Assignment 1

Implement a kernel method to predict the hourly temperatures for a date and place in Sweden. To do so, you are provided with the files `stations.csv` and `temps50k.csv`. These files contain information about weather stations and temperature measurements in the stations at different days and times. The data have been kindly provided by the Swedish Meteorological and Hydrological Institute (SMHI).

You are asked to provide a temperature forecast for a date and place in Sweden. The forecast should consist of the predicted temperatures 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 **physical** distance from a station to the point of interest. For this purpose, use the function `distHaversine` from the R package `geosphere`.
- 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. No cross-validation should be used. Instead, choose manually a width that gives large kernel values to closer points and small values to distant points. Show this with a **plot** of the kernel value as a function of distance. **Help**: 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.

Finally, repeat the exercise above by combining the three kernels into one by **multiplying** them, instead of summing them up. Compare the results obtained in both cases and elaborate on why they may differ.

Our task is to predict the temperature of a specific location on a specific date based on data from various weather stations in Sweden. This is to be done by using kernels, which allow us to regulate how much a certain stations observation will contribute to our prediction based on the distance between our observation and the observation by the station, and this distance will be measured in physical distance, date difference, and time difference.

The kernel to be used is the Gaussian Kernel, which is defined as the following, where l is the smoothing factor:

$$k(\boldsymbol{x}, \boldsymbol{x}') = exp(-||\boldsymbol{x} - \boldsymbol{x}'||^2 / 2\ell^2)$$

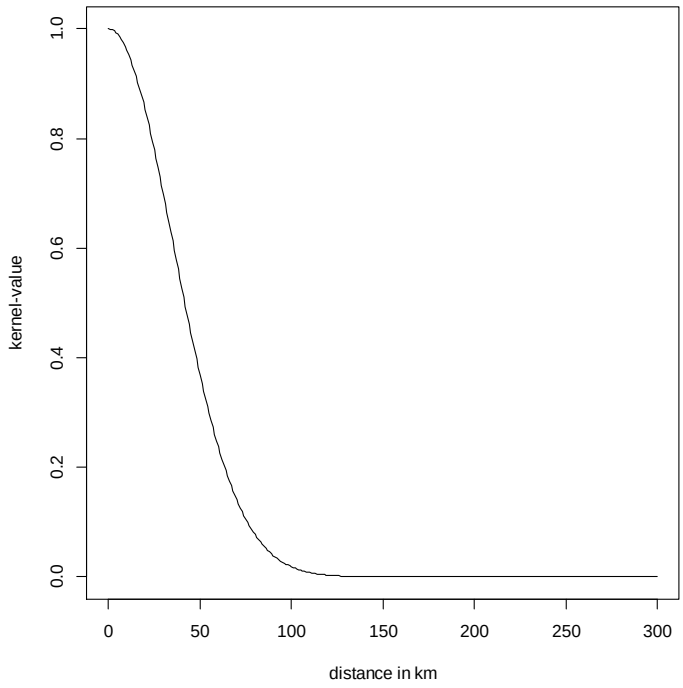
The first step is to choose appropriate smoothing coefficients (widths) for each of the three kernels to be used, where a larger value of the smoothing coefficient makes points further away from our observation have larger impact. This is due to the fact that a larger smoothing factor/width makes the tails fatter if we were to plot the gaussian kernels distribution, and thus even if an observation is not close to our observation, the value on the y-axis, i.e. the kernel-value, will still be high and this observation will still have a large impact.

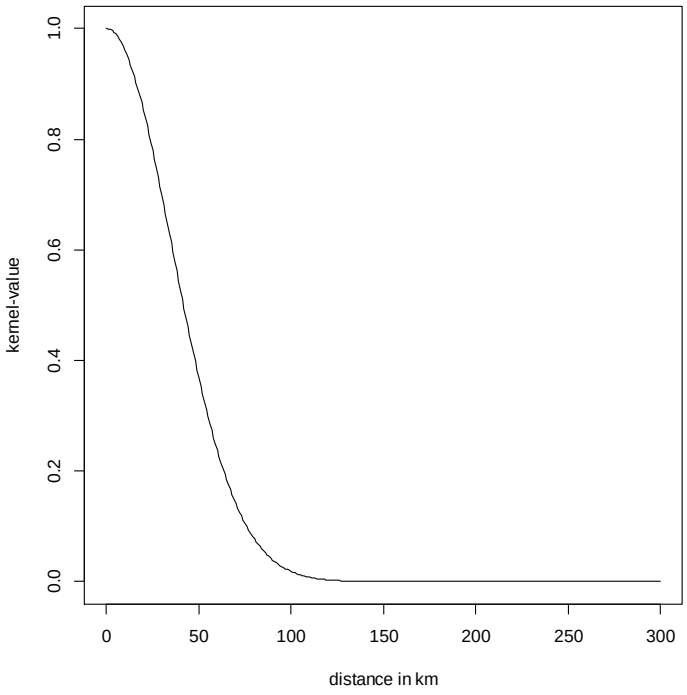
By choosing different smoothing factors to obtain reasonable behaviour, the following smoothing factors were chosen:

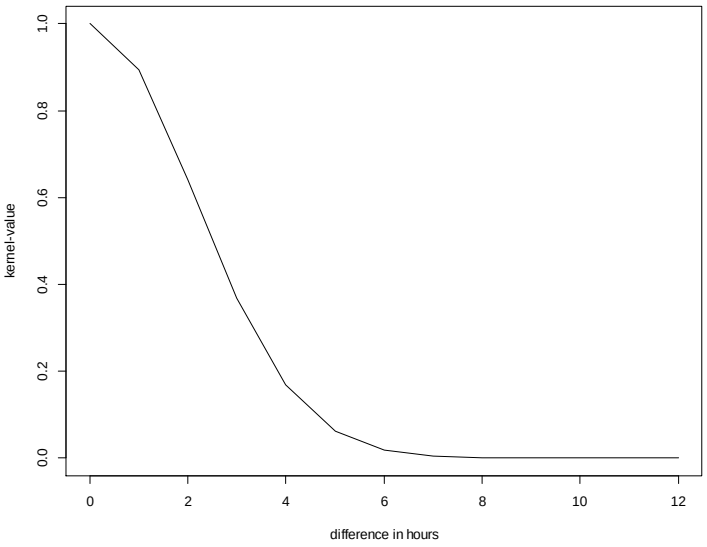
Kernel	Distance	Day	Time
Smoothing factor	50000		1.2
Kernel-value close to 0 when:	>100km	>30days	>5hours

The selection was made by finding smoothing factors that made the respective kernel-values be close to 0 when the distance was larger than 100km, the day difference was larger than 30 days, and if the time difference was larger than 5 hours. These values were chosen since hopefully will prevent both overfitting and underfitting to data, by not giving a large weight to few observations nor to many observations.

