Stat 666 Term Project: Relative Predictive Performance for Multivariate Regression and Artificial Neural Networks

Josh Christensen

April 17, 2023

1 Introduction

In this project we seek to understand and compare the robustness of predictions from multivariate regression and neural networks. We explore this phenomenon in two ways. The first is through violation of the assumption of constant variance that is part of the typical multivariate regression model. The second is through the introduction of non-linearity in approximately linear relationships. We perform two simulation studies to explore these ideas and then apply both methods to a real data set of imported cars. This data set exhibits changes in the covariance structure of the response at different levels of the covariates.

We investigate this research question for two main reasons. The first is that linear models form the backbone of many statistical models and are often the default against which other models are compared. This makes their robustness to assumption violations an interesting topic since the use of these models is often the first choice for analysts due to their simplicity, computational efficiency and interpretability.

The second reason for this project relates to neural networks. We use neural networks as the comparison case in this project because they have become commonplace in the machine learning community and are frequently touted as improved alternatives to the linear model due to their ability to automatically incorporate non-linear effects, interactions and other complexities that must be explicitly included in a linear model. While practitioners recognize that no model is perfect, neural networks are sometimes presented as a panacea for problems where prediction is the primary concern. Due to the unique positions that linear regression and neural networks occupy in the landscape of quantitative predictions we hope this project will be able to explore some of the merits of each method and provide insight on which to choose for a given analysis.

We acknowledge that a full comparison between these two models would have to be far more detailed than the prediction comparisons we share here. Indeed the interpretability and uncertainty quantification that come with multivariate regression are often reason enough to forego the black box of a neural network. On the other hand, many forms of unstructured data would effectively preclude the use of linear regression. Since these considerations are on a qualitative level we will not address them directly in this project. Instead, we will focus on providing quantitative comparisons of predictive performance to inform analysts in cases where point prediction is the primary goal.

The rest of the project proceeds as follows. First, we include an introduction to the data as well as brief explanations of multivariate regression and simple neural networks. We then describe and present results from our two simulation studies. Next we apply each method to our chosen data set. Finally, we will provide a brief discussion on the practical insights gained from the project and conclude.

1.1 Data Introduction

The data set to which we will apply the two methods comes from the University of California Irvine (UCI) Machine Learning Repository (Dua and Graff, 2019). It contains 26 variables related to the style and function of 205 different cars produced around the world. We eliminated a variable related to insurance company losses for each car due to high levels of missing values (41 of 205). We then removed all rows with missing values for any variable which left us with 203 observations of 25 variables. We used city miles per gallon (MPG), highway MPG and horsepower as our trivariate response vector. We are interested in finding

relationships to these response variables since they are among the most important functional measures of a car's performance.

2 Multivariate Multiple Regression

Multivariate multiple linear regression is a generalized case of the more common multiple linear regression. In multiple linear regression we have one response variable and multiple explanatory variables. In the case of multivariate regression we have multiple response variables which we wish to predict using one or more explanatory variables. Multivariate regression includes a host of related methods for quantifying uncertainty in parameter estimates and predictions. We will focus only on how to use multivariate regression to produce point estimates for the prediction of new data points.

The most common univariate multiple regression model can be described succinctly as

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$$

where $\boldsymbol{X}$ represents the matrix of covariates, $\boldsymbol{y}$ represents the response vector, $\boldsymbol{\beta}$ represents the regression coefficients and $\boldsymbol{\epsilon} \sim \mathcal{N}_n(0, \sigma^2 I_n)$. To predict new values of the response from this model we need only the values for our explanatory variables in the $\boldsymbol{X}$ matrix and an estimate for $\boldsymbol{\beta}$ since

$$\begin{aligned} E(y_i) &= E(\boldsymbol{x}_i\boldsymbol{\beta} + \epsilon_i) \\ &= \boldsymbol{x}_i\boldsymbol{\beta} + E(\epsilon_i) \\ &= \boldsymbol{x}_i\boldsymbol{\beta}. \end{aligned}$$

The most common estimator used for $\boldsymbol{\beta}$ is

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{y}$$

which can be obtained via minimization of the squared error loss function (generally) or via maximum likelihood estimation (using the assumption of normality). The Gauss-Markov theorem indicates that for this model $\hat{\boldsymbol{\beta}}$ is the linear unbiased estimator with minimum variance. Derivations of this estimator can be found in a majority of textbooks dealing with introductory statistics and linear models, and we recommend Rencher and Schaalje (2008) to the interested reader. Given the results above we then make predictions for a new observation using $\hat{y} = \boldsymbol{x}\hat{\boldsymbol{\beta}}$.

