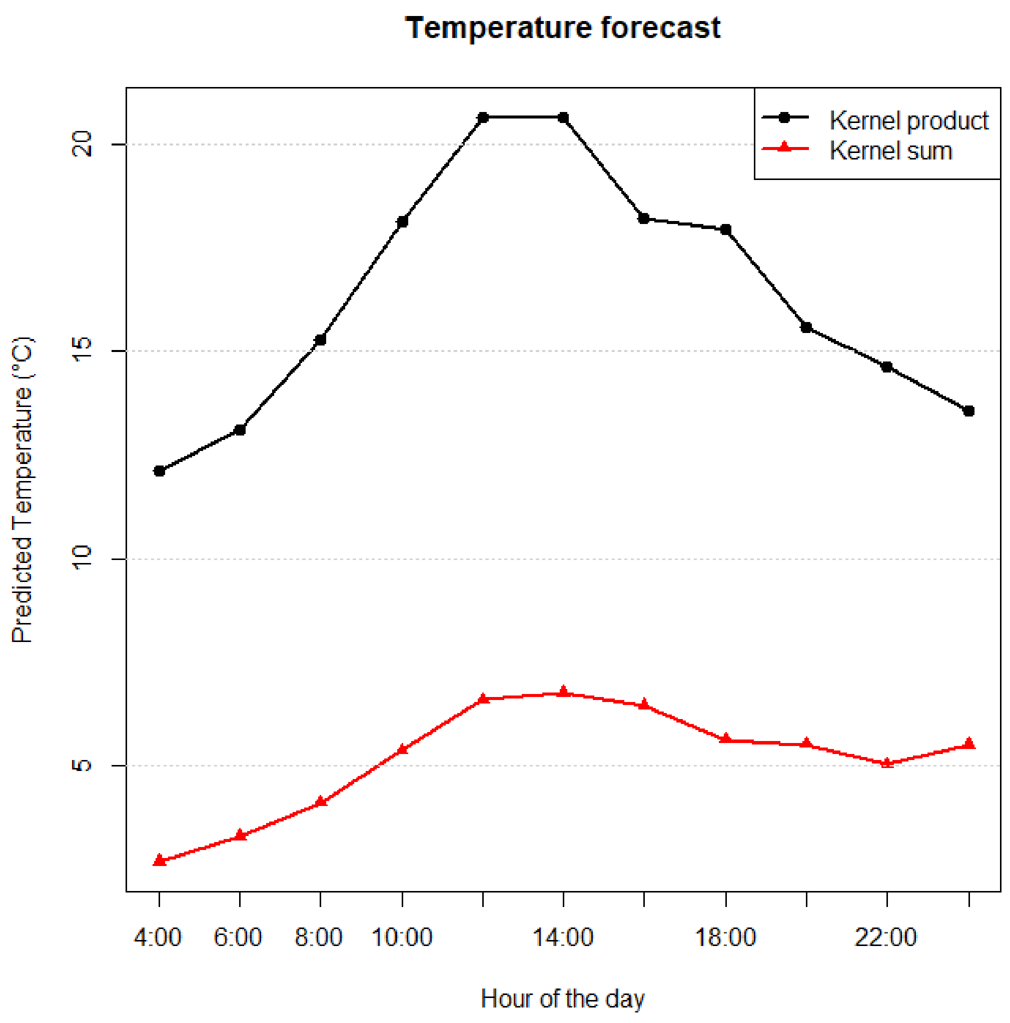

Figure 2.1: Temperature forecast for August 16, 2013, at latitude 58.0340° and longitude 14.9756° using sum and product Gaussian kernels.

Figure 2.1 shows the predicted hourly temperatures for one of several tested prediction locations in Sweden, located at latitude 58.0340° and longitude 14.9756°, on August 16, 2013. Testing was performed at a number of different spatial points to assess the behavior of the kernel methods, and this location is presented as a representative example. The sum-kernel predictor produces smoother temperature curves, as observations receive high weight if they are close in at least one of the three dimensions (space, date, or time). In contrast, the product-kernel predictor yields sharper peaks and troughs, since observations contribute significantly only when they are close in all three dimensions simultaneously. Consequently, the product kernel emphasizes highly local information, while the sum kernel provides stronger overall smoothing.

