Lab Report 3: Machine Learning 732A99/732A68/TDDE01, Group B37

Erik Lundqvist (erilu777) Axel Strid (axest556)
Oliver Solvang Stoltz (oliso325)

December 17, 2024

Statement of Contribution

- Erik Lundqvist: Wrote the report for question 3 of assignment 1. Coded assignment 2 together with Axel and Oliver. Wrote the report for assignment 2. Coded assignment 4 with Oliver, finished the code myself.
- Axel Strid: Coded the majority of assignment 2 together with Erik and Oliver. Finished the last part of the coding on assignment 2 together with Erik. Coded all of assignment 3 by myself. Wrote the report regarding assignment 3.
- Oliver Solvang Stoltz: Coded the majority of assignment 2 together with Erik and Axel. Coded assignments 4.1, 4.2, and 4.3 with Erik. Wrote the report for two first questions on assignment 1 and the whole assignment 4

1 Theory

In this assignment, three questions will be answered using the course book, Machine Learning: A First Course for Engineers and Scientists [1]

1.1 What is the kernel trick?

The kernel trick in machine learning is when you choose a kernel $\kappa(x, x')$ directly, where x and x' are two arguments from the same space. The kernel then returns a scalar value of the two arguments. The kernel trick is helpful since we do not need to design a d-dimensional vector $\phi(x)$ and derive its inner product. Instead, if x enters a model as $\phi(x)^\top \phi(x')$, we can just choose a kernel $\kappa(x, x')$. [1, pp. 194]

1.2 The purpose of hyperparameter C in SVMs

In SVMs, it is common to use a hyperparameter C for regularization of the model. C is often derived as $C = \frac{1}{2\lambda}$ or $C = \frac{1}{2n\lambda}$ and is used to either prevent overfitting of the model or allow the model to be more accurate on training data. When λ is high, C becomes low, and the model prioritizes higher accuracy on training data. Conversely, when λ is low, C becomes high, and the model focuses on preventing overfitting. [1, pp. 56, 211-212]

1.3 Mini-batch and epoch in neural networks

In neural networks, when training with large datasets, stochastic gradient descent can be used, where mini-batches and epochs are central concepts. A mini-batch is a small subsample of the training data (typically $n_b = 10$, $n_b = 100$, or $n_b = 1000$ data points) that is used to compute gradients and update model parameters. Instead of calculating gradients using all training data points before each parameter update, which is computationally expensive, the algorithm processes these smaller batches and updates parameters more frequently. After the network has processed enough mini-batches to see every training data point exactly once, it has completed one epoch. During an epoch, the network performs n/n_b parameter updates in total, where n is the total number of training examples and n_b is the mini-batch size. [1, pp. 124-125]

2 Kernel Methods

2.1 Data Preparation

The analysis was performed using two datasets provided by the Swedish Meteorological and Hydrological Institute (SMHI): `stations.csv` containing weather station information and `temps50k.csv` containing temperature measurements. These datasets were merged based on station numbers. The prediction point was set to Luleå (65.63141°N, 22.02253°E), with January 15, 2015, as the target date. Data was filtered to include only measurements occurring on or before the target date. Hour information was extracted from the time strings in the dataset for time-based kernel calculations. The prediction task was to forecast temperatures at two-hour intervals throughout the day, from 04:00 to 24:00.

2.2 Kernel Design and Smoothing Coefficients

The temperature prediction model used three Gaussian kernels to account for different aspects of the data: geographic distance, date difference, and time of day. The Gaussian kernel function used was:

$$K(x) = e^{-x^2/(2h^2)}$$

where h is the bandwidth parameter. The following bandwidths were selected:

- Geographic distance: 50,000 meters
- Date difference: 5 days
- Time distance: 4 hours

For geographic distances, the Haversine formula was used to calculate the true distance between points on Earth's surface. For time differences, a circular time calculation was implemented to correctly handle cases like the difference between 23:00 and 01:00, treating it as a two-hour difference rather than a 22-hour difference. As shown in Figure 1, each kernel's influence decreases smoothly with distance, following the Gaussian curve. The rate of decrease is controlled by the respective bandwidth parameters, which were chosen to provide appropriate weighting of nearby versus distant measurements in each dimension.