The estimator for the multivariate regression is closely related to $\hat{\boldsymbol{\beta}}$. For the multivariate regression model we have

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{B} + \boldsymbol{\Xi}$$

where $\boldsymbol{Y}$ represents the matrix of responses, $\boldsymbol{X}$ represents the matrix of covariates, $\boldsymbol{B}$ represents the matrix of regression coefficients and $\boldsymbol{\Xi}$ is the matrix of error terms. We assume that the error terms for any given response vector (values of all response variables for a single observation) are uncorrelated with all others, i.e., $\text{cov}(\boldsymbol{y}_i, \boldsymbol{y}_j) = \boldsymbol{0}$. We also assume a common covariance matrix for the response variables across observations, meaning that for all $i \in \{1, \ldots, n\}$ we have $\text{cov}(\boldsymbol{y}_i, \boldsymbol{y}_i) = \boldsymbol{\Sigma}$. Finally we once again assume that the errors have mean zero. This means that we once again only need covariates and an estimate for $\boldsymbol{B}$ in order to make predictions. We will use the least-squares estimator

$$\hat{\boldsymbol{B}} = (\boldsymbol{X}'\boldsymbol{X})^{-1}\boldsymbol{X}'\boldsymbol{Y}$$

for our analysis. This estimator has many properties closely related to those of $\hat{\boldsymbol{\beta}}$ but their details are more complex. For details on why this estimator is the default choice, how it is derived and some its properties we recommend chapter 10 of Rencher and Christensen (2012). Once we have this estimator we generate predictions using $\hat{\boldsymbol{y}} = \boldsymbol{x}\hat{\boldsymbol{B}}$.

3 Neural Networks

We now turn our attention to neural networks. Neural networks have received a great deal of attention from quantitative researchers in recent years and consequently have seen a host of innovations in their estimation,

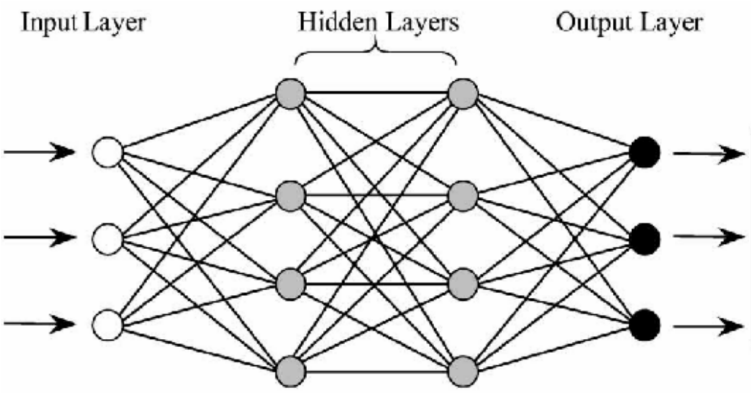

Figure 1: Modified figure 3.7 from Lesschen et al. (2005).

architecture and applications. We will focus only on feedforward neural networks for simplicity and because for simple regression problems more complex architectures are usually not needed. Feedforward neural networks are composed of layers of nodes, each of which passes information to nodes in the subsequent layer. Each neural network begins with an input layer composed of the information we are using to predict the outcome, and ends with an output layer that consists of the networks predictions of the outcomes. A basic architecture can be seen in fig. 1.

Each hidden node in a neural network can be thought of as its own multiple regression in which the explanatory variables are the outputs from preceding nodes and the response is the information passed to future nodes. The output is then modified by a secondary function, the activation function, which scales the output in some non-linear way before relaying it to future nodes. In our case we will use the rectified linear activation function which is a function that moves all negative values to zero and does not change positive values. The activation function will allow us to capture non-linear effects in our response. The scaling values for the input values of each node are called weights and the constant added to the scaled inputs is called the bias.

In supervised regression settings like the ones considered in this project neural networks are trained by evaluating predicted output from the network against known values from a training data set. A loss function is used to quantify how far off the predictions are from the true values. The loss function is then minimized via an optimization routine. In most cases this consists of backpropagation and some form of stochastic gradient descent. Predictions are obtained by giving the network new input values and then feeding those values through the network using the weights and biases found through training to produce predictions from the output nodes.

4 Simulation Studies

The first simulation study was designed to test the robustness of the two models to heteroskedasticity in a multivariate response. In each iteration of each run of the simulation we created a data set with ten covariates and three response variables. The responses consisted of two subpopulations with different correlation structures that were treated as a single population. Marginal variances were kept constant across the two subpopulations and across all runs. The covariate values were randomly generated from a standard normal distributions. The beta coefficients were $\boldsymbol{\beta}_1 = (1, 2, \ldots, 10), \boldsymbol{\beta}_2 = (.1, .2, \ldots, 1)$ and $\boldsymbol{\beta}_3 = (-.5, -.4, \ldots, 0, .1, \ldots, .4)$ which correspond to the three variables in the trivariate response respectively. The marginal variances for the three response variables were 1, 100 and 25 respectively. For each run we then selected two different sets of correlations to correspond to the two subpopulations. The first run was a control run where correlations for both subpopulations were set to 0, which generated homoskedastic data sets with no covariance between responses. For the second run, one subpopulation had no correlation while the other had $\boldsymbol{\rho} = (0.9, -0.1, 0.3)$ which represents the correlation between variables one and two, one and three and two and three respectively. For the last run the two subpopulations had correlation structures of

