Mean Estimation via Neural Network Methods in Complex Survey Imputation

---

A Thesis
Presented to
The Division of Mathematics and Statistics
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Alexander Michael Moore

May 2019

Acknowledgements

I want to thank a few people.

Preface

This is an example of a thesis setup to use the reed thesis document class (for LaTeX) and the R bookdown package, in general.

Table of Contents

List of Tables

List of Figures

Abstract

Abstract last. 100-500 words. Abstract should be able to stand alone: results and approach blatantly and simply stated. Want to sell paper at a glimpse

- main objectiuve and rationale of project
- outline of methods used
- results of the project
- conclusions about the implications of the project

Missing data is problematic and pervasive in real surveys. Complex survey imputation aims to replace missing data with approximations. Modified neural networks (naive, weighted loss, weighted resample, π-feature, derived parameter) are compared to weighted linear regression for mean estimation in artificial and real data experiements using Monte Carlo simulation on imputation of manually-induced missing data. The methods' population mean estimates are compared to the known true mean and Horvitz-Thompson estimator in both experiments. The results show the effectiveness of the neural network family in low-noise nonlinear generative functions, and superior outlier prediction in Bureau of Labour Statistics Consumer Expenditure Survey data. All neural network methods outperformed weighted linear regression for mean estimation on real data. Keras and TensorFlow are utilized for neural network models.

Key Words: Imputation, Neural Networks, Survey Statistics

Dedication

You can have a dedication here if you wish.

Chapter 1

Complex Surveys

1.1 Survey Statistics

> Researchers in the social science and health sciences are increasingly interested in using data from complex surveys to conduct the same sorts of analyses that they traditionally conduct with more straightforward data.
> - Lumley 1969

The implicit pervasiveness of survey statistics in all data motivates our exploration into its significance in imputation.

Survey statistics differ from statistics modelling in the specification of the random process that generates the data. In model-based statistics, some underlying generative model from which observations are drawn is assumed to exist. By understanding or approximating this model from data, one may draw conclusions on the nature of the generative function provided no meaningful changes to the data are made.

Contrary to model-based statistics, the analysis of complex survey samples are design based. The observations from a researcher-specified population have fixed features, and randomness is introduced when these observations are drawn from the population according to some stochastic design. This random process is under the control of the researcher, and can be known precisely. The significance of design-based methods is that the probability sample design is the procedure for taking samples from a population, not just the resulting data. This is a significant departure from the statistical analysis mindset that randomness is an element of the features and labels of a population. Using this method, the features of the population from which the observations are drawn may be estimated, but these conclusions may not generalize to other populations. With understanding of the survey sampling design from which data observations arise, the researcher may make improved estimates of the population of study compared to naive estimates (Lumley, 2011).

The probability sample design is the fundamental concept of design-based inference. Taking a random sample of 36,000 people from Oregon is an example of a survey design which implies independent and equal probability sampling of all humans in the state. The Law of Large Numbers is invoked to assume the distribution of sampled observations represents the population from which they are drawn according to any features of interest to the researcher, such as height, weight, or age.

This type of surveying can be complicated by adding unequal inclusion probability to the features to oversample from minority groups. The data created by such a design would likely not be representative of the population, since people with higher inclusion probabilities would be more likely to be sampled. However, since the probability of each person in the sample being randomly selected is known (since the population total is known), this is still a probability sample. The key point of this process is that a probability sample is the procedure for taking samples from a population, not a data set (Lumley, 2011).

There are four requirements for a data set to be a probability sample:

1. Every individual in the population must have a non-zero probability of ending up in the sample.

2. The probability of inclusion must be known for every individual who does end up in the sample.

3. Every pair of individuals in the sample must have a non-zero probability of both ending up in the sample.

4. The probability of every pair of individuals being included in the sample must be known for every pair of individuals in the sample.

1 and 2 are necessary in order to get valid population estimates, 3 and 4 are necessary to work out the accuracy of the estimates. If individuals were sampled independently of each other the first two properties would guarantee the last two (Lumley, 2011). Though 3 and 4 are requirements of a probability sample, they are often not included in datasets as they require an $n \times n$ matrix of probabilities, where n is the number of observations in the data set.

The fundamental statistical idea behind all of design-based inference is that an individual sampled with a sampling probability π_i represents $\frac{1}{\pi_i}$ individuals in the population. The value $\frac{1}{\pi_i}$ is called the sampling weight (Lumley, 2011). Since observations represent different proportions of the population, the inclusion probability must be accounted for in modeling and estimation procedures.

Data collected under a complex survey design have an additional layer of complexity and are not to be treated as independent and identically distributed (*i.i.d.*). Ignoring this complex survey design is found to create significant error in data analyses (Toth & Eltinge, 2011). This concern motivates our exploration of accounting for survey design in neural network imputation.

1.2 Imputation

Often in real-world data, there is some degree of missingness. This can be for any number of reasons, illustrated below: Item nonresponse is the focus of this thesis. Item nonresponse will be restricted to a response value of interest called a label and will be present in some of the observations. The other variables of the observation,

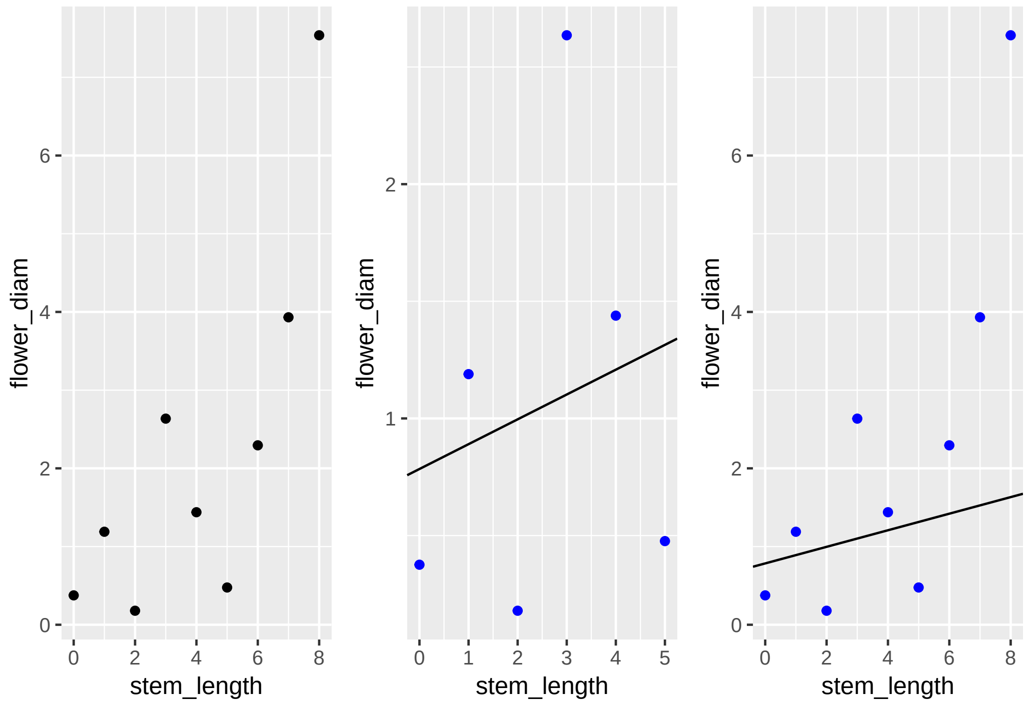
Table 1.1: Types of missingess

	Description
Noncoverage	An element in the target population is not included
Total Nonresponse	A sampled element does not participate in the survey
Item Nonresponse	A responder fails gives insatisfactory responses

called features, will be fully present. The usual form of handling item nonresponse is imputation, which fills in missing values with usable information (Brick & Kalton, 1996). Common algorithms such as Principal Component Analysis and regression require no missingness in the data set, so replacing NA values with useable information is vital for analysis.

In many statistical analyses, observations with any degree of missingness cannot be included. For example, how would one perform a linear regression with observations that have features but no label? Dropping these terms would remove potentially huge swathes of the data set, particularly in multivariate data sets, and would potentially create systematic bias.

Suppose for example a data set with information on roses had a feature with stem length and a label on flower size measured by a researcher. There might be missing values for flower size that are not randomly distributed: Before the researcher makes measurements, there has been systematic removal of the large-flowered roses by passersby. To ignore these observations would lead the analyst to draw false conclusions on the relationship of stem length to flower size, the distribution of flower sizes in the population, and estimations of the mean flower size in the population of all flowers, visualized in Example 1.1:



Imputation attempts to address this common dilemma in real-world data. Imputation is the process of replacing missingness in data sets with some value that redeems the observation for some degree of analysis. “The aim of these methods is to compensate for the missing data in such a manner that the analysis file may be subjected to any form of analysis without the need for further consideration of the missing data” (Brick & Kalton, 1996). Imputation assigns values to missing responses, which allows records with missingness to be retained during analysis. Ideally, imputation would eliminate bias in survey estimates caused by ignoring records with missing data. The catch is that imputation can destroy intervariable relationships, as well as overestimate the precision of survey estimates on fabricated data.

There are stochastic and deterministic methods for imputation. Deterministic regression imputation is the predicted value from a regression trained on complete-case data. Stochastic imputation differs due to an additional residual added term to the predicted value, taken either from a random respondent or comparable observation, and is usually preferred due to the importance of shape parameters in many analyses (Brick & Kalton, 1996).

One traditional method of imputation is the “Hot-Deck Method”, which was generally favorable when computation was less efficient. Hot Deck Imputation requires extensive knowledge of the survey variables in order to optimize performance since explicit model-based imputation needs a valid model for every survey variable (Maiti, Miller, & Mukhopadhyay, 2008). This thesis proposes naive artificial neural networks as a solution which requires minimal domain knowledge and resists the curse of dimensionality which other nonparametric methods are susceptible to, such as local polynomial regression (Maiti et al., 2008).

Due to the difficulties imposed by working with high-dimensional, complex survey data with varying degrees of domain knowledge, we turn to neural networks and their structural benefits to reliably perform these challenging imputation problems.

Chapter 2

Neural Networks

2.1 Introduction to Machine Learning

From the advent of computation, there has been a drive towards automation. (Goodfellow, Bengio, & Courville, 2016)

The capacity to derive patterns from raw data is known as machine learning, a broad term for an extremely diverse collection of algorithms. Machine learning flips the script on classical programming: whereas classical programming takes rules and data to produce answers, machine learning creates rules from data and answers by being "trained rather than explicitly programmed. It's presented with many examples relevant to a task, and it finds a structure that derives rules for automating the task" (Chollet & Allaire, 2018).

Machine learning is intended to elucidate or predict patterns in data. These algorithms handle anything from linear regression for predicting a numerical response based on a number of features, to clustering for visualization and assignment of untaught observation classes. Machine learning models are trained by minimizing error during exposure to labelled "training" data with some underlying distribution and random noise. After training, models are passed unlabelled "test" data to predict the corresponding unknown class or value. Predictive (supervised) machine learning algorithms seek to emulate and elucidate a true unknown generative function from which the data were drawn.

One key difference from statistical modelling and survey statistics methods is the point at which randomness is introduced into the data. Machine learning attempts to approximate the function from which the data was randomly generated, while survey statistics imply that randomness in data comes from the survey design. The paradox of where randomness is introduced into the data is resolved with the existence of a superpopulation U, where each observation has label $y = f(x) + \epsilon$, some generative function. From this superpopulation, a population u is created through *i.i.d.* realizations from U. From this population u, the survey is taken. Thus there still exists a generative function from which the population is drawn, but the features and label of the observations are fixed by the time the complex survey is taken on the population, reconciling the two methodologies.

For imputation purposes, the goal of machine learning methods will be to accurately estimate missing values by approximating the generative function from which they are drawn. Generative functions are of the form

$$y = f(x_1, x_2, \ldots, x_n) + \epsilon$$

where the true label y of the observation is a function of the features $x_1, ..., x_n$ perturbed by some random noise ϵ.

The estimating model will be trained via exposure to labelled observations, called the training data, then used to predict observations with missing labels, called testing data. The challenge in machine learning is learning the correct amount from training data in order to derive the underlying distribution of the observations without simply memorizing the labels of the training set. This memorizaion problem is called overfitting and is central to machine learning.