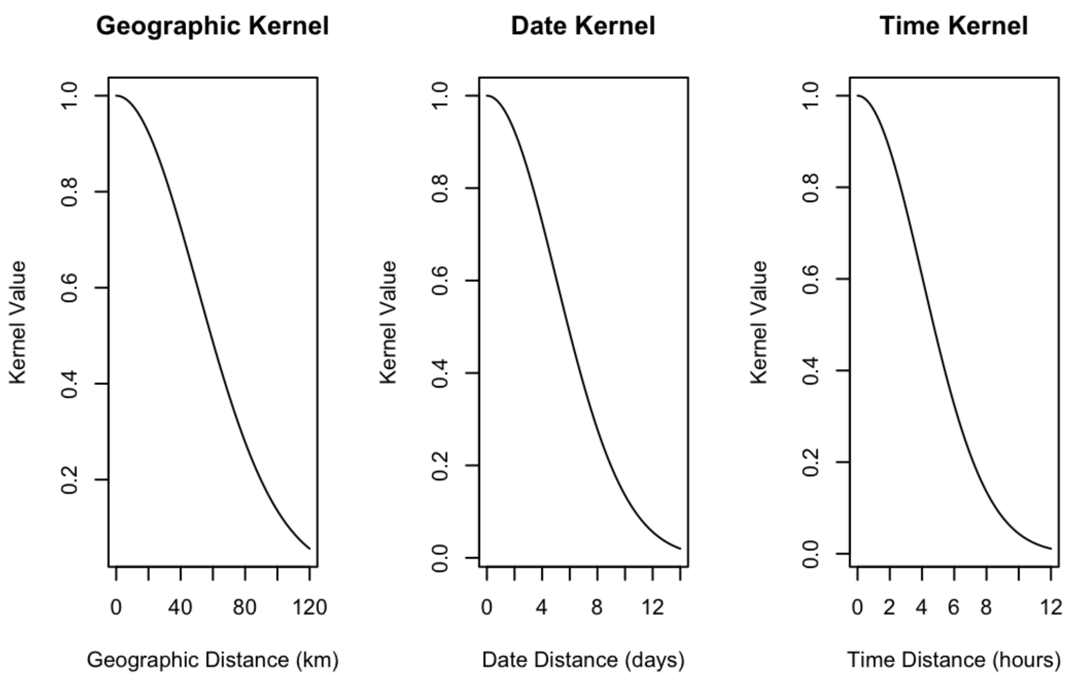

Figure 1: Kernel values as a function of distance for the three different kernels. The geographic kernel (left) shows meaningful influence up to about 80km, the date kernel (middle) up to approximately 10 days, and the time kernel (right) up to about 8 hours from the target time.

2.3 Summation of Gaussian Kernels

The first method of combining the three Gaussian kernels was through addition. For each prediction time, the combined kernel value was calculated as:

$$K_{sum}(x) = K_{geo}(x) + K_{date}(x) + K_{time}(x)$$

Using this summed kernel approach, the temperature predictions for Luleå ranged from 3.41°C to 5.55°C throughout the day. The predictions showed a clear daily temperature cycle, with the lowest temperatures in the early morning hours (3.47°C at 04:00) and highest temperatures in the early afternoon (5.55°C at 14:00). However, looking at nearby station data revealed that these predictions might be overestimating the actual temperatures. The nearest station (Boden Mo) recorded -11.5°C on January 14 (one day prior to the prediction date), suggesting that the summation method may be producing predictions that are too warm. This overestimation can be attributed to the additive nature of the kernel combination, where the influence of distant but numerous measurement points can accumulate and significantly impact the prediction.

2.4 Multiplication of Gaussian Kernels

The second approach combined the kernels through multiplication:

$$K_{prod}(x) = K_{geo}(x) \times K_{date}(x) \times K_{time}(x)$$

This method produced notably different predictions, ranging from -11.41°C to -4.49°C. These predictions align better with the observed temperature of -11.5°C from the nearby Boden Mo station. The difference between summation and multiplication results can be explained by how these operations handle kernel weights. In multiplication, a measurement point needs to be "close" in all three dimensions (geography, date, and time) to have a strong influence on the prediction. If any single dimension has a low kernel value (for instance, if a measurement is from a distant location), the overall influence of that measurement becomes very small. This creates a more selective weighting that favors measurements that are consistently relevant across all dimensions.

2.5 Results Comparison

Table 1 shows the complete set of temperature predictions from both kernel combination methods throughout the day.

Table 1: Temperature predictions using summed and multiplied kernels

Time	Summed Kernels (°C)	Multiplied Kernels (°C)
04:00	3.47	-9.51
06:00	3.92	-10.88
08:00	4.51	-11.30
10:00	5.09	-11.41
12:00	5.48	-11.40
14:00	5.55	-11.31
16:00	5.32	-11.08
18:00	4.89	-10.50
20:00	4.35	-9.13
22:00	3.81	-6.72
24:00	3.41	-4.49

3 Support Vector Machines

In this assignment, we use the `kernlab` package and its included `spam` dataset to **classify emails as spam or nonspam**. The primary objective is to fit SVM models with varying complexity parameters C using a radial basis function (RBF) kernel of width 0.05. Following the provided `Lab3Block1_2021_SVMs_St.R` script, we perform model selection based on validation error, select the best model, and then estimate its generalization error on a separate test set. Finally, we manually implement the SVM decision function to understand how new points are classified under the hood.

3.1 Model Selection and Chosen Filter

In assignment 1, we used the training set to fit multiple SVM models using various values of the *Complexity Parameter* C; C-values (from 0.3 to 5.0 in increments of 0.3) and computed their validation errors. The validation error was calculated for each model by first predicting the classes on the validation set and then forming a confusion matrix:

$$t = \begin{bmatrix} \text{TN} & \text{FP} \\ \text{FN} & \text{TP} \end{bmatrix}$$

The validation error was computed as:

$$\text{Validation Error} = \frac{\text{FP} + \text{FN}}{\text{Total Number of Observations}}$$

After evaluating all candidates, the model with $C = 1.8$ achieved the **lowest validation error** (0.16500).

Once the best C-value was found, we fitted four different models, each using $C = 1.8$:

1. **filter0**: Trained on the training set only and evaluated on the validation set. This confirmed the minimal validation error of 0.16500.

2. **filter1**: Trained on the training set only (same as filter0) and then evaluated on the test set, yielding an error of approximately 0.1673.