Below are the three plots illustrating the three different kernels impact based on different differences between two points:




kernel-value
0.0
0.2
0.4
0.6
0.8
1.0
0
50
100
150
200
250
300
distance in km


kernel-value
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8
10
12
difference in hours

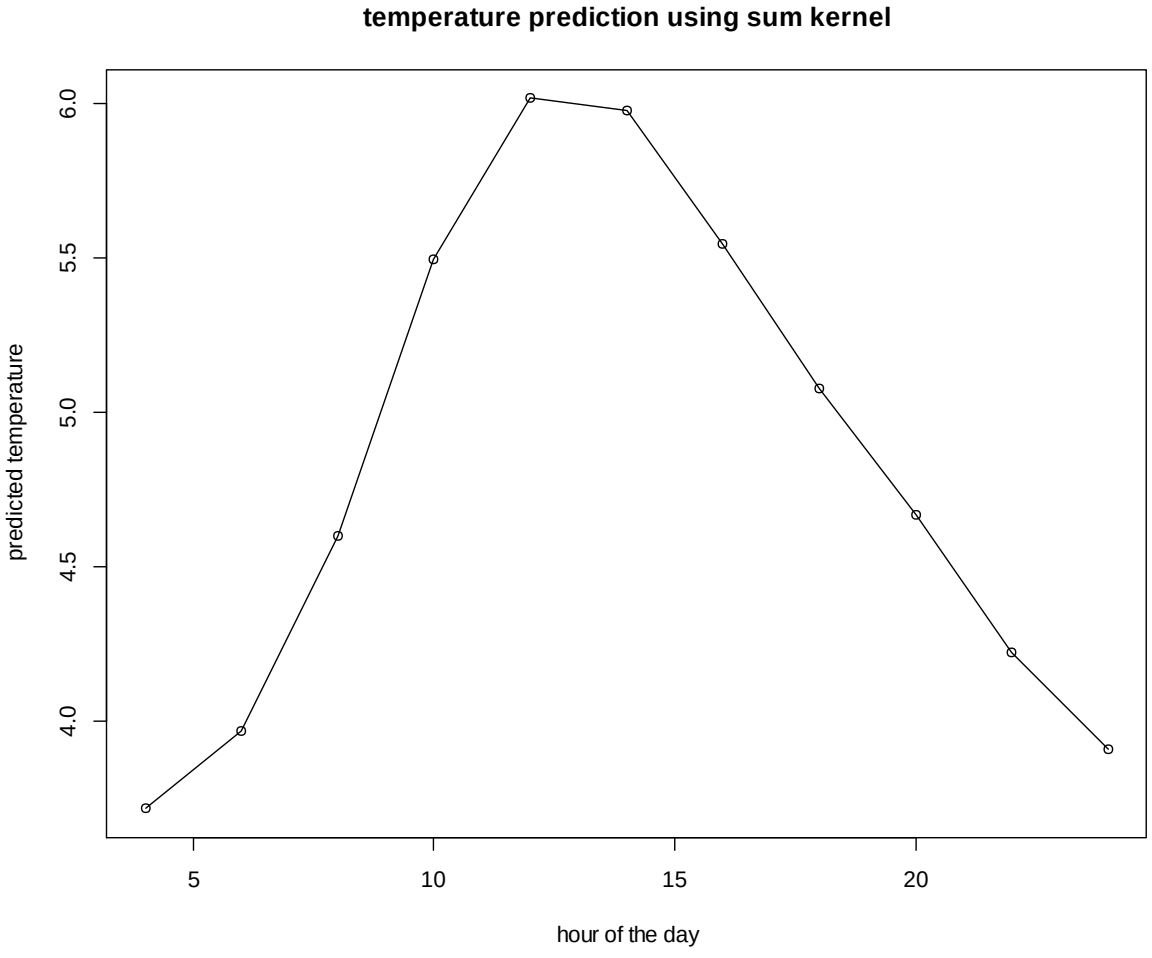
After this, we define these three kernels as functions, all of which take in 2 parameters, perform some computation to compare the distance/date/time and return respective kernel value. Now we can create a new kernel by combing the three kernels by adding them together.

We can perform predictions using a kernel in the following way:

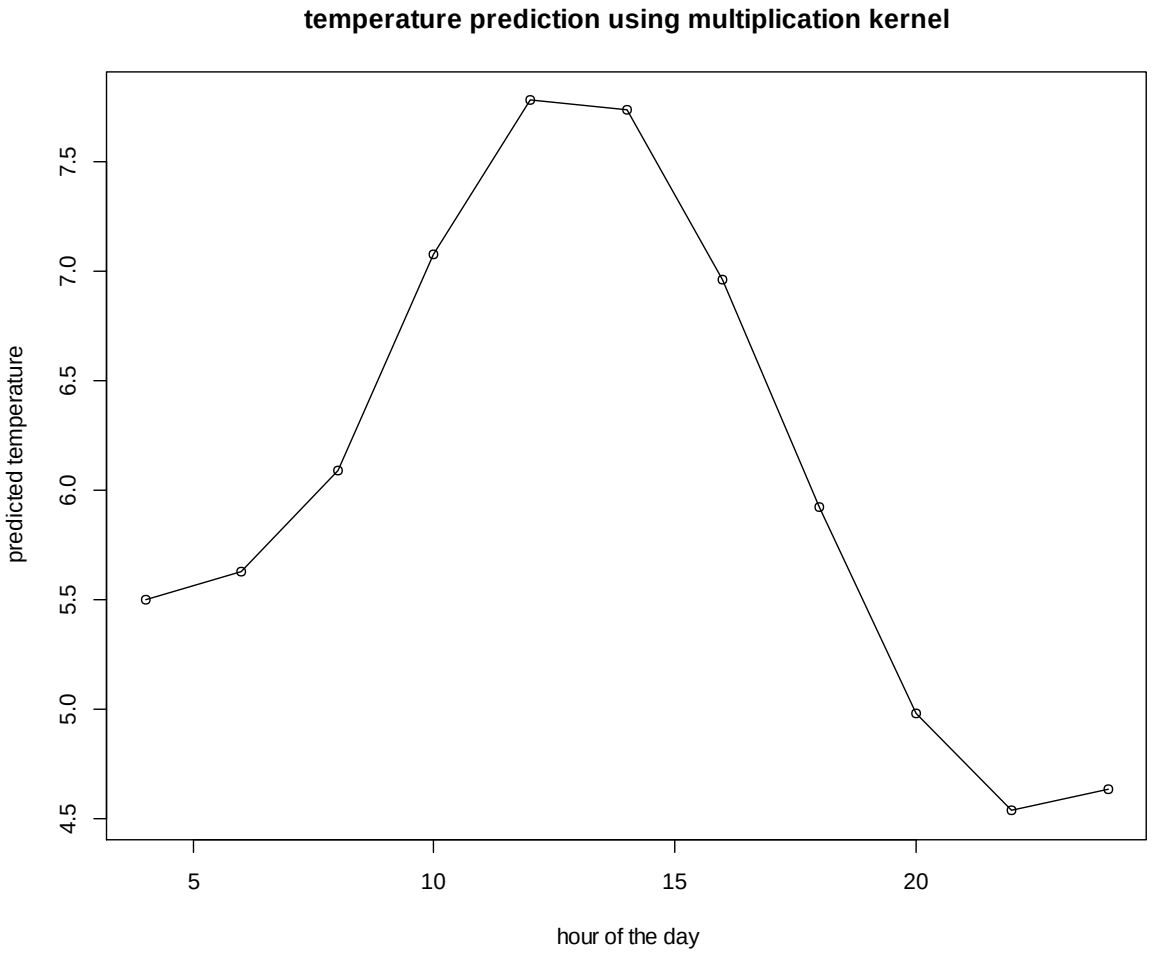
$$\widehat{y}_k(\boldsymbol{x}_*) = \frac{\sum_{i=1}^{n} k\left(\frac{\boldsymbol{x}_* - \boldsymbol{x}_i}{h}\right) y_i}{\sum_{i=1}^{n} k\left(\frac{\boldsymbol{x}_* - \boldsymbol{x}_i}{h}\right)}$$

Where x_* is our point of interest. Meaning, we make a prediction by looking at the temperature of all data points y_i, given and they will all contribute with different weights (kernel-values). These are summed up and divided by the sum of all kernel values, and this will give us our prediction for a specific location, date, and time.