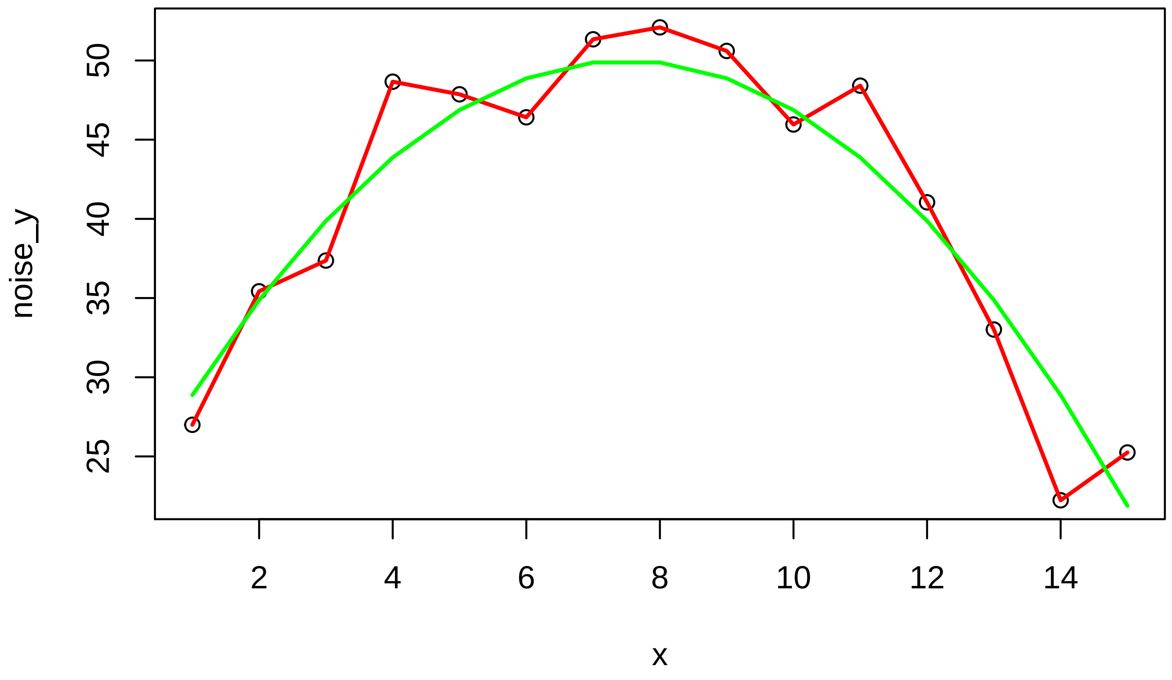


Figure 2.1: The red model has better training accuracy than the green model

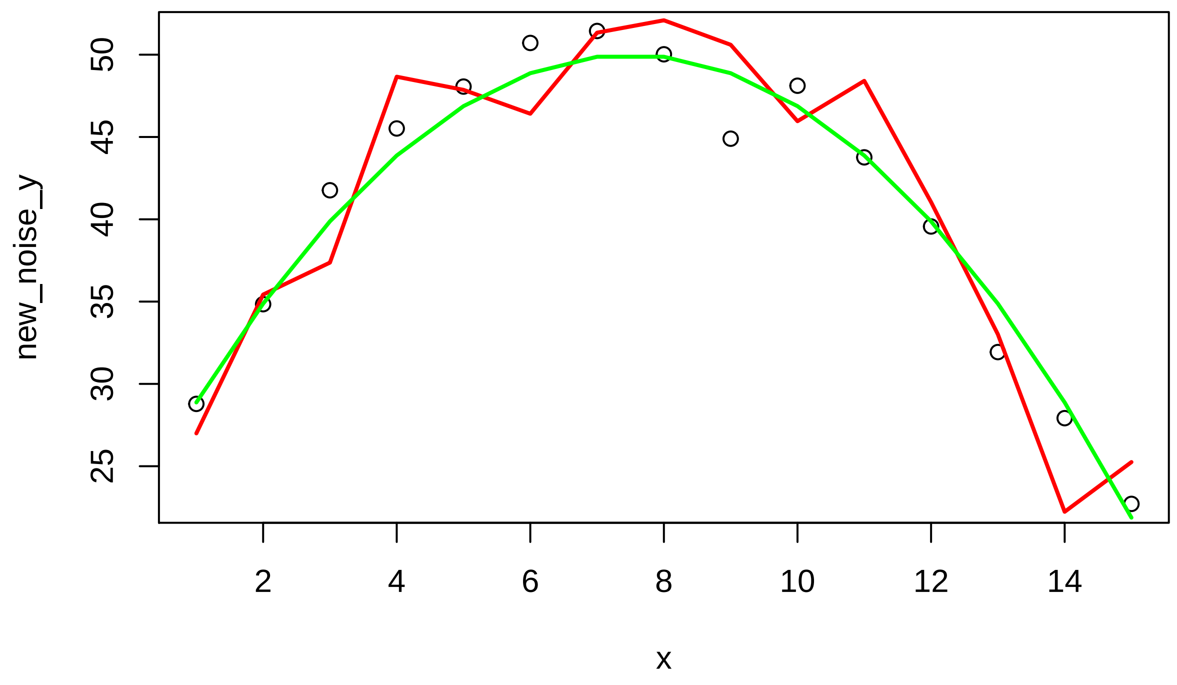


Figure 2.2: Both models on a new realization of the data

2.1 is an illustration of this pervasive principle in a regression example. An overfit (too flexible) model simply memorizes the training observation's labels, rather than the underlying distribution, in this case a second-degree polynomial.

The overfit red model does not accurately reflect the underlying distribution from which both datasets are drawn. Rather, it captures only the random noise of the data on which it was trained. It would be a mistake to assume the generative function is a degree 15 polynomial just because the training error of such a function is low.

The overfit red model has extremely low error on the realization of the data on which it was trained, but fails to learn the underlying generative function, and instead learns the random noise of the training observations (2.1). Thus this model is a poor descriptor of a new data realization from the same generative function, seen in 2.2.

The green line in 2.1 and 2.2 shows a well-trained model which straddles these extrema and captures the apparent underlying distribution of the data in a general sense by approximating the generative function from which they are drawn, and remains fairly constant under different draws from the same distribution. The aptly-flexible model has consistent performance on new realizations of the data. The ability for a model to perform well on data it was not trained on is called generalizability, and is the goal of predictive models.

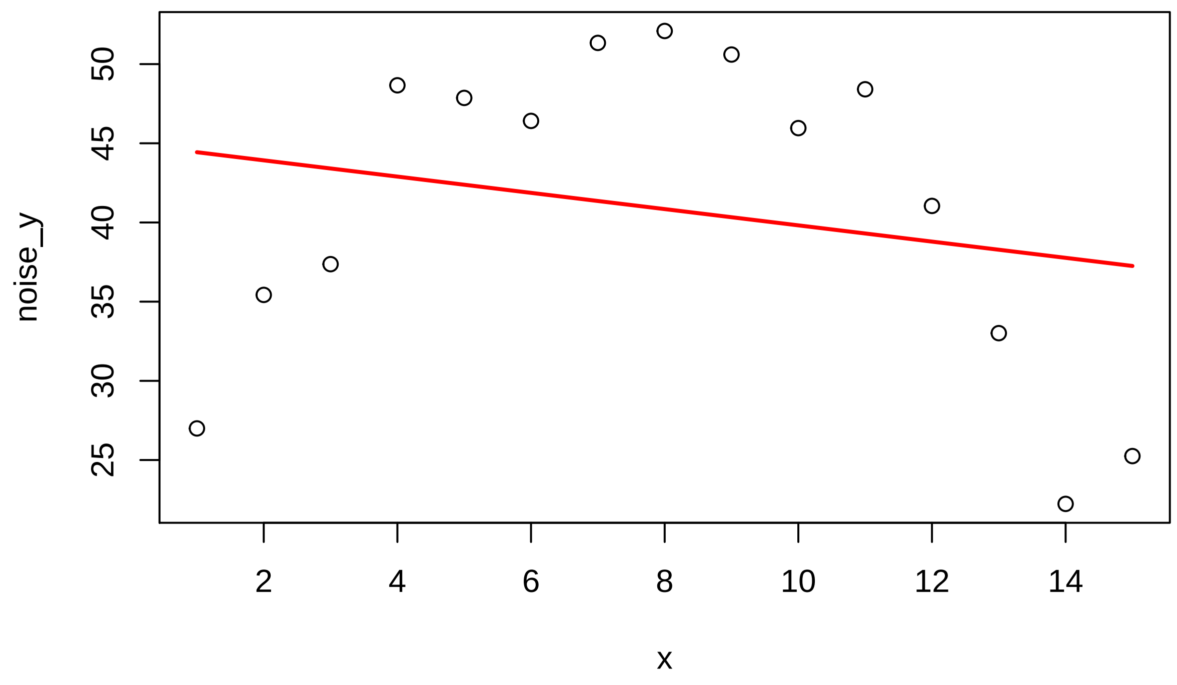


Figure 2.3: An underfit model

An underfit model, 2.3 , also fails to capture the underlying distribution, due to a lack of flexibility. A linear regression, though it minimizes its training MSE, clearly fails to capture the underlying distribution of the data.

The relationship of the fitness to the complexity of the model is described by the bias-variance tradeoff phenomena.

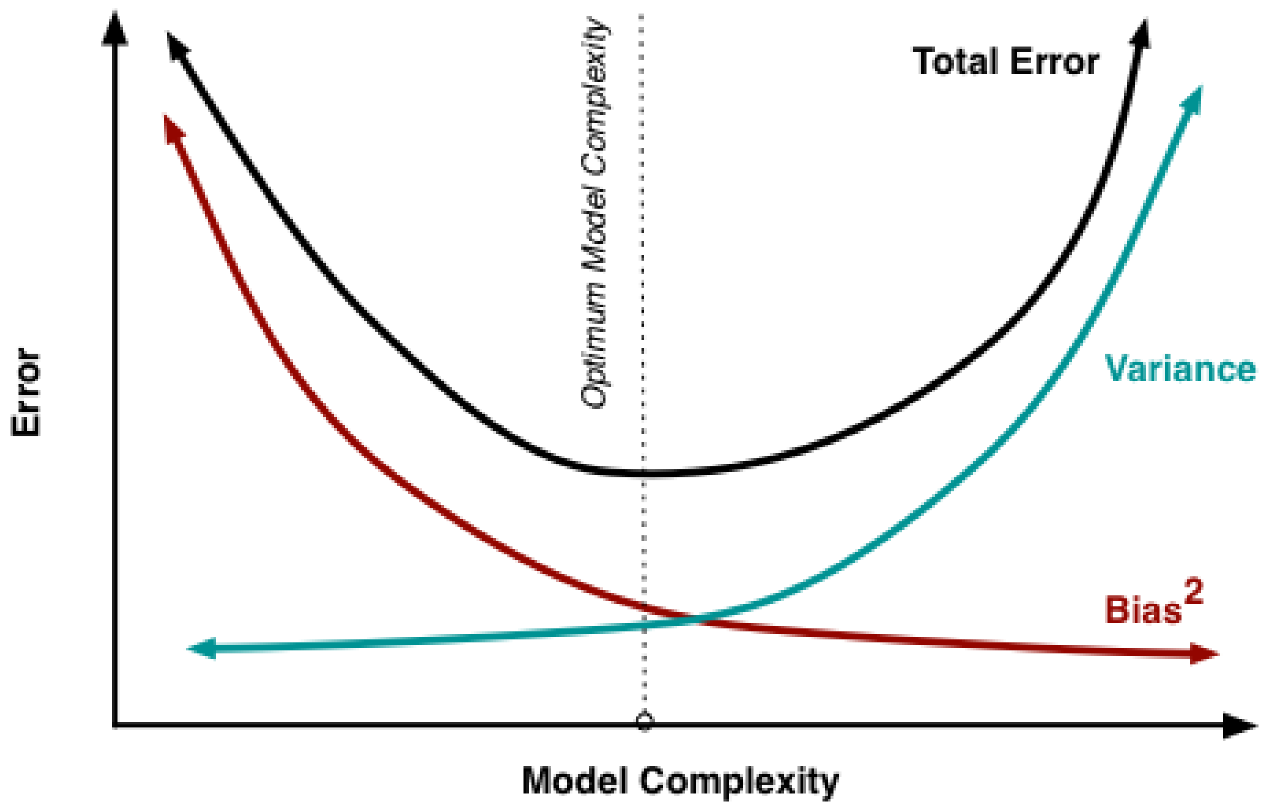


Figure 2.4: The bias-variance tradeoff

Figure 2.4[1] demonstrates the involvement of model complexity (or flexibility) in terms of training error. Model complexity refers to the ability of the model to approximate complex generative functions. We see that as the flexibility of the model increases, the bias on the training set always decreases, representing the model's performance on observations with labels. However, complexity beyond the models optimal value implies over-flexibility, in which the model is able to memorize random noise rather than stopping at the trend of the data. This increases the total error of the model when exposed to data that comes from the same generative function that the model has not been exposed to, such as a testing data set or another observation from outside the training set. Higher flexibility models create more variable models, which though trained on data from the same generative function differ greatly in appearance due to the random sample of training data. These models are unstable for inference and generalize poorly.

[1]Fortmann, Scott. "Bias and Variance." Understanding the Bias-Variance Tradeoff, June 2012, scott.fortmann-roe.com/docs/BiasVariance.html.

2.2 Neural Networks

2.2.1 Background and Context

Neural networks are a family of machine learning algorithms with an extended and diverse history of research in neuroscience, statistics, and computer science. Recently, these models have experienced great growth in attention and popularity due to the contemporary circumstance of computational capability. Neural networks thrive on large training data sets, which have tended to increase in size and availability throughout time. Neural networks also outperform competing algorithms in high-dimension feature problems, which are common in real-world machine learning applications such as image data, waveform data, and large surveys. Often utilizing derivatives of complex compositions of functions as an optimization method, deep learning training periods are computationally intensive, relying heavily on computer hardware and optimized software for reasonable implementation. Lastly, recent attention to and development of neural networks can be attributed to their proven potential in solving complicated real-world applications with promising and increasingly dominant accuracy in practice, often at the cost of a lack of inferability.

This lack of inferability is the typical downside of working with neural networks as the inference on a model can be more important than the predictive accuracy depending on the problem. Once a simple linear regression is trained, the coefficients on the predictors offer an immediately understandable interpretation of the behavior of the data. For example, a coefficient of .3 on a feature x has a simple, instant understanding: as feature x goes up by 1, the response goes up by .3. Neural networks however, lack this instant recognition due to the less intuitive layered structure of input transformations, known as representation learning.

2.2.2 Basics

Neural Networks are a composition of functions. The following describes a full neural network:

$$\hat{y} = f(\boldsymbol{x};\theta,\omega) = f^n(f^{n-1}(...f^1(\boldsymbol{x};\theta,\omega)))$$

In this function, we see the input features $x \in \mathbb{R}^n$, the learned coefficients θ, the learning rate $\omega \in \mathbb{R}$, and the output prediction $\hat{y}$. Consider one of the layers of the network, f^i. This layer is an activation function composed with a linear function:

$$f^i = \max(0, \boldsymbol{W}_i{}^T\boldsymbol{x} + c_i)$$

Where $\boldsymbol{W}_i^T\boldsymbol{x} + c_i$ is the interior linear function for $\boldsymbol{W}_i^T \in \mathbb{R}^n$, $c_i \in \mathbb{R}$.

The activation function shown above is the rectified linear unit, or $\max(0, a)$. Activation functions are significant as they introduce nonlinearity into what would otherwise be a linear function (a composition of linear functions). $W_i{}^T$ and c in f_i dictate a linear transformation on the input to the layer. An ordered list of all elements of W_i and c_i for all $i \in n$ would give the full description of the network, called θ. So the output of a 1-layer network can be expressed as

$$f(x; W, c, w, b) = w^T \max(0, \boldsymbol{W}^T \boldsymbol{x} + c) + b$$

The final activation function is another structural parameter left to the designer of the network, but differs from interior activation functions as the output of the network is restricted to the codomain of the function, limiting the number of reasonable choices. The typical output layer for scalar regression is a linear unit, on behalf of the codomain $(-\infty, \infty)$ of the activation:

$$f^n(\boldsymbol{x}) = 1 * (\boldsymbol{W}_n^T \boldsymbol{x} + c_n)$$

The learning rate ω and a loss function are meta-parameters, given by the user creating the neural network. These two parameters are used during the training of the network. During training, gradient descent is used to descend the loss function in order to find the optimal parameters for the network.

Loss functions are ways of describing the performance of a model when predicting labels of a data set. The loss function takes the model and data as inputs, and outputs a real number. Loss functions can be minimized in training by optimization which leads the network to find improvements in weights yielding accurate predictions. Loss functions allow for another degree of customization in the training of a network, such as in Ridge and LASSO regression. These algorithms add a weighting to the typical mean squared error loss function which penalizes the weights on predictors in polynomial regression. These methods introduce bias into otherwise unbiased algorithms, but reduce the variability of the model across different draws of data from the same distribution, aiming to reduce the real test loss and improve the model. The cross entropy between the data distribution and the model distribution is the typical choice (Goodfellow et al., 2016).