3. **filter2**: Trained on both the training and validation sets (combined) and evaluated on the test set. This yielded a lower test error of about 0.1498, reflecting **improved generalization due to more training data**.

4. **filter3**: Trained on all available data (training, validation, and test) and evaluated on the test set. Although it produced a very low error (about 0.0137), this estimate is not reliable as the test data was also used in training.

We ultimately select **filter2** to return to the user, as it makes use of additional training data beyond filter0 and filter1 without contaminating the test set, providing a reliable and improved generalization performance.

3.2 Generalization Error Estimation

To estimate the generalization error of the chosen model, we tested it on the untouched test set. The model we return to the user is filter2, and its error on the test set (err2) is used as the estimate of the generalization error. This error (approximately 0.1498) is **unbiased** since the test set was not involved in model selection or training of filter2.

3.3 Manual Implementation of the Decision Function

The goal of this task is to replicate the `predict()` function manually, thereby gaining insight into how an SVM classifies new points. We are basically **looking under the hood** of the SVM's prediction mechanism. After training an SVM, a new point x_{new} is classified based on the sign of:

$$f(x_{\text{new}}) = \sum_{j \in SV} \alpha_j K(x_j, x_{\text{new}}) + b,$$

where x_j are support vectors, α_j their coefficients, K the kernel function, and b the intercept.

We extract these values (`sv`, `co`, `inte`) from the fitted model `filter3`. All necessary information is already contained in the model, we just access it directly. The steps to compute the decision values for the first 10 points are:

1. Identify the support vectors (`sv`), their coefficients (`co`), and the intercept (`inte`).
2. For each new point x_{new}:
   (a) Convert the new point and each support vector into numeric vectors.
   (b) Compute the RBF kernel value $K(x_j, x_{\text{new}})$ for every support vector x_j.
   (c) Multiply each kernel value by the corresponding coefficient α_j.
   (d) Sum all these products and add the intercept to obtain $f(x_{\text{new}})$.
3. Repeat for all desired points.

After performing these steps, the manually computed decision values for the first 10 test points matched exactly those produced by `predict(filter3, ..., type="decision")`, confirming the correctness of our manual implementation.

Point	Manual Computation (k)	`predict()` Decision Value
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

Table 2: Comparison of manually computed decision values and `predict()` outputs for the first 10 points.

Conclusion: Our computed values matched exactly those obtained by the built-in `predict()` function using `type="decision"`.

4 Neural Networks

This assignment focused on neural networks and experimenting with different activation functions to observe their effects on prediction results and compare them. The dataset was generated by assigning 500 random values and passing them through a trigonometric sine function.

4.1 Learning the Sine Function

500 points were randomly sampled uniformly within the range $[0, 10]$. These points were passed through the $\sin(x)$ function to generate corresponding input-output variable pairs. The training dataset contained 25 points, while the remaining points were used as the test dataset. A Neural Network (NN) with one input node, one hidden layer containing 10 hidden nodes, and one output node was used. The activation function of this model was a sigmoid function. The model was trained on the training data and then used to make predictions on the test data.

In Figure 2, we can see the combined plot of the training data, test data, and the corresponding NN predictions. The NN model performed well, as the predictions closely match the test data. However, in the interval between 5 and 7, the model performs slightly worse, an interval were the model has no training points.

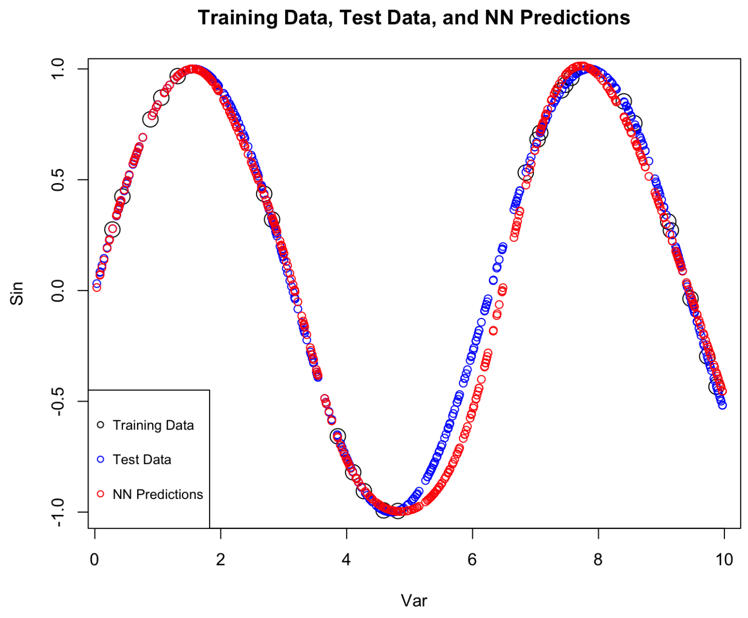

Figure 2: The plot to show Training data, Test data and the corresponding NN-prediction

4.2 Using Custom Activation Functions in Neural Network

This task involved using different activation functions in a Neural Network (NN) and comparing the results. The dataset and the NN models had the same structure as in Section 4.1, except for the activation function. The different activation functions used were:

- Linear: $h_1(x) = x$
- ReLU: $h_2(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$
- Softplus: $h_3(x) = \ln(1 + e^x)$

The three different NN models were fitted and then plotted. In figure 3 we can see the corresponding plots.