Using above definition and the summation kernel, and then predicting the temperature for our test location and test date, using time increments of 2 hours, will give us the following predictions:



We can also create a new kernel by combing the three kernels by multiplication. Using above definition and the multiplication kernel, and then predicting the temperature for our test location and test date, using time increments of 2 hours, will give us the following predictions:



Comparing the 2 kernels and their plots, in this specific case they are rather similar, however they could have differentiated by a lot. This is due to the fact in the differences between them, because the kernel which is a summation of the other 3 kernels does not require a point to be close in space, date and time, it is enough that just one or two of these aspects are close, and this point will be relevant for the prediction by having a substantial weight. In contrary to the product kernel is required to be close in space, date and time for the point to have a relevant contribution to the prediction i.e. a high weight.

Assignment 2

2. Support Vector Machines

The code in the file `Lab3Block1_2021_SVMs_St.R` performs SVM model selection to classify the `spam` dataset. To do so, the code uses the function `ksvm` from the R package `kernlab`, which also includes the `spam` dataset. All the SVM models to select from use the radial basis function kernel (also known as Gaussian) with a width of 0.05. The C parameter varies between the models. Run the code in the file `Lab3Block1_2021_SVMs_St.R` and answer the following questions.

(1) Which filter do you return to the user ? `filter0`, `filter1`, `filter2` or `filter3`? Why?

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

The data is split into 3 sets, training set (3000 observations), validation set (800 observations), and test set (800 observations).

We then train a model on training data and use the validation data to find the optimal hyperparameter C-value through cross-validation. Generally what you do when you have three data sets like this is the following: You use the training and validation sets to select hyperparameter(s) through cross validation, in which you find the optima value(s) for hyperparameter(s). Then, you retrain the whole model using all of training + validation data, since the model that was training when doing cross validation has only been trained on k-1 / k parts of the data, where k is the number of folds in k-fold cross validation. Lastly, we can use the test set to evaluate our performance and estimate the generalization error, i.e. the error we will have on new data.

We are also 4 filters which have the following properties:

Filter	Filter 0	Filter 1	Filter 2	Filter 3
Trained on:	train	train	train+validation	train+validation+test
Predicts on:	validation	test	test	test
Misclassification rate	0.068	0.085	0.082	0.021

However, we are not interested in the generalization error here, we are simply looking for the best model to return, which in thus case would be filter 3, since it has been trained on the most data.

If we were to look at filter 2, which is trained using training + validation error and then uses unseen data, test set, for evaluating the model, it should give us a good estimate of the generalization, i.e. error 2.

Error 1 is also a valid estimate, an estimate, of the generalization error, since it is produced on previously unseen data, the test set. Filter1 and filter 2 should not be that much different, if trained on train or train + validation set. Thus error 1 would also be an okay answer in the sense that it is unbiased, since you havent used test data for training, but filter 2 is better since it uses both training data and validation data for training, so there no reason to choose filter 1 over this. However, error 0 and error 3 should not be used for to estimate the generalization error, since in both these models, the data which we are predicting on in order to estimate the generalization error has previously

been used to train the model (either directly as in filter 3, or by selecting hyperparameter as in filter 0, making the validation set not previously unseen).

Thus, according to above discussion, the error we would like to return should be error 2, since this filter is trained on training + validation data, and the generalization error is estimated on previously unseen data.

However, regarding which model to actually return, should be filter 3. This is due to the fact that after we have evaluated our generalization error, we can retrain the model using ALL the data, since more data will of course give us a more accurate model, and return this model. However, this leaves no room for estimating the generalization of this model, but we have the above concluded estimation of error 2, and the generalization error from filter 3 should not perform worse than this due to the fact that it is trained using more data. So error 2 should be an upper bound for the real error of filter 3. However, error 2 is the UNBIASED generalization estiamte for filter 2, and we can not say that it is an unbiased generalization estimate for filter 3. Thus, error 2 is only and estimation of the generalization error for filter 3 (and thus biased).

Thus, return filter 3, and error 2 as estimated generalization error (upper bound for filter 3).

(3) 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. You are asked to implement this linear combination for `filter3`. You should make use of the functions `alphaindex`, `coef` and `b` that return the indexes of the support vectors, the linear coefficients for the support vectors, and the **negative** intercept of the linear combination. See the help file of the `kernlab` package for more information. You can check if your results are correct by comparing them with the output of the function `predict` where you set `type = "decision"`. Do so for the first 10 points in the `spam` dataset. Feel free to use the template provided in the `Lab3Block1_2021_SVMs_St.R` file.

Given a new input, the support vector classification performs predictions according to the following:

$$\hat{y}(\boldsymbol{x}_\star) = sign\left(\hat{\alpha}^T \boldsymbol{K}(\boldsymbol{X}, \boldsymbol{x}_\star) + b\right)$$

Where alpha is the dual parameter vector used for predictions containing the support vector coefficients, and K is the kernel value between the training points and our new point of interest, and b is the intercept.

By implementing this above equation manually and calculating the predicted value for the first 10 data points, we receive the following values:

```
> preds
 [1] -1.998999  1.560584  1.000278 -1.756815 -2.669577  1.291312 -1.068444 -1.312493  1.000184 -2.208639
```

As compared to when using our previously defined model filter3 to make predictions, where we receive the following values:

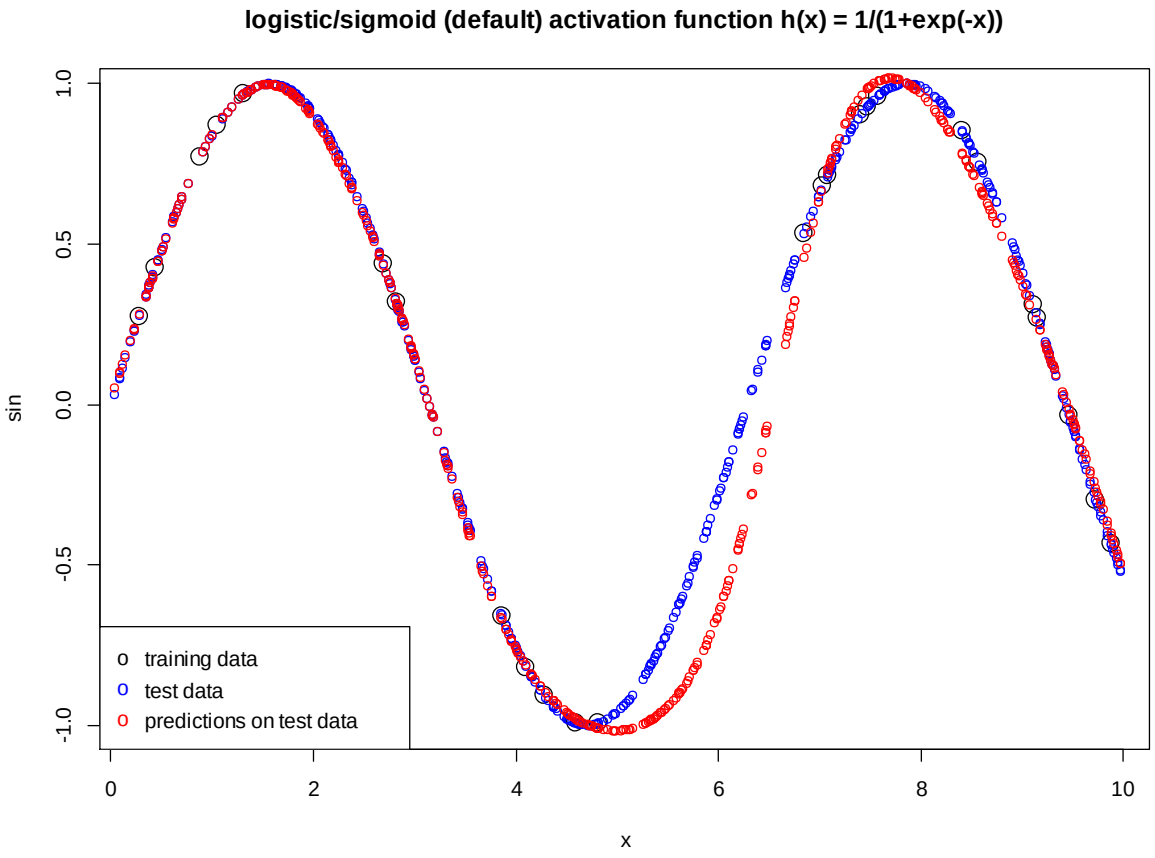
```
> SVMmodelPredictions
 [1] -1.998999  1.560584  1.000278 -1.756815 -2.669577  1.291312 -1.068444 -1.312493  1.000184 -2.208639
```