$\boldsymbol{\rho}_1 = (0.8, -0.1, -0.5)$ and $\boldsymbol{\rho}_2 = (-0.8, 0.1, 0.5) = -\boldsymbol{\rho}_1$. For each parameter combination 100 data sets of 100 points were generated. For each data set of 100 points we then fit both models to 80 data points and evaluated mean absolute error on the remaining 20 data points. The results from this simulation study are shown in table 1. Note that the two methods performed comparably well in all three runs of simulation one.

	Run 1	Run 2	Run 3
Multivariate Regression	4.282	4.275	4.271
Neural Network	4.256	4.256	4.249

Table 1: Mean absolute error results from the first simulation study.

Our original hypothesis was that the multivariate regression predictions would be biased by the different behavior in the distinct subpopulations. However, the point predictions appear to be robust to violation of this assumption, provided that the underlying mean function is linear in nature. For the neural networks in this simulation we used one hidden layer with two nodes. This neural network very close to a linear regression due to the small number of hidden nodes, and was chosen from many different architectures by evaluating performance on training and validation sets from the control example. This makes sense because the underlying function was linear and the fewer hidden nodes and hidden layers we have the closer the system of the network is to being linear.

Our second simulation introduced nonlinearity and sought to explore the competing ideas of a multivariate regression being unable to fit obvious nonlinear trends, while a neural network may be prone to overfitting noise. In this simulation we had one covariate, whose values were uniformly generated between 0 and 10. Each data set once again contained 100 data points and we generated 100 data sets per simulation. We once again fit 80 points and used 20 points as a test set to evaluate mean absolute error. Every β coefficient was equal to 2 to create a clear trend and a term equal to $\sin(8x/5)$ was added to the mean function to create a cyclical wave around the trend. We then added Gaussian noise to the 4-variate response with a constant marginal variance of σ^2 and constant correlation equal to ρ. An example of one dimension of a generated response can be seen in fig. 2. A table of the average mean absolute errors from the 100 runs at

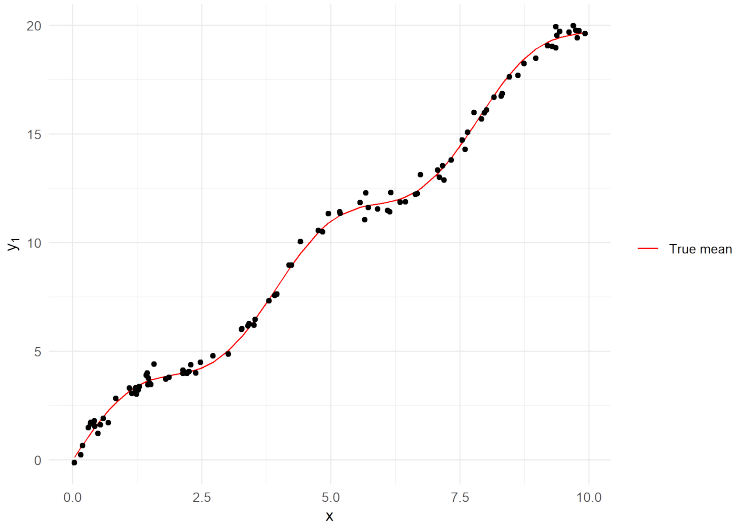

Figure 2: Demonstration of the data generating process for simulation two.

each parameter combination can be found in table 2.

First we note that it appears that neither method was affected by the correlation between the response variables. This is once again consistent with our observation from simulation one that the form the underlying mean function is far more important to the performance of our model predictions than the correlation structure. We also observe that at low observation variances where the nonlinear trends can be easily picked out, the neural network was able to outperform linear regression by fitting the sinusoidal function. However,

	σ^2							
	Multivariate Regression				Neural Network			
	0.1	1	3	8	0.1	1	3	8
$\rho = 0.1$	0.654	0.992	1.499	2.372	0.554	0.964	1.500	2.387
$\rho = 0.3$	0.658	0.985	1.501	2.333	0.545	0.948	1.505	2.366
$\rho = 0.5$	0.654	0.994	1.501	2.362	0.563	0.965	1.500	2.405
$\rho = 0.7$	0.658	1.009	1.535	2.410	0.545	0.964	1.547	2.428
$\rho = 0.9$	0.645	0.979	1.570	2.310	0.548	0.946	1.566	2.360

Table 2: Mean absolute error results from the second simulation study.

as noise increased and the underlying linear trend became the only distinguishable feature of the function, multivariate regression performed as well or better than the neural networks. At even higher levels of noise we may start to see the linear regression outperform the neural networks if we do not carefully retune the architecture and training of the neural network to avoid overfitting. For this simulation we used two hidden layers with 20 nodes each.