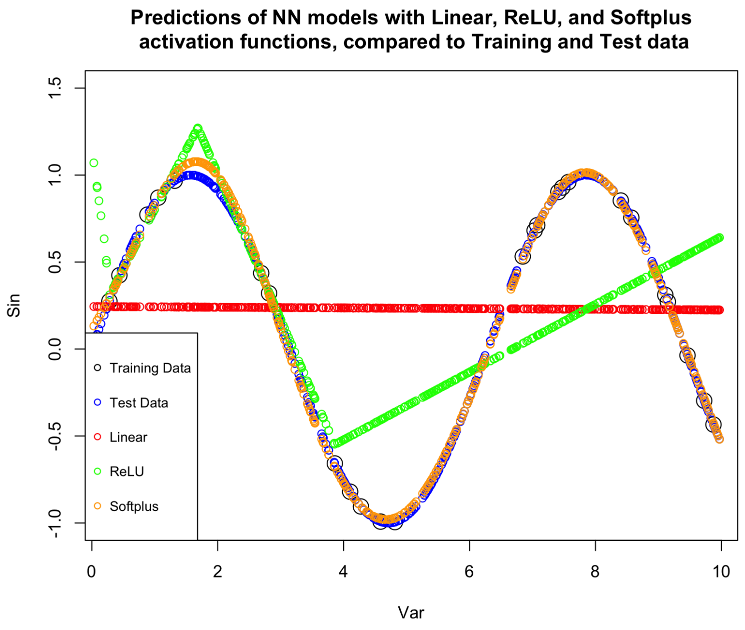

Figure 3: The plot to show Training data, Test data and the corresponding NN-predictions of the different models with linear, ReLU and softpuls activation functions.

The prediction accuracy between the different activation functions varies significantly. The least accurate model is the one with the linear activation function. Even though the model with the ReLU activation function performs better than the linear one, it still has a poor prediction quality. Finally, the model with the softplus activation function performs well and is similar to the sigmoid model shown in Figure 2.

In this task, we encountered some issues while trying to create a custom activation function for ReLU. We used the ‘neuralnet‘ function in R to solve this lab and initially attempted to use the ‘pmax‘ function to construct the ReLU function. However, since ‘pmax‘ returns vectorized values, we replaced it with an ‘ifelse‘ statement to ensure compatibility with the ‘neuralnet‘ function.

4.3 Prediction Beyond the Training Range

In this task the NN model from Section 4.1 where used to predict on a test dataset of 500 points that were randomly sampled uniformly within the range $[0, 50]$, and then passed through the $\sin(x)$ function. The results of this predictions are showed in figure 5.

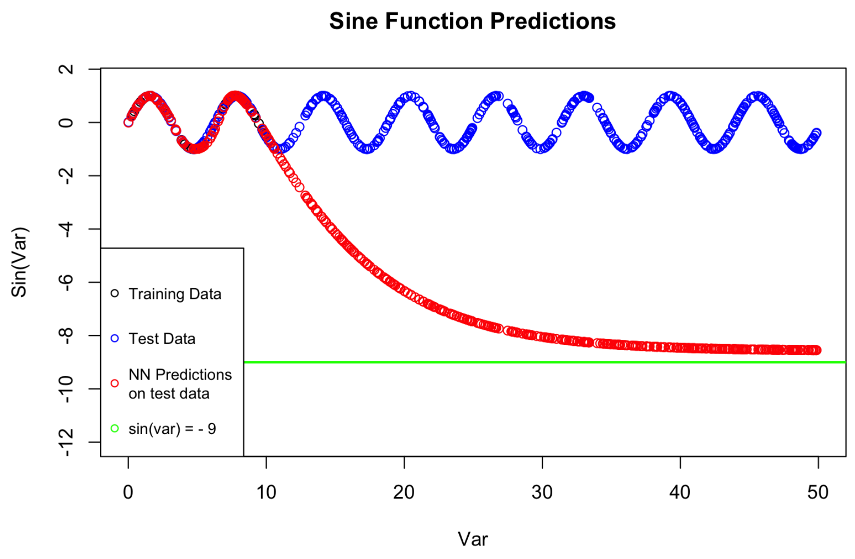

Figure 4: The plot to show Training data, Test data and the corresponding NN-predictions to the new Test dataset.

Within the range that the model was trained on, $[0, 10]$, the model makes good predictions. Outside of this range, the model performs poorly and seems to converge to a value slightly greater than -9. Note that $\sin(var)$ never can becomes -9, this line is just added to see the convergence better.

4.4 Explanation of Convergence Seen in Task 4.3

The reason why we observe convergence in 4.3 is based on the combination of the weights, biases, and the behavior of the logistic sigmoid activation function under large input values. The weights and biases for the trained model are provided in Table 3 and Table 4.