And as we can see, they are identical, meaning that the manual implementation of the SVM is correct. Note that we are specifying type=”decision”, which is then return us a numerical value rather than a prediction.

Assignment 3

(1) Train a neural network to learn the trigonometric sine function. To do so, sample 500 points uniformly at random in the interval $[0, 10]$. Apply the sine function to each point. The resulting value pairs are the data points available to you. Use 25 of the 500 points for training and the rest for test. Use **one hidden layer with 10 hidden units**. You do not need to apply early stopping. Plot the training and test data, and the predictions of the learned NN on the test data. You should get good results. Comment your results.

First we generate 25 points of training data and train our neural network using our 25 training points, start weights ~U[0,1], 1 hidden layer with 10 hidden units, and using the default sigmoid activation function. We can then make predictions on our 475 points in the test data, and we can plot the following graph where the black dots are the training data, the blue points are the test data, and the red points are the predictions on test data (predicted sin-value from x-value in test data).

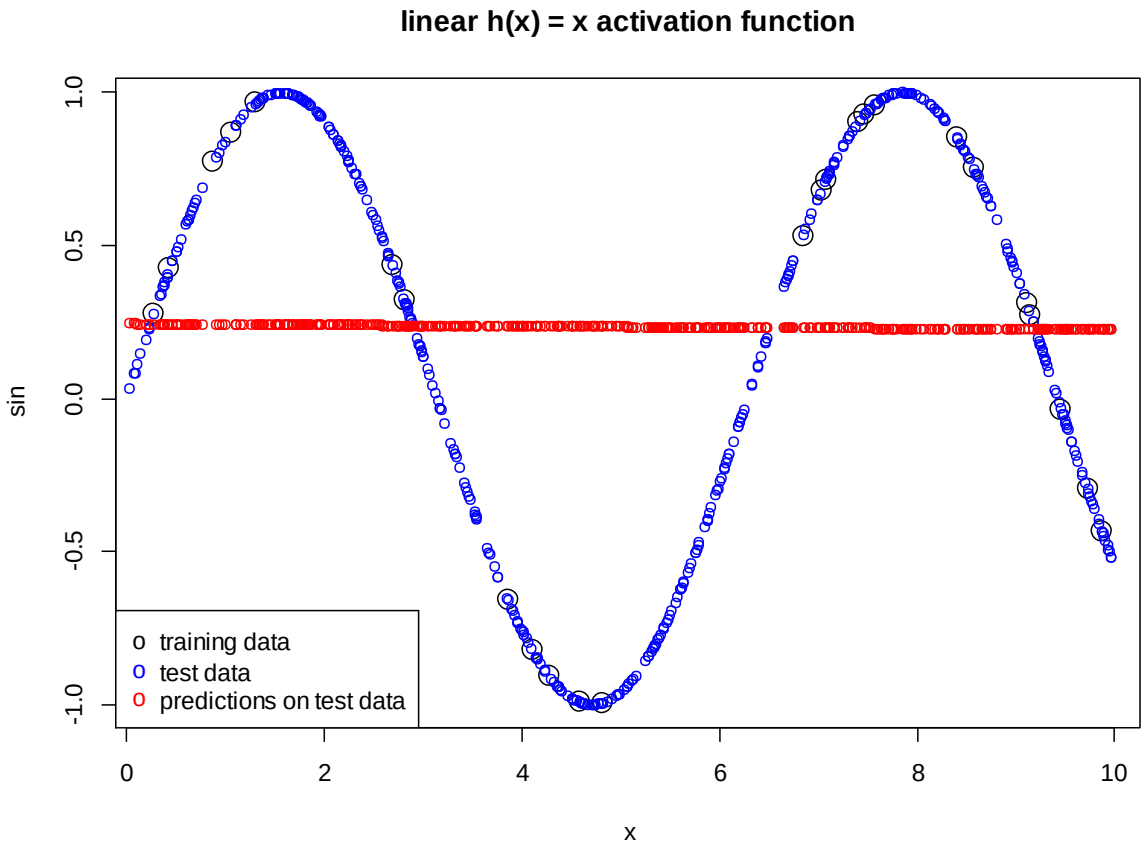


As expected, for the train and test data the points perfectly follow the sin curve. We can see the neural network model predicts sin-values from the x-values (red dots) quite well and the predictions

follow the true sin-values from the x-values (blue dots) closely in most intervals of the plot. However in some intervals the predictions are a bit worse, as can be noticed at the top of the two sin peaks where the true sin-value is around 1, especially the right peak. The model also performs bad in the interval around (5 , 6.5) where you can easily see in the graph that the model performs noticeably worse than in other regions. This is probably due to there being no data points at all in the training data which are between 4.82 and 6.84, and since the model has not been trained on this interval the model has a hard time making predictions in the interval of input variable x. The model seems to perform worse when the sin function goes from increasing to decreasing in value or the opposite, i.e. when d/dx sin(x) changes sign. Both these issues could probably be avoided by adding more training data than simply 25 training points, and by using more hidden layers and/or more hidden units, which would allow us to better train the model on the whole input space.