Cross Entropy:

$$\int_{\chi} P(x) \log Q(x) dr(x) = E_p[-\log Q]$$

Take for example the Mean Squared Error cost function:

$$MSE = \frac{1}{n} \sum_{k=1}^{n} (y - \hat{y})$$

MSE, a typical loss function for regression applications, takes the mean squared difference of the predicted label $\hat{y}$ and the true label y of the observations.

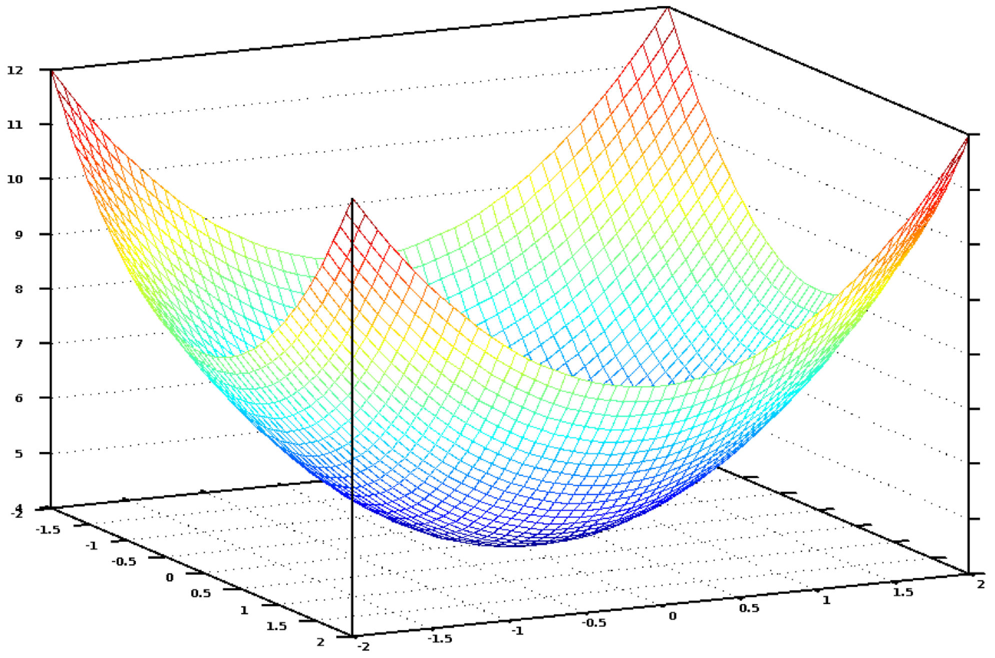


Figure 2.5: A convex loss function

In 2.5[2] We can see that the loss of a linear regression is a function of m and c given some data which is minimized at $m = 0$ and $c = 0$. The learning rate is the amount that the functions' coefficients are updated as the loss function is optimized from some initial coordinates in search of the minimum loss.

Much like training a linear regression, the training of the neural network aims to drive the approximating function to the underlying function. The important difference induced by nonparametric models, however, is the nonconvexity of the loss. The large amount of coefficients which define even a relatively small network complicate the optimization problem and make local minima a demanding distraction. Optimization is typically done with gradient learning using a loss function, or maximum likelihood estimation. Most modern neural networks are trained using maximum likelihood, in which the cost function is the negative log-likelihood between the training data and model distribution (Goodfellow et al., 2016).

2.2.3 Representation Learning

If you were handed a photograph and were asked if it contained a car, there's a good chance you would immediately know the answer. But how are you coming to this conclusion? The human mind recognizes patterns of the image it has seen before, like a round bumper, a rectangular windshield, and a few circular tires with a certain spatial

[2]Geitgey, Adam. "Machine Learning Is Fun!" Medium, Medium, 5 May 2014, `medium.com/@ageitgey/machine-learning-is-fun-80ea3ec3c471`.

relationship to come to the conclusion that there is, in fact, a car in the photograph. It is this process of representation learning that prompted early researchers to create representation-based learning algorithms to extrapolate on information relationships in the same way as the human mind. The emulation of human neural cells was the birth of neural networks, a class of machine learning algorithms which takes its name from multiple layers of connected nodes simulating a proposed structure of the neurons of the human mind. Representation learning takes observation's features as a first layer into a composition of functions, in which each layer (or function) transforms the data according to a learned representation transformation that the subsequent layer takes as an input. This composition allows for complex relationships of features to be learned in increasingly sophisticated ways, making neural networks ideal for large-dimensional datasets. For large-dimensional feature spaces such as in Consumer Expenditure Data, images, or audio, these hierarchical representations (or layered representations) are important for distilling human-like feature extraction and iterative predictor space transformation. To achieve hierarchical representation properties, subsequent layers understand more complex functions of input variables as each layer transforms inputs and and weight relationships that minimize the overall loss of the network. As each layer receives the transformed output of the previous layer, more complex relationships of features can be derived.

Successive layers of the network learn transformed representations of the data which lend themselves to an effective linear regression performed by the final layer, called the output layer. This process of successive transformation learns meaningful representations on which the output layer can be most accurate. Since the full network does not have a single functional representation, it is nonparametric. The flexibility and power of neural networks in fields demanding domain knowledge is that they can approximate any function, per the Universal Approximation Theorem (Hornik et al., 1989; Cybenko, 1989). The theorem states that a feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function can approximate any function from one finite-dimensional space to another with any desired nonzero amount of error, provided that the network is given enough hidden units. Thus the user decision required is the depth and breadth of the network, optimized through validation set meta-analysis.

Representation learning is extremely important to the broad promises of neural networks in practice. The basis for this strength is that subsequent layers of the network learn meaningful structures of the input data associated with a lower loss score. Correlations of features forming a functional relationship to the label which induce loss function descent will be internalized by the composition of subsequent functions. This property of representation learning is significant for investigating the necessity of independent and identically distributed data in deep learning algorithms. Fo survey data, it could be the case that the significance of the inclusion probability can be learned as a meaningful feature, with no special tweaks or preprocessing necessary for the algorithm. Data which require no special tweeks are extremely meaningful as this circumvents the necessity of incorporating domain knowledge and expertise in current imputation methods, which holds back expedient and lightweight research.

These advantages of Hierarchical and Distributed Representation transformation give neural networks huge advantages in accuracy and fitting capability for data with a massive hypothesis space. A hypothesis space is the space of all possible answers to a question. Image classification, for instance, represents a hypothesis space of pixels with unknown correlations that must be trained with label relationships to determine the correct transformative relationsip of millions of pixels to a most-likely class label. Thus the curse of dimensionality common throughout machine learning is mitigated through successive transformation layers.

Neural networks thrive on the interaction of many features, due to the nature of representation learning which excels in covariate relations and distilling information encoded between features. Popular modern applications are image and waveform audio data, in which machine learning problems become dominated by the Curse of Dimensionality. This common machine learning problem arises when the amount of data is insignificant when compared to the hypothesis space or feature space, and there are sparse observations for some regions. Machine learning needs many observations with each combination of values, but this becomes quickly infeasible for data with thousands of features. The peaking phenomena dictates that there is an optimal number of features to describe the data before the curse of dimensionality creates problematic sparsity and dominating observations with few neighbors . Neural networks are known to resist this commonplace issue due to the distributed learning property, wherein each node is sensitive to only particular features.

Distributed representation is a powerful implicit component of neural networks in which neurons divide feature space to better handle feature interactions: suppose an image-recognition system could recognize cars, trucks, and birds, as well as distinguish if these objects are red, green, or blue. One way of representing these inputs could be to have a separate neuron for each combination: red truck, red car, red bird, and so on for nine independent neurons. Distributed representation, however, could partition these workloads by having three neurons for color and three for object type. In addition to reducing the number of neurons required dimensionally, this also distributes the learning demands of each neuron. The neuron describing redness is able to learn about redness from any category, not one specific category as the red bird neuron must (Goodfellow et al., 2016).

Neural networks approximate nonlinear functions by applying linear models not to the features x, but to a transformed input, $\phi(x)$, where ϕ is a nonlinear transformation. ϕ provides a new, more meaningful representation for $\boldsymbol{x}$. The question then is how to choose the mapping ϕ:

1. One option is to manually engineer ϕ. This takes a huge speciality of domain knowledge and practitioner specialization, with little transfer between domains. This was the dominant method before deep learning (Goodfellow et al., 2016).

2. The strategy of neural networks comes from the learning of ϕ. In this approach, we have a model $y = f(\boldsymbol{x}; \theta, \omega)$ as specified in the neural network introduction. Due to the Universal Approximation Theorem, the nonparametric deep feed-forward network can learn a functional approximation from the input to the desired output. This method sacrifices the training convexity of the other two, but benefits from the genericity of the specifications. The human user need

only specify a general function family rather than exactly the correct function, but can still benefit from designing families of $\phi(\boldsymbol{x}; \theta)$ that they expect to be relevant (Goodfellow et al., 2016).

2.2.4 Neural Networks for Complex Survey Data

From an optimist's perspective, the need for data preprocessing or special conditions on the loss function for training the model would be unnecessary: If learning the correlations and underlying distributions associated with rare observations from complex survey design would truly lower the network's loss, it should be learned and accounted for without the need to perform special external transformations on the data to "undo" the effects of complex sample design. For this reason, it is significant to compare the potentially superior results of a naive model to one with superfluous data transformations done. A neural network model with access to an informative π feature ideally would approximate the function relating the inclusion probability and features to labels, without the need for extreme domain knowledge and manual feature engineering.

The optimism of nonparametric algorithms increases in tasks of minimal domain knowledge and feature engineering capability. Ideally, using heuristic metaparameters defining a neural network model would be enough to get reasonable predictive accuracy in conjunction with one of the methods of Chapter 3. Per the Universal Approximation Theorem, any underlying generative function is at worst approximated by a 2 layer heuristic model, potentially improving upon other naive modeling procedures such as a weighted linear regression. Additionally, in real survey data the capacity to derive the significant features from a high-dimensional space is a weakness of parametric model regression, which is highly variable in the context of the noisy features explored in Chapter 4.

Chapter 3

Methods

Missingness is an extremely common problem in real data sets. Many analyses and algorithms such as regression, classification, PCA, and clustering done throughout the sciences rely on having entirely complete observations. For this reason, some strategy must be adopted to transform a raw data set into an analyzable one.

There are multiple approaches for a researcher interested in going about this process. The researcher has a data set gathered under a complex survey design where the features $x_1, ...x_n$ and inclusion probability π_i is known for all i, though the label y_i may be missing due to item nonresponse. Sample observations with item response are the elements $r \subseteq s$, of which there are n_r, and there are n_{s-r} nonrespondents $s - r \subset s$. In order to create a data set with no missing values, the researcher may choose to adopt one of the following methods. These methods borrow notions from complex survey design about using design variables to model how characteristics of the population affects the sample (Maiti et al., 2008). Neural network methods have the advantage of flexibility in approximating any functional relationship, and can be tuned using sampling weights to account for complex sampling design using unequal sampling (Maiti et al., 2008).

There are two primary tasks for a researcher: representative complete-case data, and statistical estimation via accurate imputation. The following methods are imputation techniques designed with these goals in mind. Imputation methods utilize different techniques to create a complete-case dataset by estimating missing values based on information from the complete cases. Once the missing values are imputed, the population parameter is estimated using the approximate complete-case data set.

3.1 Mean Estimation Methods

3.1.1 Naive Mean

A common statistic of interest to a researcher is an estimate of the population mean $\mu_y = \frac{1}{N}\sum_{i \in N} y_i$ using the information from the complex sample with missingness. Taking the naive mean of complete cases is insufficient for two reasons. The first has been discussed in Chapter 1, which is the potential of systematic missingness in the data. The second is that the naive mean makes the assumption that the observations represent equal proportions of the population, as in an *i.i.d.* sample.

The naive mean estimator ignores survey design and allows equal contribution of all observations:

$$\hat{\mu}_y = \frac{1}{r}\sum_{i \in r} y_i$$

3.1.2 π-Corrected Naive Mean

The mean estimate formula weighted problem is resolved by weighting (scaling) contributions to the mean by $\frac{1}{\pi_i}$, the approximate number of population members represented by observation i. This value is the Horvitz-Thompson estimator of the true number of people who would respond in the population (Horvitz & Thompson, 1952):

$$\hat{\mu}_y = \frac{1}{\hat{N}_r}\sum_{i \in r} y_i \frac{1}{\pi_i} \tag{3.1}$$

Where $\hat{N}_r = \sum_{i \in r} \frac{1}{\pi_i}$. This resolves the problem of ignoring the survey design in estimation, but does not account for systematic bias resulting from missingness. For example, the π-corrected naive mean estimate of the population mean will be an under-estimate of the true population mean in the presence of systematic missingess of large observations.

Let $\hat{\mu}_y$ be the sample estimate of the population mean μ_y. The oracle mean can be considered the "best estimate" of μ_y, but it is only available in simulation. The oracle mean uses information that the simulation manually drops so as to create an ideal population estimate given a survey sample:

3.2 Imputation Methods

In order to combat the presence of systematic missing values, we utilize imputation to label the missing observations with the best estimate of y_i, $\hat{y}_i$. Imputation has the added benefit of approximating a complete-case dataset without dropping observations.

3.2.1 Imputation Mean Estimator:

$$\hat{\mu}_y(\text{method}) = \frac{1}{\hat{N}}(\sum_{i \in r} \frac{y_i}{\pi_i} + \sum_{i \in s-r} \frac{\hat{y}_i}{\pi_i})$$

Where $s - r$ are the missing cases, r are the respondents, and $\hat{N} = \sum_{i \in n} \frac{1}{\pi_i}$ is the Horvitz-Thompson estimator (3.1) of the population size (Horvitz & Thompson, 1952).

3.2.2 Drop NA

One option is to simply remove the observations with missing labels from the data set. This method is extremely easy to implement and serves as a sure-fire way to end up with a data set with no missing values. There are two downsides to this method: The first is that removing observations with missingness obviously decreases the size of the data set and can nontrivially reduce the fidelity of models aiming to understand the population from which the data was taken. The second problem is the assumption of random missingness. If there is any correlation of label to amount of missingness, systematic bias is introduced into the data set, as discussed in Chapter 1. For example, if there is a possibility that larger values are more likely to be dropped, then as a result the sample mean would underestimate the population mean.

3.2.3 Median Imputation

Median imputation is another easy way to get complete cases in data for analysis or estimation. Median imputation simply fills in the missing labels with the median of the respondent labels. Median imputation has multiple problems for analysis or estimation. The median offers equal weighting to all observations in a data set, meaning it destroys the informativity of the inclusion probability π. It also removes correlation of feature and label, making analyses such as PCA less informative, as covariate relations dictate axis derivations. Median imputation is extremely fast to execute and implement, but creates noninformative observations in the same manner as the Drop.NA method.