Table 3: Weights Input Node → Hidden Layer and Biases for Hidden Layer Nodes

Hidden Layer Nodes 1–5	**1**	**2**	**3**	**4**	**5**
Weights, $w_{\text{input}\to\text{hidden},j}$	3.30	-0.33	0.40	-0.77	11.96
Biases, $b_{\text{hidden},j}$	-0.80	-0.83	-0.15	-0.83	-1.80

Hidden Layer Nodes 6–10	**6**	**7**	**8**	**9**	**10**
Weights, $w_{\text{input}\to\text{hidden},j}$	-0.21	-6.47	7.90	-0.57	0.04
Biases, $b_{\text{hidden},j}$	0.80	3.08	-2.32	0.15	0.76

Table 4: Weights and Bias Parameters for Hidden Layer Nodes to Output Node

Hidden Layer Node $\to$ Output Node	**Weight Value**, $w_{\text{hidden}\to\text{output},j}$
Node 1	0.7993
Node 2	-4.9649
Node 3	-4.8295
Node 4	22.1704
Node 5	-5.5591
Node 6	-4.3748
Node 7	0.3490
Node 8	-0.7224
Node 9	1.8612
Node 10	-9.4391
Bias (Intercept), Output Node, b_{output}	0.4400

Since we use the logistic activation function, also known as the sigmoid activation function, it is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x_j}}.$$

For extreme values of x_j, the activation function saturates:

- When $x_j \ll 0$, the output approaches 0,
- When $x_j \gg 0$, the output approaches 1,

where

$$x_j = \text{input} \cdot w_{\text{input}\to\text{hidden},j} + b_{\text{hidden},j}.$$

This is referred to as sigmoid saturation. The resulting activation values (0 or 1) are then multiplied by the corresponding weights from the hidden layer to the output node, as shown in Table 4. The output of the network is calculated as:

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

where h_j are the activations of the hidden neurons. In saturated conditions, h_j can only take values of 0 or 1. Consequently, the output simplifies to a weighted sum of the activated weights and the intercept at the output node. This causes the model to converge to a specific value when presented with large input variables outside the training dataset range.

4.5 Inverse Problem: Predicting x from sin(x)

This task involved predicting x from $\sin(x)$. A total of 500 points were randomly sampled uniformly within the range $[0, 10]$. These points were passed through the $\sin(x)$ function to generate corresponding output-input variable pairs for the test dataset. A neural network (NN) model, similar to the one used in Section 4.1, was used, but with $\sin(x)$ as the input and x as the output. Additionally, a threshold of 0.1 was applied to prevent convergence errors. The model was fitted, and the corresponding predictions are shown in Figure 5.

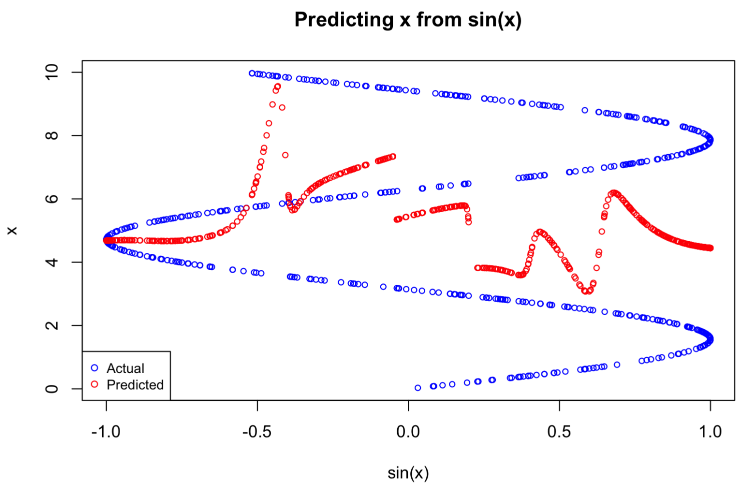

Figure 5: The plot shows actual values from and the models predictions

This model performs very poorly. This is because the sine function has values between $[-1, 1]$ and repeats itself every 2π. As a result, a single sine value can correspond to multiple x-values, introducing confusion and making it more difficult for the model to make accurate predictions.

References

[1] A. Lindholm, N. Wahlström, F. Lindsten, and T. B. Schön, *Machine Learning: A First Course for Engineers and Scientists.* Cambridge University Press, Jul. 2022. Pre-publication version. Accessed online 5 Dec 2024.