Limitations of the sum kernel. A limitation of the sum-kernel approach is that it may assign non-negligible weight to observations that are highly irrelevant in one or more dimensions. In particular, temperature measurements that are close in space and hour but far apart in date (e.g., from a different season) can still influence the prediction. This can lead to excessive smoothing and reduced seasonal specificity compared to the product-kernel approach, which requires proximity in all dimensions simultaneously.

3. Assignment 3. Support Vector Machines

The purpose of this assignment was to explore Support Vector Machines by performing model selection to classify emails as spam or non-spam using radial basis function kernels. The data used was the spam dataset included in the kernlab package.

3.1 Filters

After splitting the dataset into training, validation and test sets, different values of the parameter C were tested. A support vector machine was trained for each value, and the error rate was measured to find the best C parameter value. The best value was found to be $C = 1.8$ since it produced the lowest error rate.

Then, four different filters were fitted using $C = 1.8$. Each filter was trained and tested on different combinations of the data:

- **Filter 0:** Trained on the training set and tested on the validation set.
- **Filter 1:** Trained on the training set and tested on the test set.
- **Filter 2:** Trained on the training and validation sets and tested on the test set.
- **Filter 3:** Trained on the entire dataset and tested on the test set.

The filter that should be returned to the user is Filter 2. This filter is trained on all allowed data (training + validation sets) and tested only on unseen data (test set). This makes it the most reliable model because it uses as much training data as possible while still providing an unbiased estimate of generalization performance.

3.2 Generalization Error

For each filter, the error was calculated as follows:

- **Filter 0:** $err_0 = 0.165$
- **Filter 1:** $err_1 = 0.1672909$
- **Filter 2:** $err_2 = 0.1498127$
- **Filter 3:** $err_3 = 0.0137328$

The estimated generalization error of the filter returned to the user is given by **Filter 2**, which has an error of $err_2 = 0.1498127$. This filter is the best estimate of the true generalization error because it is evaluated on unseen data.

Filter 0 cannot be used as a generalization error estimate because its test data was already involved in choosing the optimal value of C. **Filter 1** does not use all available data for training and therefore, it does not achieve the best possible performance. **Filter 3** shows a very low error, but it cannot be used as a generalization estimate because the model was trained on the entire dataset and therefore cannot provide a valid estimate of the test error.

3.3 Manual classifications of New Data Points

The task was to manually implement the linear combination of kernel values for **filter 3**. A manual classification was performed to verify the predictions made by the support vector machine. After computing the SVM decision values manually, the results were compared with the output from the built-in `predict()` function in R.

After fitting a support vector machine, a new data point was classified using the following function:

$$f(x_{\text{new}}) = \sum_{i \in \text{SV}} \alpha_i K(x_i, x_{\text{new}}) + b,$$

where α_i are the support vector coefficients, and b is the intercept. The term $K(x_i, x_{\text{new}})$ denotes the kernel value between the support vector x_i and the new data point x_{new}, calculated using the radial basis function kernel.

The coefficients, support vector indices, and intercept are extracted from the model **filter 3**. To compute predictions for the first 10 points in the dataset, each new data point is compared to all support vectors. The kernel value between the new point and each support vector is multiplied by the corresponding coefficient α_i, and the results are summed. Finally, the intercept b is added to this sum to obtain $f(x_{\text{new}})$. The results are shown in Table 3.1.

Table 3.1: Comparison of manually computed SVM decision values and the output from `predict()` for the first 10 points in the `spam` dataset.