3.2.4 Weighted Linear Regression Imputation

Linear regression is a convex optimization problem of $p+1$ parameters, where $\hat{f}(x) = \hat{y} = (m_1 x_1 + . + m_p x_p) + b$ is the estimate of y for observation i given features $\boldsymbol{x} = [x_1, \ldots, x_p]$. Using the mean of the squared difference between the predicted and actual responses of a training data set, weighted-MSE linear regression uses the loss function (3.2) which scales the squared error contribution of each observation by $\frac{1}{\pi_i}$ to account for rare-case observations with potentially systematic missingess, and returns a scalar loss:

$$\text{MSE}(f) = \frac{1}{\hat{N}} \sum_{i \in r} (\frac{1}{\pi}(\hat{y} - y))^2 \tag{3.2}$$

Significant to this algorithm, however, is the contribution of π regardless of whether it is informative. In data known to have systematic label missingness, this is not a problem. However, it is not always the case that π is informative, and including this scaling term would be harmful to cases in which π and y are uncorrelated.

3.2.5 Naive Neural Network Imputation

Neural Network Imputation is our baseline model for imputation for estimating a population mean. The number of hidden layers, nodes, and loss is left to the user, which would be an incorporation of domain knowledge or exploratory modelling to derive a reasonable model load for learning the data's generative function. In the context of the researcher having minimal knowledge of the data, a neural network with 2 hidden layers of 32 units activated by `relu` functions is a reasonable starting point. Overtraining due to overflexible models can be stimied with a validation set, assuming the data is not so small as to create unrepresentative data by subsetting the training data further ($n > 10^3$). "Naive" neural network imputation refers to this model not having access to the π feature of the observations as a predictor or incorporate it in any way. This model uses the assumption that the data is *i.i.d* as a representative for ignoring the survey design, an assumption which is known to be a significant problem. Regardless, the neural network should approximate the generative function with some nonparametric fit, but underestimate the population mean as a result of systematic missingness and observation equity. This baseline model will be the pacemaker for the other neural network methods to improve upon by incorporating survey design information.

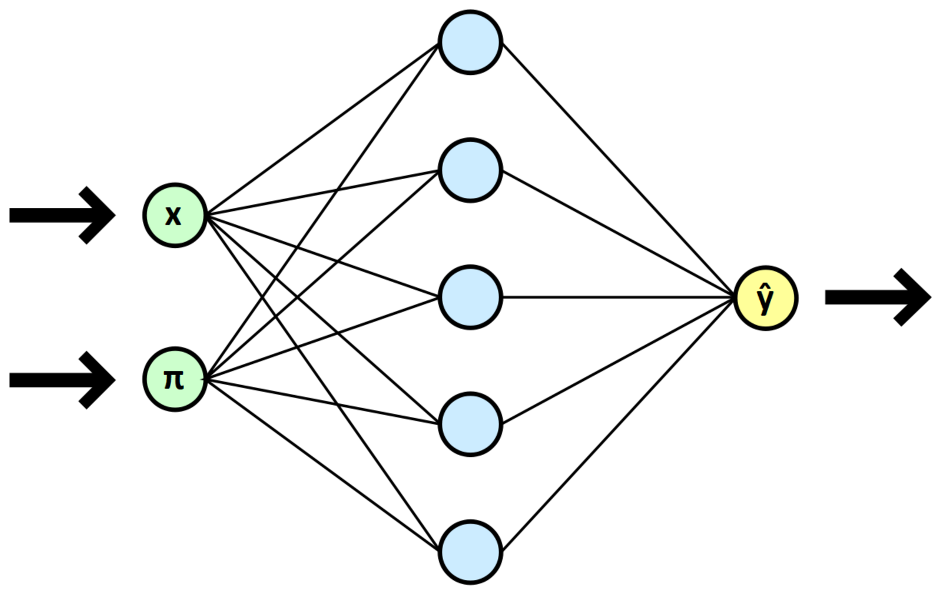
3.2.6 Weighted Loss Neural Network Imputation

Weighted Loss Neural Network Imputation takes inspiration from the weighted linear regression algorithm. This neural network training uses the same π-weighted MSE of (3.2), but sacrifices the convexity of the loss function of the linear model, which induces problematic local minima in the loss.

The existence of local minima from the high-dimensional model space comes from the flexibility of the many hidden neurons weight transformations within the model. A loss-weighted neural network hopes to account for systematic missingness in the data by heavily punishing the loss term generated from rarer observations. Since rare observations are more likely to be missing, they must be given more weight since they appear less in the training data then the population. Thus a weighting scheme attempts to un-do systematic missingness by making the rarer observations as "heavy" as they would be in the true population by making outliers be worth multiple observations to the loss contribution.

3.2.7 π-Feature Neural Network Imputation

A π-feature neural network has access to π as a predictor during training and testing. This is a realistic assumption to make as data collected under a complex survey design must have a π probability regardless of whether the label is missing. This method has the benefit of adapting to whether π is truly correlated to y, which the loss-weighted method assumes. A neural network optimist could claim that if there is a significant relationship of π to y, it will be reflected in the loss during training and the network will adapt accordingly to the information provided by the feature. However if π and y are uncorrelated and the missingess is random, the network will not still weight the observations and will correctly ignore the feature to create more accurate predictions with no need for domain knowledge on the relationship of the missingness.



3.2.8 Weighted Resample Neural Network Imputation

The weighted resample method uses the same model as the naive neural network imputation but uses a data preprocessing step to incorporate the survey design information. A weighted resample of size n is taken from the sample with replacement with observation selected by probability $\frac{1}{\pi}$. The inference of this method is an attempt to "undo" the effects of survey design in gathering the data. A weighted resample in which observations are selected by the inverse inclusion probability uses the insight that an observation with inclusion probability π represents $\frac{1}{\pi}$ members of the population. By sampling on this probability, the resampled data set should be an estimate of an *i.i.d.* sample from the population, making it viable for supervised learning ignoring the survey design elements. From this point the naive neural network method is applied (and directly compared to naive neural network estimates), since ideally this is *i.i.d* data without need for survey design information tweaks on the algorithm.

3.2.9 Derived Feature Neural Network Imputation

This method pre-processes the data by adding additional features to the data. In the simulation and `CE` experiments, this is done by multiplying two informative features with π and concatenating the result to the data. The intention of this method is expediting the training process to allow the approximation of more complex generative functions with less model capacity and training. Intuitively, should the relationship of π to x_i in the missing data be relevant, it will be more easily learned for superior prediction accuracy. This method does have an implicit "oracle" element, however, as knowing which columns to derive from a real data set is incorporation of a significant amount of domain knowledge.

Chapter 4

Simulation

4.1 Exploration of Methods Using Simulation

Simulated data is used to evaluate the methods in order to get an understanding of their performance on a data set with known characteristics.

Full insight into the feature distributions, generative function, random noise, and systematic missingness allow for controlled experimentation on when certain methods thrive. Simulation allows for parameters such as label noise and feature informativity to be changed and measure the response of the methods across any domain.

4.2 High-Dimension Simulation

Often in real-world data, there might be a large number of features, not all of which are necessarily correlated to the response label. For example in the Consumer Expenditure data, age, gender, ethnicity, and education might have a predictive relationship with household income, while a number of others such as state, reference person gender, and social security income may not. The ability for a model to discern which features are relevant and which are not is a significant benefit both for inference and predictive consistency. Inspired by research in best feature selection methods, non-correlated features are used in the model evaluation step in an attempt to more realistically simulate complex data (Hastie, Tibshirani, & Tibshirani, 2017). The addition of these features contributes to the curse of dimensionality and variability of estimates.

These additional "noisy" parameters are simulated by making the generative function of the labels y not a function of some of the features. All methods used in the simulation are accompanied by their oracle counterparts, meaning the same method is run two times: one with access to all features, the oracle restricted only to the relevant features which are inputs to the generative function.

4.3 Monte Carlo Simulation

Monte Carlo (MC) simulation attempts to overcome the inherent randomness of sampling and model training by repeated iterations of the sampling, training, and evaluation steps. A distribution of output estimates is made over many potential samples from the population in order to get a more complete interpretation of the results.

It should be noted that neural networks can be more optimized than in their performance in the following simulation study. This is because a network can be intentionally over-trained on the training data, then re-trained from scratch to the number of epochs which minimized the validation loss. This poses a challenge in MC simulation, however, where many iterations without user input are needed. Additionally, runtime becomes a vastly greater concern when networks must be over-trained through many epochs, more than doubling the execution time. If computation were no expense, the neural network methods could be expected to be slightly more accurate than the below results. This is somewhat mitigated by a weighted sampling scheme with somewhat low observation variability and a strong signal between features and label.

4.4 Creating a Simulated Population

The size of the population is $N = 10^5$ and the sample is $n = 10^3$ observations. Despite neural networks thriving in higher-dimension settings (both of features and observations), complex surveys are often implemented on real data with scarce observations, making a lower-size simulation more suitable for generalizability.

The features simulated population is made as the first step of the process. Two informative features p_1, p_2 are drawn from random uniform distribution:

$$f(x) = \frac{1}{30+30} \text{ for } x \in (-30, 30)$$

In addition to two informative features, there are 20 uninformative features drawn from the normal distribution. This distribution is arbitrary but can have unintended consequences due to accidental correlation, noted in {Hastie et al. (2017)}.

The population labels y are a nonlinear function of the informative features:

$$y = p_1^2 + p_2^2$$

The labels are then made nonpolynomial by contorting the parabola into a "bowl with flat lip":

$$y' = \begin{cases} y & y < \bar{y} \\ \bar{y} & y \geq \bar{y} \end{cases}$$

Random noise ϵ with $\sigma = 1$ is then added to the transformed labels:

$$y'' = y' + \epsilon_y$$

An inclusion probability *pi* is then assigned to each observation with a strong correlation to y, as well as random normal noise with $\sigma = 1$:

$$\pi = \sqrt{y''} + \epsilon_\pi$$

These values are then rescaled to (0,1) and rescaled so $\sum_i^N \pi_i = n$.

Combining the information y, π, p_1, p_2 and uncorrelated features gives the population on which the simulation will be performed. The true mean of the population labels μ_y is now known and recorded.

The Monte Carlo simulation is performed for 100 iterations. This is a low number of trials, but is constrained by the computation intensity of running 10 neural networks trained for hundreds of iterations each, as well as performing many weighted samples.

Within each iteration, a single estimation procedure is performed. First, a weighted sample from the population is taken with respect to the inclusion probabilities π_i on each observation. From this sample of $n = 10^3$ observations, an oracle data set is made by dropping the noise columns, keeping only p_1, p_2, π, and y. This data set will be used as a benchmark to mirror each imputation method in order to gauge the potentially harmful effects of uninformative features common to real data.

The first step is to record the oracle mean, calculated using information the other methods are not privy to:

$$\hat{\mu}_{\text{oracle}} = \frac{1}{n} \sum_{i=1}^{n} y_i \frac{1}{\pi_i}$$

All subsequent methods are the methods of study, to be performed where there is missingness in the data. 20% of the observations manually have their labels dropped weighted by y, so large labels are more likely to be absent. The result is the form of data we are using to simulate method performance: a complex sample taken from a population with some kind of systematic bias in item response.

The π-corrected mean estimate, weighted linear regression, and median imputation methods are all described fully in chapter 3, and are closed-form algorithms with no randomness in the fitting and estimation processes. The neural network models, however, have a much larger degree of flexibility and randomness involved in the implementation and execution of the estimate during simulation making monte carlo variability more pronounced.

Neural networks are trained using normalized data on non-convex loss curves in high-dimensional space, making their optimization process stochastic. This simulation uses the heuristic `adam` optimizer. Each model in this simulation uses a 2 hidden layer network with 32 hidden units per layer to make results and applications more comparable. Each layer is wrapped in the `relu` activation function:

$$\text{relu}(x) = \max(0, x)$$

The breadth and depth used here is a heuristic size which could potentially be optimized to suit the problem. This, however, creates a conflict of interest with information leakage which the other methods are not privy to, which induces bias for the success of these algorithms. To maintain the narrative of a minimal-knowledge naive fit, these models use a reasonable exploratory capacity. The models in the fitting process are manually over-trained, then re-trained from scratch to an optimal state. This is done using a validation data set of 15% of the training data. Once trained to an approximate loss minima, the models predict on holdout test data (the observations missing labels) and the population mean estimate is computed and recorded.

4.5 Evaluating models

4.5.1 Oracle Mean:

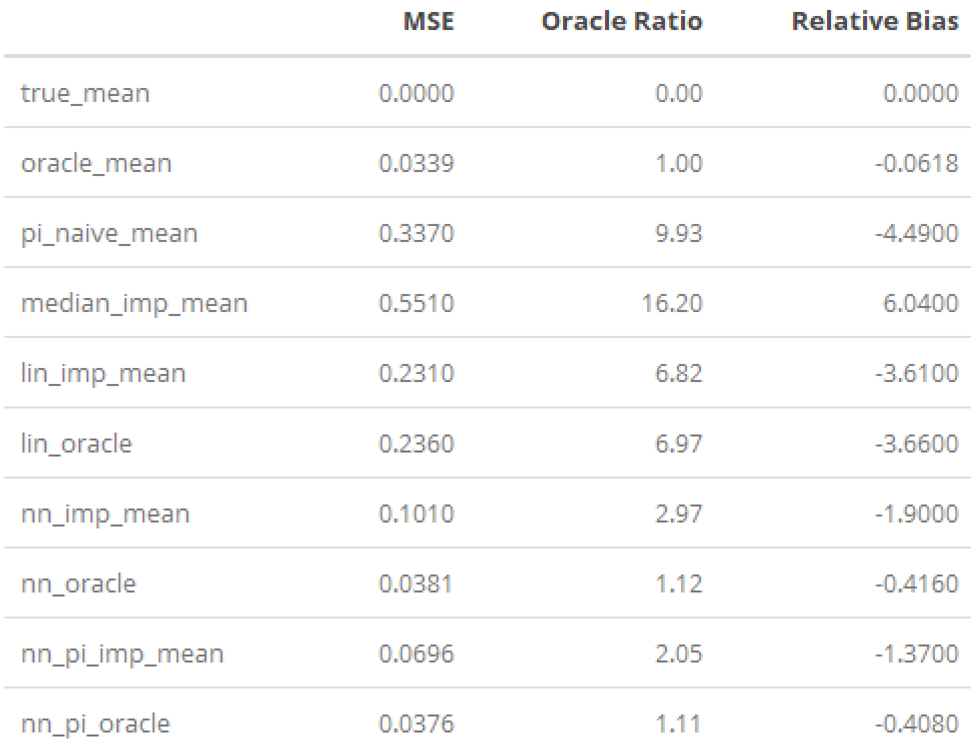
The oracle mean is the Horvitz-Thompson estimator with access to the complete sample before missingness is induced (Horvitz & Thompson, 1952). The oracle mean will be used as a benchmark against which the other methods will be compared.

$$\hat{\mu}_{\text{oracle}} = \frac{1}{N} \sum_{i=1}^{n} y_i \frac{1}{\pi_i}$$

Since the oracle mean has access to information that the other methods do not, it is the ideal estimate of a population mean given a sample. The difference between the oracle and π-corrected estimates are indices n and r over which the sum is taken.

