Investigating the applications and limitations of feed-forward artificial neural networks for multi-class classification

Etienne Latif

May 04, 2022

Abstract

In response to the rising popularity of neural networks for multi-class classification problems over the last decade, this paper aims to compare neural networks to several other popular algorithms for these tasks: multinomial logistic regression, Gaussian Naive-Bayes and K-nearest neighbours. In particular, we have investigated the classification accuracy and training/prediction times of these models on the UCI Machine Learning Repository Letter recognition data set. We found that when supplied with more than ten thousand training data points, neural networks were able to confidently classify data with upwards of 95% accuracy. However, we found that while training times were not a significant limitation on their usefulness, they required significantly more work to correctly tune the number of hidden layers and units; a model with two hidden layers each containing 100 ReLU neurons, and an output layer with 26 softmax neurons was settled on. The trade-off in implementation difficulty for accuracy was perhaps unjustified as the K-nearest neighbour classifier achieved an accuracy of 95.70% while being trivially simple to implement. We also found that for larger training data sets (containing several thousand data points) both L2 regularisation and drop out regularisation diminished the performance of the neural networks. However, for small training data sets (with less than one thousand data points) regularisation improved the performance of the networks. It it suggested that when sufficient training data is available, overfitting is not a significant problem for neural networks (and so regularisation serves only to reduce the amount of information they are able to learn in these cases). The accompanying notebooks for this paper can be viewed at `https://github.com/EtienneLatif/Letter-recognition-classification`.

1 Introduction

1.1 Multi-class classification and a brief history of neural networks

Problems which entail placing data points into strictly one category from a number of possibilities are abundant in machine learning; computer vision, natural language processing and speech recognition are all examples of fields in which such classification tasks arise. By placing the constraint that the number of possible categories that the dependent variable of any instance may take is greater than two, we obtain an informal definition of multi-class classification. As a consequence of the broad scope of tasks encompassed by multi-class classification, there is a wealth of research into methods to solve this class of problems. As a point of reference, we will study a handful of these methods that are relevant in the current era of machine learning, namely the multinomial logistic regression [1], Naive-Bayes [2, p. 153-158], and K-nearest-neighbours classifiers [3, p. 463-468]. These methods, as noted, shall serve as a reference point for us to compare against the core focus of this paper: artificial (feed-forward) neural networks. We shall begin by giving a brief overview of the history of neural networks, to frame their current position in the field of machine learning.

Modern artificial neural networks have their origins in the field of cybernetics, tracing back to the model put forward by McCulloch and Pitts (1943) [4]. This model was based on the thresholding logic of neurons in the brain; these biological neurons have a property that an incoming excitation must exceed a threshold in order to cause an impulse, which the paper descibes as 'all-or-nothing'. These early models worked by associating linear combinations of a set of input variables with a binary dependent variable, and hence required the weights on the variables in these combinations to be correctly tuned. Inspired by Hebbian theory of neuroplasticity [5], the new aim of these biologically-inspired models was to be able to learn the weights that resulted in the best accuracy by observing examples of data from each class. The perceptron, put forward by Rosenblatt (1958), was one of the first models to achieve this goal [6]. Unlike modern neural networks, these models were single-layered and linear, rendering them incapable of learning non-linear functions. One famous example of this limitation was demonstrated by Minksy and Papert (1969), who showed that the perceptron could not learn the simple XOR function (Figure 10 in Appendix A), which was damning evidence against it's usefulness. Combined with a lack of necessary processing power in computers at the time, the linearity dilemma of these primitive networks stagnated progress in their study.

Several contributing factors led to a second wave of interest in neural networks in the late 1980s. The first of these was the rising popularity of parallel distributed processing, which became known as connectionism. Connectionism, an approach to cognitive science, aims to model and explain mental phenomena with neural networks [7]. The central principle of connectionism is that simple units can be combined in networks to achieve higher intellectual capabilities than alone. To biologists, this refers to neurons in the brain combining in networks to produce mental phenomena; in machine learning it carries over to hidden units in a neural network working together to model complex functions. Models based on this new approach, as presented by Rumelhart and McClelland (1986), overcame the linearity dilemma of the perceptron by building multi-layer networks of non-linear neurons [8]. This success was confounded by the discovery and popularisation of the backpropagation algorithm in two papers by Rumelhart et al. (1986) [9, 10], an algorithm which remains the dominant solution to training neural networks today. This algorithm is the most efficient

known way to train the parameters of a neural network; it's discovery being instrumental in the ability of neural networks to learn data sets in any reasonable time frame, despite the still limited computational power of the time. The structure of the networks of the connectionist period was almost identical to those we use today, but again they rapidly fell out of favour in 1990s as their performance was surpassed by other new and popular methods such as support vector machines and random forests. The largest affliction to the success of these neural networks was the extreme difficulty in training them. These networks used a type of non-linear neuron called sigmoid neurons, which frequently caused an issue known as the vanishing gradient problem, which can make a network unfeasibly slow to train (in fact, the vanishing gradient problem shall be a topic of careful discussion and consideration later in this paper).

Throughout the 2010s, neural networks have again come to dominate the field of machine learning; what changed to permit this return to grace? Firstly, the introduction of new activation functions for neurons, such as the popular ReLU function, have been able to suppress the risks of the vanishing gradient problem, making network parameters significantly easier to train. Another large contributor has been advances in computational power. In particular, without side-tracking too far into the topic of hardware, neural networks benefit from parallel processing provided by GPUs or multiple CPUs (distributed computing). Parallel processing is ideal for neural networks, as it allows the many neurons of the network to be trained in parallel, greatly improving training times [11]. The availability of larger data sets has also proved beneficial to the training of neural networks, providing them with larger amounts of information to learn from and generalise to unseen data. Of note is the famous MNIST data set, now considered the benchmark of machine learning, used to test and tune new techniques, and consisting of 60,000 training examples [12]. Given this overview of the history of neural networks, it is obvious to question how well neural networks have been found to perform on multi-class classification problems in the literature. In the next subsection, we shall review what the literature shows for neural networks and the other aforementioned algorithms in the context of multi-class classification.

1.2 Literature review

The aim of this section is not to describe the technical workings of each algorithm, or to suggest considerations for our implementations, but strictly to give an overview of the results that can currently be observed in the literature for our chosen models. We shall arbitrarily begin with multinomial logistic regression.

From some research of multinomial logistic regression, it appears to find a wide variety of applications in the medical and social sciences fields. Liang et al. (2020) compared multinomial logistic regression and ordinal logistic regression for predicting the survival status of lung cancer patients from seven independent variables containing information about patients and their treatment procedures [13]. They found that multinomial logistic regression provided a more reliable prediction accuracy of 74.5%, although the training and testing both appear to have been run on the same data set. A further example is given by Balasubramanian (2014) who used multinomial logistic regression to investigate variables increasing the risk of breast cancer for different levels of socioeconomic status in rural and urban areas of India [14]. In this case, another impressive result of 70.1% accuracy was obtained. We can note that multinomial logistic regression appears to provide decently reliable results, although a way off from perfect accuracy.

We turn our attention next to the family of Naive-Bayes classifiers. Hajer et al. (2019) used Gaussian Naive-Bayes (the specific Naive-Bayes classifier we shall employ in this paper) to predict diagnoses of patients in data sets of breast cancer and lung cancer patients [15]. Although these are binary classification problems, we shall see later in this paper that the Naive-Bayes algorithm makes no distinction between binary and multi-class problems, so this study is indeed relevant to our paper. The results were impressive; they obtained 98% accuracy on breast cancer and 90% accuracy on lung cancer. Another study of interest was conducted by Mandal and Jana (2019) who attempted to use Naive-Bayes to classify molecules in drugs, obtaining 93% accuracy [16]. This is impressive accuracy, but of greater is the results they obtained using a K-nearest neighbour (KNN) model: 99.6%. Previously mentioned, KNN is another model we shall be examining in this paper, and this result leans strongly in it's favour. To expand on the performance of KNN, we look at a paper by Güney and Atasoy (2012) which compares KNN and support vector machines to classify n-butanol concentrations sensed by an electronic nose [17]. They found that KNN performed at 87% accuracy. Both the Naive-Bayes and KNN classifiers appear to show very promising performance on a variety of classification problems.

There are many different types and systems of neural networks that one may use, thus it is important we be precise in the method we will be implementing in this paper in order to ensure we are reviewing the appropriate literature. Hence we clarify that we shall be using a single one-against-all feed-forward neural network for classification throughout this paper (these technical terms will be explained in a later section). With this information in mind, we look at Ou and Murphey (2007), who compared a number of different approaches to implementing neural networks [18]. They found that one-against-all feed-forward neural networks are usually capable of achieving optimal classification results. They drew this conclusion from results of 10-fold cross validation on a number of data sets, finding a validation accuracy of $> 97\%$ on some data sets. Across all their tests, the lowest accuracy obtained by this method was 65.14%, with over 80% accuracy on all but two of the data sets. Reinforcing these results is the paper by LeCun et al. (1989) which used a single neural network to classify handwritten zip code digits from a data set provided by the US postal service [19]. They used 23 training epochs and 7291 training instances, obtaining a testing accuracy of 94.9%.

Summarising these results, we note that each proposed algorithm shows reliable classification performance, to varying degrees. While artificial neural networks perform consistently extremely well, we do note that both Naive-Bayes and KNN also obtain nearly comparable results, while being far simpler to implement (as we shall see). One final point is that the studies provided for neural networks used the sigmoidal and hyperbolic tangent activation funtions, which are both now outdated. We shall be using activation functions that are now known to be generally superior, and for that reason may expect even better results than observed in these papers.

1.3 Aims and contributions of this work

We now know that neural networks tend to perform extremely well at classifying unseen data instances, frequently performing at over 95% accuracy. However, as we have discussed, neural networks suffer from difficulty in training efficiently given limitations on computational power and data availability. While there are many studies comparing different types and systems of neural networks to each other in terms of accuracy [18], this paper will examine the performance of neural networks relative to other popular algorithms in the field of multi-class classification at present. We will compare the accuracy and training/prediction times required for the models, and discuss the trade-off between these metrics.

In the next section, we will introduce and examine the data set that we will be investigating in this paper.

2 The letter recognition data set

The example data set we have chosen to work on is the 'Letter Recognition' data set provided by the UCI Machine Learning Repository [20]. The purpose of this task is to classify black-and-white images of handwritten English capital letters by the correct letter, from a number of independent variables describing the pixels in the image. Some examples of instances of this data set are displayed in Figure 11 in Appendix B. This data set contains $20,000$ instances, with no missing data (simplifying the pre-processing stage). Some examples of instances of the data set are included in Figure 11 in Appendix B. The dependent variable, 'lettr', is simply the capital letter of the English language that is contained in the image. This is, of course, a categorical variable with 26 classes. As we are looking at multi-class classification, not multi-label classification, we recall that each image contains strictly one letter. The data set is relatively well balanced, with a mean of 769.2 instances per class and a standard deviation of 22.7 (frequency of each class is displayed in Figure 12 in Appendix B).

The data set contains sixteen independent variables. To understand what each of these variables means, we first need to define the 'box' of each image. As noted, the images are black-and-white, so every pixel is a binary value. Then the 'box' of the image is the smallest rectangular box that contains every black pixel in the image. An example is displayed in Figure 1.

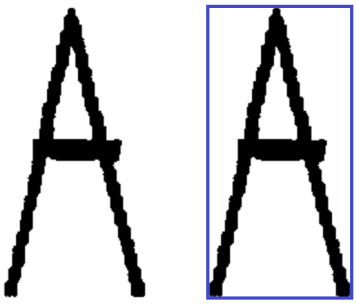

Figure 1: An example of an image in the data set (left) and the same data point with the image's box shown in blue (right). Image obtained from Frey and Slate (1991) [21] and box added manually in Photoshop.

We now describe the independent variables:

1. 'x-box': The number of horizontal pixels from the left of the image to the centre of the box.
2. 'y-box': The number of vertical pixels from the bottom of the image to the centre of the box.
3. 'width': The width of the box in pixels.
4. 'high': The height of the box in pixels.
5. 'onpix': The total number of black pixels in the image.
6. 'x-bar': The mean value over all black pixels of the horizontal distance of the pixel from the centre of the box divided by the width of the box. Pixels to the left of centre have a negative distance, pixels to the right of centre have a positive distance.
7. 'y-bar': The mean value over all black pixels of the vertical distance of the pixel from the centre of the box divided by the height of the box.
8. 'x2bar': The mean value of black pixels' horizontal distance from the centre of the box squared.
9. 'y2bar': The mean value of black pixels' vertical distance from the centre of the box squared.
10. 'xybar': The mean product of black pixels' horizontal and vertical distances from the centre of the box.
11. 'x2ybr': The mean product of black pixels' horizontal distances squared and vertical distances from the centre of the box.
12. 'xy2br': The mean product of black pixels' horizontal distances and vertical distances squared from the centre of the box.

13. 'x-ege': The mean number of horizontal edges encountered when enumerating pixels across a row over each row in the box.

14. 'xegvy': The sum of vertical positions of each horizontal edge in the box.

15. 'y-ege': The mean number of vertical edges encountered when enumerating pixels across a column over each column in the box.

16. 'yegvx': The sum of horizontal positions of each vertical edge in the box.

This data set has been used in other studies of machine learning and statistical techniques. One such paper is Pavlov et al. (2003) who used the data set to investigate mixtures of conditional maximum entropy models. They obtained an accuracy on their testing set of 72.42%. The data set was also used in a paper by Kumar et al. (1999) which proposed a Bayesian class-pair based feature selection algorithm. They tested performance of a multi-layer perceptron and two variations of their algorithm on the data set, obtaining a worst testing accuracy with the multi-layer perceptron of 79.3% and a best accuracy of 86.6%. We see that typical testing accuracy in the literature for this data set is roughly 70% to 90%. We hope that with the use of neural networks we are able to surpass 90% accuracy.

With a now clearly defined objective, we shall proceed to describe the mathematics behind the selected models.

3 Describing the models: multinomial logistic regression, Naive-Bayes and K-nearest neighbours

In this section we define the multinomial logistic regression, Naive-Bayes and KNN classifiers for a generalised multi-class classification problem. We will define the set of class labels $C = \{C^1, C^2, ..., C^K\}$ and our dependent variable $Y \in C$. We shall further associate the dependent variable for each data instance with a set of independent variables $\{x_1, x_2, ..., x_P\}$. We will express this problem for a set of N data instances in a matrix notation:

- $\mathbf{y}$ is the $N \times (K-1)$ matrix of observations, where the i-th row is the encoded response for the i-th observation. A modified one-hot encoding is used for each observation, where the i-th row is written as $\mathbf{y}_i = (y_{i1}, y_{i2}, ..., y_{iK-1})$ where:

$$y_{ik} = \begin{cases} 1 & \text{if } Y_i = C^k,\ k \in \{1, ..., K-1\} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

  which implies (a) 1 at the index corresponding to the class that the dependent variable of observation i is in, and 0 at all other indexes if $Y_i \in \{C^1, ..., C^{K-1}\}$ or (b) 0 at every index if $Y_i = C^K$.

- $\mathbf{X}$ is the $N \times P$ design matrix, where $[\mathbf{X}]_{ip} = x_{ip}$ is the value of the p-th independent variable for the i-th observation. Let $\mathbf{x}_i = (x_{i1}, x_{i2}, ..., x_{iP})^T$, then the i-th row of $\mathbf{X}$ is $\mathbf{x}_i^T$.

3.1 Multinomial logistic regression

3.1.1 Background

Perhaps the most well-known machine learning model is linear regression, which constructs a linear function of independent variables to predict a response, by minimising the least squares error between predictions and true values of the dependent variable. While this model is effective for modelling continuous responses, there are several reasons that it does not apply well when dealing with a categorical response. Considering our data set, the ordering of the letters of the alphabet is essentially arbitrary; there is no mathematical properties we can identify that seem to naturally increase or decrease as we enumerate the letters in this order. In other words, the letters are a qualitative variable that we will consider a nominal (unordered) variable from a machine learning perspective, so can not be modelled directly by a simple linear combination of dependent variables.

Instead, when we wish to use a linear model for a multi-class categorical variable, multinomial logistic regression is the appropriate choice [22]. The binary logistic regression model predates it's multinomial counterpart, the former being extended to the latter in two independent papers by Cox (1966) and Theil (1969) [23, 24]. Multinomial logistic regression models the conditional probability of the dependent variable being in each class with a linear function of the dependent variables, rather than modelling the value of the response itself. As an additional note, because multinomial logistic regression directly models the conditional probability of the response given the independent variables, it belongs to a broad subset of classification models known as 'discriminative' models. There is another major approach to classification known as the generative approach, and we shall look at a generative model later in the paper. For now it is sufficient to note that research seems to indicate that discriminative models tend to perform better than generative models, particularly in the case of large data sets, as they usually have lower asymptotic error rates (as training instances become large) [25]. This is relevant in our case, as the data set we are investigating has a large number of instances ($20,000$).