Index	**Manual Result**	**predict() Result**
1	-1.0702965	-1.0702965
2	1.0003450	1.0003450
3	0.9995908	0.9995908
4	-0.9999648	-0.9999648
5	-0.9995379	-0.9995379
6	1.0000612	1.0000612
7	-0.8585873	-0.8585873
8	-0.9997047	-0.9997047
9	0.9998209	0.9998209
10	-1.0000973	-1.0000973

As can be seen from the table, the manual calculations match exactly with the predictions produced by the `predict()` function, confirming that the implementation of the linear combination is correct.

4. Assignment 4. Neural Networks

This assignment focused on neural networks and to experiment with different activation functions.

Neural networks are parametric models where it tries to find optimal parameters by solving and optimization problem, this is called training. The parameters that are used in neural networks are weights and biases (offset).

Activation functions are non-linear scalar functions that are used to describe the non-linear relationship between $\mathbf{x}$ and $\hat{y}$.

4.1 Learn the trigonometric sine function

To learn the trigonometric sine function using a neural network a dataset was created. This was done by sampling 500 points uniformly at random in the interval [0, 10] and then the sine function was applied to these points. From these points 25 of them were used for training and the rest was used for testing.

The neural network consisted of one input node, one hidden layer with 10 hidden nodes, and one output node. The sigmoid function was used as the activation function.

The model performed well in regions containing training data, as shown in Figure 4.1. However, its performance degraded in the interval [5,7], where no training data were present.

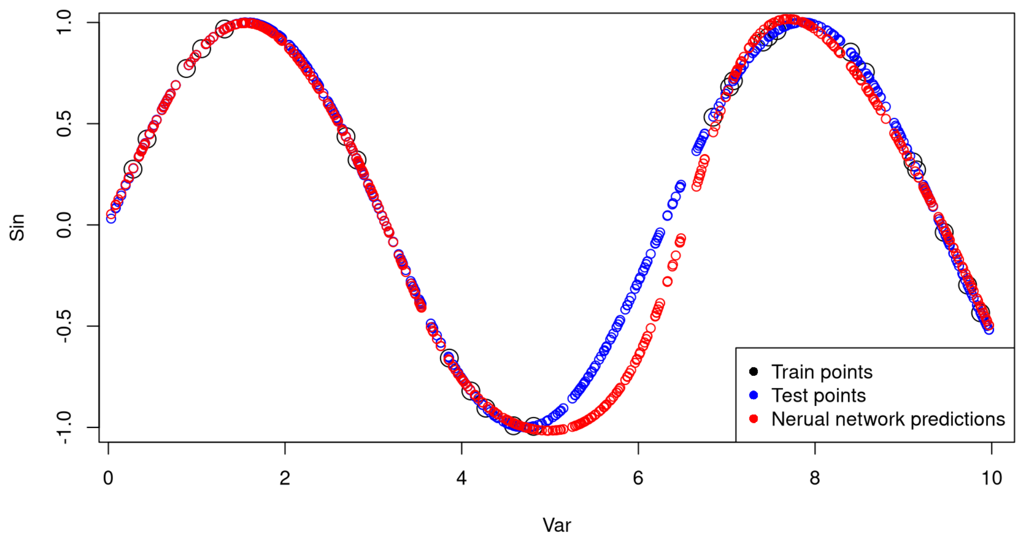

Figure 4.1: Plot of the neural network predictions.

4.2 Custom activation functions

In this task the neural network used different activation functions. The dataset was generated and partitioned in the same way as in section 4.1.

- **Linear:** $f(x) = x$
- **ReLU:** $f(x) = \begin{cases} x, & x \geq 0, \\ 0, & \text{otherwise} \end{cases}$
- **Softplus:** $f(x) = \ln(1 + e^x)$

When using different activation functions the accuracy of the predictions varies significantly (see Figure 4.2). The least accurate model is the one using the linear activation function which is expected since linear activations performs well on linear models. The ReLU activation performs better since it introduces non-linearity. The most accurate model between these are the softplus function this is likely because of the smooth non-linear behavior matches the characteristics of the sine function well.