5 Results from the Car Data

In this section we briefly summarize the results of our application of both methods to a real data set. This exercise revealed many practical considerations relevant to the two methods. Originally we wished to compare predictions using a train data set of consisting of 80% of the data and a test set of 20% of the data. We decided on a neural network architecture of two hidden layers with 40 nodes each after comparing training and validation set performance for many different architectures. This highlights one of the practical considerations of the neural network: there are many tuning parameters and if tuning is not performed predictions may be unreliable. In addition to the architecture of the network one must also decide on an optimization algorithm, learning rate, number of training passes, etc. While these decisions can be informed by quantitative metrics many of them are ultimately left to the discretion of the modeler. We obtained a mean absolute error of 3.86 across all response values in the test set.

We then attempted to apply the same methodology to the multivariate regression, but encountered a problem where some factor values were present in the test set, but not the training set. Predictions from multivariate regression require estimates for all relevant beta values. With no data for some values we could not predict. Where the neural net could simply train nodes not to fire if no information was given, the multivariate regression failed to produce results. This could be a blessing and a curse as we can still produce predictions, but those predictions may not represent the information we think they do.

We ultimately changed our train/test split to 90% and 10% for the multivariate regression and sampled the training set such that all levels of the factors were present in the training set. We then obtained a mean absolute error of 35.57. Notice that this is an order of magnitude larger than the mean absolute error of the neural network. Based on our simulation results we expect this is a result of nonlinear relationships among the variables being detected by the neural network, but overlooked by the linear model. With the large number of variables this is a very likely explanation. Based on the simulation studies the changing correlation structure across levels of the covariates for which the data set was originally selected is likely less of a factor in the predictive difference observed.

6 Discussion

Based on the quantitative results from our simulations and our cars data set we now provide some interpretation of the results and some practical suggestions.

In comparing the predictive power of neural networks to multivariate regression for multivariate responses we discovered that the correlation structure of the response variables played little role in the models ability to produce accurate point predictions. Indeed it appears that a choice by a modeler to completely ignore underlying correlation among responses will have little to no effect on point estimate accuracy.

Since the multivariate regression is robust to violations of this assumption it is an excellent choice for any predictive problem in which it is reasonable to assume that the relationship between explanatory and response variables is linear in nature. This would be easy to investigate in smaller dimensional systems, but becomes difficult once we reach the number of variables we see in the cars data. If this assumption is reasonable one can greatly reduce the programming, computation and modeling time required compared to the neural network for predictions of similar accuracy.

On the other hand, if there is doubt as to the linearity of the underlying system and it would be difficult to produce informed hypotheses about the true nature of the underlying relationships we suggest the use of a neural network. Neural networks are much more involved to program, take more time to estimate and take time and care to tune. However, their extreme flexibility makes them an appealing choice in cases where it would be difficult or impossible to investigate the relationships between all of the variables at play. This is evident by the neural network's outperformance on the cars data where we had a much higher dimensional problem than our simulations and where the underlying relationships may not have been even approximately linear.

7 Conclusion

In this project we compared predictive capabilities of multivariate regression and neural networks for scenarios that included non-homogeneous covariance structures in the response variables and differing levels of apparent non-linearity in a multivariate response. We found that point predictions from both methods are robust to heteroskedasticity. The neural network outperforms the linear model when non-linear trends are clear, but as noise increases the neural network's advantage disappears and the predictions from the two methods become comparable.

In future work we hope to expand our second simulation study to a wider variety of non-linear settings and observe whether extremely high levels of noise cause the neural network to overfit the data. Also of future interest is a sensitivity analysis for the various tuning parameters of the neural network, including learning rate, optimization routine, number of hidden layers, number of hidden nodes, etc. The selection of these values is non-trivial and can result in drastic differences in a network's predictive performance.

References

Dua, D. and Graff, C. (2019). UCI machine learning repository.

Lesschen, J. P., Verburg, P., and Staal, S. (2005). Statistical methods for analysing the spatial dimension of changes in land use and farming systems.

Rencher, A. C. and Christensen, W. F. (2012). *Methods of multivariate analysis*. Wiley, Hoboken, New Jersey, third edition. edition.

Rencher, A. C. and Schaalje, G. B. (2008). *Linear models in statistics*. John Wiley & Sons.