3.1.2 Model definition

To set up our probabilistic model, we will select the K-th class as our baseline without loss of generality (recall that we are considering the dependent variable to be nominal so this choice is arbitrary). We shall make a slight notational adaption to our design matrix for this model, though it should be noted that this new notation applies only to this section on multinomial logistic regression. This adaption is the introduction of a dummy 'one' variable to our vector of independent variables for every data instance, such that $\mathbf{X}$ becomes an $N \times (P+1)$ matrix with $[\mathbf{X}]_{i0} = 1, i = 1, ..., N$. Then the transpose of the i-th row of the design matrix is the vector of independent variables for observation i, given by $\mathbf{x}_i = (1, x_{i1}, x_{i2}, ..., x_{iP})^T$. The purpose of this variable will become clear after we introduce the two following matrices that we will need for this model:

- $\boldsymbol{\beta}$ is the $(P+1) \times (K-1)$ matrix of coefficients, where $[\boldsymbol{\beta}]_{pk} = \beta_{pk}$ is the coefficient of the p-th independent variable for the k-th class. Let $\boldsymbol{\beta}_k = (\beta_{0k}, \beta_{1k}, ..., \beta_{pk})^T$, the k-th column of the matrix. We call β_{pk} a bias if $p = 0$, and a weight otherwise.
- $\mathbf{P}$ is the $N \times (K-1)$ matrix of conditional class probabilities for each observation, where $[\mathbf{P}]_{ik} = p_{ik}(\mathbf{x}_i; \boldsymbol{\beta}) = p_{ik} = Pr(Y_i = C^k | \mathbf{X}_i = \mathbf{x}_i)$.

Now for any data instance i and class k we can construct a linear combination of the independent variables with the vector multiplication $\mathbf{x}_i^T \boldsymbol{\beta}_k$, where β_{0k}, the coefficient of the dummy variable, will represent the intercept term.

The multinomial logistic regression model is generally given by $K-1$ logits:

$$log \frac{p_k}{p_K} = \mathbf{x}^T \boldsymbol{\beta}_k, \quad \forall\, k \in \{1, 2, ..., K-1\} \tag{2}$$

By exponentiating both sides of the logit model and multiplying both sides by p_K we obtain each of our class probabilities for an observation of Y as:

$$p_k = p_K e^{\mathbf{x}^T \boldsymbol{\beta}_k} \tag{3}$$

Where p_K is easily found by:

$$\begin{aligned} p_K &= 1 - \sum_{k=1}^{K-1} p_k \\ &= 1 - \sum_{k=1}^{K-1} p_K e^{\mathbf{x}^T \boldsymbol{\beta}_k} \\ \implies p_K &= \frac{1}{1 + \sum_{l=1}^{K-1} e^{\mathbf{x}^T \boldsymbol{\beta}_l}} \end{aligned} \tag{4}$$

Plugging this result back into Eq. 3 we are able to fully specify the conditional class probabilities for every possible class in C:

$$\begin{aligned} p_k &= \frac{e^{\mathbf{x}^T \boldsymbol{\beta}_k}}{1 + \sum_{l=1}^{K-1} e^{\mathbf{x}^T \boldsymbol{\beta}_l}}, \quad \forall\, k \in \{1, 2, ..., K-1\} \\ p_K &= \frac{1}{1 + \sum_{l=1}^{K-1} e^{\mathbf{x}^T \boldsymbol{\beta}_l}} \end{aligned} \tag{5}$$

Each of these probabilities is a linear function of the independent variables. They are all in the set $[0, 1]$, and when they are all added together total one. For a new data point, these probabilities are simple to calculate, and the data is classified into the class with highest conditional probability.

3.1.3 Coefficient estimation

The remaining issue is to estimate the coefficients $\boldsymbol{\beta}$, which is done by maximum likelihood estimation [26, p. 271]. We note that the dependent variable takes a multinomial distribution, which models the probability of the frequency of data points in each class in a sample. For our sample of N observations, using the previously obtained conditional probabilities we write the probability mass function of the multinomial distribution as:

$$f(\mathbf{y}_i; \boldsymbol{\beta}) = \frac{N!}{\prod_{k=1}^{K-1} y_{ik}!} \prod_{k=1}^{K-1} p_{ik}^{y_{ik}} p_{iK}^{1 - \sum_{k=1}^{K-1} y_{ik}} \tag{6}$$

Then the joint probability mass function is given as:

$$f(\mathbf{y}; \boldsymbol{\beta}) = \prod_{i=1}^{N} \left(\frac{N!}{\prod_{k=1}^{K-1} y_{ik}!} \prod_{k=1}^{K-1} p_{ik}^{y_{ik}} p_{iK}^{1 - \sum_{k=1}^{K-1} y_{ik}} \right) \tag{7}$$

In order to obtain our maximum likelihood estimate for $\boldsymbol{\beta}$ we take Eq. 7 as a function of $\boldsymbol{\beta}$ for fixed values of the response variable. We can omit the terms $N!$ and $\prod_{k=1}^{K-1} y_{ik}!$ as they have no dependence on $\boldsymbol{\beta}$, to obtain the joint likelihood function as follows:

$$\begin{aligned}
L(\boldsymbol{\beta}) &= \prod_{i=1}^{N} \left(\prod_{k=1}^{K-1} p_{ik}^{y_{ik}} p_{iK}^{1-\sum_{k=1}^{K-1} y_{ik}} \right) \\
&= \prod_{i=1}^{N} \left(\prod_{k=1}^{K-1} p_{ik}^{y_{ik}} \frac{p_{iK}}{p_{iK}^{\sum_{k=1}^{K-1} y_{ik}}} \right) \\
&= \prod_{i=1}^{N} \left(\prod_{k=1}^{K-1} p_{ik}^{y_{ik}} \frac{p_{iK}}{\prod_{k=1}^{K-1} p_{iK}^{y_{ik}}} \right) \\
&= \prod_{i=1}^{N} \left(\prod_{k=1}^{K-1} \left(\frac{p_{ik}}{p_{iK}} \right)^{y_{ik}} p_{iK} \right) \\
&= \prod_{i=1}^{N} \left(\prod_{k=1}^{K-1} \left(e^{\mathbf{x}_i^T \boldsymbol{\beta}_k} \right)^{y_{ik}} \left(1 + \sum_{k=1}^{K-1} e^{\mathbf{x}_i^T \boldsymbol{\beta}_k} \right)^{-1} \right)
\end{aligned} \tag{8}$$

We look to find the value of $\boldsymbol{\beta}$ that maximises this likelihood function as our estimated coefficients. As the logarithmic function is monotone increasing, we know that this is equivalent to maximising the log-likelihood function, which we shall denote $\ell(\boldsymbol{\beta}; \mathbf{y}) = \ell(\boldsymbol{\beta}) \equiv logL(\boldsymbol{\beta})$. Taking the natural log on both sides of Eq. 8, we get:

$$\begin{aligned}
\ell(\boldsymbol{\beta}) &= \sum_{i=1}^{N} \left(\sum_{k=1}^{K-1} y_{ik} \mathbf{x}_i^T \boldsymbol{\beta}_k - log \left(1 + \sum_{k=1}^{K-1} e^{\mathbf{x}_i^T \boldsymbol{\beta}_k} \right) \right) \\
&= \sum_{i=1}^{N} \left(\sum_{i=1}^{K-1} y_{ik} \left(\sum_{p=0}^{P} x_{ip} \beta_{pk} \right) - log \left(1 + \sum_{k=1}^{K-1} \exp \left(\sum_{p=0}^{P} x_{ip} \beta_{pk} \right) \right) \right)
\end{aligned} \tag{9}$$

Note that for convenience here we haven chosen to express vector products as summations of their elements, and exponential expressions as exp(.). We can get the score function $u_{pk}(\boldsymbol{\beta})$ by taking the first partial derivative of ℓ in terms of $\boldsymbol{\beta}_{pk}$, for $p = 1, ..., P; k = 1, ..., K$.

$$\begin{aligned}
u_{pk}(\boldsymbol{\beta}) &= \frac{\partial \ell(\boldsymbol{\beta})}{\partial \beta_{pk}} \\
&= \frac{\partial}{\partial \beta_{pk}} \sum_{i=1}^{N} \left(\sum_{i=1}^{K-1} y_{ik} \left(\sum_{p=0}^{P} x_{ip} \beta_{pk} \right) - log \left(1 + \sum_{k=1}^{K-1} \exp \left(\sum_{p=0}^{P} x_{ip} \beta_{pk} \right) \right) \right) \\
&= \sum_{i=1}^{N} \left(y_{ik} x_{ip} - \frac{\partial}{\partial \beta_{pk}} log \left(1 + \sum_{k=1}^{K-1} \exp \left(\sum_{p=0}^{P} x_{ip} \beta_{pk} \right) \right) \right) && \text{(Linearity in differentiation)} \\
&= \sum_{i=1}^{N} \left(y_{ik} x_{ip} - \left(1 + \sum_{k=1}^{K-1} \exp \left(\sum_{p=0}^{P} x_{ip} \beta_{pk} \right) \right)^{-1} \frac{\partial}{\partial \beta_{pk}} \left(1 + \sum_{k=1}^{K-1} \exp \left(\sum_{p=0}^{P} x_{ip} \beta_{pk} \right) \right) \right) && \text{(Log derivative rule)} \\
&= \sum_{i=1}^{N} \left(y_{ik} x_{ip} - \frac{\exp \left(\sum_{p=0}^{P} x_{ip} \beta_{pk} \right)}{\left(1 + \sum_{k=1}^{K-1} \exp \left(\sum_{p=0}^{P} x_{ip} \beta_{pk} \right) \right)} \frac{\partial}{\partial \beta_{pk}} \left(\sum_{p=0}^{P} x_{ip} \beta_{pk} \right) \right) && \text{(Exponential derivative rule)} \\
&= \sum_{i=1}^{N} \left(y_{ik} x_{ip} - p_{ik} x_{ip} \right) && \text{(Definition of } p_{ik}\text{)}
\end{aligned} \tag{10}$$

We know that the MLEs solve the $(P+1) \times (K-1)$ equations, $u_{pk}(\boldsymbol{\beta}) = 0$. Note that we place each score function in the score vector $\mathbf{u}(\boldsymbol{\beta})$. We use the Newton-Raphson algorithm for this, which requires us to find the Hessian matrix of second partial derivatives. To get this matrix, we differentiate Eq. 10 in terms of each $\beta_{p'k'}$, where $p = 1, ..., P; k = 1, ..., K$:

$$\begin{aligned}
\frac{\partial^2 \ell(\boldsymbol{\beta})}{\partial \beta_{pk} \partial \beta_{p'k'}} &= \frac{\partial}{\partial \beta_{p'k'}} \sum_{i=1}^{N} \left(y_{ik} x_{ip} - p_{ik} x_{ip} \right) \\
&= - \sum_{i=1}^{N} x_{ip} \frac{\partial}{\partial \beta_{p'k'}} \left(\frac{\exp \left(\sum_{p=0}^{P} x_{ip} \beta_{pk} \right)}{1 + \sum_{k=1}^{K-1} \exp \left(\sum_{p=0}^{P} x_{ip} \beta_{pk} \right)} \right) && \text{(Linearity in differentiation)}
\end{aligned} \tag{11}$$

Here we apply the quotient rule:

$$f(x) = \frac{g(x)}{h(x)} \implies \frac{\partial}{\partial x} f(x) = \frac{\frac{\partial}{\partial x} g(x)h(x) - \frac{\partial}{\partial x} h(x)g(x)}{(h(x))^2} \tag{12}$$

It is necessary for us to differentiate between the cases in which $k' = k$ and $k' \neq k$, so we shall proceed separately for each case.

Case 1: $k' = k$

From (8), we have:

$$\begin{aligned} g(x) = \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right) &\implies \frac{\partial}{\partial \beta_{p'k'}} g(x) = x_{ip'} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right) \quad \text{(Exponential derivative rule)} \\ h(x) = 1 + \sum_{k=1}^{K-1} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right) &\implies \frac{\partial}{\partial \beta_{p'k'}} h(x) = x_{ip'} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right) \quad \text{(Linearity in differentiation)} \end{aligned} \tag{13}$$

Hence:

$$\begin{aligned} &\frac{\partial}{\partial \beta_{p'k'}} \left(\frac{\exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)}{1 + \sum_{k=1}^{K-1} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)} \right) \\ &= \frac{x_{ip'} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right) \left(1 + \sum_{k=1}^{K-1} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)\right) - x_{ip'} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right) \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)}{\left(1 + \sum_{k=1}^{K-1} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)\right)^2} \\ &= \frac{x_{ip'} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right) \left(\left(1 + \sum_{k=1}^{K-1} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)\right) - \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)\right)}{\left(1 + \sum_{k=1}^{K-1} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)\right)^2} \\ &= x_{ip'} p_{ik} \frac{\left(1 + \sum_{k=1}^{K-1} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)\right) - \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)}{1 + \sum_{k=1}^{K-1} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)} \quad \text{(Definition of } p_{ik}\text{)} \\ &= x_{ip'} p_{ik} \left(\frac{1 + \sum_{k=1}^{K-1} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)}{1 + \sum_{k=1}^{K-1} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)} - \frac{\exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)}{1 + \sum_{k=1}^{K-1} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)} \right) \\ &= x_{ip'} p_{ik}(1 - p_{ik}) \quad \text{(Definition of } p_{ik}\text{)} \end{aligned} \tag{14}$$

Case 2: $k' \neq k$

From (8), we have:

$$\begin{aligned} g(x) = \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right) &\implies \frac{\partial}{\partial \beta_{p'k'}} g(x) = 0 \qquad \text{(Function is constant w.r.t } \beta_{p'k'}\text{)} \\ h(x) = 1 + \sum_{k=1}^{K-1} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right) &\implies \frac{\partial}{\partial \beta_{p'k'}} h(x) = x_{ip'} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk'}\right) \quad \text{(Linearity in differentiation)} \end{aligned} \tag{15}$$

Hence:

$$\begin{aligned} &\frac{\partial}{\partial \beta_{p'k'}} \left(\frac{\exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)}{1 + \sum_{k=1}^{K-1} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)} \right) \\ &= \frac{-x_{ip'} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right) \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk'}\right)}{\left(1 + \sum_{k=1}^{K-1} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)\right)^2} \\ &= -x_{ip'} \frac{\exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)}{1 + \sum_{k=1}^{K-1} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)} \frac{\exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk'}\right)}{1 + \sum_{k=1}^{K-1} \exp\left(\sum_{p=0}^{P} x_{ip}\beta_{pk}\right)} \quad = -x_{ip'} p_{ik} p_{ik'} \end{aligned} \tag{16}$$

Now that the partial derivative has been obtained in both cases, we can plug our results back into Eq. 11 to find our matrix of second derivatives:

$$[\mathbf{H}(\boldsymbol{\beta})]_{(pk)(p'k')} = \frac{\partial^2 \ell(\boldsymbol{\beta})}{\partial \beta_{pk} \partial \beta_{p'k'}} = \begin{cases} -\sum_{i=1}^{N} x_{ip} x_{ip'} p_{ik}(1 - p_{ik}), & \text{if } k' = k \\ \sum_{i=1}^{N} x_{ip} x_{ip'} p_{ik} p_{ik'}, & \text{otherwise} \end{cases} \tag{17}$$

Finally, we shall find our MLEs by using the Newton-Raphson formula. For the purposes of this formula, we shall define $\boldsymbol{\beta}_v = \left(\boldsymbol{\beta}_1^T, \boldsymbol{\beta}_2^T, ..., \boldsymbol{\beta}_{K-1}^T\right)^T$ by which we consider the parameters as a column vector, obtained by concatenating together all the columns of the matrix $\boldsymbol{\beta}$. Note that this vector formulation is just used for convenience in constructing the Newton-Raphson formula, and the parameters are shared between $\boldsymbol{\beta}$ and $\boldsymbol{\beta}_v$, so that by updating the values in the column vector we also update the values in the matrix. Then the Newton-Raphson formula for obtaining the MLEs is given by:

$$\boldsymbol{\beta}_v^{(m+1)} = \boldsymbol{\beta}_v^{(m)} - \mathbf{H}\left(\boldsymbol{\beta}_v^{(m)}\right)^{-1} \mathbf{u}\left(\boldsymbol{\beta}_v^{(m)}\right) \tag{18}$$

From this formula the Newton-Raphson algorithm can be used to find the coefficient values by the following steps:

1. Select a small threshold $\epsilon > 0$ and an initial $\boldsymbol{\beta}_v^{(0)}$. Set $\boldsymbol{\beta}_v^{(m)} \leftarrow \boldsymbol{\beta}_v^{(0)}$.
2. Calculate $\boldsymbol{\beta}_v^{(m+1)}$ by Eq. 18.
3. If $\|\boldsymbol{\beta}_v^{(m+1)} - \boldsymbol{\beta}_v^{(m)}\| < \epsilon$ then we have converged to a solution $\hat{\boldsymbol{\beta}}_v \leftarrow \boldsymbol{\beta}_v^{(m+1)}$. Otherwise set $\boldsymbol{\beta}_v^{(m)} \leftarrow \boldsymbol{\beta}_v^{(m+1)}$ and return to step 2.