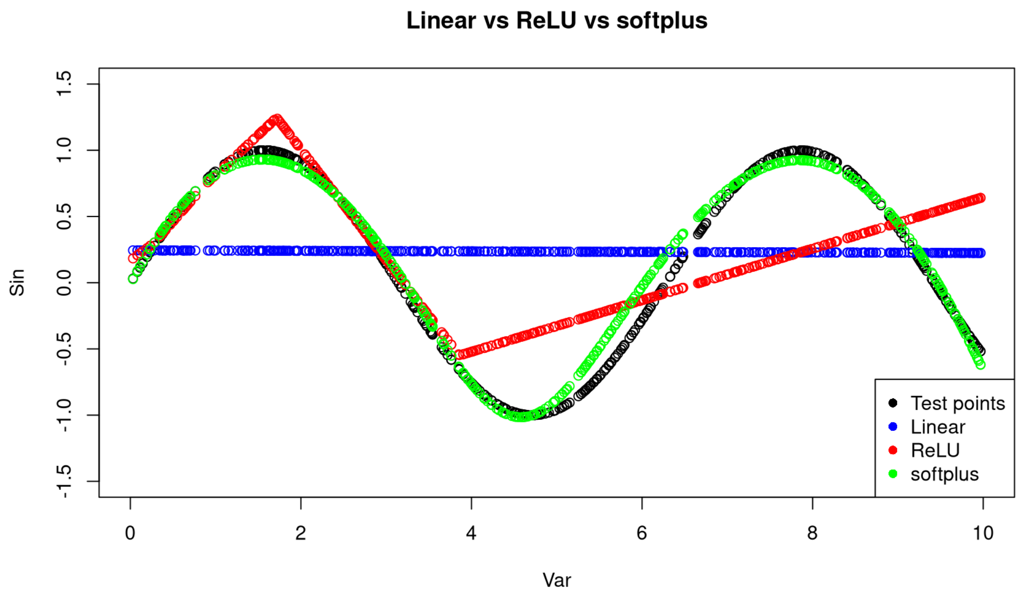

Figure 4.2: Plot of the neural network with different activation functions.

4.3 500 new points

The neural network that was learned in section 4.1 using the sigmoid activation function was used again to predict the sine function for new values. The new values were created by sampling 500 points uniformly at random in the interval [0, 50] and then passed through the sine function. Figure 4.3 shows the predicted values for the new points.

What can be seen is that the model performs very well in the interval [0, 10] but for the interval [10, 50] the model starts to converge to -7.5.

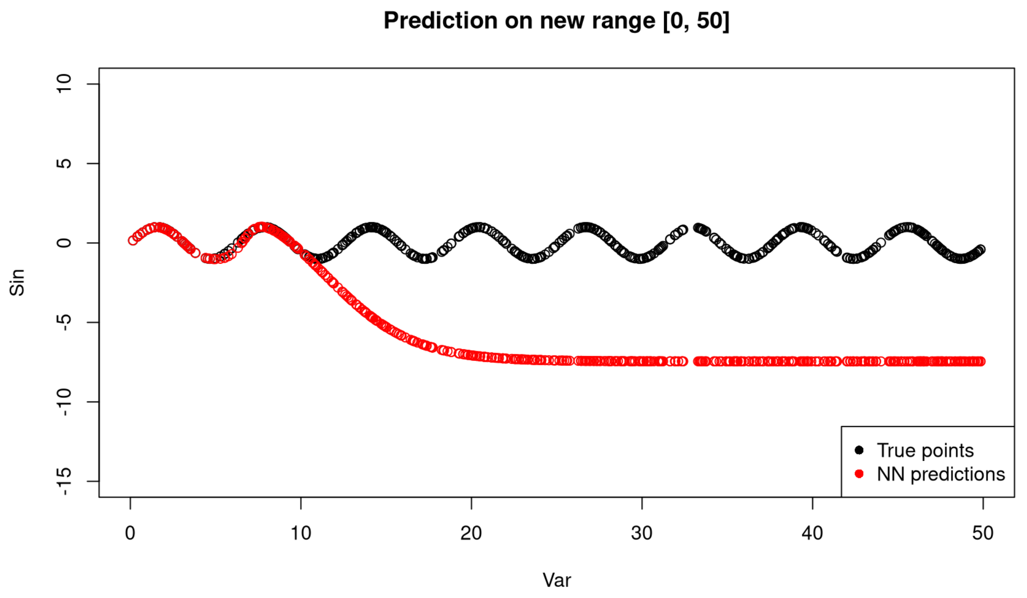

Figure 4.3: Plot of the neural network predictions with 500 new data points.