Appendix: Analysis Code

Simulation 1 R code (multivariate regression)

```r
# Simulate data frame
run_simulation <- function(n_sim = 100, sigmas) {
  # Extract sigmas
  sigma_1 <- sigmas[[1]]
  sigma_2 <- sigmas[[2]]

  # Set sample size for each simulated dataset
  n <- 100

  # set the betas for the simulation
  beta <- matrix(c(1:10,
                   seq(0.1,1,by = 0.1),
                   seq(-0.5, 0.4, by = 0.1)),
                 nrow = 10, ncol = 3)

  # Create storage
  linear_model_mae <- numeric(n_sim)

  # Storage for the datasets so they can be fit with the neural net later
  generated_datasets <- vector("list", n_sim)

  for(i in 1:n_sim) {
    # Generate the covariate matrices
    x_1 <- matrix(rnorm(10 * n),
                  nrow = n, ncol = 10)
    x_2 <- matrix(rnorm(10 * n),
                  nrow = n, ncol = 10)

    # Generate the first dataset
    y_1 <- x_1 %*% beta + t(replicate(n, crossprod(chol(sigma_1), rnorm(3)), simplify = "
    matrix"))

    # Generate the second dataset
    y_2 <- x_2 %*% beta + t(replicate(n, crossprod(chol(sigma_2), rnorm(3)), simplify = "
    matrix"))

    # Aggregate
    y <- rbind(y_1, y_2)
    x <- rbind(x_1, x_2)

    # Split train and test
    test_sample_size <- round(n * .2)
    test_index <- sample.int(n * 2, test_sample_size)
    y_train <- y[-test_index,]
    y_test <- y[test_index,]
    x_train <- x[-test_index,]
    x_test <- x[test_index,]

    # Fit the linear model
    betahat <- coef(lm(y_train ~ x_train))

    # Evaluate MAE for predictions
    linear_model_mae[i] <- mean(abs(y_test - cbind(1,x_test) %*% betahat))

    # Store the dataset
    generated_datasets[[i]] <- list(x_train = x_train,
                                    y_train = y_train,
                                    x_test = x_test,
                                    y_test = y_test)
  }

  return(
    list(
```

```
      linear_model_mae = linear_model_mae,
      generated_datasets = generated_datasets
    )
  )
}

make_sigmas <- function(cor1, cor2){
  sigma_1 <- matrix(c(1,cor1[1],cor1[2],
                      cor1[1],1,cor1[3],
                      cor1[2],cor1[3],1), nrow = 3, ncol = 3, byrow = TRUE)

  sigma_2 <- matrix(c(1,cor2[1],cor2[2],
                      cor2[1],1,cor2[3],
                      cor2[2],cor2[3],1), nrow = 3, ncol = 3, byrow = TRUE)

  stdevs <- c(1,10,5)

  scaling_matrix <- matrix(stdevs, nrow = 3, ncol = 3)
  scaling_matrix <- scaling_matrix * t(scaling_matrix)
  scaling_matrix

  sigma_1 <- sigma_1 * scaling_matrix
  sigma_2 <- sigma_2 * scaling_matrix

  return(
    list(
      sigma_1,
      sigma_2
    )
  )
}

control_results <- run_simulation(sigmas = make_sigmas(rep(0,3), rep(0,3)))
sim_results_ind_mixed <- run_simulation(sigmas = make_sigmas(rep(0,3), c(.9,-.1,.3)))
sim_results_swap <- run_simulation(sigmas = make_sigmas(c(.8,-.1,-.5), c(-.8,.1,.5)))

c(
  mean(control_results$linear_model_mae),
  mean(sim_results_ind_mixed$linear_model_mae),
  mean(sim_results_swap$linear_model_mae)
)
```

Simulation 1 Python code (neural networks)

```
# Re run the same simulations that we ran in R with a neural net
# Import packages
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Make NumPy printouts easier to read.
np.set_printoptions(precision = 3, suppress = True)

print(tf.__version__)

control_sets = r.control_results['generated_datasets']
mixed_sets = r.sim_results_ind_mixed['generated_datasets']
swap_sets = r.sim_results_swap['generated_datasets']

# This is where I could begin a function that would take first_set
def fit_nn(generated_set):
  normalizer = tf.keras.layers.Normalization(axis=-1)

```

```
  normalizer.adapt(np.array(generated_set['x_train']))

  model = keras.Sequential([
      normalizer,
      layers.Dense(2, activation='relu'),
      # layers.Dense(2, activation='relu'),
      layers.Dense(3)
  ])

  model.compile(loss='mean_absolute_error',
                optimizer=tf.keras.optimizers.Adam(0.01))

  dnn_model = model

  history = dnn_model.fit(
      generated_set['x_train'],
      generated_set['y_train'],
      # validation_split=0.2,
      verbose=0, epochs = 300)

  return dnn_model.evaluate(generated_set['x_test'], generated_set['y_test'], verbose=0)

fit_nn(control_sets[0])

def nn_sim(dataset_list):
  n_sim = len(dataset_list)
  nn_mae = [None] * n_sim
  for i in range(n_sim):
    nn_mae[i] = fit_nn(dataset_list[i])

  return nn_mae

control_results = nn_sim(control_sets)
mixed_results = nn_sim(mixed_sets)
swap_results = nn_sim(swap_sets)

print([np.mean(control_results), np.mean(mixed_results), np.mean(swap_results)])
```