4.6 Results

Not a place to list findings. Rather, convince that what I did is correct, worthy of study, and impactful. Be honest about the generalizability of the work. If the findings are negative, take care to explain why the results still matter (discussion of why linear regression is so dank at mean estimation). Cast the negative result as a useful insight, rather than a problem with the methodology.



	MSE	Oracle Ratio	Relative Bias
true_mean	0.0000	0.00	0.0000
oracle_mean	0.0339	1.00	-0.0618
pi_naive_mean	0.3370	9.93	-4.4900
median_imp_mean	0.5510	16.20	6.0400
lin_imp_mean	0.2310	6.82	-3.6100
lin_oracle	0.2360	6.97	-3.6600
nn_imp_mean	0.1010	2.97	-1.9000
nn_oracle	0.0381	1.12	-0.4160
nn_pi_imp_mean	0.0696	2.05	-1.3700
nn_pi_oracle	0.0376	1.11	-0.4080

The weighted-MSE and derived-parameter methods were left out of the simulation results for runtime concerns and unpromising results on this generative function.

The oracle counterpart for each method outperforms the noisy-parameter method, as is expected. The discrepancy in the linear regression method can be attributed to monte carlo variability, but the finding is significant that these are so similar. Whereas the neural network methods were anticipated to have consistent results despite noisy parameters while the linear regression would struggle, the opposite is true. This could be attributed to the training duration, as a higher number of epochs may have dropped the weights of the noisy parameters to be ignored.

All methods underestimated the population mean, described by the relative bias column of the results. EXCEPT FOR MEDIAN? HOW IS MEDIAN IMPUTATION OVER ESTIMATING! This is reasonable on behalf of the systematic missingness of the data, but each non-naive method attempted to mitigate this amount. The results indicate that the negative bias was significantly larger for non-oracle methods, meaning noisy parameters could contribute to mean under-estimation. For this reason, a researcher with some level of domain knowledge should hand-select features known to have a relationship to the label to be used as training data, rather than including all available features.

In the case of a nonlinear generative function with little noise, it should come as no surprise that most nonlinear neural network methods outperformed the linear.

The π-feature neural network's fantastic performance can be largely attributed to the significant relationship of π to the label y: $\pi = \sqrt{y} + \epsilon$. This strong relationship is not unreasonable, however, as the positive correlation of inclusion probability to label is known to occur in real-world data on salary-related data across BLS business surveys.

Despite a highly nonlinear generative function, the nature of weighted linear regression is highly successful for mean estimation. In the context of the bowl-with-lip shape generative function, the linear regression looks like a flat plane relative to the x_1, x_2 axes slightly above halfway up the bowl. This is an unfortunate side effect of the parabola-shaped true function, as the label predictions of the linear model are bound to this plane. A more realistic nonparabola context would lend itself to considerably more successful linear imputation estimates, as the weighted high-label outliers would be properly represented by the plane's extrema on the domain.

When estimating the population mean, accurate imputation results do not factor into the score. Rather, the mean of the distribution of imputed labels solely defines the score of the algorithm. In the case of mean estimation, linear regression provides an extremely lightweight estimation method in terms of computational time and implementation that cannot overfit to the respondent data. Though among the largest MSE ratios, linear regression should be considered a success in this experiment and taken seriously as the imputation method to beat.

Chapter 5

Consumer Expenditure Surveys

5.1 Data

The Consumer Expenditure (CE) Surveys[1] provide data on "expenditures, income, and demographic characteristics of consumers in the United States. The CE data are collected by the Census Bureau for the Bureau of Labor Statistics (BLS). The data are primary used to revise the relative importance of goods and services in the market basket of the Consumer Price Index".

The data will be the `fmli171x` survey data, one of the quarterly surveys for 2017 which contains household information such as specific incomes, expenditures, home description, and family description. The data has a low degree of missingness as quarterly information is propagated between surveys to create complete observations.

The mean estimation methods from chapter 3 will be used to estimate the population mean household income before tax, `FINCBTAX`. The features used to predict this label are: - `pi`, the inclusion probability (1 / `FINLWT21`, the survey weight provided by BLS) - `AGE_REF`, the age of the reference person (the person who answered the survey) - `BATHRMQ`, the number of bathrooms in the home - `BEDROOMQ`, the number of bedrooms in the home - `EDUC_REF`, the education of the reference person (some highschool, etc) - `FAM_SIZE`, the number of family members in the home - `TOTEXPCQ`, the total household expenditures last quarter

Like all data gathered by BLS `FINCBTAX` and some features are subject to missingness, but have been in the data sets. This means for the purpose of the experiment on method imputation quality, we must artifically impose missingness then compare the imputation results to a combination of true labels and BLS-derived labels. This is an unavoidable pitfall of working with real data, as we paradoxically require "truly missing" labels to impute, but need the true label to compare the quality of the imputation to.

According IRS Statistics of Income, the average household adjusted gross income (AGI) was \$67,565 in 2015. This is the last year the population mean household income is available. This numbercould be considered the true population mean μ_y, but is derived differently than the BLS CE data and is slightly different than the

[1] `https://www.bls.gov/cex/pumd-getting-started-guide.htm`

Horvitz-Thompson estimator for the sample on `FINCBTAX`.

5.2 Procedure

The CE data method performance comparison will be performed in much the same way as the simulated data method comparison.

The `fmli171x` survey data is first pre-processed to remove features with large swathes of missingness. For this study, features with more than 33% missing values are dropped from the data set. Features with missingness less than 33% are then median-imputed, where missing values are replaced with the median value of the feature. The median in this case is used for reasons: it returns only reasonable values, is uninformative, and does not rely on multiple imputation.

The label to be used is `FINCBTAX`, the financial income of the response household before taxes. This label has no missingness as it is BLS-derived (already imputed).

This process returns a complete-case data set of 6208 family samples across 607 variables. There are 127 million households in the US, whcih can be used as an approximate population size N.

The problem of assessing model performance in real-world data is the paradox of missing labels: ideally, we would impute a missing label, then learn the true value, and score the model accordingly. For this data, we will again rely on Monte Carlo simulation to create a distribution of population mean estimates (U.S. mean household income) by inducing missingness in the known labels, imputing, and comparing results via MSE to the true mean.

The following process is repeated a number of times to create a distribution of mean estimates for each method:

1. Record the true (sample) mean and Horvitz-Thompson mean estimate
2. Induce missingness on 20% of the labels, weighted to larger labels
3. Perform and record each model's mean estimate. For this experiment, a more accurate neural network training method is adopted in which each network undergoes two trainings: the first finds the ideal train duration by overtraining on the training data, the second re-trains the model to the validation minimum of the first model. This is the correct training method which was eschewed from Chapter 4 due to the computational intensity of double-training.
4. The results are compared using mean squared error to $'mu_y$, oracle ratio, and percent relative bias.

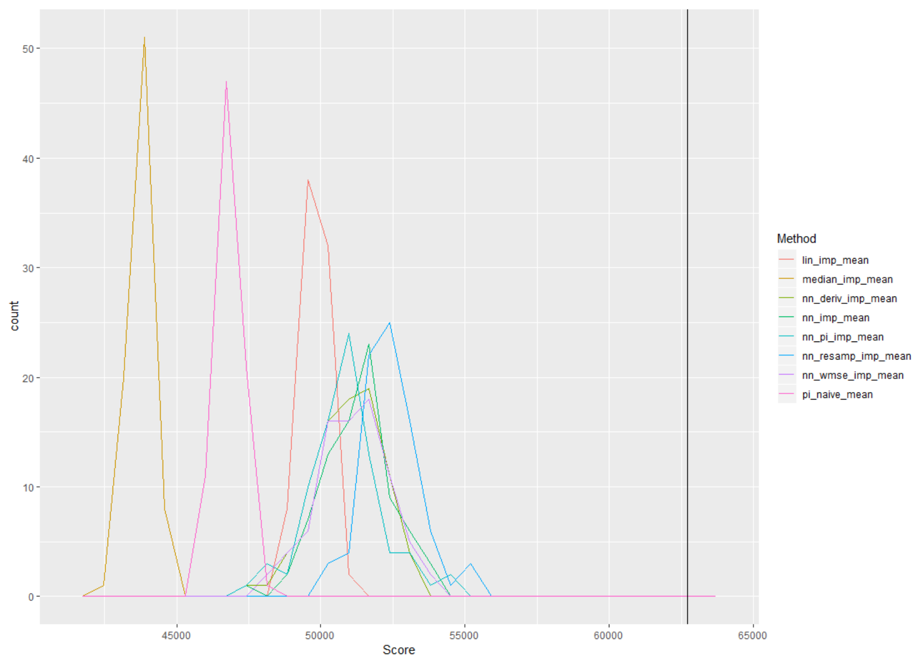
The dimension of the neural networks has changed somewhat to account for the new data. Since the number of informative parameters has increased along with the complexity of the generative function transforming them to the label, the size of the model has increased. The standard real-data neural network uses the same two hidden layers activated by `relu` and a linear output layer, but now has 64 hidden units per layer. This describes a significantly higher dimension model as the input vector $\boldsymbol{x}$ undergoes multiple high dimension linear transformations wrapped in the `relu` activation:

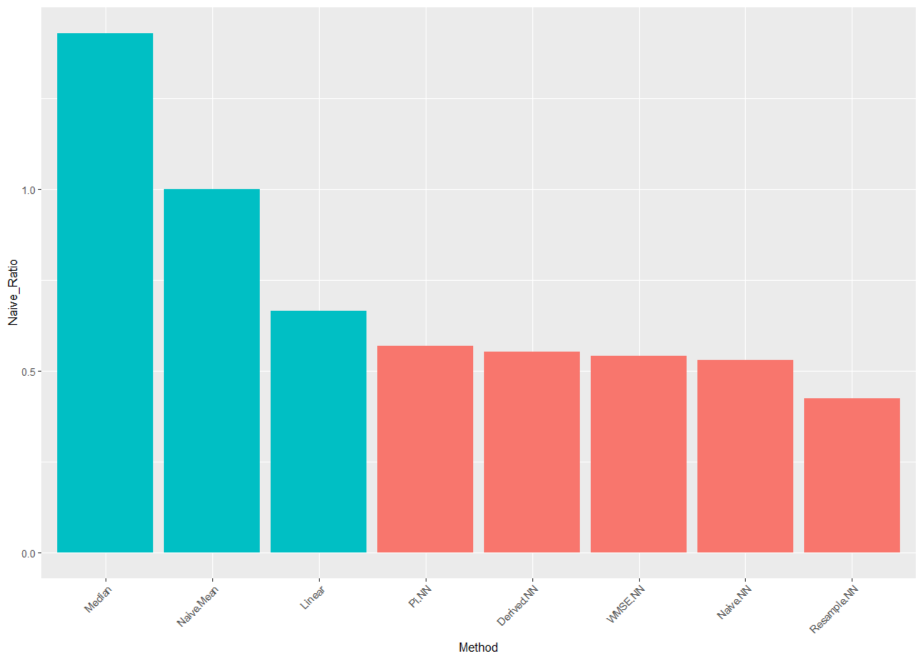
$$\boldsymbol{x} \in \mathbb{R}^6 \to \mathbb{R}^{64} \to \mathbb{R}^{64} \to \mathbb{R}$$

Parameterized by matrices of dimension 6×64, 64×64, 64×1, instead of 6×32, 32×32, 32×1 as in the 32 hidden unit simulation.

To accomidate the significantly higher number of trainable features in this network the hyperparameter ω, the learning rate, must be initialized higher in the `adam` optimization algorithm. This can expedite the number of epochs (training data exposures) needed for the algorithm to converge, but can sacrifice predictive fidelty, a constraint imposed by computational demand. This tradeoff is made in the name of a more statistically informative monte carlo simulation.

5.3 Results




Naive_Ratio
1.0
0.5
0.0
Median
Naive.Mean
Linear
PI.NN
Derived.NN
WMSE.NN
Naive.NN
Resample.NN
Method

Chapter 6

Conclusion

6.1 Discussion

Early simulation results showed neural networks struggling to compete with the stability of weighted linear regression mean estimation. The highly nonlinear relationship of the simulation is a product of extensive neural network testing to produce a positive result indicative of potential for neural network capability. The CE results are suprising in this manner: though more variable, neural networks consistently outperform the parametric linear model for outlier prediction, contributing to accuracy and, consequentially, mean estimations.

These results show clear promise to the capability of neural network imputation for systematic missingness in labels with π correlation. The significant improvement of each neural network method from weighted linear regression is a positive finding, but one that is slightly undermined due to the total success of neural networks. When all nonlinear methods outperform the linear regression, it is clear that there is a strong nonlinearity between the features and label for the CE data given the selected features.

However, that does not mean these results are unimpactful. The success of the neural network methods is that they thrived in the same minimal-knowledge enviroment that linear regression does. The only change from the simulated neural networks to real-data experiments was a larger network breadth and learning rate, both hyperparameters changed for computational time, not superior accuracy. This is highly promising as the assumptions of NN success were derived from the nonparametric benefits of a more flexible model which is highly adaptable to data relationships. The hypotheses on the advantages of this properly trained family of algorithms are demonstratably upgrades for mean estimation and imputation accuracy, even when ignoring survey design.

The lower scores of the π-feature and derived-parameter methods compared to naive neural network imputation is a suprising result. This outcome is at odds with the hypotheses on the feature-selection property outlined in Chapter 2. If the π feature and its derivatives were informative to the label, the methods should have superior accuracy, and if the π feature was uninformative, the weighted-MSE should have underperformed.

These results altogether show the benefits of neural networks in minimal domain knowledge imputation: using heuristic feedforward networks and a validation set, the models unilaterally outperformed a typical weighted regression for mean imputation.

If a researcher is interested in an easy to implement, computationally inexpensive heuristic algorithm to impute complex survey data for mean estimation, weighted linear regression is likely the most appealing method. The reality of real data is that it is more likely noisy than functional.

Neural network methods are promising, however, because of their many degrees flexibility. The nonparametric models are adaptable to more complex intervariable relationships of the data. If a domain expert hypothesized a sophisticated function on a subset of variables which make an excellent label predictor without knowing its specific form, there is room for that structure in the neural network architecture.

Likewise, the massive space of hyperparameters for even a simple feedforward neural network gives immeasurable potential to a savvy designer. Understanding of the shape of the loss function, necessary load of the model, and relationship of features are all examples of knowledge lending itself to specific selections of model load, activations, optimizers, losses, weightings, connectivity, and countless other tweaks letting a knowledgable designer maximize predictive performance. Weighted linear regression on real data has shown demonstratable inferiorty to the parametric family, and lacks this degree of tuning potential. The results of the CE experiment could diverge even further given algorithmic tweaks optimized to validaiton data.