We can also impose weight regularisation in the form of an additive penalty term on the log-likelihood function. The purpose of this penalty term is to prevent the weight parameters from becoming too large, to prevent the model becoming overly complex and overfitting to the training data. The ideas of overfitting and regularisation are discussed in more depth in a later chapter. For now it is sufficient to note that with L2 regularisation the maximisation problem is reformulated to the following:

$$\hat{\boldsymbol{\beta}} = \arg\max_{\boldsymbol{\beta}} \left(\sum_{i=1}^{N} \left(\sum_{i=1}^{K-1} y_{ik} \left(\sum_{p=0}^{P} x_{ip} \beta_{pk} \right) - log\left(1 + \sum_{k=1}^{K-1} \exp\left(\sum_{p=0}^{P} x_{ip} \beta_{pk} \right) \right) \right) - \lambda \sum_{k=1}^{K-1} \sum_{p=1}^{P} \beta_{pk}^2 \right) \tag{19}$$

Where $\lambda > 0$ controls the regularisation strength of the penalty term. Note that we have not included parameters where $p = 0$, as it is conventional to only regularise the weights and not the biases. We denote the new function to be maximised by:

$$\tilde{\ell}(\boldsymbol{\beta}) \equiv \ell(\boldsymbol{\beta}) - \lambda \sum_{k=1}^{K-1} \sum_{p=1}^{P} \beta_{pk}^2 \tag{20}$$

Clearly as we increase the values of the weights, the regularisation term increases in strength. As the weights in the regularisation are of a higher power than those in $\ell(\boldsymbol{\beta})$, the rate of growth of the regularisation term is higher, and so if the weights are to keep being increased, the regularisation term will eventually outweigh $\ell(\boldsymbol{\beta})$. Hence the weights are forced to remain small, as values that are too large will not result in maximum solutions to $\tilde{\ell}(\boldsymbol{\beta})$. Notice that higher values of λ force lower values of the weights; selecting a λ too small can lead to overfitting, but selecting a λ too large can lead to underfitting as the weights are forced to be too small to capture relevant information. The optimal value of λ can be found using cross-validation over a set of potential values. Cross-validation is performed by holding-out a portion of the training set when training the model, and then evaluating the performance of the model after training with the partial training set on the held out data. The value of λ for which the best validation accuracy is achieved can then be taken as the value that we use for training on the total training set.

3.2 Naive-Bayes

3.2.1 Background

In the previous section we looked at a discriminative model. Here we look at a popular group of generative classifiers known as Naive-Bayes classifiers. Rather than directly model the conditional probability of the response given the independent variables, generative classifiers learn a model of the density $Pr(\mathbf{x}, y)$ that generated the data and apply Bayes theorem to calculate the conditional probability $Pr(y|\mathbf{x})$. The defining assumption of the Naive-Bayes family of classifiers, is that within each class the distributions of each independent variable are independent of one another. Although this assumption sounds unrealistic in most cases, the model shows surprisingly impressive results in practice [27, 28].

This raises the immediate question of whether this is a reasonable assumption in our data set. From Figure 2 we can identify immediately that the variables *onpix*, *high*, *width*, *y-box* and *x-box* all highly correlate with one another. We also see that *x-ege* and to a lesser degree *y-ege* show some notable correlation with the previously mentioned variables. Empirical research has found that Naive-Bayes classifiers tend to perform best with large numbers of training instances [28] and when the independent variables are either entirely independent or functionally dependent, with the worst performance being seen for cases in between these two extremes [27]. As a result, we should consider dropping the highly dependent independent variables when fitting the model, as this will leave us with a mostly independent feature set.

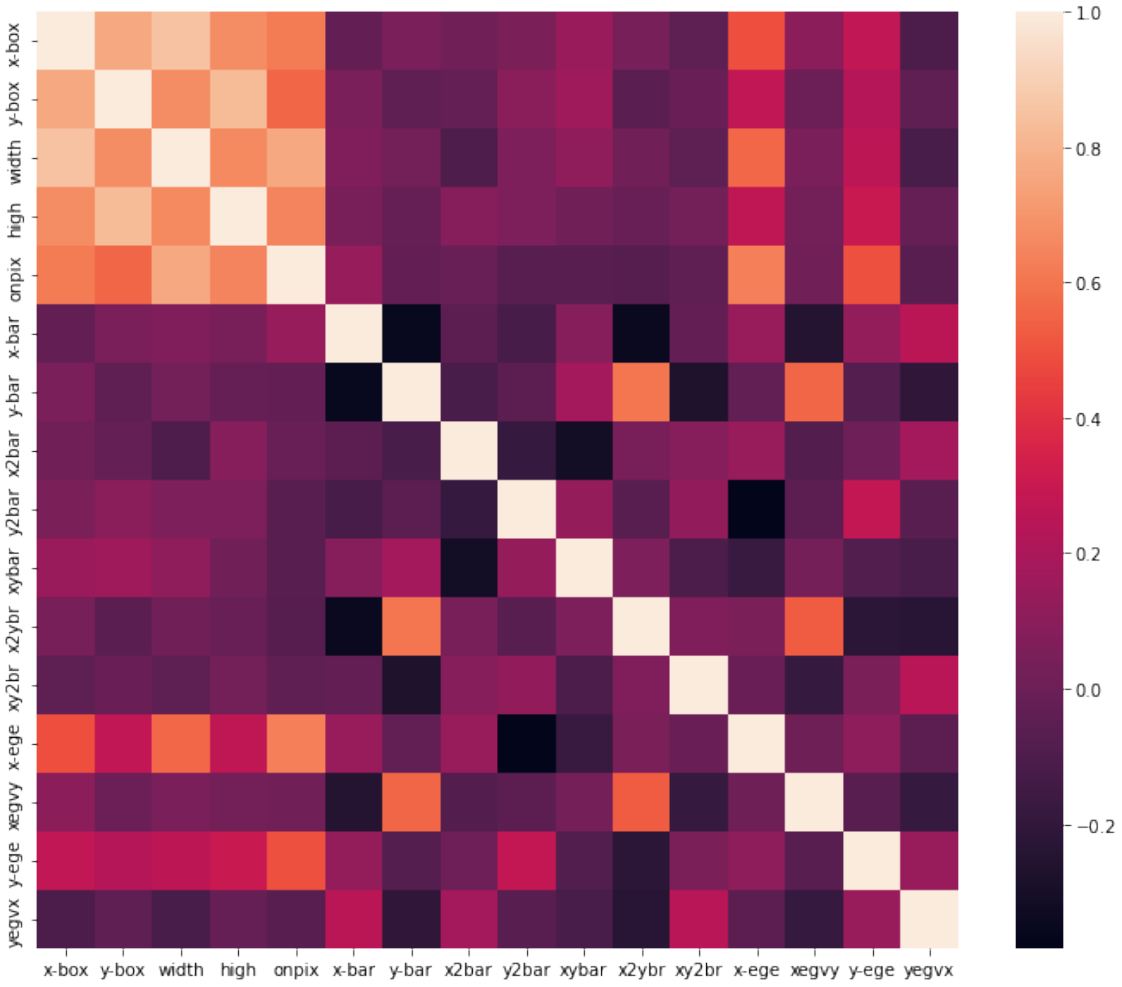

Figure 2: Heat map representation of the correlation matrix for each of the 16 dependent variables (lighter squares correspond to higher positive correlation)

3.2.2 Model definition

The aim of Naive-Bayes classifiers is to model the distribution of the dependent variable by combining the distribution of the dependent variables conditional on the response and prior information about the distribution of the dependent variable. We call this conditional distribution the posterior distribution, denoted $Pr(Y = C^k|\mathbf{x}) \equiv p_k$.

We first estimating the following distributions:

- $f_k(\mathbf{X}) = Pr(\mathbf{X}|Y = C^k)$: the conditional distribution of the independent variables for a given class. Note that these distributions are independent between classes.
- π_k: the prior distribution of the dependent variable.

This information is then combined using Bayes theorem to model the posterior distribution of each class (which we shall continue to denote p_k):

$$p_k = \frac{\pi_k f_k(\mathbf{x})}{\sum_{j=1}^{K} \pi_j f_j(\mathbf{x})} \tag{21}$$

Aforementioned, Naive-Bayes classifiers make the assumption that within each class, the distributions of each independent variable are independent of eachother. Mathematically, we have the assumption:

$$\rho_{x_i,x_j} = 0, \quad i \neq j \implies f_k(\mathbf{x}) = \prod_{p=1}^{P} f_{kp}(x_p) \tag{22}$$

Under this assumption, we plug Eq. 22 into Eq. 21 to get our updated posterior distribution model for Naive-Bayes classifiers:

$$p_k = \frac{\pi_k \prod_{p=1}^{P} f_{kp}(x_p)}{\sum_{j=1}^{K} \pi_j \prod_{p=1}^{P} f_{jp}(x_p)} \tag{23}$$

We may note that the denominator of the posterior proability of each class is the same, so we may express Eq. 23 as:

$$p_k \propto \pi_k \prod_{p=1}^{P} f_{kp}(x_p) \tag{24}$$

And finally we obtain our estimated class for a given input vector of independent variables by maximising Eq. 24 as a function of k:

$$C^{\hat{k}}, \quad \hat{k} = \arg\max_{k \in \{1,\ldots,K\}} p_k = \arg\max_{k \in \{1,\ldots,K\}} \pi_k \prod_{p=1}^{P} f_{kp}(x_p) \tag{25}$$

This leaves the question of how to estimate the marginal distributions of independent variables for each class and the prior distribution of the dependent variable.

3.2.3 Estimating the prior and marginal distributions

The issue of estimating the prior class probabilities $\hat{\pi}_k$ is as simple as calculating the proportion of observations with response variable in the k-th class:

$$\hat{\pi}_k = \frac{\sum_{i=1}^{N} [y_i = C^k]}{N} \tag{26}$$

For estimating the marginal distributions of each independent variable in a given class, we have to make an assumption about the distribution of the independent variables. A popular approach we shall take is to model each independent variable with a Gaussian distribution. This is the only implementation of Naive Bayes for continuous independent variables that is given by the popular Scikit-learn library. It is common to examine the distribution of independent variables with histograms. Examining Figure 3, we can see that the Gaussian assumption seems reasonable (although imperfect) for the majority of our independent variables.

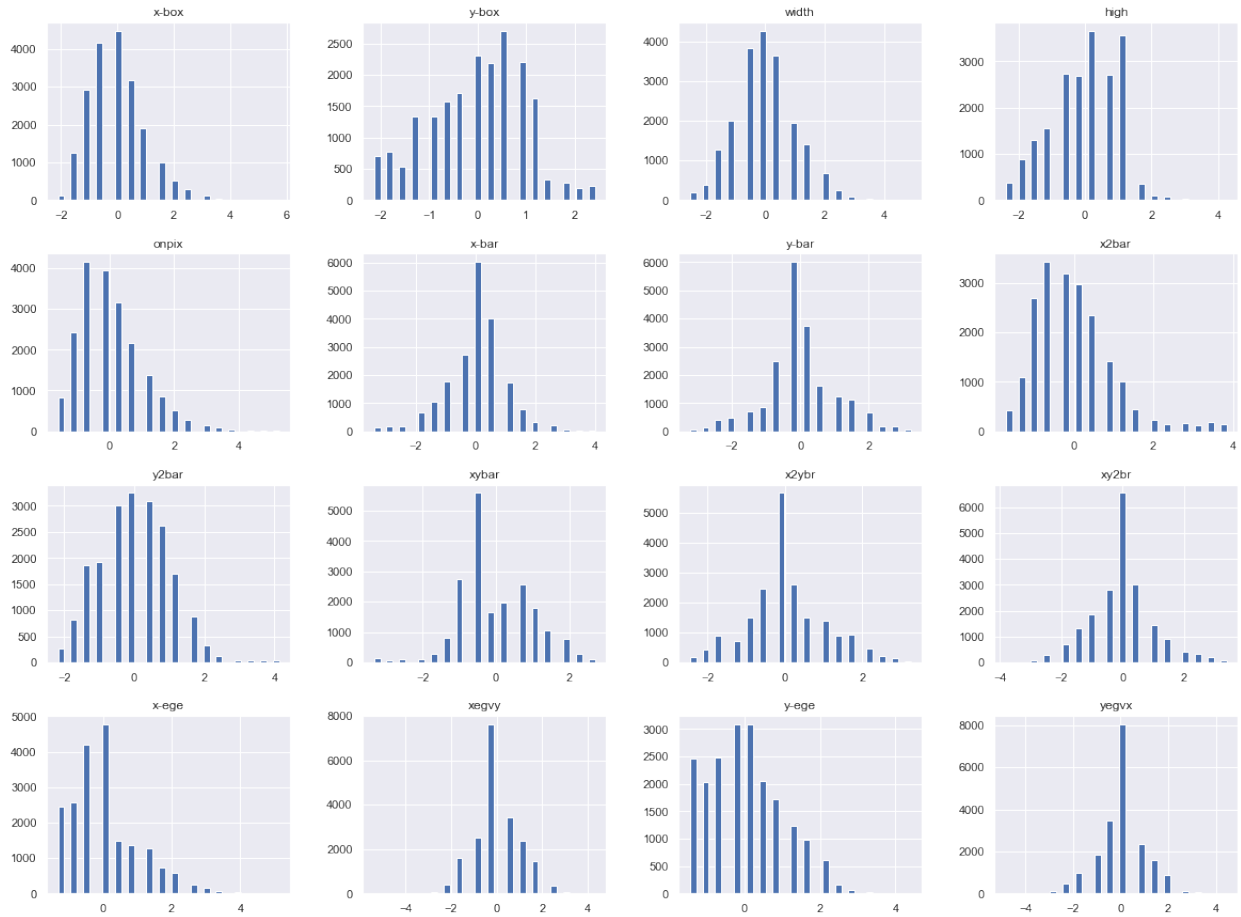

Figure 3: Histograms showing the distributions of each of the (standardised) independent variables.

There are several further measures we can take to scrutinise the assumption of normality more closely. A more informative graphical approach is to create normal quantile-quantile (QQ) plots for each independent variable. These are plots of the quantiles of the sample distribution against the theoretical quantiles of a standard Gaussian distribution, and so a perfect match is obtained when the quantiles are plotted along the line $y = x$ [29]. It is important that we standardise each of our variables for these plots

to be meaningful, which is done by subtracting the mean of each variables from itself, and dividing by the standard deviation. The QQ plots for the independent variables are given in Fig. 4. From this graphical analysis it appears that almost none of the independent variables follow a Gaussian distribution. Interestingly, several variables for which the Gaussian assumption seemed to hold reasonably well by examining their distribution as a histogram, seem to be decisively poorly modelled by a Gaussian distribution when we look at their normal QQ plot. One such example is 'width', which appears to neatly follow a Gaussian distribution based on the histogram we plotted, but actually seems to be positively skewed when we examine it's QQ plot, meaning it's distribution is asymmetrical and more heavily concentrated below the mean.