Simulation 2 R code (multivariate regression)

```
# Simulate data frame
run_simulation <- function(n_sim = 100, rho = 0.5, sigma2 = 1) {
  # Create sigma matrix
  sigma <- (matrix(rho, nrow = 4, ncol = 4) + diag(1 - rho, 4)) * sigma2

  # Set sample size for each simulated dataset
  n <- 100

  # set the betas for the simulation
  beta <- matrix(1,
                 nrow = 1, ncol = 4)

  # Create storage
  linear_model_mae <- numeric(n_sim)

  # Storage for the datasets so they can be fit with the neural net later
  generated_datasets <- vector("list", n_sim)

  for(i in 1:n_sim) {
    # Generate the covariate vector
    x <- runif(n, 0, 10)

    # Generate the y sequences
    f <- function(x) x + sin(x * 1.6)
    cyclical_component <- matrix(rep(f(x),4), nrow = n, ncol = 4)
```

```r
    y <- x %*% beta + cyclical_component + t(replicate(n, crossprod(chol(sigma), rnorm(4)),
    simplify = "matrix"))

    # Split train and test
    test_sample_size <- round(n * .2)
    test_index <- sample.int(n, test_sample_size)
    y_train <- y[-test_index,]
    y_test <- y[test_index,]
    x_train <- x[-test_index]
    x_test <- x[test_index]

    # Fit the linear model
    betahat <- coef(lm(y_train ~ x_train))

    # Evaluate MAE for predictions
    linear_model_mae[i] <- mean(abs(y_test - cbind(1,x_test) %*% betahat))

    # Store the dataset
    generated_datasets[[i]] <- list(x_train = x_train,
                                    y_train = y_train,
                                    x_test = x_test,
                                    y_test = y_test)
  }

  return(
    list(
      linear_model_mae = linear_model_mae,
      generated_datasets = generated_datasets
    )
  )
}

sims <- list(
  sim_1 = run_simulation(rho = 0),
  sim_2 = run_simulation(rho = .3),
  sim_3 = run_simulation(rho = .5),
  sim_4 = run_simulation(rho = .7),
  sim_5 = run_simulation(rho = .9),
  sim_6 = run_simulation(rho = 0, sigma2 = 3),
  sim_7 = run_simulation(rho = .3, sigma2 = 3),
  sim_8 = run_simulation(rho = .5, sigma2 = 3),
  sim_9 = run_simulation(rho = .7, sigma2 = 3),
  sim_10 = run_simulation(rho = .9, sigma2 = 3),
  sim_11 = run_simulation(rho = 0, sigma2 = 0.1),
  sim_12 = run_simulation(rho = .3, sigma2 = 0.1),
  sim_13 = run_simulation(rho = .5, sigma2 = 0.1),
  sim_14 = run_simulation(rho = .7, sigma2 = 0.1),
  sim_15 = run_simulation(rho = .9, sigma2 = 0.1),
  sim_16 = run_simulation(rho = 0, sigma2 = 8),
  sim_17 = run_simulation(rho = .3, sigma2 = 8),
  sim_18 = run_simulation(rho = .5, sigma2 = 8),
  sim_19 = run_simulation(rho = .7, sigma2 = 8),
  sim_20 = run_simulation(rho = .9, sigma2 = 8)
)

unlist(lapply(lapply(sims, \(x) x$linear_model_mae), mean))
```