6.2 Conclusion

Neural networks are extremely promising in the realm of imputation. Each neural network method using only heuristic default parameters outperformed the weighted linear regression model significantly in the simulation and Consumer Expenditure mean estimation experiments.

Nonparametric modelling with validation data should outperform linear models, even when survey design information is not incorporated. Weighted resample neural networks outperformed all other algorithms despite sampling and resampling variability, lending them great promise for further investigation, discussed in **??** .

6.3 Future Work

There is massive potential in exploring the applications of neural networks to complex survey imputation. A more competitive model such as local polynomial smoothing using a validation set could be a superior general model for unknown generative functions. The linear regression proved limiting and underfitting in both simulation and CE applications. Any stable heuristic method with some potential to encode survey weights is worth investigating.

Multiple imputation is a world worth exploring due to the superior predictive capability of neural networks and their intervarriable relationship preservation property.

Weighted lienar regression is an excellent mean estimator due to the stability of its predictions to extrema, but lacks in predictive accuracy. Neural networks, however, could offer superior multiple imputations as relationships are nonlinearized through successive predictions.

Though it performed very well on real data with the algorithm outlined in Chapter 3, the weighted resample neural network method could be improved in various ways. The method struggled in low-n simulations due to highly variable resampling. This would be somewhat alleviated by resampling to N weighted by $\frac{1}{\pi}$, but was cut for computational concerns. In addition to N-observation resampling, bootstrapping methods inspired by random forests are a promising exploration. There would be many tweaks herewithin, but the simple algorithm is to create k data frames using the weighted resample `sample_n(sample, weight = 1\pi)`, then train a naive neural network to each of these datasets, and take the mean or median of the resulting imputed labels from testing data as the output prediction. This would alleviate both network and resample variability of the already-dominant algorithm.

Appendix A

The First Appendix

This first appendix includes all of the R chunks of code that were hidden throughout the document (using the `include = FALSE` chunk tag) to help with readibility and/or setup.

This appendix contains every line of code necessary to replicate the results of the two experiments detailed in this thesis. The bulk of each experiment is testing structures used to derive optimal runtime and epoch-time for each of the many methods, and evaluate their performance across a variety of standards, seen at the end of each iteration loop.

A.1 Data Processing

```r
# This chunk ensures that the thesisdown package is
# installed and loaded. This thesisdown package includes
# the template files for the thesis.
if(!require(devtools))
  install.packages("devtools", repos = "http://cran.rstudio.com")
if(!require(thesisdown))
  devtools::install_github("ismayc/thesisdown")
library(thesisdown)
```

A.2 Simulated Data Experiment {sim_data_exp}

```r
# model selection 2
## Adding noise features, new generative function, and
# model selection oracle

# Compare distributions of mean via different imputation methods on artifical da
```

```
library(keras)
library(dplyr)
library(ggplot2)
library(tensorflow)

N = 10^5
n = 10^3
it <- 100

p_y_ep_control <- 1

rescale <- function(vec) { # rescale to 0,1
  (vec - min(vec)) / (max(vec) - min(vec))
}

create_validation_split <- function(x_train, y_train) {
  # Validation Set
  # THIS HAS BEEN UPDATED TO SAMPLE 20% OF X_TRAIN ROWS AS INDECES
  val_indices <- sample(1:nrow(x_train), .15*nrow(df))

  x_val <<- x_train[val_indices,]
  partial_x_train <<- x_train[-val_indices,]

  y_val <<- y_train[val_indices]
  partial_y_train <<- y_train[-val_indices]
}

normalize_data <- function(train_data, test_data) {
  mean <- apply(train_data, 2, mean)
  std <- apply(train_data, 2, sd)

  x_train <<- scale(train_data, center = mean, scale = std)
  x_test <<- scale(test_data, center = mean, scale = std)
}

p_1 <- runif(N, -30, 30)
p_2 <- runif(N, -30, 30)

nulls <- matrix(rnorm(20*N, 0, 8), ncol = 20)

p_y_ep <- rnorm(N, mean = 0, sd = p_y_ep_control)
#p_y <- p_1*p_2 + p_3 + p_y_ep

p_y <- p_1 + abs(p_2) + p_y_ep #added noise for reasonability
```

```
#mpy <- mean(p_y)
#p_y[p_y > mpy] <- p_y[p_y > mpy] + mpy

p_y[p_y > mean(p_y)] <- mean(p_y)

p_y <- p_y + p_y_ep
p_y <- abs(p_y)

p_pi_ep <- rnorm(N, mean = 0, sd = 1)
temp_pi <- sqrt(p_y) + p_pi_ep
temp_pi <- rescale(temp_pi)
p_pi <- temp_pi * (n / sum(temp_pi))

p_df <- cbind(p_1, p_2, p_y, p_pi, nulls)
p_tbl <- as_tibble(p_df)

# Verify sum of pi over population approx n
sum(p_tbl$p_pi) == n#= 10000 = n

statistic_tracker <- data.frame(true_mean = numeric(it),
                                oracle_mean = numeric(it),
                                pi_naive_mean = numeric(it),
                                median_imp_mean = numeric(it),
                                lin_imp_mean = numeric(it),
                                lin_oracle = numeric(it),
                                nn_oracle = numeric(it),
                                nn_pi_oracle = numeric(it),
                                nn_resamp_oracle = numeric(it),
                                nn_wmse_oracle = numeric(it),
                                nn_deriv_oracle = numeric(it))
# Want to estimate:
mu_y <- (1 / N) * sum(p_tbl$p_y)
mu_y
# monte carlo simulation of draws from population with comparison methods
for (i in 1:it) {
  statistic_tracker$true_mean[i] <- mu_y

  sample_population_by_pi <- sample_n(tbl = p_tbl, size = n, replace = FALSE,
                                      weight = p_pi)

  df <- sample_population_by_pi %>% rename(x_1 = p_1, #Since we are not dealing w
                                           x_2 = p_2,
                                           pi = p_pi,
                                           y = p_y)
```

```
#########
# Oracle mean
# this is a standard estimator which is done on the full (nonmissing) data to compar
hat_N_sample <- sum(1 / df$pi)
statistic_tracker$oracle_mean[i] <- (1 / hat_N_sample) * sum((1 / df$pi) * df$y)

#########
# Drop some labels - weighted to high y

indices <- sample(1:nrow(df), .2*nrow(df), prob = df$y) #weights are stdized to 1 by
dropped_obs <- df[indices,]
reduced_df <- df[-indices,] # These have a large differnce: 296 for dropped and 276 f

# Make oracle df without noisy parameters
odf <- df[,1:4]
o_dropped_obs <- odf[indices,]
o_reduced_df <- odf[-indices,]

#########
# pi-corrected naive mean
# should be unbiased but very large variance. should come forward when gen func chan
## NOTE DIFFERENCE OF HAT N RESPOND AND HAT N SAMPLE
hat_N_respondent <- sum(1 / (reduced_df$pi)) # largely underestimates the true N
statistic_tracker$pi_naive_mean[i] <- (1 / hat_N_respondent)*sum((1 / reduced_df$pi) *
                                                                  reduced_df$y)

#########
# Median imputation: fill missing values with median
len <- dim(dropped_obs)[1]
median_list <- rep(median(reduced_df$y), len)
labels <- as.vector(reduced_df$y)
median_list <- as.vector(median_list)

imputed_list <- c(labels,median_list)

statistic_tracker$median_imp_mean[i] <- (1 / hat_N_sample) * sum((1 / df$pi) *
                                                                  imputed_list)

#########
# mean according to imputed data via linear regression (1/pi-weighted)
red_weight <- 1 / reduced_df$pi
lin_dat <- select(reduced_df, -c(pi))
lin_dropped <- select(dropped_obs, -c(pi))
```

```
linear_model <- lm(y ~ ., data = lin_dat, weights = red_weight)

lm_y_hat <- predict(linear_model, lin_dropped)
hat_N_sample <- sum(1/df$pi)
statistic_tracker$lin_imp_mean[i] <- (1 / hat_N_sample)*(sum(reduced_df$y / redu
                                                          + sum(lm_y_hat / droppe

# Linear Regression Oracle Method
red_weight <- 1 / o_reduced_df$pi
lin_dat <- select(o_reduced_df, -c(pi))
lin_dropped <- select(o_dropped_obs, -c(pi))

linear_model <- lm(y ~ ., data = lin_dat, weights = red_weight)

lm_y_hat <- predict(linear_model, lin_dropped)
hat_N_sample <- sum(1/odf$pi)
statistic_tracker$lin_oracle[i] <- (1 / hat_N_sample)*(sum(o_reduced_df$y / o_re
                                                        + sum(lm_y_hat / o_droppe



# nn oracle
y_train <- o_reduced_df$y
reduced_df_nolab <- select(o_reduced_df, -c(pi, y))

y_test <- o_dropped_obs$y
dropped_obs_nolab <- select(o_dropped_obs, -c(pi,y))

reduced_df_nolab <- as.matrix(reduced_df_nolab)
dropped_obs_nolab <- as.matrix(dropped_obs_nolab)

x_train <- reduced_df_nolab
x_test <- dropped_obs_nolab

normalize_data(x_train, x_test)
create_validation_split(x_train, y_train)

model <- keras_model_sequential() %>%
  layer_dense(units = 32, activation = "relu",
              input_shape = dim(x_train)[[2]]) %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1)

model %>% compile(
  optimizer = "adam",
  loss = "mse",
  metrics = c("mae")
)

history <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 170,
  verbose = 0,
  validation_data = list(x_val, y_val)
)

x_test <- as.matrix(x_test)
nn_y_hat <- predict(model, x_test)

hat_N_sample <- sum(1/df$pi)
statistic_tracker$nn_oracle[i] <- (1 / hat_N_sample)*(sum(o_reduced_df$y / o_reduced_d
                                                       + sum(nn_y_hat / o_dropped_obs$p


#oracle nn pi feature
y_train <- o_reduced_df$y
reduced_df_nolab <- select(o_reduced_df, -c(y))
reduced_df_nolab$pi <- 1 / reduced_df_nolab$pi

y_test <- o_dropped_obs$y
dropped_obs_nolab <- select(o_dropped_obs, -c(y))
dropped_obs_nolab$pi <- 1 / dropped_obs_nolab$pi

reduced_df_nolab <- as.matrix(reduced_df_nolab)
dropped_obs_nolab <- as.matrix(dropped_obs_nolab)

x_train <- reduced_df_nolab
x_test <- dropped_obs_nolab

normalize_data(x_train, x_test)
create_validation_split(x_train, y_train)

model <- keras_model_sequential() %>%
  layer_dense(units = 32, activation = "relu",
              input_shape = dim(x_train)[[2]]) %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1)
```

```
model %>% compile(
  optimizer = "adam",
  loss = "mse",
  metrics = c("mae")
)

history <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 300,
  verbose = 0,
  batch_size = 32,
  validation_data = list(x_val, y_val)
)

x_test <- as.matrix(x_test)
nn_pi_y_hat <- predict(model, x_test)

hat_N_sample <- sum(1/df$pi)
statistic_tracker$nn_pi_oracle[i] <- (1 / hat_N_sample)*(sum(o_reduced_df$y / o_
                                                        + sum(nn_pi_y_hat / o_d

# oracle resample
dropped_obs_NA_lab <- o_dropped_obs
dropped_obs_NA_lab$y <- NA

orig_df <- rbind(o_reduced_df, dropped_obs_NA_lab)

weight_vec <- 1 / as.numeric(orig_df$pi)

orig_tbl <- as_tibble(orig_df)

# RESAMPLE ON DF NOT REDUCED_DF
resamp_df <- sample_n(tbl = orig_tbl, size = nrow(orig_tbl), replace = TRUE, wei

# re-partition into complete cases, and cases to be imputed
resamp_reduced_df <- resamp_df[-which(is.na(resamp_df$y)),]
resamp_dropped_obs <- resamp_df[which(is.na(resamp_df$y)),]

y_train <- resamp_reduced_df$y
resamp_reduced_df_nolab <- select(resamp_reduced_df, -c(y))

y_test <- resamp_dropped_obs$y
resamp_dropped_obs_nolab <- select(resamp_dropped_obs, -c(y))
```

```
resamp_reduced_df_nolab <- as.matrix(resamp_reduced_df_nolab)
resamp_dropped_obs_nolab <- as.matrix(resamp_dropped_obs_nolab)

x_train <- resamp_reduced_df_nolab
x_test <- resamp_dropped_obs_nolab

create_validation_split(x_train, y_train)

model <- keras_model_sequential() %>%
  layer_dense(units = 32, activation = "relu",
              input_shape = dim(x_train)[[2]]) %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1)

model %>% compile(
  optimizer = "adam",
  loss = "mse",
  metrics = c("mae")
)

history <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 200,
  verbose = 0,
  batch_size = 32,
  validation_data = list(x_val, y_val)
)

x_test <- as.matrix(x_test)
nn_resamp_y_hat <- predict(model, x_test)

hat_N_sample <- sum(1/df$pi)
statistic_tracker$nn_resamp_oracle[i] <- (1 / hat_N_sample)*(sum(reduced_df$y / reduce
                                                                + sum(nn_resamp_y_hat /

# WSME NN
#######
# New Split
# Inherit train and test
train <- reduced_df
test <- dropped_obs

# record training data pi
```

```r
pi_vec <- train$pi

# Inherit train and test
train <- reduced_df
y_train <- train$FINCBTAX
x_train <- select(train, -c(FINCBTAX))

test <- dropped_obs
y_test <- test$FINCBTAX
x_test <- select(test, -c(FINCBTAX))

# Split train into train' and val
val_indices <- sample(1:nrow(train), .20*nrow(df))

x_val <- x_train[val_indices,]
partial_x_train <- x_train[-val_indices,]

y_val <- y_train[val_indices]
partial_y_train <- y_train[-val_indices]


# Extract partial pi from full train x
#range01 <- function(x){(x-min(x))/(max(x)-min(x))}

pi_prime <- pi_vec[-val_indices]
obs_weights <- 1 / pi_prime

full_obs_weights <- 1 / pi_vec

# remove pi from x_test and x_train, partial_x_train, x_val
x_test <- select(x_test, -c(pi))
x_train <- select(x_train, -c(pi))
x_val <- select(x_val, -c(pi))
partial_x_train <- select(partial_x_train, -c(pi))

# Normalize all WRT train
mean <- apply(x_train, 2, mean)
std <- apply(x_train, 2, sd)

x_train <- scale(x_train, center = mean, scale = std)
x_test <- scale(x_test, center = mean, scale = std)
x_val <- scale(x_val, center = mean, scale = std)
partial_x_train <- scale(partial_x_train, center = mean, scale = std)
```

```
model <- keras_model_sequential() %>%
  layer_dense(units = 32, activation = "relu",
              input_shape = dim(x_train)[[2]]) %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(units = 1)

model %>% compile(
  optimizer = "adam",
  loss = "mse",
  metrics = c("mae")
)

history <- model %>% fit(
  partial_x_train,
  partial_y_train,
  sample_weight = obs_weights,
  epochs = 250,
  verbose = 0,
  batch_size = 32,
  validation_data = list(x_val, y_val)
)

x_test <- as.matrix(x_test)
x_test <- x_test[,-3]
nn_y_hat <- predict(model, x_test)

hat_N_sample <- sum(1/df$pi)
statistic_tracker$nn_wmse_oracle[i] <- (1 / hat_N_sample)*(sum(o_reduced_df$y / o_redu
                                                           + sum(nn_y_hat / o_dropped_


# Oracle derived params
y_train <- o_reduced_df$y
reduced_df_nolab <- select(o_reduced_df, -c(y))

y_test <- o_dropped_obs$y
dropped_obs_nolab <- select(o_dropped_obs, -c(y))

x1pi <- (reduced_df_nolab$x_1)*(reduced_df_nolab$pi)
x2pi <- (reduced_df_nolab$x_2)*(reduced_df_nolab$pi)

dx1pi <- (dropped_obs_nolab$x_1)*(dropped_obs_nolab$pi)
dx2pi <- (dropped_obs_nolab$x_2)*(dropped_obs_nolab$pi)
```

```
  reduced_df_nolab <- cbind(reduced_df_nolab, x1pi, x2pi)
  dropped_obs_nolab <- cbind(dropped_obs_nolab, dx1pi, dx2pi)

  reduced_df_nolab <- as.matrix(reduced_df_nolab)
  dropped_obs_nolab <- as.matrix(dropped_obs_nolab)

  x_train <- reduced_df_nolab
  x_test <- dropped_obs_nolab

  normalize_data(x_train, x_test)
  create_validation_split(x_train, y_train)

  model <- keras_model_sequential() %>%
    layer_dense(units = 32, activation = "relu",
                input_shape = dim(x_train)[[2]]) %>%
    layer_dense(units = 32, activation = "relu") %>%
    layer_dense(units = 1)

  model %>% compile(
    optimizer = "adam",
    loss = "mse",
    metrics = c("mae"))

  history <- model %>% fit(
    partial_x_train,
    partial_y_train,
    epochs = 300, #high variance; study further
    verbose = 0,
    batch_size = 32,  #what should this be
    validation_data = list(x_val, y_val)
  )

  x_test <- as.matrix(x_test)
  nn_y_hat <- predict(model, x_test)

  hat_N_sample <- sum(1/df$pi)
  statistic_tracker$nn_deriv_oracle[i] <- (1 / hat_N_sample)*(sum(reduced_df$y / r
                                                                + sum(nn_y_hat / d


  print(i)
}

# Save the mean table so that we don't have to always re-run it
```

```
write.csv(statistic_tracker, file = "C:\\Users\\Alexander\\Documents\\thesis stat tracke
dat <- statistic_tracker

#dat <- read.csv("c:/Users/Alexander/Documents/thesis stat tracker/10_full.csv")
#dat <- dat[,-1]

#### COMPUTING MSE (and making table)
# MSE against TRUE mean for each method:
# take dif of each with true. square each. sum all. divide by iterations

mse_table <- dat[1,]
for (i in 1:dim(dat)[2]) {

  matdat <- as.matrix(dat)

  mse_table[i] <- mean( (matdat[,i] - matdat[,1])^2 )
}

mse_table

# then do MSE of method divided by MSE of oracle to get measure of goodness
# (closer to 1 means closer to oracle means good)

oracle_ratio_table <- dat[1,]

for (i in 1:dim(dat)[2]) {

  oracle_ratio_table[i] <- mse_table[i] / mse_table[2]
}

oracle_ratio_table

# compute bias:
# mean of statistic vector - true mean
# divide this differnce by true mean
# multiple by 100 to get "percent relative bais", a normalized bias measure

prb_table <- dat[1,]
mu_y <- mean(dat[,1])

for (i in 1:dim(dat)[2]) {

  prb_table[i] <- 100 * ((mean(dat[,i]) - mu_y) / mu_y)
}
```

```r
prb_table

library(kableExtra)
library(knitr)

mse_t <- transpose(mse_table)
oracle_t <- transpose(oracle_ratio_table)
prb_t <- transpose(prb_table)

binded <- cbind(mse_t, oracle_t, prb_t)
colnames(binded) <- c("MSE", "Oracle Ratio", "Relative Bias")
rownames(binded) <- colnames(statistic_tracker)

test <- binded
test <- signif(test, digits = 3)

test %>%
  kable() %>%
  kable_styling()
```