4.4 Convergence when predicting beyond the training range

In section 4.3 it can be seen that the model converges to -7.5. Tables 4.1 and 4.2 shows the weight and biases from input layer to hidden layer and from hidden layer to output layer.

Table 4.1: Weights from input layer to hidden layer (including biases)

	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
Bias	-1.049	-4.309	-5.875	-4.135	-3.837	7.329	-15.570	1.005	-0.175	3.388
Input weight	1.651	0.383	0.949	2.110	2.934	-2.224	2.345	-1.265	1.223	-2.895

Table 4.2: Weights from hidden layer to output node (including bias)

Connection	Weight
Bias	-0.735
H1 → Output	0.493
H2 → Output	-11.295
H3 → Output	1.543
H4 → Output	-1.490
H5 → Output	0.525
H6 → Output	1.665
H7 → Output	2.832
H8 → Output	-2.623
H9 → Output	0.678
H10 → Output	0.763

The model used the sigmoid activation function which is defined as: $\sigma(x) = \frac{1}{1+e^{-x}}$. For extreme values of x_j the sigmoid function approaches different values. So that $\sigma(x) \to 0$ when $x \to -\infty$ and $\sigma(x) \to 1$ when $x \to \infty$. Where

$$x_j = input \cdot w_{input \to hiddenj} + b_{hiddenj}$$

and the output of the neural network is calculated as:

$$\text{output} = \sum_{j=1}^{10} w_{hidden \to output,j} \cdot h_j + b_{output}$$

where h_j are the activations of the hidden nodes. So for values predictions outside the training range the output simplifies to the sum of the weights and the bias at the output node. This causes the model to converge to a certain value when predicting values outside the training range.

4.5 Inverse neural network

In this task the goal was to use a neural network to predict x from sin(x). Earlier we used the neural network to predict sin(x) from x. The dataset was generated as in section 4.1. When predicting x from sin(x) the result is very bad (see Figure 4.4). This is because the sine function is periodic and therefore a single sine value can correspond to multiple x-values and therefore the network cannot learn a unique mapping from sin(x) to x.