Assignment 2 Code

```r

#### LAB 3, ASSIGNMENT 2 ####

#### SET SEED ####
set.seed(1234567890)

#### Install necessary packages ####
install.packages("geosphere")

library(geosphere)

#### Code given in the lab ####
stations <- read.csv("stations.csv", fileEncoding = "latin1" )
temps <- read.csv("temps50k.csv")
st <- merge(stations,temps,by="station_number")
bandwidth_geo <- 50000 # distance in meters (..?)
bandwidth_date <- 5 # number of days
bandwidth_time <- 4 # number of hours

# The point to predict
latitude <- 65.63141
longitude <- 22.02253
interest_date <- as.Date("2015-01-15")
prediction_hours <- c(4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24)

# the vectors that will be filled with predictions
pred_temperatures_sum <- vector(length=length(prediction_hours))
pred_temperatures_prod <- vector(length=length(prediction_hours))

# Ensure the `date` column is in Date format
st$date <- as.Date(st$date)
# Filter the data to keep rows with dates on or before the interest date
filtered_data <- subset(st, date <= interest_date)

# Function used to calculate the gaussian
gaussian_kernel <- function(x, h){
  return(exp(-(x^2) / (2 * h^2)))
}

### Gaussian kernel for distance
geo_distances <- distHaversine(
  cbind(filtered_data$longitude, filtered_data$latitude),
  c(longitude, latitude)
)

K_geo <- gaussian_kernel(geo_distances, bandwidth_geo)

### Gaussian kernel for date
date_distances <- (as.numeric(difftime(interest_date,
    ↪ filtered_data$date, units = "days")))
K_date <- gaussian_kernel(date_distances, bandwidth_date)

# Extract hours from the time strings in filtered_data
filtered_data$hour <- as.numeric(substr(filtered_data$time,
    ↪ 1, 2))

# Calculate predictions for each hour we want to predict
for(i in 1 : length(prediction_hours)){
  target_hour <- prediction_hours[i] # The hour we're trying
      ↪  to predict

  # Calculate time differences
  time_distances <- abs(filtered_data$hour - target_hour)
  # for example, 23 and 01 should have a 2 hour difference,
      ↪ not 23-1=22
  time_distances <- pmin(time_distances, 24 - time_distances
      ↪ )

  # Calculate time kernel
  K_time <- gaussian_kernel(time_distances, bandwidth_time)

  # Combine kernels by summing
  K_sum_combined <- K_geo + K_date + K_time

  # Combine kernels by multiplying
  K_prod_combined <- K_geo * K_date * K_time

  # Calculate weighted average temperatures (both methods)
  pred_temperatures_sum[i] <- sum(K_sum_combined * filtered_
     ↪ data$air_temperature) / sum(K_sum_combined)
  pred_temperatures_prod[i] <- sum(K_prod_combined *
     ↪ filtered_data$air_temperature) / sum(K_prod_combined
     ↪ )

  # Product prediction lower, more "strict" (needs to be
     ↪ close in all directions)
  cat("Time:", target_hour, "- Sum:", round(pred_
     ↪ temperatures_sum[i], 4),
      " degrees C, Product:", round(pred_temperatures_prod[i
         ↪ ], 4), "degrees C\n")
}

### PLOT THE KERNEL VALUE AS A FUNCTION OF DISTANCE

# Create sequence of distances/differences to plot
geo_seq <- seq(0, 120000, length.out=100)  # 0 to 100km
date_seq <- seq(0, 14, length.out=100)     # 0 to 14 days
time_seq <- seq(0, 12, length.out=100)     # 0 to 12 hours

# Calculate kernel values
geo_kernel_values <- gaussian_kernel(geo_seq, bandwidth_geo)
date_kernel_values <- gaussian_kernel(date_seq, bandwidth_
   ↪ date)
time_kernel_values <- gaussian_kernel(time_seq, bandwidth_
   ↪ time)

# Create plots
par(mfrow=c(1,3))  # 1 row, 3 columns of plots

# Geographic distance kernel
plot(geo_seq/1000, geo_kernel_values, type="l",
     xlab="Geographic Distance (km)", ylab="Kernel Value",
     main="Geographic Kernel")

# Date distance kernel
plot(date_seq, date_kernel_values, type="l",
     xlab="Date Distance (days)", ylab="Kernel Value",
     main="Date Kernel")

# Time distance kernel
plot(time_seq, time_kernel_values, type="l",
```

```r
     xlab="Time Distance (hours)", ylab="Kernel Value",
     main="Time Kernel")


# EXTRA, not part of lab but to see the and compare to
   ↪ actual values
# close in geographical distance and date:
# Sort all measurements by distance to our point
st$distance_to_lulea <- distHaversine(
  cbind(st$longitude, st$latitude),
  c(22.02253, 65.63141)
)

# Find stations close to Lule (within 100km)
nearby_stations <- subset(st, distHaversine(cbind(longitude,
   ↪  latitude), c(22.02253, 65.63141)) < 100000)

# Look at measurements near our date of interest (within 2
   ↪ weeks)
nearby_data <- subset(nearby_stations,
                      abs(as.numeric(difftime(date, as.Date(
                         ↪ "2015-01-15"), units="days"))) <
                         ↪  14)

# Print relevant information
print("Nearby measurements:")
print(nearby_data[, c("station_number", "station_name", "
   ↪ date", "time", "air_temperature")])

# Looking at this data, the product predictions seem better,
# the sum predictions guessed too high
```

Assignment 3 Code

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

####################################################

# This template is from the file Lab3Block1_2021_SVMs_St.R
# ALL COMMENTS ARE MADE OF US STUDENTS IN ORDER TO FULLY UNDERSTAND THIS SCRIPT!

# TASK: Classify mail as spam/nonspam using Support Vector Machines

# Data Setup
install.packages("kernlab")
library(kernlab)
set.seed(1234567890)
data(spam) # spam dataset comes with kernlab package

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

# Split the data
tr <- spam[1:3000, ]     # train: first 3000 observations
va <- spam[3001:3800, ] # validation: next 800 observations
trva <- spam[1:3800, ]  # combined train+validation: first 3800 observations
te <- spam[3801:4601, ] # test: last 801 observations

# Idea:
# 1. Use tr to fit models with different C values (complexities).
# 2. Select best model using va (min validation error)
# 3. Use te as a independent test set to estimate generalization error of chosen model

by <- 0.3
err_va <- NULL

```

```
# Trying different values of C, from 0.3 - 5.0 in increments
    ↪ of 0.3
for(i in seq(by,5,by)){

  # Fit a SVM (=filter) on train set, using Radial Basis
      ↪ Function kernel (rbfdot) with fixed width=0.05
  filter <- ksvm(type~.,data=tr,kernel="rbfdot",kpar=list(
      ↪ sigma=0.05),C=i,scaled=FALSE) # Different C