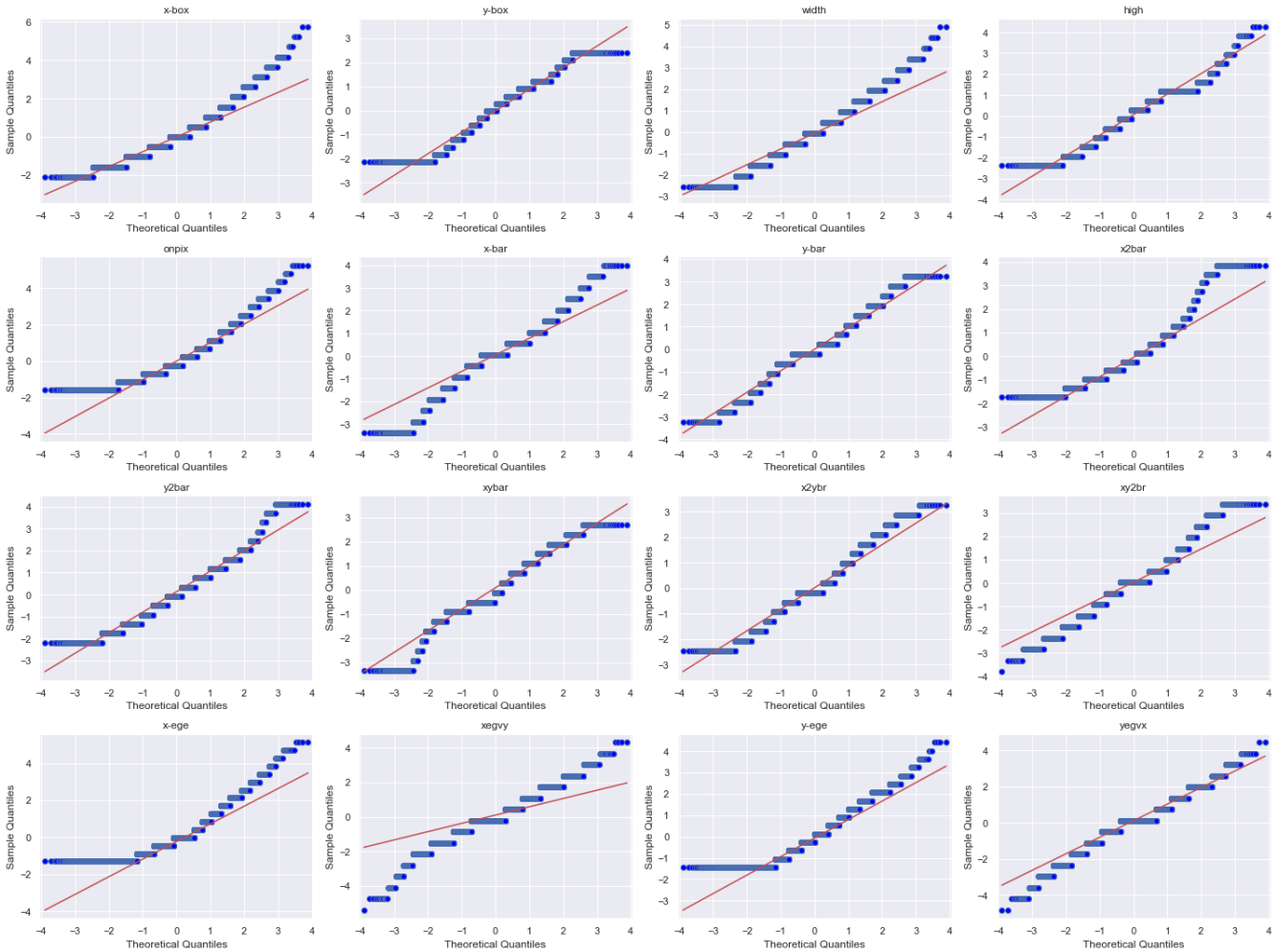

Figure 4: Normal QQ plots for each of the (standardised) independent variables.

While it is possible to individually observe the QQ plot for every independent variable in order to make a judgement on the appropriateness of the Gaussian assumption, it is far more efficient and reliable to perform formal statistical tests [30]; there are several tests we can perform to test the assumption [29]. Although the Shapiro-Wilks test is often described as one of the most powerful tests for symmetric and asymmetric distributions, it was originally only suitable for sample sizes smaller than 50 data points, and even with improvement can only be used for samples of up to $5,000$ data points [30]. As our data set contains $20,000$ points, the Shapiro-Wilks test is inadequate. A more appropriate group of tests for a data set of this size are empirical distribution function (EDF) tests. This group of tests work by comparing the EDF of the sample to the cumulative distribution function (CDF) of the standard normal distribution, denoted $\Phi(X)$. Denoting the ordered observations of a single variable $\mathbf{X}$ by $x_{(1)} \leq x_{(2)} ... \leq x_{(N)}$, the EDF of $\mathbf{X}$ is given by:

$$F_N(x) = \frac{\sum_{i=1}^{N}[x_{(i)} \leq x]}{N} \tag{27}$$

For each x in $\{x_{(1)}, x_{(2)}, ..., x_{(N)}\}$, where $[x_{(i)} \leq x] = 1$ if $x_{(1)} \leq x$ and 0 otherwise. The Anderson-Darling (AD) test is usually recommended as the most powerful EDF test [31], and is given by the test statistic:

$$\begin{aligned} AD &= N \int_{-\infty}^{\infty} (F_N(x) - \Phi(x))^2 (\Phi(x)(1 - \Phi(x)))^{-1} d\Phi(x) \\ &= -\sum_{i=1}^{N} \left(\frac{(2i-1)\left(\Phi\left(y_{(i)}\right) + log(1 - \Phi\left(y_{(N+1-i)}\right)\right)}{N} \right) - N \end{aligned} \tag{28}$$

Where $y_{(i)} = \left(x_{(i)} - \bar{x}\right)/s$, $\bar{x}$ is the sample mean and s is the sample standard deviation. For this test we take the null hypothesis to be that the variable follows a Gaussian distribution, and we reject this hypothesis for a given significance level when the AD test statistic surpasses the critical value. By running this test on each independent variable, we found that the Gaussian assumption was conclusively rejected at the $1\%, 2.5\%, 5\%, 10\%$ and 15% significance levels.

From the graphical and analytical evidence available, it appears that the Gaussian assumption we may have made from looking only at the histograms for the distribution of the independent variables is incorrect. We do however proceed under this assumption as we wish to compare the usefulness of the Gaussian Naive-Bayes classifier with other popular classifiers, and such cases where the model assumptions are not properly met are not excluded as useful examples of the potential limitations of Gaussian Naive-Bayes.

Under the Gaussian assumption we obtain following:

$$X_p|Y = C^k \sim N(\mu_{pk}, \sigma^2_{pk}) \implies f_{kp}(x_p) = \frac{1}{\sqrt{2\pi\sigma^2_{pk}}} \exp\left(-\frac{(x_p - \mu_{pk})^2}{2\sigma^2_{pk}}\right) \tag{29}$$

Estimating the values of μ_{pk} and σ^2_{pk} for $p = 1, ..., P$ and $k = 1, ..., K$ is a simple maximum likelihood estimation task. Within a given class, we will denote the number of data instances as N_k such that $\sum_{k=1}^{K} N_k = N$. Then we can take the sub-set of data for which $Y = C^k$ and obtain the likelihood for a class C^k and independent variable p:

$$\begin{aligned} L(\mu_{pk}, \sigma^2_{pk}) &= \prod_{i=1}^{N_k} \left(2\pi\sigma^2_{pk}\right)^{-\frac{1}{2}} \exp\left(-\frac{(x_i p - \mu_{pk})^2}{2\sigma^2_{pk}}\right) \\ &= \left(2\pi\sigma^2_{pk}\right)^{-\frac{N_k}{2}} exp\left(-\frac{1}{2\sigma^2_{pk}} \sum_{i=1}^{N_k} (x_{ip} - \mu_{pk})^2\right) \end{aligned} \tag{30}$$

As was noted in 3.1.3, maximising the function $\ell(\mu_{pk}, \sigma^2_{pk}) \equiv logL(\mu_{pk}, \sigma^2_{pk})$ is equivalent to maximising $L(\mu_{pk}, \sigma^2_{pk})$ so we proceed by taking the natural logarithm of both sides of (22):

$$\ell(\mu_{pk}, \sigma^2_{pk}) = -\frac{N_k}{2} log\left(2\pi\sigma^2_{pk}\right) - \frac{1}{2\sigma^2_{pk}} \sum_{i=1}^{N_k} (x_{ip} - \mu_{pk})^2 \tag{31}$$

Then we can take first and second partial derivatives to obtain the score vector and Hessian matrix:

$$\begin{aligned} \mathbf{u}(\mu_{pk}, \sigma^2_{pk}) &= \begin{bmatrix} \frac{\partial \ell(\mu_{pk}, \sigma^2_{pk})}{\partial \mu_{pk}} \\ \frac{\partial \ell(\mu_{pk}, \sigma^2_{pk})}{\partial \sigma^2_{pk}} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sigma^2_{pk}} \sum_{i=1}^{N_k} (x_{ip} - \mu_{pk}) \\ -\frac{N_k}{2\sigma^2_{pk}} + \frac{1}{2\left(\sigma^2_{pk}\right)^2} \sum_{i=1}^{N_k} (x_{ip} - \mu_{pk})^2 \end{bmatrix} \\ \mathbf{H}(\mu_{pk}, \sigma^2_{pk}) &= \begin{bmatrix} \frac{\partial^2 \ell(\mu_{pk}, \sigma^2_{pk})}{\partial \mu^2_{pk}} & \frac{\partial^2 \ell(\mu_{pk}, \sigma^2_{pk})}{\partial \mu_{pk} \partial \sigma_{pk}}^2 \\ \frac{\partial^2 \ell(\mu_{pk}, \sigma^2_{pk})}{\partial \sigma^2_{pk} \partial \mu_{pk}} & \frac{\partial^2 \ell(\mu_{pk}, \sigma^2_{pk})}{\partial \left(\sigma^2_{pk}\right)^2} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial}{\partial \mu_{pk}} \left(\frac{1}{\sigma^2_{pk}} \sum_{i=1}^{N_k} (x_{ip} - \mu_{pk})\right) & \frac{\partial}{\partial \mu_{pk}} \left(\frac{1}{\sigma^2_{pk}} \sum_{i=1}^{N_k} (x_{ip} - \mu_{pk})\right) \\ \frac{\partial}{\partial \mu_{pk}} \left(-\frac{N_k}{2\sigma^2_{pk}} + \frac{1}{2\left(\sigma^2_{pk}\right)^2} \sum_{i=1}^{N_k} (x_{ip} - \mu_{pk})^2\right) & \frac{\partial}{\partial \mu_{pk}} \left(-\frac{N_k}{2\sigma^2_{pk}} + \frac{1}{2\left(\sigma^2_{pk}\right)^2} \sum_{i=1}^{N_k} (x_{ip} - \mu_{pk})^2\right) \end{bmatrix} \\ &= \begin{bmatrix} -\frac{N_k}{\sigma^2_{pk}} & -\frac{1}{(\sigma^2_{pk})^2} \sum_{i=1}^{N_k} (x_{ip} - \mu_{pk})^2 \\ -\frac{1}{(\sigma^2_{pk})^2} \sum_{i=1}^{N_k} (x_{ip} - \mu_{pk})^2 & \frac{N_k}{2\left(\sigma^2_{pk}\right)^2} - \frac{1}{\left(\sigma^2_{pk}\right)^3} \sum_{i=1}^{N_k} (x_{ip} - \mu_{pk})^2 \end{bmatrix} \end{aligned} \tag{32}$$

So we know that $(\hat{\mu_{pk}}, \hat{\sigma^2_{pk}})$ solve the two equations $u_1(\mu_{pk}, \sigma^2_{pk})$ and $u_2(\mu_{pk}, \sigma^2_{pk})$. These solutions are easy to find:

$$\hat{\mu_{pk}} = \frac{1}{N_k} \sum_{i=1}^{N_k} x_{ip} = \overline{x_p}^{(k)}, \quad \hat{\sigma^2_{pk}} = \frac{1}{N_k} \sum_{i=1}^{N_k} (x_{ip} - \overline{x_p}^{(k)})^2, \quad \mathbf{H}(\hat{\mu_{pk}}, \hat{\sigma^2_{pk}}) = \begin{bmatrix} -\frac{N_k}{\hat{\sigma^2_{pk}}} & 0 \\ 0 & -\frac{N_k}{2\left(\hat{\sigma^2_{pk}}\right)^2} \end{bmatrix} \tag{33}$$

Note that we have chosen $\overline{x_p}^{(k)}$ to denote the sample mean of x_p within the observations of response k. The Hessian matrix at $(\hat{\mu_{pk}}, \hat{\sigma^2_{pk}})$ is negative definite, so we have shown that our solution is not only a stationary point, but also a maximum [32, p. 212-213]. In summary, we model each independent variable as a Gaussian distribution, and take the sample mean and sample variance to be the parameters of this distribution.

3.3 K-nearest neighbours

In contrast to the models we have discussed so far, the KNN classifier, first described by Fix and Hodges (1951) [33], is neither generative nor discriminative. In fact, KNN does not build a probabilistic model of the dependent variable at all; it is a non-parametric, memory-based algorithm that simply stores the training data for access at the time of classification. Training data is stored as labelled P-dimensional vectors; given a new query vector of independent variables $\mathbf{x}^{(q)}$ the L (changed from K to avoid

confusion with class labels) training data instances with smallest Euclidean distance $\|\mathbf{x}^{(t)} - \mathbf{x}^{(q)}\|$ in P-dimensional space are found, and the modal class label among these L 'nearest neighbours' is taken as the predicted class of the query input. In the case of a tie in the most common class label, a label is selected at random from the tied classes. Intuitively, we will expect this to result in quite long prediction times.

4 Deep learning

4.1 What are neural networks, and what are they used for?

In mathematics, a function is a mapping of a domain of inputs to a range of outputs, such that every possible input maps to one single output. At the most abstract level, we can view neural networks simply as function approximators. In fact, neural networks are known as 'universal function approximators', meaning they are capable of learning any continuous function to any desired degree of accuracy. To make this concept precise, given any continuous function $f(x)$ with domain of inputs $\mathcal{D}$ and a sufficient number of data points $(x, f(x))$, a neural network can be found with output $\hat{f}(x)$ such that for any desired accuracy $\epsilon > 0$ we have $|\hat{f}(x) - f(x)| < \epsilon, \forall x \in \mathcal{D}$ [34]. The implications of this property in machine learning, and specifically in this paper in the case of multi-class classification, are enormous, as we are presented with a potential method by which to predict an output from any input data, to as high of a degree of accuracy as we wish.

Before moving on, let us define the notation that shall be used as we explore the mathematical workings behind how neural networks are able to accomplish such tasks. We shall directly inherit the notation of the design matrix $\mathbf{X}$ as described in Section 3, and shall modify the notation of our observation matrix $\mathbf{y}$ into a more traditional form one-hot encoding as follows:

- $\mathbf{y}$ is the $N \times K$ matrix of observations where the i-th row is the one-hot encoded response of the i-th observation, written as $\mathbf{y}_i = (y_{i1}, y_{i2}, ..., y_{iK})$ where:

$$y_{ik} = \begin{cases} 1 & \text{if } Y_i = C^k,\ k \in \{1, ..., K\} \\ 0 & \text{otherwise} \end{cases} \tag{34}$$

  This implies a 1 at the index matching the class that the instance is in, and 0 at every other index.

4.2 Describing feed-forward neural networks in the context of multi-class classification

4.2.1 How neural networks make predictions

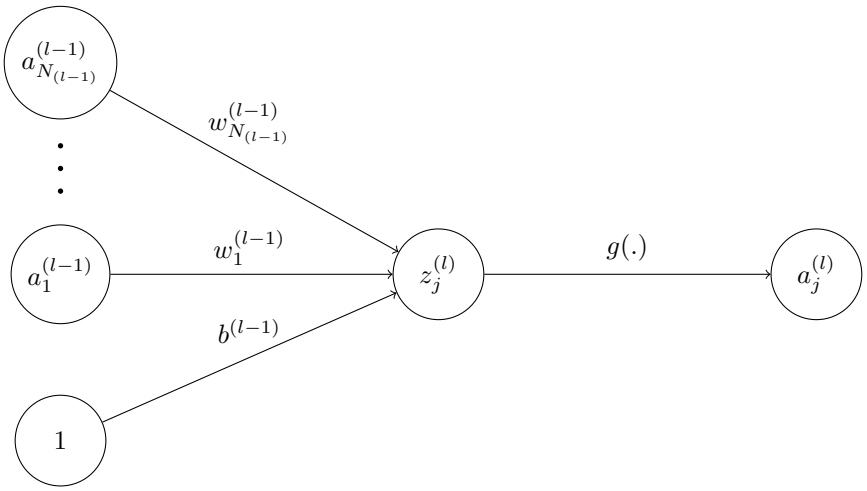

Figure 5: An example of a single neuron in an abitrary layer of a neural network

We shall demonstrate the workings of a neural network with an arbitrarily sized network of L hidden layers, and N_l neurons in layer l, $\quad l = 1, ..., L$. The most simple concept that we first examine is the basic units from which neural networks are built from: hidden neurons. A neuron can be described simply as a non-linear transformation $g(.)$ (known as the activation function) of a linear combination of variables. Let us begin with a generic input, $\mathbf{x}$, which forms an input layer of $P + 1$ nodes, each corresponding to a single value from the vector. The neurons in the first hidden layer take a linear combination of our input vector which we shall call $z(\mathbf{x})$, and pass the result to our activation function, as follows:

$$a_i^{(1)} = g\left(b_i^{(1)} + w_{i1}^{(1)}x_1 + ... + w_{iP}^{(1)}x_P\right) = g\left(b_i^{(1)} + \sum_{j=1}^{P} w_{ij}^{(1)}x_j\right) = g\left(z_i^{(1)}\right), \quad i = 1, ..., N_1 \tag{35}$$

Our choice of activation function $g(.)$ shall be discussed in more depth later in this section, but for now it is sufficient to note that it is a non-linear function, differentiable over it's entire domain. We proceed to each consecutive layer of the neural network in

the same way, feeding the N_{l-1} neurons of each hidden layer into the N_l neurons of the next, noting that we introduce an $a_0^{(l-1)} = 1$ dummy neuron in each layer, so that we may add the bias term $b_i^{(l)}$ in each $z_i^{(l)}$:

$$a_i^{(l)} = g\left(b_i^{(l)} + w_{i1}^{(l)} a_1^{(l-1)} + ... + w_{iN_{l-1}}^{(l)} a_{N_{l-1}}^{(l-1)}\right) = g\left(b_i^{(l)} + \sum_{j=1}^{N_{l-1}} w_{ij}^{(l)} a_j^{(l-1)}\right) = g\left(z_i^{(l)}\right), \quad i = 1, ..., N_l, \quad l = 2, ..., L-1, \tag{36}$$

We have graphically displayed the process of feeding layer $l-1$ into l in order to compute a non-linear transformation in Figure 5.