MSE 0.01388677

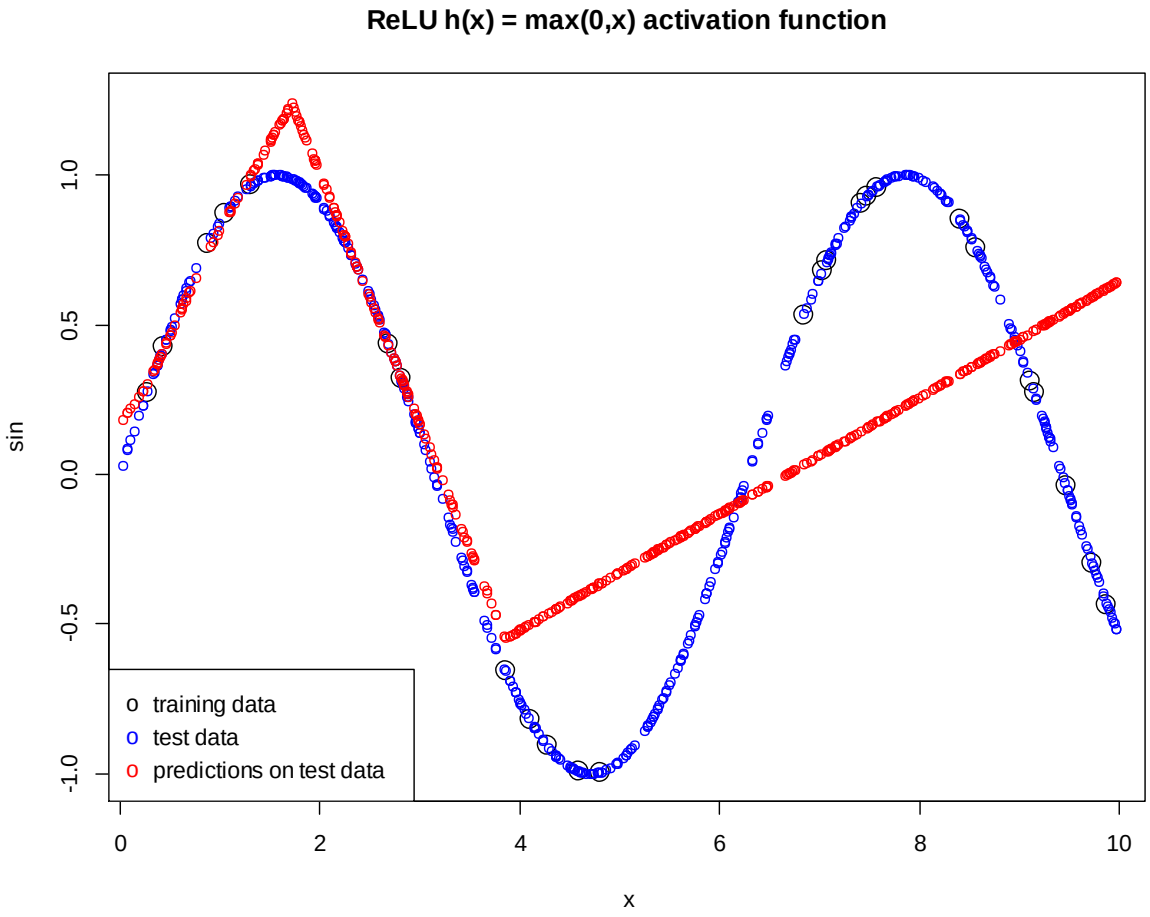
(2) In question (1), you used the default logistic (a.k.a. sigmoid) activation function, i.e. `act.fct = "logistic"`. Repeat question (1) with the following custom activation functions: $h_1(x) = x$, $h_2(x) = \max\{0, x\}$ and $h_3(x) = \ln(1 + \exp x)$ (a.k.a. linear, ReLU and softplus). See the help file of the `neuralnet` package to learn how to use custom activation functions. Plot and comment your results.



As can be seen in the plot, using a linear activation function makes our model have very bad performance. When using a linear activation function the hidden layers have no effect other than adding the corresponding weights to the corresponding input as well as adding the intercept, so the

output from each hidden layer will be the input*weight + intercept. Thus the only thing affecting our input are the weights and intercepts, so we get a mapping from input to output only depending on weights and intercepts that is similar to linear regression in a way, so we get the predictions in a straight line as in the plot. If we removed the hidden layer and summed up our weights and intercept, we could make a linear regression model that does the exact same as above model. As the neural network minimizes the error function (SSE by default when using the nerualnet package) using an activation function like this, the prediction line seems to predict closely to the average sin-value of all the test points. This is why we get the predictions on a line like this around the average sin-value of the true values.

MSE 0.4467372

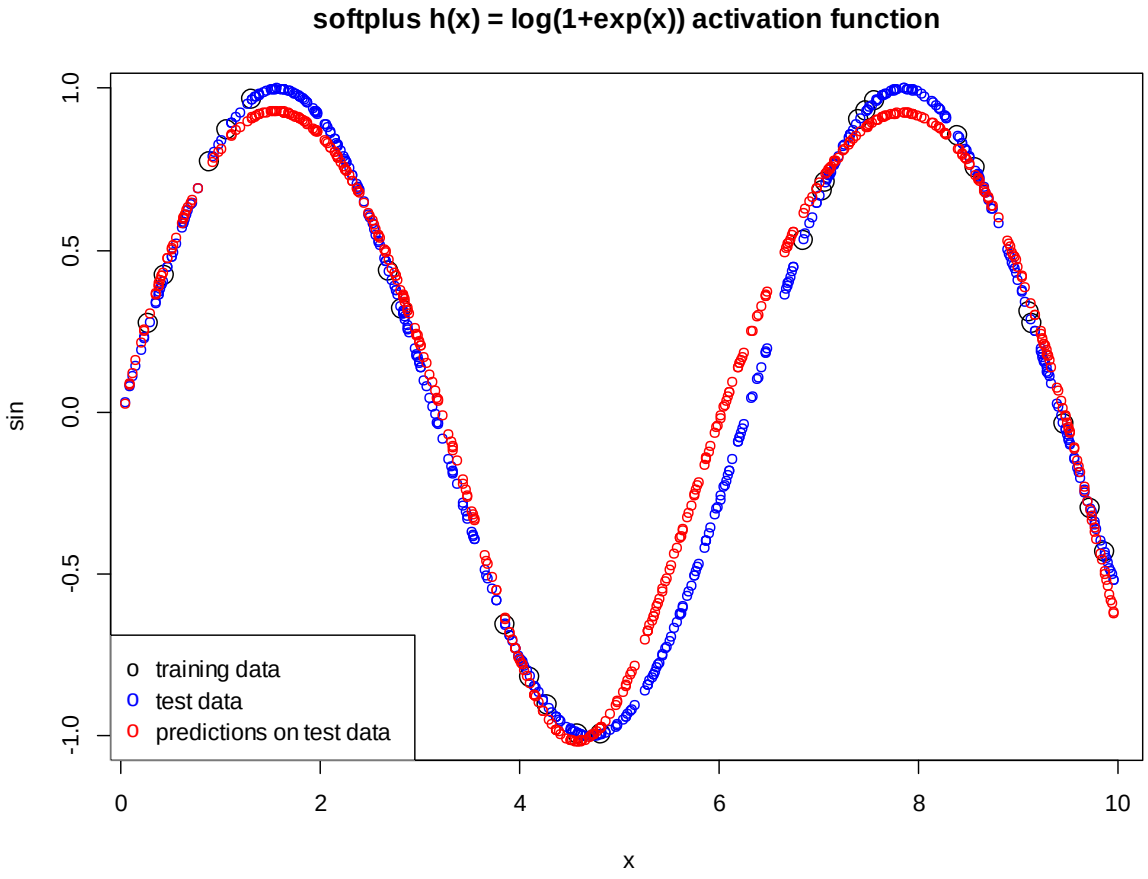


Using the ReLU activation function here also results in a model making bad predictions, but still better than using the linear activation function. Since ReLU outputs 0 or x some of the hidden units will output 0, compared to the linear activation function which always output x, now instead of getting the predictions on a straight line we get these segments of lines which result in better predictions. This is due to the fact that ReLU is a piece wise linear function. But the overall performance is still bad, especially in the interval (4,10) where its prediction seem to be an average of the true values in that interval, similar to the how the linear activation function acted across all data points.