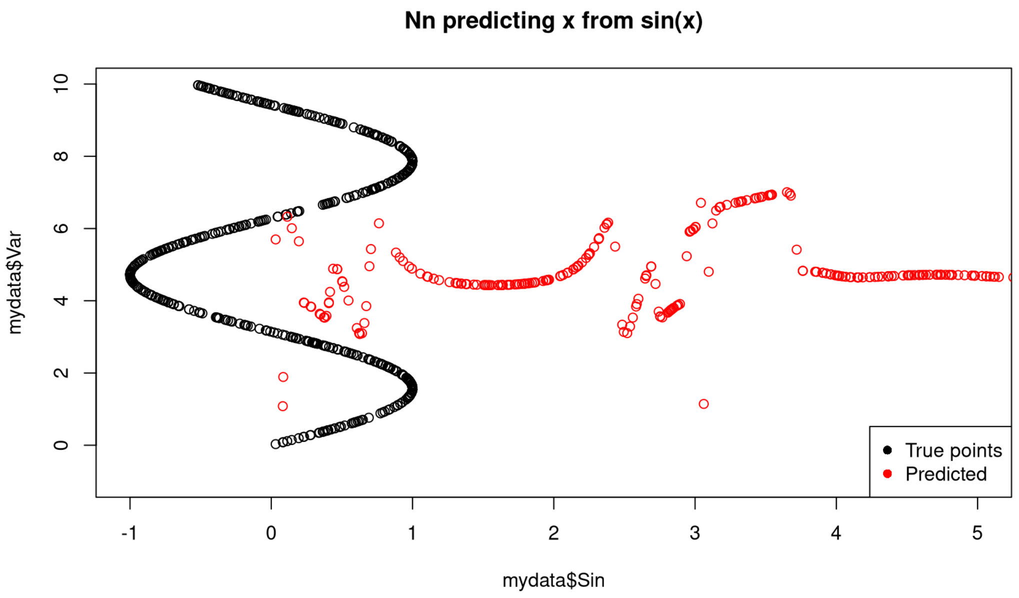

Figure 4.4: Plot of the inverse neural network.

A. Code

A.1 Full R code for assignment 2

```
set.seed(12345)
library(geosphere)
library("dplyr")



stations <- read.csv("stations.csv", fileEncoding = "latin1")
temperatures <- read.csv("temps50k.csv")
st <- merge(stations, temperatures, by="station_number")


#date and time pre-processing
st$date <- as.Date(st$date)
st$time <- substr(st$time, 1, 5)
st$hour <- as.numeric(substr(st$time, 1, 2))

#position pre-processing, calculating haversine distance
positions <- stations%>%select(longitude, latitude)



h_space <- 80000
h_date <- 20
h_time <- 2

#a <- 67.91130 remote point
#b <-  19.60680
#
# a <- 59.8953
# b <- 17.5935# high freq

a <- 58.0340 # Lat
b <- 14.9756 # Long external

# a <-58.4274 #59.8953   17.5935 high frequency point
# b <- 14.826

space_distances <- distHaversine(positions, c(b,a))

#date <- as.Date("2013-07-04") #date to predict
date <- as.Date("2013-08-16")
```

```
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"
)

times <- as.numeric(substr(times, 1, 2))

temp_sum <- vector(length = length(times))     # sum of kernels
temp_prod <- vector(length = length(times))    # product of kernels


#scaled_distance <- function(data, prediction, smoothing) (data-prediction)/h
gaussian_kernel <- function(u) exp(-u^2)


for(i in seq_along(times)){
  pred_time <- times[i]

  st_filtered <- st %>%
    filter(date < date | (date == date & hour <= pred_time))

  #calculate the distances in each domain
  d_space <- distHaversine(cbind(st_filtered$longitude, st_filtered$latitude),
      c(b,a))
  d_date <- abs(as.numeric(st_filtered$date - date))
  d_time <- abs(st_filtered$hour - pred_time)

  #scale the distances
  u_space <- d_space/h_space
  u_date <- d_date/h_date
  u_time <- d_time/h_time

  #calculate the kernel values
  k_space <- gaussian_kernel(u_space)
  k_date <- gaussian_kernel(u_date)
  k_time <- gaussian_kernel(u_time)

  w_sum <- k_space + k_date + k_time
  w_prod <- k_space * k_date * k_time #product of kernels

  temp_sum[i] <- sum(w_sum * st_filtered$air_temperature) / sum(w_sum)
  temp_prod[i] <- sum(w_prod * st_filtered$air_temperature) / sum(w_prod)
}

# Plot sum vs product kernel predictions on same figure
plot(times, temp_prod,
     type = "o",
     pch = 19,
     col = "black",
     lwd = 2,
     xlab = "Hour of the day",
     ylab = "Predicted Temperature ( C )",
     main = "Temperature forecast",
     ylim = range(c(temp_sum, temp_prod), na.rm = TRUE),
     xaxt = "n")  # suppress x-axis to format manually

# Add sum-kernel predictions
points(times, temp_sum,
       type = "o",
       pch = 17,
       col = "red",
       lwd = 2)

# Format x-axis
axis(1, at = times, labels = paste0(times, ":00"))

# Add light grid
grid(nx = NA, ny = NULL, col = "lightgray", lty = "dotted")

# Add legend
legend("topright",
       legend = c("Kernel product", "Kernel sum"),
       col = c("black", "red"),
       pch = c(19, 17),
       lwd = 2)

# oral defence: comparison of sum vs product of kernels:
# Using the sum of kernels gives smooth temperature predictions because a
   measurement
# gets high weight if it is close in any one of the three dimensions (space,
   date, or time).
# Using the product of kernels gives more variable predictions because a
   measurement
# only contributes significantly if it is close in all three dimensions
   simultaneously.
# Therefore, the sum kernel smooths over gaps in data, while the product
   kernel emphasizes
# only the most relevant nearby measurements, leading to sharper peaks and
   dips.
```