  # Predict on validation set
  mailtype <- predict(filter,va[,-58]) # Target variable "
      ↪ type" in column 58 (=spam/nonspam)

  # Confusion matrix
  t <- table(mailtype,va[,58])

  # Validation error = (FP+FN)/n
  err_va <-c(err_va,(t[1,2]+t[2,1])/sum(t))
}

# err_va contains validation errors for each C (16 values)
err_va # [0.30375 0.22375 0.20125 0.17375 0.16750 0.16500
    ↪ 0.16625 0.16875 0.17000 0.16875 0.16750 0.16750
    ↪ 0.16875 0.16750 0.16750 0.16750]

min_err_va_pos <- which.min(err_va)  # Position 6 in the
    ↪ array
min_err_va <- err_va[min_err_va_pos] # 0.16500
C_min_err_va <- min_err_va_pos * by  # 1.8

# Select best model SVM with C = 1.8

# filer0 = SVM with C = 1.8 (based on minimum validation
    ↪ error = 0.16500 when C=1.8)
# Fitted on train dataset, prediction on validation dataset
filter0 <- ksvm(type~.,data=tr,kernel="rbfdot",kpar=list(
    ↪ sigma=0.05),C=which.min(err_va)*by,scaled=FALSE)
mailtype <- predict(filter0,va[,-58])
t <- table(mailtype,va[,58])
err0 <- (t[1,2]+t[2,1])/sum(t)
err0 # 0.16500 (obviously)

# filter1 = same model, now prediction on test
```

```r
filter1 <- ksvm(type~.,data=tr,kernel="rbfdot",kpar=list(
    ↪ sigma=0.05),C=which.min(err_va)*by,scaled=FALSE)
mailtype <- predict(filter1,te[,-58])
t <- table(mailtype,te[,58])
err1 <- (t[1,2]+t[2,1])/sum(t)
err1 # 0.1672909

# Idea: Now that we've selected best C, train on more data
    ↪ to (hopefully) get a better model
# filter2 = same C, fitted on train+validation, prediction
    ↪ on test
filter2 <- ksvm(type~.,data=trva,kernel="rbfdot",kpar=list(
    ↪ sigma=0.05),C=which.min(err_va)*by,scaled=FALSE)
mailtype <- predict(filter2,te[,-58])
t <- table(mailtype,te[,58])
err2 <- (t[1,2]+t[2,1])/sum(t)
err2 # 0.1498127

# filter3 = same C, fitted on all data, prediction on test
filter3 <- ksvm(type~.,data=spam,kernel="rbfdot",kpar=list(
    ↪ sigma=0.05),C=which.min(err_va)*by,scaled=FALSE)
mailtype <- predict(filter3,te[,-58])
t <- table(mailtype,te[,58])
err3 <- (t[1,2]+t[2,1])/sum(t)
err3 # 0.01373283




# Questions

# 1. Which filter do we return to the user ? filter0,
    ↪ filter1, filter2 or filter3? Why?
# Answer: filter2 - because best generalization error, and
    ↪ predicted on independent dataset. With C chosen from
    ↪ training+validation.
# filter3 lower error, however tested on same data as it has
    ↪  been fitted to. Overfitting + biased gerneralization
    ↪ error.

# 2. What is the estimate of the generalization error of the
    ↪  filter returned to the user? err0, err1, err2 or err3
    ↪ ? Why?
```

```r
# Answer: err2 - generalization error estimate from a model not trained on test data. Corresponds to testing filter2.

# 3. Implementation of SVM predictions.




# Goal: Understand how an SVM makes predictions and replicate this process manually.
# i.e., Replicate what predict() does "under the hood"

# Theory: Once a SVM has been fitted to the training data, a  new point is essentially classified
# according to THE SIGN of a linear combination of the KERNEL FUNCTION VALUES between the
# support vectors and the new point.

# From lab instructions:
# alphaindex: Return the indexes of the support vectors
# coef: The linear coefficients for the support vectors
# b: The negative intercept of the linear combination

# Extract constants from the filter3 model
sv <- alphaindex(filter3)[[1]] # Index of Support Vectors of  model filter3
co <- coef(filter3)[[1]]       # Coefficients for each Support Vector
inte <- - b(filter3)           # Intercept of the linear combination (b() in kernlab reutrns negative intercept )
k <- NULL                      # Kernel function

# SVM with kernel K classifies new point x_new based of the sign of this function:
# f(x_i) = sum_j(a_j * K(sv_j, x_i)) + intercept, where a_j = coefficients for SV j

# Predict first 10 points in spam dataset
for(i in 1:10){ # We produce predictions for just the first 10 points in the dataset.
  k2 <- NULL

  # Compute the kernel values BETWEEN each support vector and the new point
  # The new point being spam[i,-58] meaning row i, column 58 (target variable = type)
  x_new <- spam[i, -58] # x_new is the new point

  # Iterate through all Support Vectors
  for(j in 1:length(sv)){
    # Define support vector
    sv_x <- spam[sv[j], -58] # Support Vector on index j

    # Convert data frame rows into numeric vectors
    # rbfkernel() function requires this
    sv_x <- as.numeric(sv_x)
    x_new <- as.numeric(x_new)