If we had a larger NN, the ReLU activation function would perform much better. Here we are only using one layer with 10 hidden units, which is not enough since due to the fact that ReLU is piece wise linear, we do not introduce much non linearity using only one hidden layer, since the whole hidden layer will just be different linear combinations of the inputs depending on what units will be "on or off", i.e. will be 0 or x. If we would have more hidden layers (and more hidden units), although ReLU for each individual hidden unit and is piece wise linear, we would with multiple layers get a more nonlinear behaviour which could better capture the underlying data distribution by this increase of model complexity and flexibility. We also know that NN are universal approximators, so given enough complexity of the model (hidden layers and hidden units), as well as data, we will get very good performance with the ReLU activation fucntion.

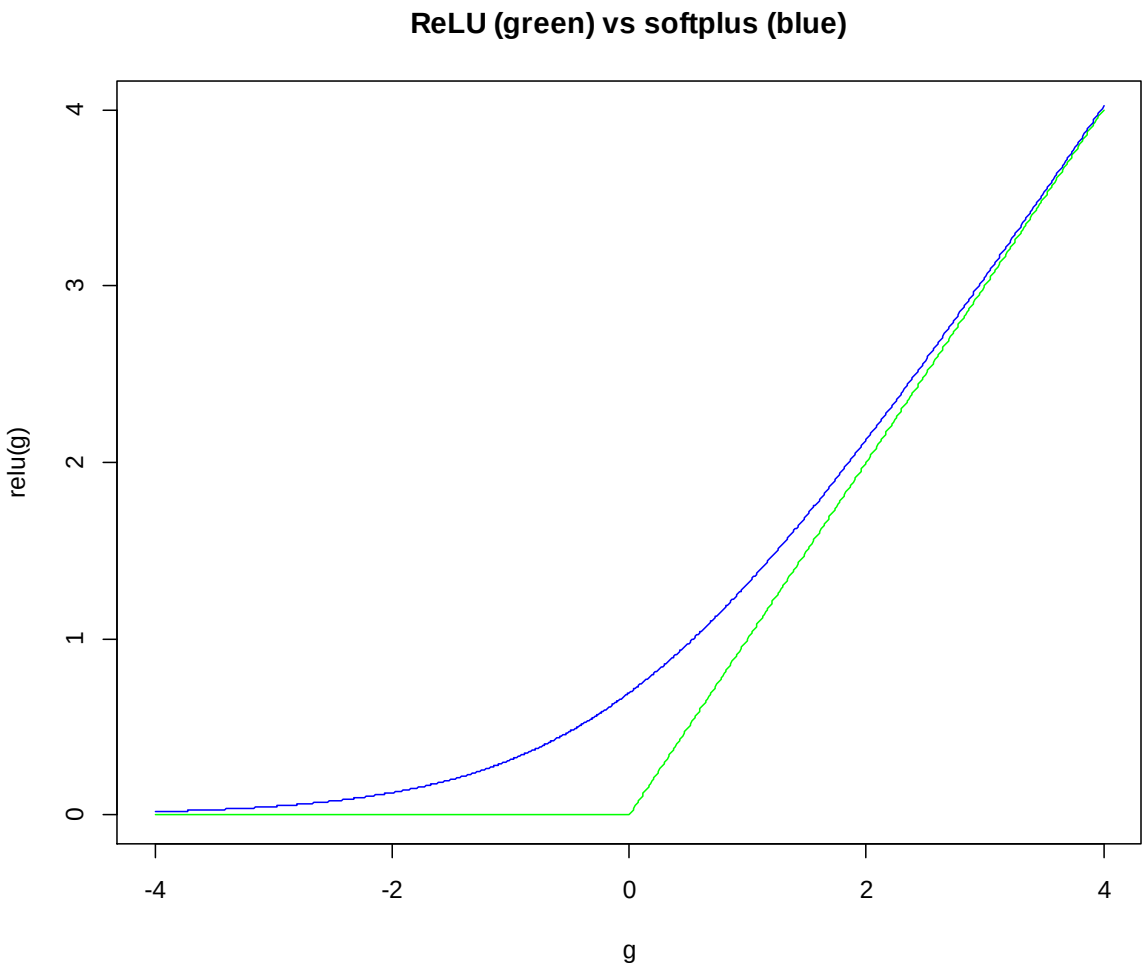
One thing worth mentioning about the ReLU function is that it is not differentiable for x=0, since the derivative at x=0 is undefined.

MSE 0.1388677



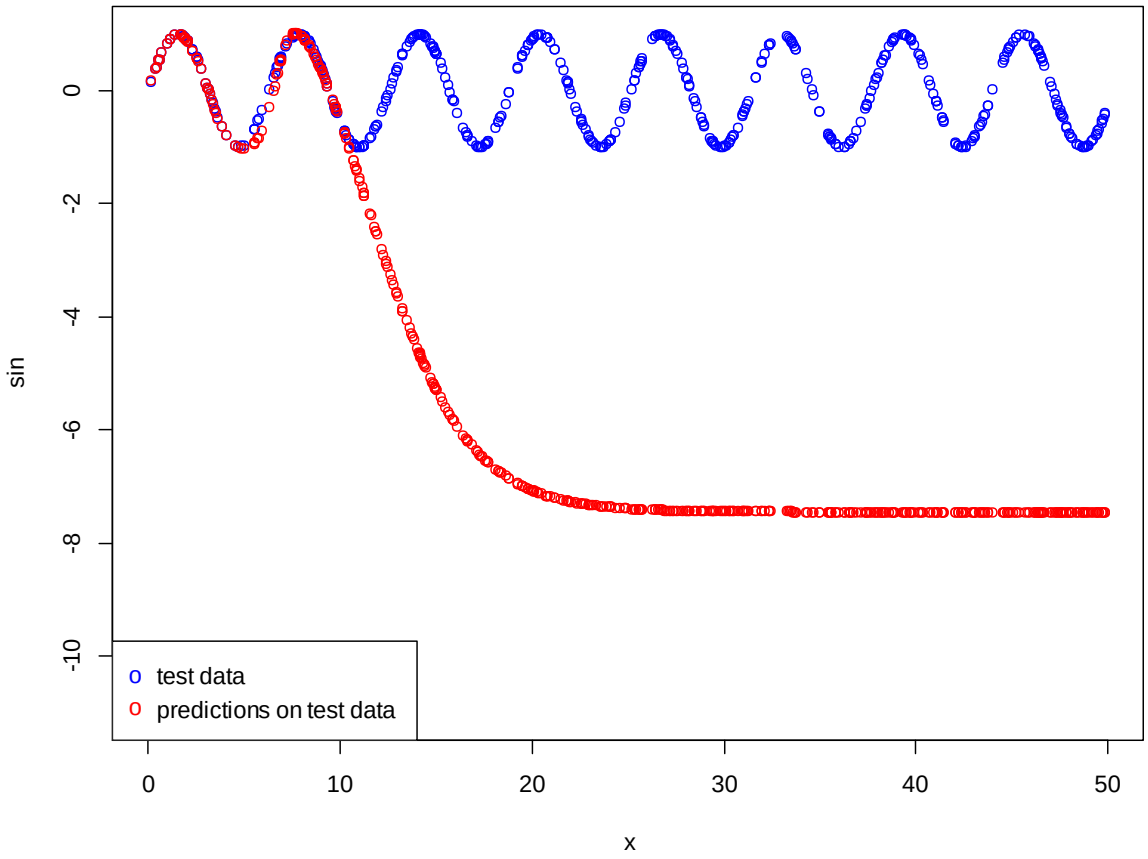
The softplus activation function has as similar shape to the ReLU function but it is "softer" since it is a differentiable function, i.e. it does not go from outputting 0 to being linear at x = 0 but curves at x = 0. Because of this we already using only 1 layer introduce more nonlinearity and thus we produce much better predictions, and a performance which seems to be even better than the default sigmoid activation function, since this model makes better predictions in the interval 5-6.5, although it makes worse predictions around x=8 and x=2. Comparing MSE, this model has a MSE of 0.009255613compared to 0.01388677 for the model using the sigmoid activation function, so this indicates that this is the best model we have seen so far.