A.3 CE Data Experiment {ce_data_exp}

```r
## Real Experiments in CE Data
# Documentation:
# https://www.bls.gov/cex/2017/csxintvwdata.pdf

library(readr)
library(dplyr)
library(keras)
library(kableExtra)
library(knitr)
library(ggplot2)
library(tensorflow)
library(data.table)

# Relevant functions
count_nas <- function(df) {
  isna <- apply(df, 2, is.na)
  tots <- apply(isna, 2, sum)
  tots
}
```

```r
rescale <- function(vec) { # rescale to 0,1
  (vec - min(vec)) / (max(vec) - min(vec))
}

create_validation_split <- function(x_train, y_train) {
  # Validation Set
  val_indices <- sample(1:nrow(x_train), .2*nrow(df))

  x_val <<- x_train[val_indices,]
  partial_x_train <<- x_train[-val_indices,]

  y_val <<- y_train[val_indices]
  partial_y_train <<- y_train[-val_indices]
}

normalize_data <- function(train_data, test_data) {
  mean <- apply(train_data, 2, mean)
  std <- apply(train_data, 2, sd)

  x_train <<- scale(train_data, center = mean, scale = std)
  x_test <<- scale(test_data, center = mean, scale = std)
}

# custom `adam` optimizer, learning rate up for grabs
# .001 default, .005 testing (methods were using 600 epo)
adam_lr <- optimizer_adam(lr = 0.1, beta_1 = 0.9, beta_2 = 0.999,
                          epsilon = NULL, decay = 0, amsgrad = FALSE, clipnorm = NULL,
                          clipvalue = NULL)

dat <- read_csv("Data/imputed_CE.csv")

# Assuming data is in the correct shape, here's how the real testing goes down
it <- 20

statistic_tracker <- data.frame(true_mean = numeric(it),
                                oracle_mean = numeric(it),
                                pi_naive_mean = numeric(it),
                                median_imp_mean = numeric(it),
                                lin_imp_mean = numeric(it),
                                nn_imp_mean = numeric(it),
                                nn_pi_imp_mean = numeric(it),
                                nn_resamp_imp_mean = numeric(it),
                                nn_wmse_imp_mean = numeric(it),
                                nn_deriv_imp_mean = numeric(it))
```

```
# Prep label
df <- select(dat, c(FINCBTAX, pi, AGE_REF, BATHRMQ, BEDROOMQ, EDUC_REF, FAM_SIZE,
                    TOTEXPCQ))

#df <- df[1:1000,]

#label <- "FINCBTAX"
#label_index <- which(colnames(df) == label)

mu_y <- mean(df$FINCBTAX)
mu_y # == mean american household income :)

testing <- TRUE
#epo <- 200

for (i in 1:it) {

  # Split the CE data into training and testing. Might want some kind of scheme
  # get the right amount of high-pi and low-pi observations in each
  # (testing is unlabelled, but we know the right answer)

  #########
  # True mean
  statistic_tracker$true_mean[i] <- mu_y

  #########
  # Oracle mean
  # this is a standard estimator which is done on the full (nonmissing) data to
  hat_N_sample <- sum(1 / df$pi)
  statistic_tracker$oracle_mean[i] <- (1 / hat_N_sample) * sum((1 / df$pi) * df$FI

  #########
  # Drop some labels - weighted to high y
  zeros <- rep(0, nrow(df))
  rec <- pmax(zeros, df$FINCBTAX)

  indices <- sample(1:nrow(df), .20*nrow(df), prob = rec)

  # Make df with all features (noisy)
  dropped_obs <- df[indices,]
  reduced_df <- df[-indices,]

  #########
  # pi-weighted naive mean
# non-imputation of estimate which accounts for complex design but ignores systemati
hat_N_respondent <- sum(1 / (reduced_df$pi))
statistic_tracker$pi_naive_mean[i] <- (1 / hat_N_respondent)*sum((1 / reduced_df$pi) *
                                                                   reduced_df$FINCBTAX

# Compute median imputation mean estimate

#########
# Median imputation: fill missing values with median
len <- dim(dropped_obs)[1]
median_list <- rep(median(reduced_df$FINCBTAX), len)
labels <- as.vector(reduced_df$FINCBTAX)
median_list <- as.vector(median_list)

imputed_list <- c(labels,median_list)

statistic_tracker$median_imp_mean[i] <- (1 / hat_N_sample) * sum((1 / df$pi) *
                                                                   imputed_list)

#########
# Linear regression imputation: MSE 1/pi weighted
red_weight <- 1 / reduced_df$pi
lin_dat <- select(reduced_df, -c(pi))
lin_dropped <- select(dropped_obs, -c(pi))

linear_model <- lm(lin_dat$FINCBTAX ~ ., data = lin_dat, weights = red_weight)

lm_y_hat <- predict(linear_model, lin_dropped) # THROWS WARN
hat_N_sample <- sum(1/df$pi)
statistic_tracker$lin_imp_mean[i] <- (1 / hat_N_sample)*(sum(reduced_df$FINCBTAX / red
                                                          + sum(lm_y_hat / dropped_obs$

#########
# Mean accoridng to imputed data via naive neural network (ignore complex design)
y_train <- reduced_df$FINCBTAX
reduced_df_nolab <- select(reduced_df, -c(pi, FINCBTAX))

y_test <- dropped_obs$FINCBTAX
dropped_obs_nolab <- select(dropped_obs, -c(pi, FINCBTAX))

reduced_df_nolab <- as.matrix(reduced_df_nolab)
dropped_obs_nolab <- as.matrix(dropped_obs_nolab)

x_train <- reduced_df_nolab
```

```
x_test <- dropped_obs_nolab

normalize_data(x_train, x_test)
create_validation_split(x_train, y_train)

if(testing){
  # Intentionally over-train on the training data to find the validation minim
  model <- keras_model_sequential() %>%
    layer_dense(units = 64, activation = "relu",
                input_shape = dim(x_train)[[2]]) %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 1)

  model %>% compile(
    optimizer = adam_lr,
    loss = "mse",
    metrics = c("mae")
  )

  epo <- 150
  history <- model %>% fit(
    partial_x_train,
    partial_y_train,
    epochs = epo,
    verbose = 0,
    validation_data = list(x_val, y_val)
  )

  goodtrain <- which.min(history$metrics$val_loss)
  print(goodtrain)
  print("of")
  print(epo)

  # Train a new model for the validation-minimizing number of epochs
  model <- keras_model_sequential() %>%
    layer_dense(units = 64, activation = "relu",
                input_shape = dim(x_train)[[2]]) %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 1)

  model %>% compile(
    optimizer = adam_lr,
    loss = "mse",
    metrics = c("mae")
    )

    history <- model %>% fit(
      x_train,
      y_train,
      epochs = goodtrain,
      verbose = 0
      #batch_size = 32,
      #validation_data = list(x_val, y_val)
    )
    plot(history)
  } else {

    # Train a new model for the validation-minimizing number of epochs
    model <- keras_model_sequential() %>%
      layer_dense(units = 64, activation = "relu",
                  input_shape = dim(x_train)[[2]]) %>%
      layer_dense(units = 64, activation = "relu") %>%
      layer_dense(units = 1)

    model %>% compile(
      optimizer = adam_lr,
      loss = "mse",
      metrics = c("mae")
    )

    history <- model %>% fit(
      x_train,
      y_train,
      epochs = 419,
      verbose = 0
      #callbacks = EarlyStopping()
      #batch_size = 32,
      #validation_data = list(x_val, y_val)
    )
  }

  x_test <- as.matrix(x_test)
  nn_y_hat <- predict(model, x_test)

  hat_N_sample <- sum(1/df$pi)
  statistic_tracker$nn_imp_mean[i] <- (1 / hat_N_sample)*(sum(reduced_df$FINCBTAX / redu
                                                          + sum(nn_y_hat / dropped_obs$p
```

```r
print("Finished nn imp mean")

#########
# Mean according to imputed data via neural network w pi feature
y_train <- reduced_df$FINCBTAX
reduced_df_nolab <- select(reduced_df, -c(FINCBTAX))

y_test <- dropped_obs$FINCBTAX
dropped_obs_nolab <- select(dropped_obs, -c(FINCBTAX))

reduced_df_nolab <- as.matrix(reduced_df_nolab)
dropped_obs_nolab <- as.matrix(dropped_obs_nolab)

x_train <- reduced_df_nolab
x_test <- dropped_obs_nolab

normalize_data(x_train, x_test)
create_validation_split(x_train, y_train)

if(testing){
  # intentionally over-train the model
  model <- keras_model_sequential() %>%
    layer_dense(units = 64, activation = "relu",
                input_shape = dim(x_train)[[2]]) %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 1)


  model %>% compile(
    optimizer = adam_lr,
    loss = "mse",
    metrics = c("mae")
  )

  epo <- 100
  history <- model %>% fit(
    partial_x_train,
    partial_y_train,
    epochs = epo,
    verbose = 0,
    validation_data = list(x_val, y_val)
  )

  goodtrain <- which.min(history$metrics$val_loss)
```

```
    print(goodtrain)
    print("of")
    print(epo)

    # re-train the new model to the val-min
    model <- keras_model_sequential() %>%
      layer_dense(units = 64, activation = "relu",
                  input_shape = dim(x_train)[[2]]) %>%
      layer_dense(units = 64, activation = "relu") %>%
      layer_dense(units = 1)

    model %>% compile(
      optimizer = adam_lr,
      loss = "mse",
      metrics = c("mae")
    )

    history <- model %>% fit(
      x_train,
      y_train,
      epochs = goodtrain,
      verbose = 0
    )
    plot(history)

  } else {

    # re-train the new model to the val-min
    model <- keras_model_sequential() %>%
      layer_dense(units = 64, activation = "relu",
                  input_shape = dim(x_train)[[2]]) %>%
      layer_dense(units = 64, activation = "relu") %>%
      layer_dense(units = 1)

    model %>% compile(
      optimizer = adam_lr,
      loss = "mse",
      metrics = c("mae")
    )

    history <- model %>% fit(
      x_train,
      y_train,
      epochs = 218,
    verbose = 0
    #batch_size = 32,
    #validation_data = list(x_val, y_val)
  )
}

x_test <- as.matrix(x_test)
nn_pi_y_hat <- predict(model, x_test)

hat_N_sample <- sum(1/df$pi)
statistic_tracker$nn_pi_imp_mean[i] <- (1 / hat_N_sample)*(sum(reduced_df$FINCBT
                                                               + sum(nn_pi_y_hat / d

print("Finished nn pi imp mean")


# WSME NN
#######
# New Split
# Inherit train and test
train <- reduced_df
test <- dropped_obs

# record training data pi
pi_vec <- train$pi

# Inherit train and test
train <- reduced_df
y_train <- train$FINCBTAX
x_train <- select(train, -c(FINCBTAX))

test <- dropped_obs
y_test <- test$FINCBTAX
x_test <- select(test, -c(FINCBTAX))

# Split train into train' and val
val_indices <- sample(1:nrow(train), .20*nrow(df))

x_val <- x_train[val_indices,]
partial_x_train <- x_train[-val_indices,]

y_val <- y_train[val_indices]
partial_y_train <- y_train[-val_indices]
```

```r
# Extract partial pi from full train x
#range01 <- function(x){(x-min(x))/(max(x)-min(x))}

pi_prime <- pi_vec[-val_indices]
obs_weights <- 1 / pi_prime

full_obs_weights <- 1 / pi_vec

# remove pi from x_test and x_train, partial_x_train, x_val
x_test <- select(x_test, -c(pi))
x_train <- select(x_train, -c(pi))
x_val <- select(x_val, -c(pi))
partial_x_train <- select(partial_x_train, -c(pi))

# Normalize all WRT train
mean <- apply(x_train, 2, mean)
std <- apply(x_train, 2, sd)

x_train <- scale(x_train, center = mean, scale = std)
x_test <- scale(x_test, center = mean, scale = std)
x_val <- scale(x_val, center = mean, scale = std)
partial_x_train <- scale(partial_x_train, center = mean, scale = std)

#######

if(testing){
  # intentionally overtrain
  model <- keras_model_sequential() %>%
    layer_dense(units = 64, activation = "relu",
                input_shape = dim(partial_x_train)[[2]]) %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 1)

  model %>% compile(
    optimizer = adam_lr,
    loss = "mse",
    metrics = c("mae")
  )

  epo <- 100
  history <- model %>% fit(
    partial_x_train,
    partial_y_train,
    sample_weight = obs_weights,
      epochs = epo,
      verbose = 0,
      validation_data = list(x_val, y_val)
    )

    goodtrain <- which.min(history$metrics$val_loss)
    print(goodtrain)
    print("of")
    print(epo)

    # retrain
    model <- keras_model_sequential() %>%
      layer_dense(units = 64, activation = "relu",
                  input_shape = dim(x_train)[[2]]) %>%
      layer_dense(units = 64, activation = "relu") %>%
      layer_dense(units = 1)

    model %>% compile(
      optimizer = adam_lr,
      loss = "mse",
      metrics = c("mae")
    )

    history <- model %>% fit(
      x_train,
      y_train,
      epochs = goodtrain,
      verbose = 0,
      sample_weight = full_obs_weights
    )
    plot(history)

  } else {

    # retrain
    model <- keras_model_sequential() %>%
      layer_dense(units = 64, activation = "relu",
                  input_shape = dim(x_train)[[2]]) %>%
      layer_dense(units = 64, activation = "relu") %>%
      layer_dense(units = 1)

    model %>% compile(
      optimizer = adam_lr,
      loss = "mse",
    metrics = c("mae")
  )

  history <- model %>% fit(
    x_train,
    y_train,
    epochs = 209,
    verbose = 0,
    sample_weight = full_obs_weights
  )
}

nn_y_hat <- predict(model, x_test)

hat_N_sample <- sum(1/df$pi)
statistic_tracker$nn_wmse_imp_mean[i] <- (1 / hat_N_sample)*(sum(reduced_df$FINCBTAX /
                                                              + sum(nn_y_hat / dropped_

print("Finished nn wmse mean")


##########
# Mean according to imputed dataset via neural network with weighted resample
# on the full sample. the missing values in the resample are then imputed and
# the imputation mean is taken on the new data set
# (without pi feature)
dropped_obs_NA_lab <- dropped_obs
dropped_obs_NA_lab$FINCBTAX <- NA

orig_df <- rbind(reduced_df, dropped_obs_NA_lab)

# re-sample by inclusion probability
weight_vec <- 1 / as.numeric(orig_df$pi)

orig_tbl <- as_tibble(orig_df)

# RESAMPLE ON DF NOT REDUCED_DF
resamp_df <- sample_n(tbl = orig_tbl, size = nrow(orig_tbl), replace = TRUE, weight =

# re-partition into complete cases, and cases to be imputed
resamp_reduced_df <- resamp_df[-which(is.na(resamp_df$FINCBTAX)),]
resamp_dropped_obs <- resamp_df[which(is.na(resamp_df$FINCBTAX)),]

y_train <- resamp_reduced_df$FINCBTAX
```

```
resamp_reduced_df_nolab <- select(resamp_reduced_df, -c(FINCBTAX))

y_test <- resamp_dropped_obs$FINCBTAX
resamp_dropped_obs_nolab <- select(resamp_dropped_obs, -c(FINCBTAX))

resamp_reduced_df_nolab <- as.matrix(resamp_reduced_df_nolab)
resamp_dropped_obs_nolab <- as.matrix(resamp_dropped_obs_nolab)

x_train <- resamp_reduced_df_nolab
x_test <- resamp_dropped_obs_nolab

normalize_data(x_train, x_test)
create_validation_split(x_train, y_train)

if(testing){
  model <- keras_model_sequential() %>%
    layer_dense(units = 64, activation = "relu",
                input_shape = dim(x_train)[[2]]) %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 1)

  model %>% compile(
    optimizer = adam_lr,
    loss = "mse",
    metrics = c("mae")
  )

  epo <- 200
  history <- model %>% fit(
    partial_x_train,
    partial_y_train,
    epochs = epo,
    verbose = 0,
    validation_data = list(x_val, y_val)
  )

  goodtrain <- which.min(history$metrics$val_loss)
  print(goodtrain)
  print("of")
  print(epo)

  # retrain
  model <- keras_model_sequential() %>%
    layer_dense(units = 64, activation = "relu",
                  input_shape = dim(x_train)[[2]]) %>%
      layer_dense(units = 64, activation = "relu") %>%
      layer_dense(units = 1)

    model %>% compile(
      optimizer = adam_lr,
      loss = "mse",
      metrics = c("mae")
    )

    history <- model %>% fit(
      x_train,
      y_train,
      epochs = goodtrain,
      verbose = 0
    )
    plot(history)

  } else {

    # retrain
    model <- keras_model_sequential() %>%
      layer_dense(units = 64, activation = "relu",
                  input_shape = dim(x_train)[[2]]) %>%
      layer_dense(units = 64, activation = "relu") %>%
      layer_dense(units = 1)

    model %>% compile(
      optimizer = adam_lr,
      loss = "mse",
      metrics = c("mae")
    )

    history <- model %>% fit(
      x_train,
      y_train,
      epochs = 800,
      verbose = 0
      #batch_size = 32,
      #validation_data = list(x_val, y_val)
    )
  }

  x_test <- as.matrix(x_test)
```

```
nn_resamp_y_hat <- predict(model, x_test)

hat_N_sample <- sum(1/df$pi)
statistic_tracker$nn_resamp_imp_mean[i] <- (1 / hat_N_sample)*(sum(reduced_df$FI
                                                               + sum(nn_resamp_y

print("Finished resamp imp mean")


########
# Derived-parameter NN imputation, where derived parameters are ??

y_train <- reduced_df$FINCBTAX
reduced_df_nolab <- select(reduced_df, -c(FINCBTAX))

y_test <- dropped_obs$FINCBTAX
dropped_obs_nolab <- select(dropped_obs, -c(FINCBTAX))

x1pi <- (reduced_df_nolab$TOTEXPCQ)*(reduced_df_nolab$pi)
x2pi <- (reduced_df_nolab$AGE_REF)*(reduced_df_nolab$pi)

dx1pi <- (dropped_obs_nolab$TOTEXPCQ)*(dropped_obs_nolab$pi)
dx2pi <- (dropped_obs_nolab$AGE_REF)*(dropped_obs_nolab$pi)

reduced_df_nolab <- cbind(reduced_df_nolab, x1pi, x2pi)
dropped_obs_nolab <- cbind(dropped_obs_nolab, dx1pi, dx2pi)

reduced_df_nolab <- as.matrix(reduced_df_nolab)
dropped_obs_nolab <- as.matrix(dropped_obs_nolab)

x_train <- reduced_df_nolab
x_test <- dropped_obs_nolab

normalize_data(x_train, x_test)
create_validation_split(x_train, y_train)

if(testing){
  model <- keras_model_sequential() %>%
    layer_dense(units = 64, activation = "relu",
                input_shape = dim(x_train)[[2]]) %>%
    layer_dense(units = 64, activation = "relu") %>%
    layer_dense(units = 1)

  model %>% compile(
      optimizer = adam_lr,
      loss = "mse",
      metrics = c("mae"))

    epo <- 100
    history <- model %>% fit(
      partial_x_train,
      partial_y_train,
      epochs = epo,
      verbose = 0,
      #batch_size = 32,
      validation_data = list(x_val, y_val)
    )

    goodtrain <- which.min(history$metrics$val_loss)
    print(goodtrain)
    print("of")
    print(epo)

    # retrain
    model <- keras_model_sequential() %>%
      layer_dense(units = 64, activation = "relu",
                  input_shape = dim(x_train)[[2]]) %>%
      layer_dense(units = 64, activation = "relu") %>%
      layer_dense(units = 1)

    model %>% compile(
      optimizer = adam_lr,
      loss = "mse",
      metrics = c("mae")
    )

    history <- model %>% fit(
      x_train,
      y_train,
      epochs = goodtrain,
      verbose = 0
      #batch_size = 32,
      #validation_data = list(x_val, y_val)
    )
    plot(history)

  } else {
    # retrain
    model <- keras_model_sequential() %>%
      layer_dense(units = 64, activation = "relu",
                  input_shape = dim(x_train)[[2]]) %>%
      layer_dense(units = 64, activation = "relu") %>%
      layer_dense(units = 1)

    model %>% compile(
      optimizer = adam_lr,
      loss = "mse",
      metrics = c("mae")
    )

    history <- model %>% fit(
      x_train,
      y_train,
      epochs = 165,
      verbose = 0
      #batch_size = 32,
      #validation_data = list(x_val, y_val)
    )
  }

  x_test <- as.matrix(x_test)
  nn_y_hat <- predict(model, x_test)

  hat_N_sample <- sum(1/df$pi)
  statistic_tracker$nn_deriv_imp_mean[i] <- (1 / hat_N_sample)*(sum(reduced_df$FIN
                                                                + sum(nn_y_hat / d
  print("Finished derived param imp mean")

  print(i)
}

write.csv(statistic_tracker, file = "C:\\Users\\Alexander\\Documents\\thesis stat


################################################
# Compare results
# oracle only
```

```
dat <- statistic_tracker

mse_table <- dat[1,]
for (i in 1:dim(dat)[2]) {

  matdat <- as.matrix(dat)

  mse_table[i] <- mean( (matdat[,i] - matdat[,1])^2 )
}

mse_table

# then do MSE of method divided by MSE of oracle to get measure of goodness
# (closer to 1 means closer to oracle means good)

oracle_ratio_table <- dat[1,]

for (i in 1:dim(dat)[2]) {

  oracle_ratio_table[i] <- mse_table[i] / mse_table[2]
}

oracle_ratio_table


# compute bias:
# mean of statistic vector - true mean
# divide this differnce by true mean
# multiple by 100 to get "percent relative bais", a normalized bias measure

prb_table <- dat[1,]
mu_y <- mean(dat[,1])

for (i in 1:dim(dat)[2]) {

  prb_table[i] <- 100 * ((mean(dat[,i]) - mu_y) / mu_y)
}

prb_table

mse_t <- transpose(mse_table)
oracle_t <- transpose(oracle_ratio_table)
prb_t <- transpose(prb_table)
```

```
binded <- cbind(mse_t, oracle_t, prb_t)
colnames(binded) <- c("MSE", "Oracle Ratio", "Relative Bias")
rownames(binded) <- colnames(statistic_tracker)

test <- binded
test <- signif(test, digits = 3)

test %>%
  kable() %>%
  kable_styling()

if(FALSE){
  # mean and SD ratios
  df <- read.csv("Data/AWS_running_short.csv")
  df <- df[,-1]

  means <- apply(df, FUN = mean, MARGIN = 2)
  means

  sds <- apply(df, FUN = sd, MARGIN = 2)
  sds

  mean_ratio <- means / means[2]
  mean_ratio

  sd_ratio <- sds / sds[2]
  sd_ratio

  # fULL
  df <- read.csv("Data/AWS_running_full.csv")
  df <- df[,-1]

  means <- apply(df, FUN = mean, MARGIN = 2)
  means

  sds <- apply(df, FUN = sd, MARGIN = 2)
  sds

  mean_ratio <- means / means[2]
  mean_ratio

  sd_ratio <- sds / sds[2]
  sd_ratio
}
```