The output layer is a special case, containing K neurons (one for each potential output class), and uses a special activation function called the softmax function, that we shall denote $S(.)$. The details of this function will be discussed later, but it is for now important to note that it maps each of the K linear combinations of incoming neurons to a value in the set $(0, 1)$, such that the sum total of the values is one. These outputs are the class probabilities for an input vector, and so we classify the instance into the class with the highest probability. The output layer is constructed as follows:

$$\hat{y}_k = a_k^{(L)} = S\left(b_k^{(L)} + w_{k1}^{(L)} a_1^{(L-1)} + ... + w_{kN_{L-1}}^{(L)} a_{N_{L-1}}^{(L-1)}\right) = S\left(b_k^{(L)} + \sum_{j=1}^{N_{L-1}} w_{kj}^{(L)} a_j^{(L-1)}\right) = S\left(z_k^{(L)}\right), \quad k = 1, ..., K \tag{37}$$

For any neuron given by $a_j^{(l)} = g\left(b_j^{(l)} + \sum_{j=1}^{N_{l-1}} w_{ij}^{(l)} a_j^{(l-1)}\right)$ we call $b_i^{(l)}$ the biases, and $w_{ij}^{(l)}$ the weights. From this generalised form, we are able to construct neural networks with the demonstrated structure of any size we wish. An example of a neural network with two hidden layers can be viewed in Figure 6. By restricting ourselves to neural networks containing no closed directed cycles, as we have throughout this section, we can denote these neural networks as 'feed-forward' neural networks [35].

When predicting the class label of a new data point, the values of the independent variables are passed to the input layer, the linear combinations of neurons are fed through the activation functions in each layer of the network to construct complex non-linear combinations of the input values, and finally the values of the output layer are passed through the softmax function to obtain the probabilities of the data point belonging to each class. The result is return as a one-hot encoded vector (as described in Eq. 34), with a 1 at the index k corresponding to the class C^k with the highest class probability, and 0 at every other index. We are left with the issue of how to determine the best values for the weights and biases within the network. Motivated by this, we provide two rules that a cost function for neural networks must follow, and how the backpropagation algorithm is used to minimise this cost function.

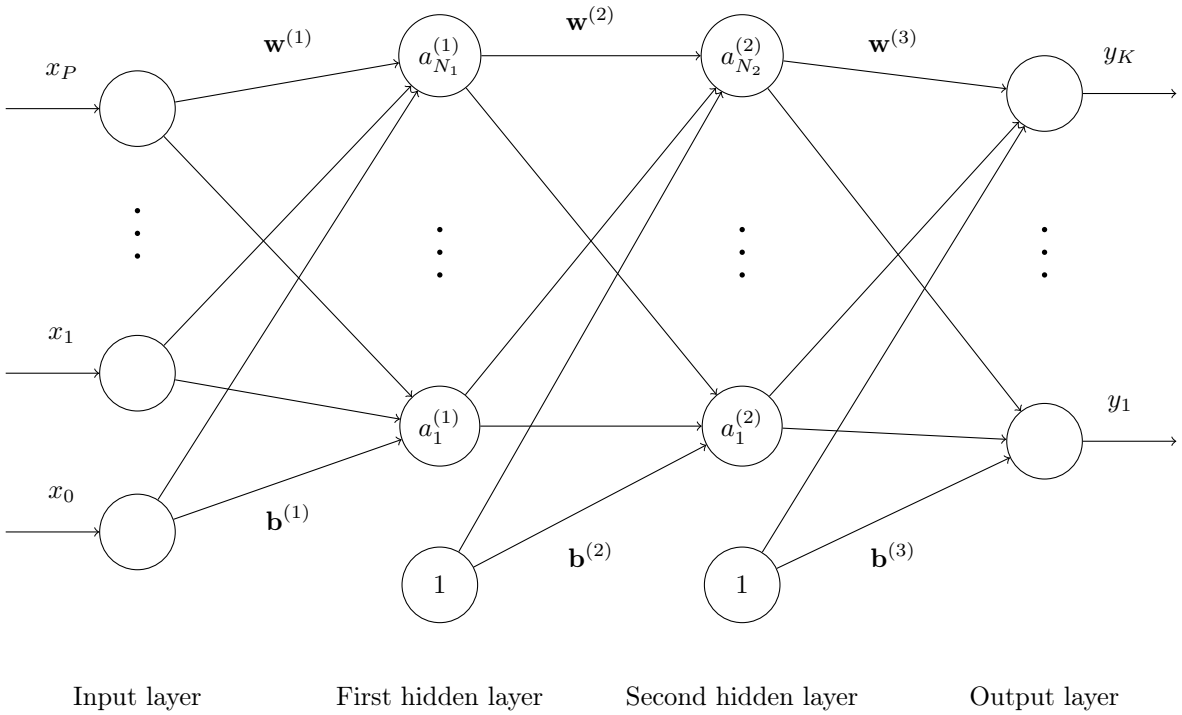

Figure 6: An example of a neural network with two hidden layers

4.2.2 How neural networks learn: mini-batch gradient descent and backpropagation

Despite the large number of coefficients in large neural networks, the problem of finding the coefficients that give the greatest accuracy remains as simple as minimising a chosen cost function (we will interchangeably refer to this as a loss function) [36].

Specifically, given a cost function $C(\boldsymbol{\theta})$, we look for the values of all the weights and biases that minimise C. We have some freedom in choice of C, but our choice must necessarily satisfy two assumptions [34]:

- The cost function may be calculated by averaging over the cost function of each training instance: $C = \frac{1}{N}\sum_{i=1}^{N} C_i$. This assumption is necessary for us to obtain the partial derivatives of the cost function with respect to all of the weights and biases by averaging over the partial derivatives of the cost function with respect to these coefficients for some number of training examples.
- The cost function is a function of $\mathbf{a}^{(L)}$, the network outputs. The network outputs are necessary variables of the cost function as we wish to optimise their value by minimising the cost function with respect to them. We will choose frequently the notation of $\boldsymbol{\theta} = \left(\boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(2)}, ..., \boldsymbol{\theta}^{(L)}\right)$ where $\boldsymbol{\theta}^{(l)} = \left(b_1^{(l)}, ..., b_{N_{L-1}}^{(l)}, w_{11}^{(l)}, w_{12}^{(l)}, ..., w_{N_l N_{l-1}}^{(l)}\right)$ for $l = 1, ..., L$, such that $\boldsymbol{\theta}$ is the vector of every weight and bias in the network. Since $\mathbf{a}^{(L)}$ is itself a function of $\boldsymbol{\theta}$, we can further say that the cost function is therefore a function of $\boldsymbol{\theta}$.

Our aim is to find the set of weights and biases in the network that minimise the chosen cost function. We must select a variant of gradient descent for this task. Gradient descent is simply the process of updating the networks parameters in the direction that is opposite to the gradient of the cost function, in order to 'descend' to the cost function's global or local minimum [37]. There are three alternative version of gradient descent, given as follows:

- Batch gradient descent: Computing the average of the cost functions and partial derivatives for every training instance before every update of the network parameters.
- Stochastic gradient descent: Computing the cost function and partial derivatives for a single training instance and updating the network parameters for every instance.
- Mini-batch gradient descent: Computing the cost function for a number of training examples (called a mini-batch) and updating the parameters for each mini-batch.

We select mini-batch gradient descent for this task over batch gradient descent and stochastic gradient descent as the former is very slow and usually performs many redundant computations while the latter struggles to converge to the exact minimum [37]. For generality, we will consider mini-batches of size $n < N$ and a learning rate of $\eta > 0$, then we begin by randomly assigning the initial weights and biases, and then updating each parameter by the following rule:

$$\boldsymbol{\theta}^{(m+1)} = \boldsymbol{\theta}^{(m)} - \eta \nabla_{\boldsymbol{\theta}^{(m)}} C_n \left(\boldsymbol{\theta}^{(m)}; \mathbf{x}_{(i:i+n)}, \mathbf{y}_{(i:i+n)}\right) \tag{38}$$

Where C_n is the cost function averaged over the mini-batch.

In order to find $\nabla_{\boldsymbol{\theta}} C\left(\boldsymbol{\theta}; \mathbf{x}, \mathbf{y}\right)$ for any single training instance, we need to compute the partial derivative of the cost function for that instance with respect to every weight and bias in the network, which can become a very large number for dense neural networks with many neurons. The most efficient way to compute all of these partial derivatives is by the backpropagation algorithm.

The backpropagation algorithm relies on four equations to recursively propagate backwards from the output layer through the network. We begin by defining the error of the i-th neuron in the l-th layer as $\frac{\partial C}{\partial z_i^{(l)}}$. By using the notation of $\delta_i^{(l)} \equiv \frac{\partial C}{\partial z_i^{(l)}}$ to denote a neuron's error, as is common in the literature [35, 34, 36], we will simplify the notation of each step in the final line, after clearly demonstrating the necessary calculus. Then the steps of the backpropagation algorithm are given as follows:

1. Computing the errors for the neurons of the output layer:

$$\begin{aligned}
\frac{\partial C}{\partial z_j^{(L)}} &= \frac{\partial C}{\partial a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \quad \text{(Chain rule)} \\
&= \frac{\partial C}{\partial a_j^{(L)}} g'\left(z_j^{(L)}\right) \\
&\equiv \delta_j^{(L)}
\end{aligned} \tag{39}$$

2. Computing the errors for the neurons in layer $l \in \{1, ..., L-1\}$ given the errors of the neurons in layer $l+1$:

$$\begin{aligned}
\frac{\partial C}{\partial z_j^{(l)}} &= \sum_{i=1}^{N_l} \frac{\partial C}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial z_j^{(l)}} \quad \text{(Chain rule)} \\
&= \sum_{i=1}^{N_l} \frac{\partial C}{\partial a_i^{(l+1)}} g'\left(z_i^{(l+1)}\right) \frac{\partial z_i^{(l+1)}}{\partial z_j^{(l)}} \quad \text{(Applying the first step of the algorithm)} \\
&= \sum_{i=1}^{N_l} \frac{\partial C}{\partial a_i^{(l+1)}} g'\left(z_i^{(l+1)}\right) w_{ij}^{(l+1)} g'\left(z_j^{(l)}\right) \\
&= \sum_{i=1}^{N_l} \delta_i^{(l+1)} w_{ij}^{(l+1)} g'\left(z_j^{(l)}\right)
\end{aligned} \tag{40}$$

3. Computing the partial derivative of the cost function with respect to the network weights:

$$\begin{aligned}\frac{\partial C}{\partial w_{ij}^{(l)}} &= \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}}\frac{\partial C}{\partial z_j^{(l)}} \quad \text{(Chain rule)} \\ &= \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}}\sum_{i=1}^{N_l}\frac{\partial C}{\partial a_i^{(l+1)}}g'\left(z_i^{(l+1)}\right)w_{ij}^{(l+1)}g'\left(z_j^{(l)}\right) \quad \text{(Applying the second step)} \\ &= a_j^{(l-1)}\sum_{i=1}^{N_l}\frac{\partial C}{\partial a_i^{(l+1)}}g'\left(z_i^{(l+1)}\right)w_{ij}^{(l+1)}g'\left(z_j^{(l)}\right) \\ &= a_j^{(l-1)}\delta_i^{(l)}\end{aligned} \tag{41}$$

4. Computing the partial derivative of the cost function with respect to the network biases:

$$\begin{aligned}\frac{\partial C}{\partial b_j^{(l)}} &= \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}}\frac{\partial C}{\partial z_j^{(l)}} \quad \text{(Chain rule)} \\ &= \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}}\sum_{i=1}^{N_l}\frac{\partial C}{\partial a_i^{(l+1)}}g'\left(z_i^{(l+1)}\right)w_{ij}^{(l+1)}g'\left(z_j^{(l)}\right) \quad \text{(Applying the second step)} \\ &= \sum_{i=1}^{N_l}\frac{\partial C}{\partial a_i^{(l+1)}}g'\left(z_i^{(l+1)}\right)w_{ij}^{(l+1)}g'\left(z_j^{(l)}\right) \\ &= \delta_j^{(l)}\end{aligned} \tag{42}$$

From these steps we see that by calculating the errors in the output layer of the network, we can recursively calculate the errors for the neurons in each previous layer. We then obtain the partial derivatives with respect to weights and biases by simply applying the chain rule. The backpropagation algorithm provides the fastest way to calculate all of these partial derivatives, requiring a single forwards pass through the network to calculate the outputs, and then a single backwards pass to calculate every partial derivative.

Putting everything we have explained, we are able to train the network parameters by the following process:

1. Randomly assign the initial weights and biases.
2. Use backpropagation to calculate the partial derivatives of the chosen cost function with respect to the cost function for a mini-batch of training instances.
3. Update the weights and biases using mini-batch gradient descent.
4. Repeat steps 2 and 3 for each mini-batch in the training data. Each time steps 2 and 4 are repeated is known as a training 'iteration'.
5. Repeat steps 2-4 for a selected number of 'epochs'.

By this algorithm we allow the network's parameters to be updated using information from every training data instance as many times as the number of epochs we select. We note that a training iteration is defined as each time a mini-batch is processed in training, and a training epoch is each time the entire training set is processed. Although we have demonstrated the fastest known way to train a feed-forward neural network, as these networks become large, containing large numbers of neurons, the numbers of weights and biases to estimate becomes huge, meaning training times are to be expected to be significantly longer than the models we have described in the previous section.

For the sake of generality, we have described the backpropagation algorithm for an unspecified cost function $C(\boldsymbol{\theta})$, but it is worth noting that the cost function we shall use for our implementation is known as the categorical cross-entropy function, generally considered the most appropriate choice for multi-class classification problems [38, p. 84]. To derive this function, we begin modelling each instance by a categorical distribution. We shall define an $N \times K$ matrix $\hat{\mathbf{y}}$ where the i-th row is given by $\hat{\mathbf{y}}_i = (\hat{y}_{i1}, \hat{y}_{i2}, ..., \hat{y}_{iK})$ such that $\hat{y}_{ik}$ is the predicted class probability of the i-th observation belonging to class k using the softmax function in the output layer. Then the probability mass function of the categorical distribution is:

$$f(\mathbf{y}_i; \boldsymbol{\theta}) = \prod_{k=1}^{K}\hat{y}_{ik}^{y_{ik}} \tag{43}$$

By multiplying together the probability mass function of each instance and taking the resulting function as a function of $\mathbf{w}$ and $\mathbf{b}$ we can obtain the likelihood function and hence the log-likelihood function:

$$L(\boldsymbol{\theta}) = \prod_{i=1}^{N}\prod_{k=1}^{K} \hat{y}_{ik}^{y_{ik}}$$
$$\ell(\boldsymbol{\theta}) = \sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} log(\hat{y}_{ik}) \tag{44}$$

Rather than maximising $\ell(\boldsymbol{\theta})$ we instead minimise $-\ell(\boldsymbol{\theta})$, which we call the categorical cross-entropy function. Hence our cost function is formally given as:

$$C(\boldsymbol{\theta}) = -\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik} log(\hat{y}_{ik}) \tag{45}$$

The summation bounds can of course be adapted so that the function can be calculated for a mini-batch rather than the total data set.