MSE 0.009255613



(3) Sample 500 points uniformly at random in the interval $[0, \mathbf{50}]$, and apply the sine function to each point. Use the NN learned in question (1) to predict the sine function value for these new 500 points. You should get mixed results. Plot and comment your results.

If we sample 500 new points but now uniformly distributed as U(0,50), and use our neural network using the sigmoid activation function (first NN-model) to make predictions, we get the below plot.



In the interval (0,10), our model will perform just as well as previously. However, similar to what was discussed in part 1, since our model is so simple (1 hidden layer, 10 hidden units) and due to the fact that all our training points are in the interval (0,10), we will get very bad performance on our predictions if we try to make predictions on an interval the model is not trained on. In general, predicting in a range that you have not trained on is difficult. The NN is only able to predict well in between points, i.e. interpolate, but in a range outside of the training data of the model like above, i.e. extrapolation, the model does not predict well, since we have not shown any data in the interval (10,50), so the NN has a difficult time predicting here. There is no reason for the NN to assume that the sine will continue as it should, extrapolation outside of the range the NN has been trained on is in principle impossible, especially with a simple model like this. So since the model is only trained on the interval (0,10), and the simple model has not learned the oscillating behaviour of the sin function, the model predicts that the predictions will continue in a downwards trend for x-values larger than 10 like how they have for x-values in the interval (8,10).

(4) In question (3), the predictions seem to converge to some value. Explain why this happens. To answer this question, you may need to get access to the weights of the NN learned. You can do it by running nn or $nn\$weights$ where nn is the NN learned.

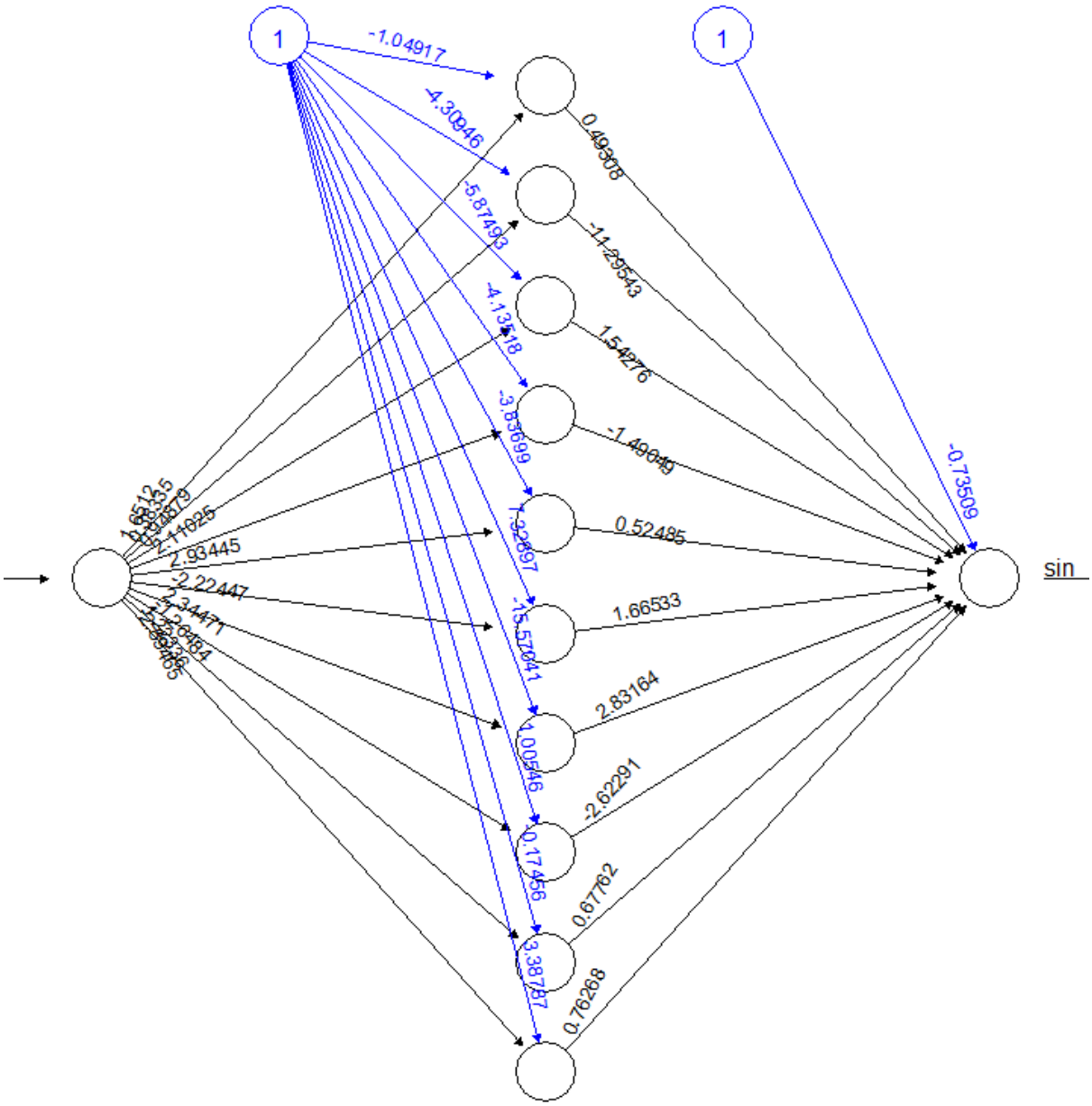
If we sort our predictions, we will see that our data points converge to ~ -7.451

This has to do with how the sigmoid function used as activation fucntion behaves. The sigmoid activation function h(x) = 1 / (1 + exp(-x)) is S-shaped and outputs a value between 0 and 1.

For large values of x as input into the sigmoid function, already at x-values around 6, we get an output value very close to 0 (0.0025), and for negative x-values, already around -6 we get an output very close to 1 (0.9975). Thus, if the absolute value of x is large, the output from the hidden units using the sigmoid activation function will be very close to 0 or 1, and even though all of our x-values are positive, since we can have negative weights we may have inputs to the hidden units that are both negative and positive.

So in this case, where we have large x-values, h(x) will output a value very close to 0 or 1. Since the sigmoid outputs 0 or 1 for large x-values, at some point it will saturate and no matter how much we increase x, we will get the same outputs that are 0 or 1. Thus we will converge to some value as x increase, since the outputs from the hidden units will remain 0 or 1, and the value of the linear combination from the hidden units + intercept that is the outputted sin-value does not change.

If we extract our weights and intercepts that can be seen in the below plot from our neural network, we can test and see what happens for increasing values of x.