A.2 Full R code for assignment 3

```
# Using template:
# Lab 3 block 1 of 732A99/TDDE01/732A68 Machine Learning
# Author: jose.m.pena@liu.se
# Made for teaching purposes

library(kernlab)
set.seed(1234567890)

# Read data
data(spam)
foo <- sample(nrow(spam))
spam <- spam[foo, ]

# Split data into training, validation and test sets
train <- spam[1:3000, ]
valid <- spam[3001:3800, ]
train_valid <- spam[1:3800, ]
test <- spam[3801:4601, ]

# Find best C parameter value
by <- 0.3
err_va <- NULL

for(i in seq(from = by, to = 5, by = by)){
  # Fit SVM with RBF kernel and sigma = 0.05
  filter <- ksvm(type ~ .,
                 data = train,
                 kernel = "rbfdot",
                 kpar = list(sigma = 0.05),
                 C = i,
                 scaled = FALSE)
  mailtype <- predict(filter, valid[,-58])

  # Error rate on validation set
  t <- table(mailtype, valid[,58])
  err_va <-c(err_va, (t[1,2]+t[2,1])/sum(t))
}

# Train on training set. Test on validation set.
filter0 <- ksvm(type ~ .,
                data = train,
                kernel = "rbfdot",
                kpar = list(sigma = 0.05),
                C = which.min(err_va) * by,
                scaled = FALSE)
mailtype <- predict(filter0, valid[,-58])
t <- table(mailtype, valid[,58])
err0 <- (t[1,2]+t[2,1])/sum(t)
```

```
err0

# Train on training set. Test on testing set.
filter1 <- ksvm(type ~ .,
                data = train,
                kernel = "rbfdot",
                kpar = list(sigma = 0.05),
                C = which.min(err_va) * by,
                scaled = FALSE)
mailtype <- predict(filter1, test[,-58])
t <- table(mailtype, test[,58])
err1 <- (t[1,2]+t[2,1])/sum(t)
err1

# Train on training and validation set. Test on test set.
filter2 <- ksvm(type ~ .,
                data = train_valid,
                kernel = "rbfdot",
                kpar = list(sigma = 0.05),
                C = which.min(err_va) * by,
                scaled = FALSE)
mailtype <- predict(filter2, test[,-58])
t <- table(mailtype, test[,58])
err2 <- (t[1,2]+t[2,1])/sum(t)
err2

# Train on the entire data set. Test on test set.
filter3 <- ksvm(type ~ .,
                data = spam,
                kernel = "rbfdot",
                kpar = list(sigma = 0.05),
                C = which.min(err_va) * by,
                scaled = FALSE)
mailtype <- predict(filter3, test[,-58])
t <- table(mailtype, test[,58])
err3 <- (t[1,2]+t[2,1])/sum(t)
err3



# Implementation of SVM predictions
sv_indexes = alphaindex(filter3)[[1]] # Support vector indexes
coefficients = coef(filter3)[[1]]     # alpha * y
intercept = - b(filter3)              # Intercept (b() in kernlab returns
    negative intercept)
k = NULL
rbf_kernel = rbfdot(sigma = 0.05)


# We produce predictions for the first 10 points in the dataset
```

```
for(i in 1:10){
  k2 = NULL
  x_new = as.numeric(spam[i, -58])  # New data point

  for(j in 1:length(sv_indexes)){
    x_j = as.numeric(spam[sv_indexes[j], -58]) # Support vector on index j
    kernel_value = rbf_kernel(x_j, x_new)      # Compute K(x_j, x_new)
    k2 = c(k2, coefficients[j] * kernel_value)
  }
  k = c(k, sum(k2) + intercept)
}

# Compare result with this prediction
answ = predict(filter3, spam[1:10, -58], type = "decision")
round(answ, digits = 7)
k
```