4.2.3 Activation functions

So far we have referred to the activation functions in each layer of the network generically as a non-linear differentiable function, (except int the output layer, where we specified a softmax function). However, activation functions are an important and largely researched topic [39, 40] and the selection of an appropriate activation function can be vital for efficient and accurate training of the network. Here we discuss the issue in limited depth, as it relates to practical implementation. For simplicity, we will limit our discussion to the most relevant activation functions for our purpose: the sigmoid function, the ReLU function, the PReLU function and the softmax function. Although other activation functions exist such as the binary step function, used in multilayer perceptrons [34, 39, 41], most such functions are either outdated or inappropriate choices for multi-class classification problems.

The sigmoid function (also called the logistic function) was once the most popular choice of activation function for neurons in the hidden layers and output layer of networks, as described in Jordan (1995) [42]. The paper actually concluded that the sigmoid function was not in particular a better choice of activation function than many other general non-linear functions. This belief is reaffirmed by Stinchcombe and White (1989), who rigorously proved that sigmoid activation functions are not a requirement for the universal approximation theorem for neural networks to hold [43]. The logistic sigmoid function and is given by the following formula:

$$\sigma(z) = \frac{1}{1+\exp(-z)} \tag{46}$$

Shown in Figure 13 of Appendix C, the sigmoid function transforms the real line to the range $(0, 1)$. The result of transforming every input to such a small range is that large changes in input to the function result in very small changes in output, and hence small gradients, as illustrated again by Figure 13, and given by the following formula:

$$\begin{aligned}\sigma'(z) &= \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \sigma(z)(1-\sigma(z))\end{aligned} \tag{47}$$

From Equation 40 we know that backpropagation multiplies the gradient of each layer back through the network as it passes back through the network to earlier layers. When these gradients are less than one (which is very often the case, especially with conventional weight/bias initialisation procedures [34, Ch. 5]), the multiplicative effect of the backpropagation algorithm causes gradients to get smaller as the layers get closer to the input layer, resulting in only very tiny changes to the weights and biases of these shallow layers in gradient descent. This is known as the *vanishing gradient problem* [44], and can greatly increase training time. In fact the vanishing gradient problem is the most common manifestation of the larger issue presented by the sigmoid function, the *unstable gradient problem*, which also encompasses the related *exploding gradient problem* [34]. As we aim to demonstrate the optimal performance of neural networks in this paper, we are motivated to look for a better alternative activation function than the sigmoid function.

The rectified linear unit function, or ReLU, is one of the most popular activation functions for modern neural networks [39, 45]. The function is given as follows:

$$g(z) = \max(0, z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases} \tag{48}$$

The function and it's first derivative are displayed in Figure 14 of Appendix C. Being a piece-wise function, ReLU introduces an interesting non-linearity in that it outputs zero for all non-positive inputs, resulting in only neurons with positive linear transformations being active [39]. The first derivative of the ReLU function is given as follows:

$$g'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases} \tag{49}$$

We note the property that $g'(z) = 1, \forall z > 0$, which prevents the error of shallow layers from shrinking towards zero in backpropagation, and hence providing a partial solution to the vanishing gradient problem of the sigmoid function. In fact neural networks using ReLU activation functions (and it's variants, which we shall discuss next) empirically outperform and benefit more from increased depth than those using sigmoid activation functions [46, 47].

However, ReLU is not a complete solution, as a result of the property that $g'(z) = 0, \forall z \leq 0$. For this reason, it is possible for ReLU neurons to 'die' if the gradient becomes zero, as the weights and biases will stop updating in gradient descent. This is known as the *dying ReLU problem*, and one potential solution is to carefully select our procedure for initialising the weights and biases of the network [48, 49]. Specifically, we will use the initialisation method put forward by He et al. (2015) which uses a zero-mean Gaussian with variance $2/N_l$ for the weights, and zero for the biases [46]. An improvement is obtained by slightly modifying ReLU to be given by:

$$g(z) = \max(0, z) = \begin{cases} z & \text{if } z > 0 \\ az & \text{otherwise} \end{cases} \tag{50}$$

By fixing $a = 0.01$ (or another very small constant) we obtain the leaky ReLU function (LReLU) [39, 46, 47], which has been found to reduce training time, but have little impact on accuracy as compared with the standard ReLU activation function [46, 47]. When a is treated as a learnable parameter, we have the parameterised ReLU function (PReLU) [39, 46]. Without loss of generality, we can define the first derivative for both these cases by the following formula:

$$g'(z) = \begin{cases} 1 & \text{if } z > 0 \\ a & \text{otherwise} \end{cases} \tag{51}$$

Clearly in the case that $a \neq 0$ we no longer have a zero gradient for non-positive inputs, mitigating the risk of the zero-gradient problem as the weights and biases will still receive (small) updates, maintaining the possibility of converging to an optimal value. In fact, we can see that the standard ReLU function which we first defined is a special case of this more general form, with $a = 0$ fixed. To avoid the need to tune additional parameters, we keep PReLU as a fall-back if we encounter issues with using our preferred activation function of ReLU.

The final activation function that requires discussion is the softmax activation function. Aforementioned, the softmax function maps each of the K linear combinations of incoming neurons in the output layer to a probability of an instance belonging to that class. The function is given by the formula below:

$$S(z_k) = \frac{e^{z_k}}{\sum_{i=1}^{K} e^{z_i}}, \forall k \in \{1, ..., K\} \tag{52}$$

4.3 Regularisation

When implementing a neural network, we begin by building a small baseline model which is unlikely to be able to capture enough important features of the data to make accurate predictions, which we call underfitting. This underfit model will have high bias as it is not able to learn the features of the data sufficiently to make accurate predictions, but low variance as we expect to find similar network parameters when training on different subsets of the data, making predictions precise. As we increase the number of hidden layers and neurons, we increase the information about the training data that the model is able to learn, reducing the model bias and improving performance on the training data, assuming we have allocated a sufficient amount of training data and epochs for the network to effectively learn. Initially, as the bias of the model decreases, we will usually see an improvement in predictions on unseen data. However, as we continue to increase the size of the model, we eventually reach the issue of overfitting. When we include too many hidden layers and neurons, we may find that after several training epochs the performance on the unseen data begins to diminish, as the model is now overfit. The reason for this is that when the model becomes overly complex, it is able to learn the training data set too closely, potentially learning patterns that only appear in a specific training set as a result of random noise and may not be present in unseen data, leading to poor generalisation power [38, p. 104]. Hence, although the overfit model has lower bias than the underfit model, as it is able to more accurately learn the true patterns in the data set, it suffers from high variance as the model will learn different parameters for different training sets as a result of random noise. We would expect such an issue to be exacerbated by smaller training sets. This is known in the machine learning literature as the bias-variance tradeoff, and is usually characterised by the 'U-shaped' curve shown in Figure 7.

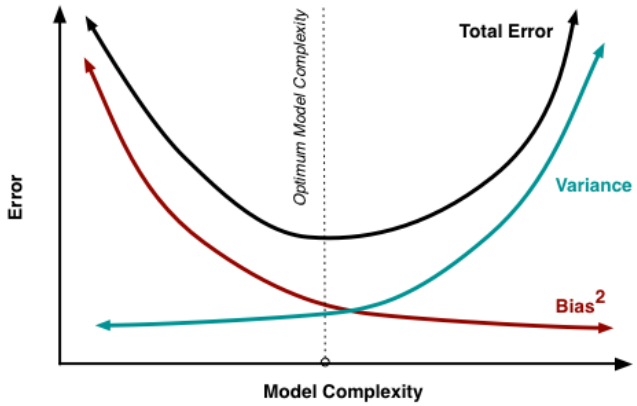

Figure 7: Graph depicting how bias (squared; it can take negative values), variance and total error vary as model complexity is increased [50].

Theoretically, there may be a specific number of hidden layers and units that optimises this trade-off, but there is no set formula to find this model; finding such a model would be infeasible. A far more practical approach is to increase our model to a reasonable complexity and then impose regularisation techniques which restrict the quantity of information that the network is able to learn. The hope is that by limiting the information that the network can learn, it will be forced to focus on the most important patterns which are most likely to generalise to unseen data.

A very common approach for regularisation is weight regularisation, which involves adding an extra term to the cost function in order to force the weights and biases to take small values [38, p. 107]. In particular, we shall choose to use L2 regularisation, which entails adding the L2 norm of the weight vector to the cost (we also used this regularisation method in multinomial logistic regression). The new cost function with L2 regularisation imposed is given by:

$$\tilde{C} = C + \lambda \sum_{l=1}^{L} \sum_{i=1}^{N_l} \sum_{j=1}^{N_{l-1}} \left(w_{ij}^{(l)} \right)^2 \tag{53}$$

$\lambda > 0$ again controls the strength of the penalty term, so making this value too large can result in an underfit network. To understand more clearly why this method works, note that while the partial derivatives of $\tilde{C}$ with respect to the network biases is identical to that for C in Eq. 42, the partial derivatives of $\tilde{C}$ with respect to the weights becomes:

$$\frac{\partial \tilde{C}}{\partial w_{ij}^{(l)}} = \frac{\partial C}{\partial w_{ij}^{(l)}} + 2\lambda w_{ij}^{(l)} \tag{54}$$

Which can be evaluated with Eq. 41. Now considering a single element $w_{ij}^{(l)}$ of $\boldsymbol{\theta}$ in Eq. 38, and assuming a mini-batch size of one for simplicity, we obtain the weight updates for the original cost function C as:

$$w_{ij}^{(l)}{}^{(m+1)} = w_{ij}^{(l)}{}^{(m)} - \eta \frac{\partial C}{\partial w_{ij}^{(l)}} \tag{55}$$

Considering the same element but updating with the regularised cost function we now get the expression:

$$\begin{aligned} w_{ij}^{(l)}{}^{(m+1)} &= w_{ij}^{(l)}{}^{(m)} - \eta \frac{\partial C}{\partial w_{ij}^{(l)}} - 2\eta\lambda w_{ij}^{(l)} \\ &= (1 - 2\eta\lambda)\, w_{ij}^{(l)} - \eta \frac{\partial C}{\partial w_{ij}^{(l)}} \end{aligned} \tag{56}$$

We see that we are rescaling the weights by a factor of $1 - 2\eta\lambda$ at each training iteration, which explains why the method is often referred to as weight decay.

Another regularisation method which is used for neural networks is known as drop out (or dilution), and was well detailed by Srivastava et al. (2014) [51]. The idea is that a layer with dropout assigns a constant probability ϕ to each unit of 'dropping out' of each training iteration, or in other words outputting a zero. After training, it is necessary to multiply the weights of the input edges to each neuron directly succeeding a drop out by $(1 - \phi)$, as during testing such neurons will be connected to ϕ times more neurons than they were on average during training [52, p. 367]. It is suggested that values of ϕ close to zero are generally optimal for the input units, while $\phi = 0.5$ is close to optimal for the hidden layers in the majority of tasks [51]. These values are a useful reference to experiment around using cross-validation to find a value of ϕ that yields good results. The reason that such an algorithm may improve performance is that by removing a sample of neurons at each training iteration, each neuron is forced to learn useful information without co-adaption to any particular set of neighbouring neurons, and without reliance on any particular set of incoming neurons [52, p. 366]. A motivational biological analogy is given by Srivastava et al. (2014), explaining that in sexual

reproduction there is random recombination of genes, usually splitting up co-adapted sets of genes. The reason, then, that sexually reproducing organisms evolve to become well-adapted to their environment is that genes have to adapt to provide useful functions in the presence of a variety of other states of it's surrounding genes. This concept is carried over to drop out regularisation; neurons must learn to provide useful information to the network without co-adaption on other neurons, and therefore become less sensitive to small random noise in the data that provides less information than the true patterns.

5 Empirical analysis

5.1 Implementation strategy

The practical analysis and modelling of the data was carried out with the Pandas, Matplotlib, Scikit-learn and Keras with TensorFlow libraries. Details of how to reproduce these experiments are given in Appendix D. The implementation was divided into three main stages: data analysis, model fitting and evaluation. As previously described when introducing the data set in Section 2, the data set contains $20,000$ data points, no missing data, a categorical response variable with twenty-six possible class labels (which are relatively evenly represented) and sixteen independent variables. We also displayed a correlation matrix and the independent variable distributions in Section 3.2 on Naive-Bayes classifiers. There is no further relevant information for the data analysis, so we will move on to describing the model fitting stage.

5.1.1 Fitting the models

Our first step to fitting the models was to use Z-score standardisation on the independent variables and randomly split the data into training and testing sets. Three different training/testing splits were used: 16,000 training instances to 2,000 testing instances, 8,000 training instances to 11,000 testing instances and 500 training instances to 19, 500 testing instances. For each model we used 5-fold cross-validation with the training set before fitting the model with the full training set, to ensure the model was valid. This process works by first splitting the training data into 5 equally sized sub-samples, then training the data on 4 of those subsets and testing the accuracy of the model on the subset that was not used in training. This is repeated for each subset and the results are averaged to give a cross-validation accuracy. If a cross-validation accuracy of $> 50\%$ was obtained, then the model was considered valid and so it was fit to the total training set and tested on the testing data. In addition, each model was fit and used for prediction five times, and the median computational time for both steps as well as the accuracy score was stored for evaluation. The reason for repeating this process and taking the median was to account for outliers, as the computational time fluctuates depending on the perforance of the hardware at any given moment (for example external demands from other programs may result in an abnormally long training or testing time).

Each of multinomial logistic regression, Gaussian Naive-Bayes and KNN was implemented with the scikit-learn library in Python.

The first model to be fit was multinomial logistic regression. L2 regularisation was introduced to reduce the risk of overfitting to the data, and a grid search, which performs cross-validation over a set of possible values of a hyperparameter, was used to tune the hyperparameter C which defines the inverse of the regularisation strength. With all three training and testing splits multinomial model passed the cross-validation stage.

Next we attempted to fit the Gaussian Naive-Bayes model. The initial cross-validation accuracy was sufficient to pass the model to the training stage for each training/testing split, but was the lowest of all the models in every case, and was not very much improved by dropping the variables that we found to be highly correlated. For the training set of size $16,000$, we found a cross-validation accuracy of 63.52%, which increased only to 65.45% after removing the highly correlated variables ('x-box', 'y-box', 'width', 'high', 'onpix'). A similarly negligible improvement was also found for the smaller training sets.

For K-nearest neighbour the first step was to perform a grid search tuning of the number of nearest neighbours to use for voting. We first search over the range $[1, 10]$, and found that 1 was the optimal value, eliminating a need to search over larger values. This model consistently yielded far higher cross-validation accuracy results than the two previous models, achieving an impressive cross-validation accuracy of 94.41% on the largest training set.

For selecting the number of hidden units for our neural network we began by creating a simple baseline model with 2 hidden layers, each containing 20 hidden RELU neurons. We then 5-fold cross-validated the model with 25 training epochs, and plotted a graph for validation loss against epochs for the first fold. Categorical cross-entropy was selected as our loss function. We used a graph so that we could easily visually compare the results of our larger models with the baseline, and so that validation loss could be tracked over the training epochs to identify whether our models were overfitting. We then created three larger models than the baseline model, each with 100 hidden neurons in both hidden layer. These three models respectively had no regularisation, L2 regularisation and drop out regularisation. We selected a small value of $\lambda = 0.001$ for the L2 regularisation, as we noted that for each training set the large unregularised model outperformed the baseline model and so was capturing relevant information from the data. A large value of λ is more likely to limit the expressiveness of the network and reduce the information it is able to learn from the data. For drop out regularisation we followed the general guidelines given by Srivastava et al. (2018) [51], and by experimenting around the recommended values found that $\phi = 0.4$ for the hidden layers and $\phi = 0.01$ for the input layer yielded best results. The cross-validation loss results for the training set with $16,000$ instances is displayed in Figure 8. show that the unregularised model had the lowest cross-validation loss results, but it also appeared to locally trend upwards at some epochs, likely a result of overfitting. It should be noted that the validation loss at the final epoch for the L2 regularised model is only approximately 0.1 higher than that of the unregularised model, so we may prefer this model due to the fact it does not show sharp fluctuations in

validation loss across the epochs. All three of the larger models returned validation accuracy scores of $94 - 95\%$ so we chose to fit them to the total training set and record their performance metrics.