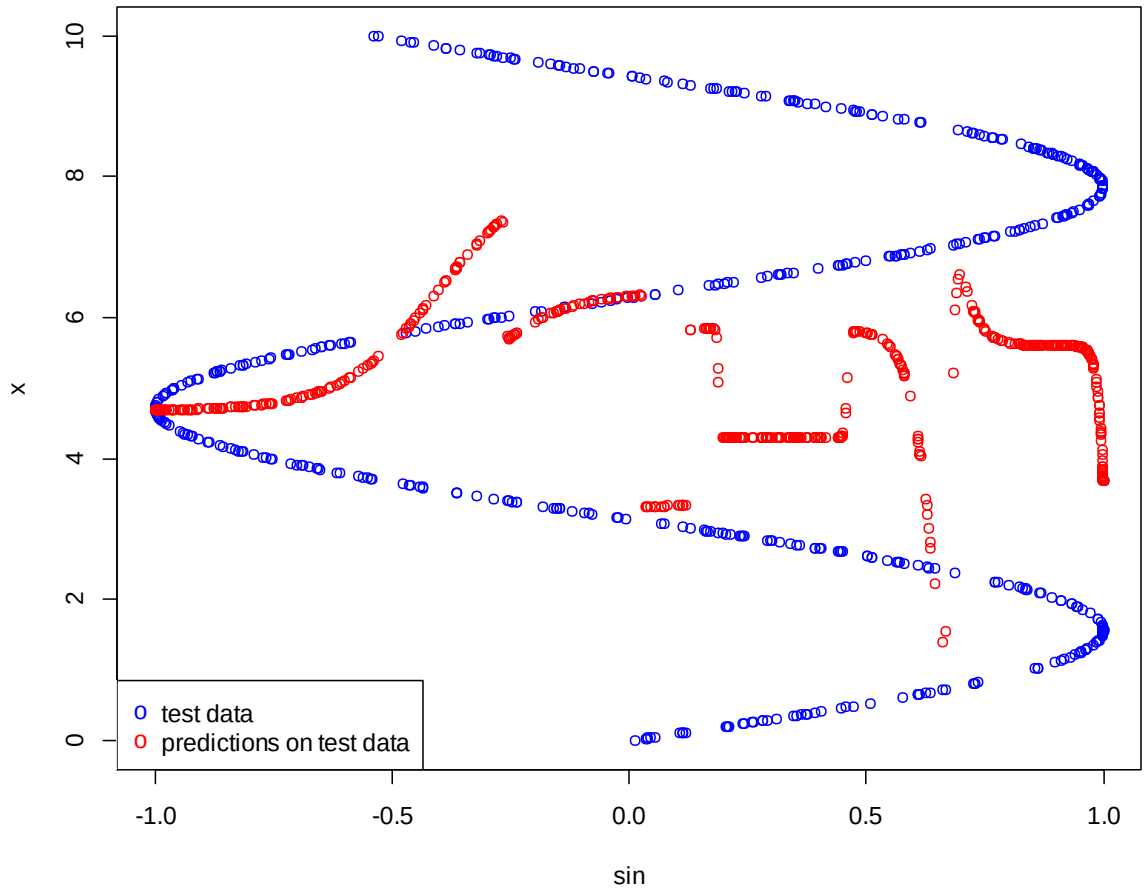
The converge()-function prints the output from the hidden units, and the final output (sin-value) from the model. See below image for results for x = 20, 30, 40, 50, 100, 200

```
> converge(20)
     [,1]      [,2]     [,3] [,4] [,5]         [,6] [,7]         [,8] [,9]         [,10]
[1,]    1 0.9663529 0.999998    1    1 7.267756e-17    1 2.820846e-11    1 2.131898e-24
          [,1]
[1,] -7.070999
> converge(30)
     [,1]      [,2] [,3] [,4] [,5]         [,6] [,7]         [,8] [,9]         [,10]
[1,]    1 0.9992473    1    1    1 1.587211e-26    1 9.062244e-17    1 5.721143e-37
          [,1]
[1,] -7.442553
> converge(40)
     [,1]      [,2] [,3] [,4] [,5]         [,6] [,7]         [,8] [,9]         [,10]
[1,]    1 0.9999837    1    1    1 3.466324e-36    1 2.911335e-22    1 1.535321e-49
          [,1]
[1,] -7.450871
> converge(50)
     [,1]      [,2] [,3] [,4] [,5]         [,6] [,7]         [,8] [,9]         [,10]
[1,]    1 0.9999996    1    1    1 7.570133e-46    1 9.352948e-28    1 4.120172e-62
          [,1]
[1,] -7.451051
> converge(100)
     [,1] [,2] [,3] [,4] [,5]         [,6] [,7]         [,8] [,9]         [,10]
[1,]    1    1    1    1    1 3.760759e-94    1 3.200593e-55    1 5.734515e-125
          [,1]
[1,] -7.451055
> converge(200)
     [,1] [,2] [,3] [,4] [,5]          [,6] [,7]          [,8] [,9]         [,10]
[1,]    1    1    1    1    1 9.281527e-191    1 3.747954e-110    1 1.11086e-250
          [,1]
[1,] -7.451055
```

When we send our input through the sigmoid function, we multiply by some weight and add intercept, which will then pass through the sigmoid and the sigmoid gets saturated (value very close to 0 or 1) quick for large inputs of x. Then, after the hidden layer, we just have a linear combination of these outputs from the hidden units which will be 0's or 1's with their corresponding weight + intercept, and this will be the value we converge towards. As we can see, for x=40 we get a value close to the value we could see in the plot (-7.451). Already for x=50, all the outputs from the hidden units are very close to 1 or 0. As we increase x, this output will also eventually become very close to 0, which we can see happens for x=200. Thus, as we increase x further past x=200 we will see very little change to the predicted sin-value, and by testing some even larger x-values it seems that the predicted sin-value is converging to –7.451055 as we increase the x-value which is in line with what we discussed above. This value can also be calculated by observing which hidden unit outputs, which will be 1 or 0 for large values, and then calculate the sin value using the hidden layer output with the extracted weights and intercept for the second layer.

(5) Sample 500 points uniformly at random in the interval $[0, 10]$, and apply the sine function to each point. Use all these points as training points for learning a NN that tries to predict x **from** $sin(x)$, i.e. unlike before when the goal was to predict $sin(x)$ from x. Use the learned NN to predict the **training data**. You should get bad results. Plot and comment your results. **Help**: Some people get a convergence error in this question. It can be solved by stopping the training before reaching convergence by setting `threshold = 0.1`.

If we now train a new model and try to predict the x-value from the sin-value and instead plot the independent variable (sin) on the x-axis and x on the y-axis, we get the below plot.



We can see that we get very bad predictions when the model is trying to learn to predict target x from feature sin, even though we are using 500 training points now, and we are trying to predict data we have previously seen (the training data). Since we now have a 1 to many mapping in our training data, where our sin-values correspond to 2-4 different x-values, the model can not learn this and we get some type of mean value between these x-values instead. Another way to view this is that we are trying to learn arcsin here, but arcsin is not an injective function outside of its defined interval [-pi/2, pi/2], due to the fact outside of the range arcsin is defined on for one value of x we have several values of y. When trained the model is trying to minimize the error and thus we get this type of predictions that are the mean over points in an interval of the plot. This becomes quite

apparent if we divide the graph into 3 intervals as below, and we can see that for each of these intervals the predicted values are around the mean value of true x-values in these intervals.