Simulation 2 Python code (neural networks)

```python
# Re run the same simulations that we ran in R with a neural net
# Import packages
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
```

```
from tensorflow.keras import layers

# Make NumPy printouts easier to read.
np.set_printoptions(precision = 3, suppress = True)

print(tf.__version__)

sims = r.sims

# This is where I could begin a function that would take first_set
def fit_nn(generated_set):
  normalizer = tf.keras.layers.Normalization(input_shape=[1,], axis=None)

  normalizer.adapt(np.array(generated_set['x_train']))

  model = keras.Sequential([
      normalizer,
      layers.Dense(20, activation='relu'),
      layers.Dense(20, activation='relu'),
      layers.Dense(4)
  ])

  model.compile(loss='mean_absolute_error',
                optimizer=tf.keras.optimizers.Adam(0.01))

  dnn_model = model

  history = dnn_model.fit(
      np.array(r.x), # generated_set['x_train']
      np.array(r.y), # generated_set['y_train']
      validation_split=0.2,
      verbose = 0, epochs = 100)

  return dnn_model.evaluate(np.array(r.x), np.array(r.y), verbose=0)

fit_nn(sims['sim_1']['generated_datasets'][0])

def nn_sim(dataset_list):
  n_sim = len(dataset_list)
  nn_mae = [None] * n_sim
  for i in range(n_sim):
    nn_mae[i] = fit_nn(dataset_list[i])

  return nn_mae

results_1 = nn_sim(sims['sim_1']['generated_datasets'])
results_2 = nn_sim(sims['sim_2']['generated_datasets'])
results_3 = nn_sim(sims['sim_3']['generated_datasets'])
results_4 = nn_sim(sims['sim_4']['generated_datasets'])
results_5 = nn_sim(sims['sim_5']['generated_datasets'])
results_6 = nn_sim(sims['sim_6']['generated_datasets'])
results_7 = nn_sim(sims['sim_7']['generated_datasets'])
results_8 = nn_sim(sims['sim_8']['generated_datasets'])
results_9 = nn_sim(sims['sim_9']['generated_datasets'])
results_10 = nn_sim(sims['sim_10']['generated_datasets'])
results_11 = nn_sim(sims['sim_11']['generated_datasets'])
results_12 = nn_sim(sims['sim_12']['generated_datasets'])
results_13 = nn_sim(sims['sim_13']['generated_datasets'])
results_14 = nn_sim(sims['sim_14']['generated_datasets'])
results_15 = nn_sim(sims['sim_15']['generated_datasets'])
results_16 = nn_sim(sims['sim_16']['generated_datasets'])
results_17 = nn_sim(sims['sim_17']['generated_datasets'])
results_18 = nn_sim(sims['sim_18']['generated_datasets'])
results_19 = nn_sim(sims['sim_19']['generated_datasets'])
results_20 = nn_sim(sims['sim_20']['generated_datasets'])

print([np.mean(results_1), np.mean(results_2), np.mean(results_3), np.mean(results_4),
np.mean(results_5), np.mean(results_6), np.mean(results_7), np.mean(results_8),
```

```
np.mean(results_9), np.mean(results_10), np.mean(results_11), np.mean(results_12),
np.mean(results_13), np.mean(results_14), np.mean(results_15), np.mean(results_16),
np.mean(results_17), np.mean(results_18), np.mean(results_19), np.mean(results_20)])
```

Car Data R code (multivariate regression)

```
# Set up the multivariate regression on the real data

# Load necessary libraries
library(dplyr)
library(magrittr)

# Import the data
url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/autos/imports-85.data'
column_names = c('symboling', 'normalized-losses', 'make', 'fuel-type', 'aspiration',
                 'num-of-doors', 'body-style', 'drive-wheels', 'engine-location',
                 'wheel-base', 'length', 'width', 'height', 'curb-weight', 'engine-type',
                 'num-of-cylinders', 'engine-size', 'fuel-system', 'bore', 'stroke',
                 'compression-ratio', 'horsepower', 'peak-rpm', 'city-mpg', 'highway-mpg',
                 'price')

dataset <- read.csv(url, header = FALSE, col.names = column_names, na.strings = "?")

dataset <- dataset[,-2]
ncol(dataset)

sum(is.na(dataset))

dataset <- na.omit(dataset)

dataset %<>% mutate(num.of.cylinders = case_when(num.of.cylinders == "three" ~ 3,
                                                 num.of.cylinders == "four" ~ 4,
                                                 num.of.cylinders == "five" ~ 5,
                                                 num.of.cylinders == "six" ~ 6,
                                                 num.of.cylinders == "eight" ~ 8,
                                                 num.of.cylinders == "twelve" ~ 12))

train_index <- sample.int(nrow(dataset), size = round(nrow(dataset) * .5))
train <- dataset[train_index,]
test <- dataset[-train_index,]

regression_fit <- lm(cbind(city.mpg, highway.mpg, horsepower) ~ ., data = train)

predictions <- predict(regression_fit, test)

mean(abs(as.matrix(predictions[,3] - dataset[c("city.mpg", "highway.mpg", "horsepower")])))
```