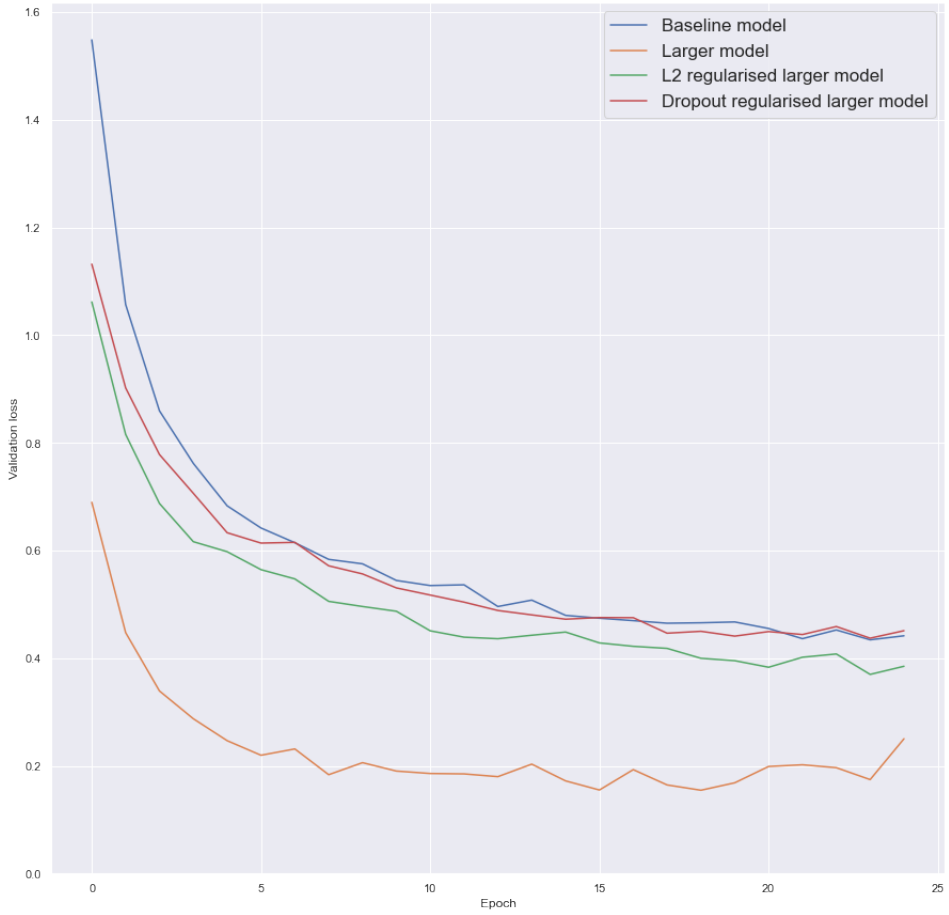

Figure 8: The results of cross-validation loss against epoch for the trained neural networks

5.1.2 Results

In order to properly critique and improve on the the models we implement, it is important that we use informative metrics to measure their performance.

We have chosen to measure how good our models were at classifying instances using an accuracy score, which is defined as the number of correctly classified data points over the total number of data points that have been classified:

$$\frac{1}{N}\sum_{i=1}^{N}[y_i = \hat{y}_i] \tag{57}$$

This result was multiplied by 100 to obtain a percentage of correctly classified results. There are metrics such as the F1 score available to provide an optimal trade-off between precision and accuracy. These more sophisticated metrics amend the problems that can arise from using accuracy scores, such as misleading results for imbalanced data sets [52, p. 87-90]. However, our data set shows no significant class imbalances, so is robust to the potential issues that may arise when using a simple accuracy score.

Another consideration for this study is computational time required to train the model. As we have discussed, one reason that neural networks were considered impractical in the past was the infeasible amount of computational time that was required to train them. We have already mentioned that we recorded the median training and prediction times for each model, however these results come with the caveat that computational time to perform operations varies greatly depending on the machine that code is run on. To ensure our experiments were fair, all code was run on the same CPU, an AMD Ryzen 7 5800H on a Dell G15 computer with 16GB RAM. GPU computing was avoided as it may have resulted in negligible run times for this data that would be difficult to compare, and we have mentioned that the differences in computational time between the models is of more interest to us than the absolute time. The results collected for the training set of size $16,000$ are displayed in Table 5.1.2.

Model	Training time (s)	Prediction time (s)	Accuracy (%)
Multinomial logistic regression	1.34	0.00	72.63
Gaussian Naive-Bayes	0.02	0.00	66.68
1-nearest neighbour	0.01	0.74	95.70
Unregularised neural network	1.58	0.11	97.15
L2 regularised neural network	1.55	0.10	94.25
Dropout regularised neural network	2.15	0.11	91.87

Table 1: The recorded median training time, median prediction time and accuracy of each model for 16,000 training instances.

We found that the unregularised neural network obtained higher accuracy than every other model (97.15%), including, surprisingly, both of the regularised neural networks. The 1-nearest neighbour model also obtained greater accuracy than both regularised neural networks (95.70). Gaussian Naive-Bayes achieved the poorest accuracy, with only 66.68%, though this was not unexpected as we established that the Gaussian assumption did not hold for the independent variables. The neural networks had the longest training times, though <2.5 seconds is still negligible in practical terms. The 1-nearest neighbour model had the longest prediction time by some margin with 0.74 seconds, 6-7 times longer than the neural networks. Next, the results collected for the training set of size $8,000$ are displayed in Table 5.1.2.

Model	Training time (s)	Prediction time (s)	Accuracy (%)
Multinomial logistic regression	0.71	0.00	71.89
Gaussian Naive-Bayes	0.02	0.03	65.77
1-nearest neighbour	0.00	1.33	92.92
Unregularised neural network	1.30	0.48	95.48
L2 regularised neural network	1.39	0.45	93.23
Dropout regularised neural network	1.77	0.48	91.58

Table 2: The recorded median training time, median prediction time and accuracy of each model for 8,000 training instances.

We find that the accuracy of each model diminished by only 1-3% when we halved the size of the training set. The multinomial logistic regression model was the most robust to this decrease in training data availability, having a loss of only 0.97% in accuracy. We find that when reducing the training set size the difference in accuracy between the unregluarised neural network and the L2 regularised neural network falls from 2.9% to 2.25%, and the difference between the unregularised network and the dropout network falls from 5.28% to 3.9%. We also notice that the training times for the neural networks drops. Finally, the results for the 500 instance training set is displayed in Table 5.1.2.

Model	Training time (s)	Prediction time (s)	Accuracy (%)
Multinomial logistic regression	0.14	0.01	63.04
Gaussian Naive-Bayes	0.00	0.04	60.96
1-nearest neighbour	0.00	0.49	68.64
Unregularised neural network	0.10	0.52	74.50
L2 regularised neural network	0.11	0.51	75.53
Dropout regularised neural network	0.16	0.53	76.41

Table 3: The recorded median training time, median prediction time and accuracy of each model for 500 training instances.

We immediately notice that for the smallest training set the accuracy of the unregularised neural network is surpassed by that of both regularised networks, with the dropout network performing better than all other models with an accuracy of 76.41%. Also worth noting is the massive decrease in accuracy for the 1-nearest neighbour model, decreasing by 27.06% from the accuracy obtained with the $16,000$ instance training set, making it the least robust of the models to a reduction in training data availability.

Finally, we also chose to generate a confusion matrix [52, p. 84-85] for the multinomial logistic regression and Gaussian Naive-Bayes models trained with the 16,000 instance training set, so as to gain a better understanding of their poorer performance. At every index (i, j) the confusion matrix displays the number of times a data point belonging to class i was classified as belonging to class j, for $i = 1, ..., K$, $j = 1, ..., K$, with $K = 26$ for our data. The diagonal entries of the matrix are the numbers of correctly classified instances, so a confusion matrix of a perfect classifier would be a diagonal matrix with all off-diagonal entries as 0, and diagonal entries totalling N (the number of testing instances). In the cases of the multinomial logistic regression and Gaussian Naive-Bayes confusion matrices we can note that we see small numbers of cells in the matrix with values that are >10 (Figure 9. This indicates systematic misclassifications of data from certain classes, implying that these two model were not able to learn the true patterns in the data, and so are inappropriate models for this task. To contrast, we generated the confusion matrix for the

1NN model which shows very small values of <5 in many cells, implying that their misclassifications are more likely a result of random noise in the data than the models being incorrectly fit.

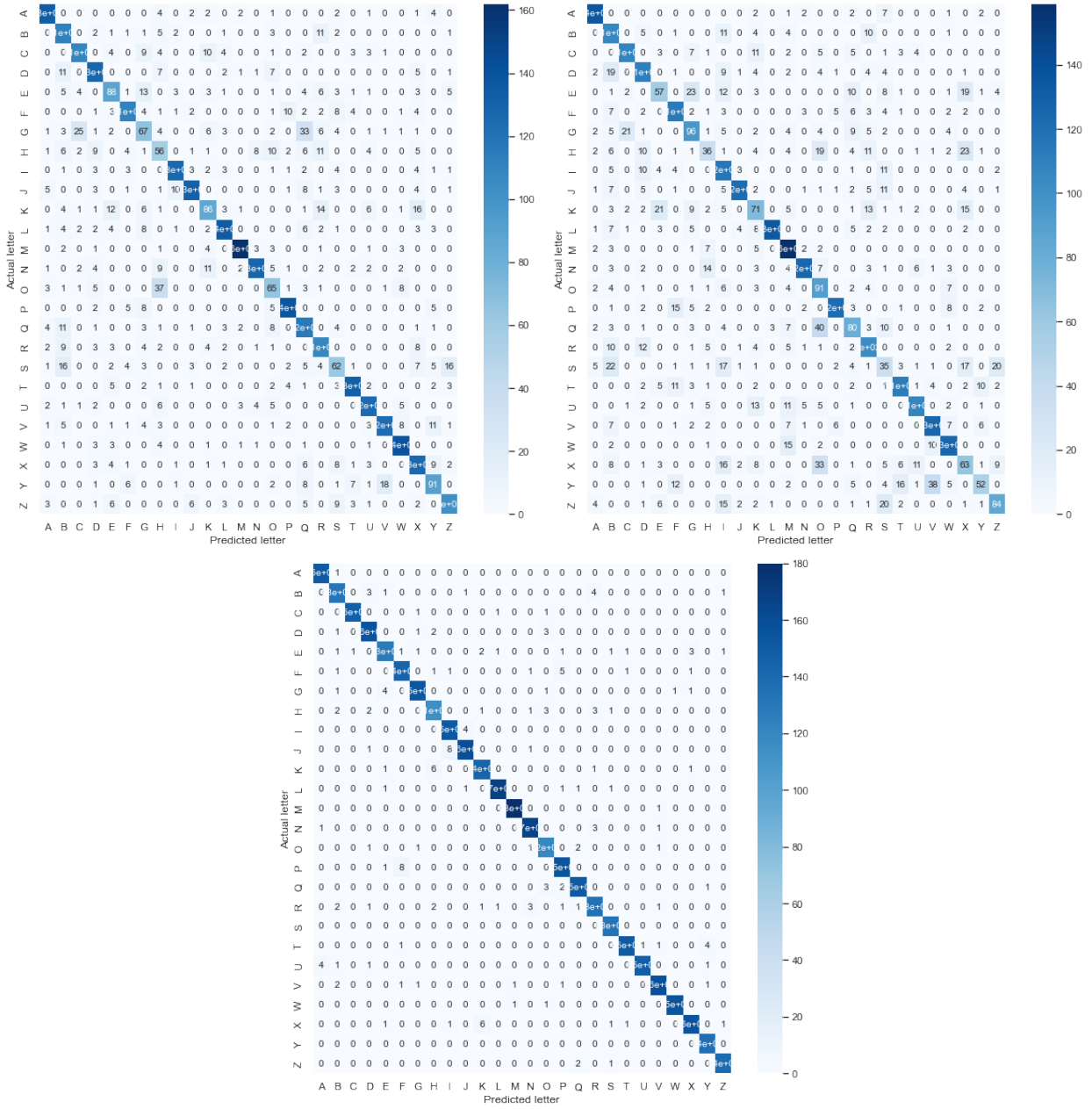

Figure 9: The confusion matrices for multimomial logistic regression (top left) and Gaussian Naive-Bayes (top right) and 1-nearest neighbour (bottom)

5.2 Discussions

Considering first the neural networks trained with the larger training sets, we had expected to find that regularisation would reduce variance and potential overfitting in the model, and hence lead to an increase in accuracy. In attempt to understand why regularisation reduced the model performance we have researched the bias-variance tradeoff as it applies specifically to neural networks in more depth. We found that Neal et al. (2019) provide a compelling case with analytical and empirical evidence that it is possible for both bias and variance to decrease as the number of hidden layers and neurons in a neural network is increased [53]. It is possible then that the unregularised neural network trained in our empirical research was not overfitting significantly to the data set, and that implementing regularisation served only to limit the information that the model was able to capture about the data. It is also possible that the larger data sets provided enough information to the neural network to achieve good generalisation to unseen data without the need for regularisation. This idea is supported by the observations that the difference in accuracy between the unregularised and regularised models fell when the training data set was halved, and the unregularised model was outperformed by the regularised networks when only 500 training data instances were available. This suggests that regularisation should be used when the amount of training data available is limited, but may inhibit the performance of a neural network when there is large amounts of data available for training.

The neural networks had impressive performance in the experiments, but required significantly more effort than the other models to correctly tune and run. The question of whether this trade-off is justified appears to depend on the specifics of the task. For the larger training sets we found comparable performance with the 1-nearest neighbour model, which was very simple to tune and took less combined computational time for training and prediction than the neural networks. However, when the available data was limited, the 1-nearest neighbour model was outperformed by the drop out regularised neural network by 7.77%. It would appear then that the 1-nearest neighbour model is preferred when several thousand instances of training data is available, while the additional effort to set up the neural network is worthwhile when less than a thousand instances of training data is available.

6 Conclusions and proposed future work

We have now described and fit the multinomial logistic regression, Gaussian Naive-Bayes, 1-nearest neighbour and feedforwad neural network classifiers to the UCI letter recognition data set. We have recorded the training and prediction times and accuracy scores of each model on the data. From our results (shown in Tables 5.1.2, 5.1.2 and 5.1.2) we have drawn the following conclusions:

- While the best accuracy score on the $16,000$ instance data set was obtained by the unregularised neural network (97.15%), the KNN classifier obtained an accuracy score only slightly lower (95.70%) and was significantly easier to train. This classifier required the tuning of only a single hyperparameter (the number of nearest neighbours) which was easy to run automatically with grid search. For this reason we are likely to prefer the KNN algorithm for large data sets.
- When the training data available is limited (<1,000) instances, the KNN algorithm suffers a greater reduction in accuracy than neural networks. In this case, a neural network with drop out regularisation appears to outperform KNN significantly enough (a 7.7% improvement in accuracy) to justify the difficulties in setting up the model.
- When large amounts of training data are available, introducing regularisation (both L2 and drop out) appears to diminish the performance of a neural network, suggesting that in such cases the neural network is able to learn the data closely while still retaining generalisation power.
- For the largest training data set, the 1NN classifier required no time to train and took 0.77 seconds to predict on the testing set. The neural networks each required more than 1.5 seconds for training, and $0.10 - 0.11$ seconds to make predictions. Although the neural networks required more time to train and predict, this was still ultimately only of the order of a few seconds, essentially trivial. Hence we can conclude that neural networks require greater computational time to run than the KNN classifier (and the models with poorer performance) but are ultimately computationally feasible on multi-class classification tasks with data sets of size comparable to $20,000$. By observing the changes in training time across the different training set sizes, we can see that there is a direct relationship between training set size and training time for neural networks, and this time may become large for enormous data sets that are common in many modern machine learning problems. For tasks where optimising the accuracy to as high as can possibly be achieved is not a strict constraint we may find that the KNN classifier is a superior choice due to the reduction in computational time and impressive accuracy results.
- The neural networks and KNN classifier required no assumption about the distribution of the data, while the Gaussian Naive-Bayes classifier depended on the assumption that independent variables were Gaussian distributed, which we showed to be an incorrect assumption across the board. Our data showed that the Gaussian Naive-Bayes performed poorly in every stage of the experiment. This implies strong limitations on the breadth of use cases for Gaussian Naive-Bayes to very specific cases where the independent variables follow a Gaussian distribution, where we find that neural networks and KNN have no such constraints.

Some further work may be needed for greater clarity on some of the results we have found. A comparison of neural networks with Gaussian Naive-Bayes in the context of data where the Gaussian assumption holds for the independent variables may also be warranted to assess the usefulness of Gaussian Naive-Bayes where it's model assumptions are properly met.

Furthermore, an investigation into the computational time required for training of and prediction with neural networks when using far larger data sets, perhaps with millions of data points would provide greater insight into the uses of neural networks in a time when huge data sets are becoming more easily available. While the data set we used contained tens of thousands of data instances, this number appeared negligible for modern computing power. We may find that neural networks become infeasibly slow for data sets of this size, but this conclusion can not be drawn from this paper.