References

Angel, E. (2000). *Interactive computer graphics : A top-down approach with opengl*. Boston, MA: Addison Wesley Longman.

Angel, E. (2001a). *Batch-file computer graphics : A bottom-up approach with quicktime*. Boston, MA: Wesley Addison Longman.

Angel, E. (2001b). *Test second book by angel*. Boston, MA: Wesley Addison Longman.

Brick, J. M., & Kalton, G. (1996). Handling missing data in survey research. *Statistical Methods in Medical Research*, *5*(3), 215–238.

Chollet, F., & Allaire, J. (2018). Deep learning with r. manning publications.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

Hastie, T., Tibshirani, R., & Tibshirani, R. J. (2017). Extended comparisons of best subset selection, forward stepwise selection, and the lasso. *arXiv Preprint arXiv:1707.08692*.

Horvitz, D. G., & Thompson, D. J. (1952). A generalization of sampling without replacement from a finite universe. *Journal of the American Statistical Association*, *47*(260), 663–685.

Lumley, T. (2011). *Complex surveys: A guide to analysis using r* (Vol. 565). John Wiley & Sons.

Maiti, T., Miller, C. P., & Mukhopadhyay, P. K. (2008). Neural network imputation: An experience with the national resources inventory survey. *Journal of Agricultural, Biological, and Environmental Statistics*, *13*(3), 255.

Toth, D., & Eltinge, J. L. (2011). Building consistent regression trees from complex sample data. *Journal of the American Statistical Association*, *106*(496), 1626–1636.