    # We need the kernel function value K(sv_x, x_new)
    # The kernel is RBF (Radial Basis Function) with sigma =0.05
    # Define RBF kernel function:
    rbfkernel <- rbfdot(sigma = 0.05)

    # Compute K(sv_x, x_new)
    k_val <- rbfkernel(sv_x, x_new)

    # Multiply by the corresponding coefficient
    k2 <- c(k2, k_val * co[j])
  }
  # Sum over all Support Vector and add intercept
  k <- c(k, sum(k2) + inte)
}

# k now holds the decision value for the first 10 points in spam dataset
k
predict(filter3,spam[1:10,-58], type = "decision")

# The two lines above indeed has the same output!
```

Assignment 4 Code

```

#### Install packages ####
install.packages("neuralnet")
library(neuralnet)

#Set SEED
set.seed(1234567890)

#### Assignment 4.1 ####
#Given code from assignment
Var <- runif(500, 0, 10)
mydata <- data.frame(Var, Sin=sin(Var))
tr <- mydata[1:25,] # Training
te <- mydata[26:500,] # Test
# Random initialization of the weights in the interval [-1,
    ↪ 1]
winit <- runif(10,-1,1)
nn_logi <- neuralnet(Sin ~ Var, tr, hidden = 10,
    ↪ startweights = winit)

# Plot of the training data (black), test data (blue), and
    ↪ predictions on the test data (red)
plot(tr, cex=2, main="Training Data, Test Data, and NN
    ↪ Predictions")
points(te, col = "blue", cex=1)
points(te[,1],predict(nn_logi,te), col="red", cex=1)
legend("bottomleft",
       legend = c("Training Data", "Test Data", "NN
           ↪ Predictions"),
       col = c("black", "blue", "red"),
       pch = 1,
       cex = 0.8)

#### Assignment 4.2 ####
# Neural network with the activation function to be linear f
    ↪ (x) = x
nn_linear <- neuralnet(Sin ~ Var, tr, hidden = 10, act.fct =
    ↪  function(x) x, startweights = winit)

#ReLU function
ReLU <- function (x){
  ifelse(x >= 0, x, 0) #in the report explain why we did
     not use return or pmax()
}
# Neural network with the activation function ReLU f(x)=x,
   ifelse(x >= 0, x, 0)
nn_ReLU <- neuralnet(Sin ~ Var, tr, hidden = 10, act.fct =
   ReLU, startweights = winit)

#Softplus function
softplus <- function(x){
  log(1+exp(x))
}
# Neural network with the activation function softplus f(x)=
   log(1+exp(x)
nn_softplus <- neuralnet(Sin ~ Var, tr, hidden = 10, act.fct
   = softplus, startweights = winit)

plot(tr, cex=2, ylim = c(-1, 1.5), main="Predictions of NN
   models with Linear, ReLU, and Softplus\n activation
   functions, compared to Training and Test data"
)
points(te, col = "blue") # Test data points
points(te[,1], predict(nn_linear, te), col="red") # Linear
   model predictions
points(te[,1], predict(nn_ReLU, te), col="green") # ReLU
   model predictions
points(te[,1], predict(nn_softplus, te), col="orange") #
   Softplus model predictions
legend("bottomleft",
       legend = c("Training Data", "Test Data", "Linear", "
          ReLU", "Softplus"),
       col = c("black", "blue", "red", "green", "orange"),
       pch = 1, cex = 0.8)


#### Assignment 4.3 ####

Var<- runif(500, 0, 50)

mydata2 <- data.frame(Var, Sin=sin(Var))
test2 <- mydata2[1:500,]

# Set up the plot with x-axis limits between 0 and 50
```

```r
plot(tr, cex = 1, xlim = c(0, 50), ylim = c(-12, 1.5), main = "Sine Function Predictions", xlab = "Var", ylab = "Sin(Var)")

# Add test data points in blue
points(test2, col = "blue", cex = 1)

# Add predictions from the neural network in red
points(test2[, 1], predict(nn_logi, test2), col = "red", cex =1)
abline(h = -9, col = "green", lwd = 2)

legend("bottomleft", legend = c("Training Data", "Test Data", "NN Predictions
on test data", "sin(var) = - 9"),
       col = c("black", "blue", "red", "green"), pch = c(1, 1, 1), cex = 0.8)

#### Assignment 4.4 ####
# Print the weights
print(nn_logi$weights)

# Look at the weights

#### Assignment 4.5 ####

# Sample points
set.seed(1234567890)  # Reset seed for reproducibility
Var <- runif(500, 0, 10)

# Create data frame but now sin(x) is input and x is output!
mydata_inverse <- data.frame(Var = Var, Sin = sin(Var))

# Train neural network with flipped relationship
# Sin ~ Var (from earlier) becomes Var ~ Sin
nn_inverse <- neuralnet(Var ~ Sin, mydata, hidden = 10, threshold = 0.1)

# Plot results
plot(mydata_inverse$Sin, mydata_inverse$Var,
     main = "Predicting x from sin(x)",
     xlab = "sin(x)",
     ylab = "x",
     col = "blue",
     cex = 0.7)
points(mydata_inverse$Sin, predict(nn_inverse, mydata_inverse),
       col = "red",
       cex = 0.7)
legend("bottomleft",
       legend = c("Actual", "Predicted"),
       col = c("blue", "red"),
       pch = 1,
       cex = 0.8)
```