A.3 Full R code for assignment 4

```
install.packages('readxl')
install.packages('kknn')
install.packages('neuralnet')

library(readxl)
library(kknn)
library(neuralnet)

######Exercise 1###########
set.seed(1234567890)
Var <- runif(500, 0, 10)
mydata <- data.frame(Var, Sin=sin(Var)) #calculates sin for each data point.
tr <- mydata[1:25,] # Training
te <- mydata[26:500,] # Test
# Random initialization of the weights in the interval [-1, 1]
set.seed(1234567890)
winit <- runif(31, -1, 1)
nn <- neuralnet(Sin ~ Var, data=tr, hidden=10, startweights = winit, act.fct =
     "logistic")
# Plot of the training data (black), test data (blue), and predictions (red)
print(nn$weights)
plot(tr, cex=2, col="black")
points(te, col = "blue", cex=1)
points(te[,1],predict(nn,te), col="red", cex=1)
legend("bottomright", legend = c("Train points", "Test points", "Nerual
    network predictions"),
       col=c("black", "blue", "red"), pch=16)
```

```
#####Exercise 2#########
linear <- function(x) x
ReLU <- function(x) ifelse(x > 0, x, 0.01 * x)
softplus <- function(x) log(1+exp(x))

nn_linear <- neuralnet(Sin ~ Var, data=tr, hidden=10, startweights = winit,
    act.fct = linear)
nn_ReLU <- neuralnet(Sin ~ Var, data=tr, hidden=10, startweights = winit, act.
    fct = ReLU)
nn_softplus <- neuralnet(Sin ~ Var, data=tr, hidden=10, startweights = winit,
    act.fct = softplus)

plot(te, cex=1, col="black", ylim= c(-1.5, 1.5), main="Linear vs ReLU vs
    softplus")
points(te[,1], predict(nn_linear, te), col="blue", cex=1)
points(te[,1], predict(nn_ReLU, te), col="red", cex=1)
points(te[,1], predict(nn_softplus, te), col="green", cex=1)
legend("bottomright", legend = c("Test points", "Linear", "ReLU", "softplus"),
       col=c("black", "blue", "red", "green"), pch=16)


######Exercise 3###########
set.seed(1234567890)
Var <- runif(500, 0, 50)
mydata <- data.frame(Var, Sin=sin(Var)) #calculates sin for each data point.

set.seed(1234567890)
winit <- runif(31, -1, 1)
plot(mydata, cex=1, col="black", main="Prediction on new range [0, 50]", ylim=
    c(-15, 10))
points(mydata[,1],predict(nn,mydata), col="red", cex=1)
legend("bottomright", legend = c("True points", "NN predictions"), col=c("
    black", "red"),
       , pch=16)

########Exercise 4#######
# nn$result.matrix
# nn$weights
#

#########Exercise 5#######
set.seed(1234567890)
Var <- runif(500, 0, 10)
mydata <- data.frame(Var, Sin=sin(Var)) #calculates sin for each data point.

set.seed(1234567890)
winit <- runif(31, -1, 1)

#before we used Sin~Var, for inverse we use Var~Sin (predict x from sin)
```

```
nn_inverse <- neuralnet(Var ~ Sin, data=mydata, hidden=10, startweights =
    winit, threshold = 0.1)

plot(mydata$Sin,mydata$Var, cex=1, col= "black", main = "Nn predicting x from
    sin(x)",
     ylim=c(-1, 10), xlim = c(-1.0, 5))
points(mydata[,1], predict(nn_inverse, mydata), col="red", cex=1)
legend("bottomright", legend = c("True points", "Predicted"), col = c("black",
     "red"), pch=16)
```