Car Data Python code (neural network)

```
# Import packages
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Make NumPy printouts easier to read.
np.set_printoptions(precision = 3, suppress = True)

print(tf.__version__)

url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/autos/imports-85.data'
column_names = ['symboling', 'normalized-losses', 'make', 'fuel-type', 'aspiration',
```

```
                'num-of-doors', 'body-style', 'drive-wheels', 'engine-location',
                'wheel-base', 'length', 'width', 'height', 'curb-weight', 'engine-type',
                'num-of-cylinders', 'engine-size', 'fuel-system', 'bore', 'stroke',
                'compression-ratio', 'horsepower', 'peak-rpm', 'city-mpg', 'highway-mpg',
                'price']

raw_dataset = pd.read_csv(url, names=column_names,
                          na_values='?', comment='\t',
                          sep=',', skipinitialspace=True)

dataset = raw_dataset.copy()
dataset = dataset.drop("normalized-losses", axis = 1)
len(dataset.columns)

dataset.isna().sum()

dataset = dataset.dropna()

dataset.loc[:,'num-of-cylinders'] = dataset['num-of-cylinders'].map({'three': 3, 'four': 4,
    'five': 5, 'six': 6, 'eight': 8, 'twelve':12})

dataset = pd.get_dummies(dataset, columns=['make', 'fuel-type', 'aspiration', 'num-of-doors'
    , 'body-style', 'drive-wheels', 'engine-location', 'engine-type', 'fuel-system'],
prefix='', prefix_sep='')

train_dataset = dataset.sample(frac=0.8, random_state=0)
test_dataset = dataset.drop(train_dataset.index)

train_dataset.describe().transpose()

train_features = train_dataset.drop(['city-mpg', 'highway-mpg', 'horsepower'], axis = 1)
test_features = test_dataset.drop(['city-mpg', 'highway-mpg', 'horsepower'], axis = 1)

train_labels = train_dataset[['city-mpg', 'highway-mpg', 'horsepower']]
test_labels = test_dataset[['city-mpg', 'highway-mpg', 'horsepower']]

train_dataset.describe().transpose()[['mean', 'std']]

normalizer = tf.keras.layers.Normalization(axis=-1)

normalizer.adapt(np.array(train_features))

print(normalizer.mean.numpy())

first = np.array(train_features[:1])

with np.printoptions(precision=2, suppress=True):
  print('First example:', first)
  print()
  print('Normalized:', normalizer(first).numpy())

def plot_loss(history):
  plt.figure()
  plt.plot(history.history['loss'], label='loss')
  plt.plot(history.history['val_loss'], label='val_loss')
  plt.ylim([0, 10])
  plt.xlabel('Epoch')
  plt.ylabel('Error')
  plt.legend()
  plt.grid(True)
  plt.show()
  plt.close()

test_results = {}

def build_and_compile_model(norm):
  model = keras.Sequential([
      norm,
```

```
      layers.Dense(40, activation='relu'),
      layers.Dense(40, activation='relu'),
      layers.Dense(3)
  ])

  model.compile(loss='mean_absolute_error',
                optimizer=tf.keras.optimizers.Adam(0.001))
  return model

dnn_model = build_and_compile_model(normalizer)
dnn_model.summary()

# %%time
history = dnn_model.fit(
    train_features,
    train_labels,
    validation_split=0.2,
    verbose=0, epochs=1500)

plot_loss(history)

test_results['dnn_model'] = dnn_model.evaluate(test_features, test_labels, verbose=0)

pd.DataFrame(test_results, index=['Mean absolute error']).T

test_predictions = dnn_model.predict(test_features)

error = test_predictions - test_labels
plt.hist(error, bins=25)
plt.xlabel('Prediction Error')
_ = plt.ylabel('Count')
plt.show()
plt.close()

dnn_model.save('term_project/project_dnn_model')

reloaded = tf.keras.models.load_model('term_project/project_dnn_model')

test_results['reloaded'] = reloaded.evaluate(
    test_features, test_labels, verbose=0)

pd.DataFrame(test_results, index=['Mean absolute error']).T

# Now fit to all of the data

dataset_features = dataset.drop(['city-mpg', 'highway-mpg', 'horsepower'], axis = 1)
dataset_labels = dataset[['city-mpg', 'highway-mpg', 'horsepower']]

full_normalizer = tf.keras.layers.Normalization(axis=-1)

full_normalizer.adapt(np.array(dataset_features))

print(full_normalizer.mean.numpy())

first = np.array(dataset_features[:1])

with np.printoptions(precision=2, suppress=True):
  print('First example:', first)
  print()
  print('Normalized:', full_normalizer(first).numpy())

full_dnn_model = build_and_compile_model(full_normalizer)
full_dnn_model.summary()

# %%time
history = full_dnn_model.fit(
    train_features,
    train_labels,
```

```
    validation_split = 0.2,
    verbose = 0, epochs = 1500)

plot_loss(history)

test_results['full_dnn_model'] = full_dnn_model.evaluate(dataset_features, dataset_labels,
    verbose=0)

test_predictions = full_dnn_model.predict(test_features)

error = test_predictions - test_labels
plt.hist(error, bins=25)
plt.xlabel('Prediction Error')
_ = plt.ylabel('Count')
plt.show()
plt.close()
```