A Graphs

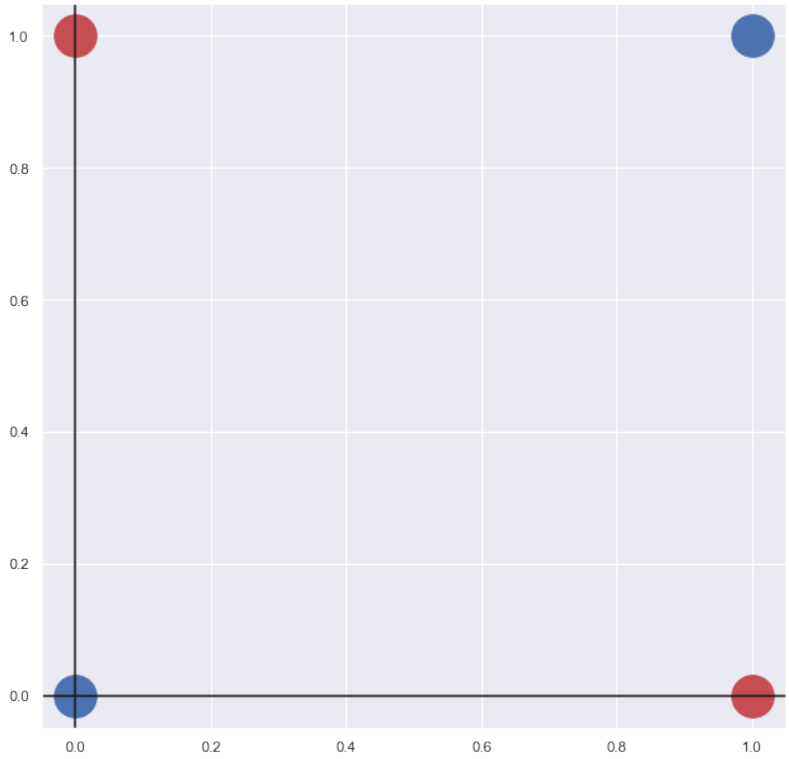

Figure 10: The XOR function, defined by $f(a, b) = 0$ if $a = b$ and $f(a, b) = 1$ otherwise, for $a = 0, 1$ and $b = 0, 1$

B Figures related to the data set

Figure 11: Examples of the data points in the data set, obtained from Frey and Slate (1991) [21]

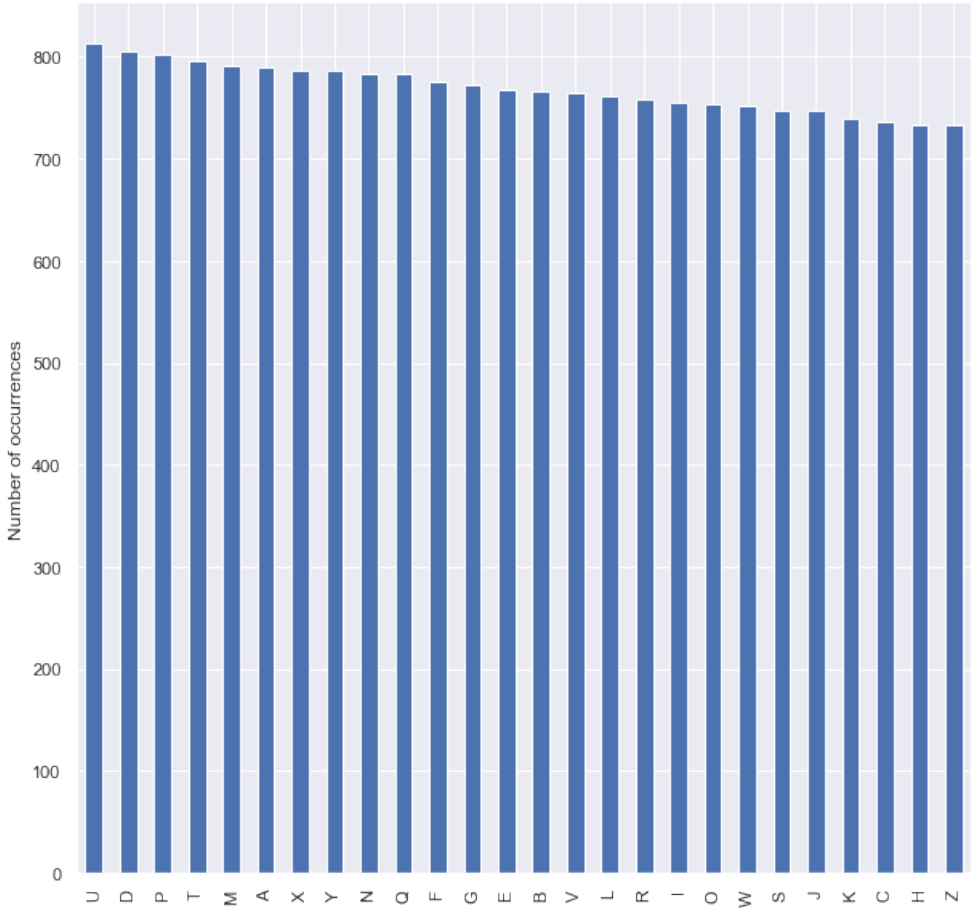

Figure 12: The representation of classes in the data set, ordered by descending frequency

C Graphs of activation functions

This appendix contains graphs of the activation functions discussed in Section 4.2.3. These graphs were generated using Matplotlib.

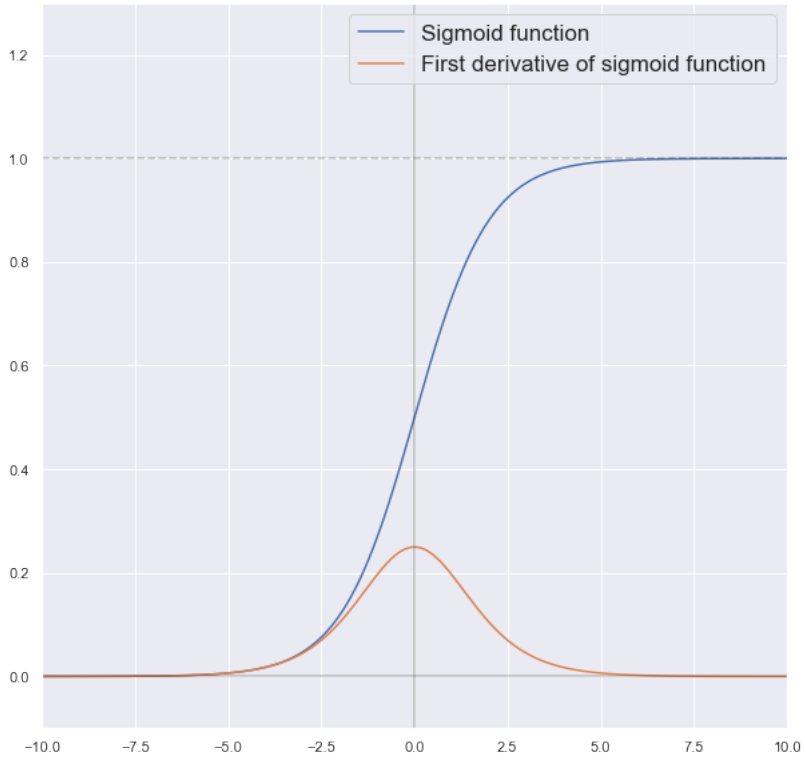

Figure 13: The sigmoid activation function and it's first derivative

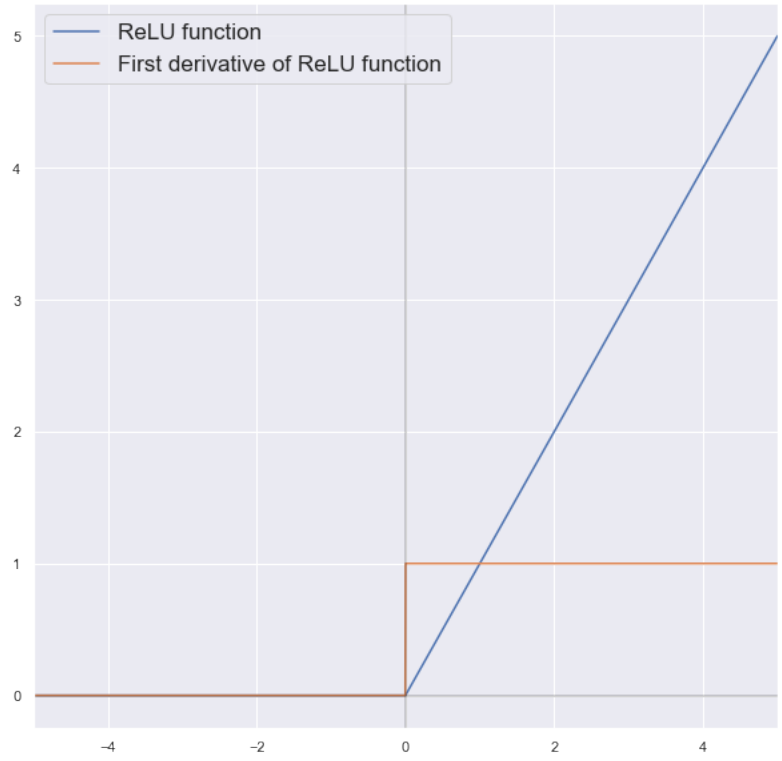

Figure 14: The ReLU activation function and it's first derivative

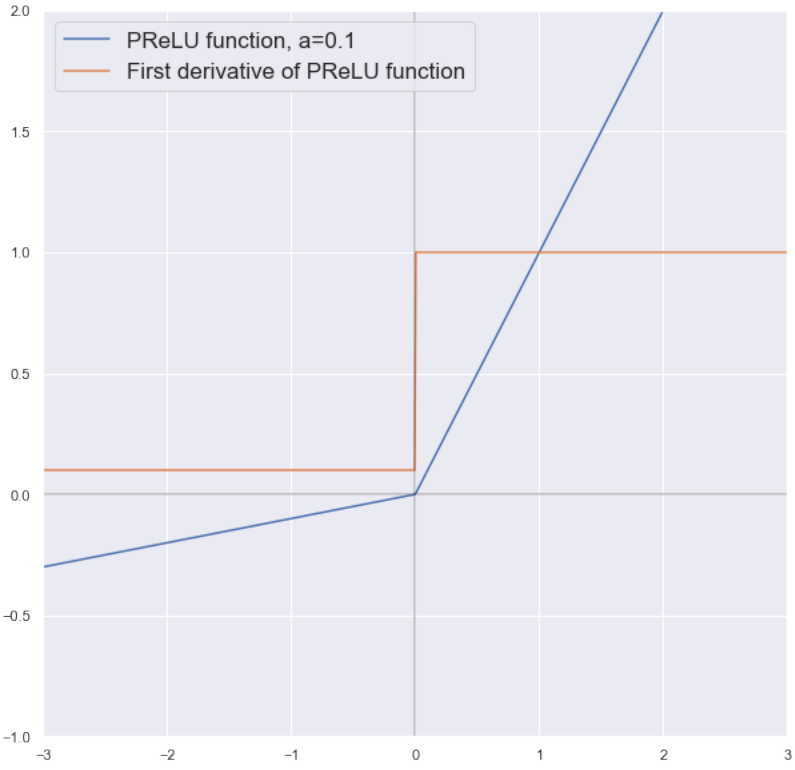

Figure 15: The PReLU activation function and it's first derivative, with $a = 0.1$ fixed

D Reproducing the data

The notebooks accompanying this paper are available at https://github.com/EtienneLatif/Letter-recognition-classification. The 'Letter-Recognition-Classification.ipynb' contains the relevant code to run the models. Under subheading 'Creating the training and testing sets' there is a variable 'train_size' which can be changed to adjust the size of the training set used when the code is run. It should be noted that while the observed accuracy values will be the same as those reported in this paper, the training and

prediction times will vary depending on the machine the code is run on, and from run to run.

References

[1] D. Böhning, "Multinomial logistic regression algorithm," *Annals of the institute of Statistical Mathematics*, vol. 44, no. 1, pp. 197–200, 1992.

[2] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: With Applications in R*, 2nd ed. Springer Publishing Company, Incorporated, 2021.

[3] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed. Springer Publishing Company, Incorporated, 2017.

[4] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, p. 115–133, 1943.

[5] D. Hebb, *The Organization of Behavior: A Neuropsychological Theory*. New York: Wiley, 1943.

[6] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, pp. 65–386, 1958.

[7] C. Buckner and J. Garson, "Connectionism," in *The Stanford Encyclopedia of Philosophy*, Fall 2019 ed., E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2019.

[8] J. Mcclelland, D. Rumelhart, and P. Group, *Parallel Distributed Processing, Volume 2: Explorations in the Microstructure of Cognition: Psychological and Biological Models*, ser. Bradford book. MIT Press, 1987.

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[10] ——, *Parallel distributed processing: explorations in the microstructure of cognition. Volume 1. Foundations*, 01 1986, ch. 8. Learning Internal Representations by Error Propagation.

[11] B. Topping, J. Sziveri, A. Bahreinejad, J. Leite, and B. Cheng, "Parallel processing, neural networks and genetic algorithms," *Advances in Engineering Software*, vol. 29, no. 10, pp. 763–786, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0965997897000628

[12] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[13] J. Liang, G. Bi, and C. Zhan, "Multinomial and ordinal logistic regression analyses with multi-categorical variables using r," *Annals of translational medicine*, vol. 8, p. 982, 2020.

[14] S. Balasubramanian, "A multinomial logistic regression analysis to study the influence of residence and socio-economic status on breast cancer incidences in southern karnataka," 2014.

[15] H. Kamel, D. Abdulah, and J. M. Al-Tuwaijari, "Cancer classification using gaussian naive bayes algorithm," in *2019 International Engineering Conference (IEC)*, 2019, pp. 165–170.

[16] L. Mandal and N. D. Jana, "A comparative study of naive bayes and k-nn algorithm for multi-class drug molecule classification," in *2019 IEEE 16th India Council International Conference (INDICON)*. IEEE, 2019, pp. 1–4.

[17] S. Güney and A. Atasoy, "Multiclass classification of n-butanol concentrations with k-nearest neighbor algorithm and support vector machine in an electronic nose," *Sensors and Actuators B: Chemical*, vol. 166-167, pp. 721–725, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925400512003103

[18] G. Ou and Y. L. Murphey, "Multi-class pattern classification using neural networks," *Pattern Recognit.*, vol. 40, pp. 4–18, 2007.

[19] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[20] D. Dua and C. Graff, "UCI machine learning repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[21] P. W. Frey and D. J. Slate, "Letter recognition using holland-style adaptive classifiers," *Machine learning*, vol. 6, no. 2, pp. 161–182, 1991.

[22] C. Kwak and A. Clayton-Matthews, "Multinomial logistic regression," *Nursing Research*, vol. 51, pp. 404–410, 2002.

[23] D. R. Cox, "Some procedures associated with the logistic qualitative response curve," *Research papers in statistics: Festschrift for J. Neyman*, pp. 55–71, 1966.

[24] H. Theil, "A multinomial extension of the linear logit model," *International economic review*, vol. 10, no. 3, pp. 251–259, 1969.

[25] A. Ng and M. Jordan, "On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes," in *Advances in Neural Information Processing Systems*, T. Dietterich, S. Becker, and Z. Ghahramani, Eds., vol. 14. MIT Press, 2001. [Online]. Available: https://proceedings.neurips.cc/paper/2001/file/7b7a53e239400a13bd6be6c91c4f6c4e-Paper.pdf

[26] D. L. Hosmer, Jr, S. Lemeshow, and R. X. Sturdivant, *Applied Logistic Regression*, 3rd ed. Wiley, 2013, ch. 8.1.

[27] I. Rish, "An empirical study of the naïve bayes classifier," *IJCAI 2001 Work Empir Methods Artif Intell*, vol. 3, 01 2001.

[28] T. J. Watson, "An empirical study of the naive bayes classifier," 2001.

[29] K. R. Das and A. Imon, "A brief review of tests for normality," *American Journal of Theoretical and Applied Statistics*, vol. 5, no. 1, pp. 5–12, 2016.

[30] B. W. Yap and C. H. Sim, "Comparisons of various types of normality tests," *Journal of Statistical Computation and Simulation*, vol. 81, no. 12, pp. 2141–2155, 2011.

[31] S. Engmann and D. Cousineau, "Comparing distributions: the two-sample anderson-darling test as an alternative to the kolmogorov-smirnoff test." *Journal of applied quantitative methods*, vol. 6, no. 3, 2011.

[32] D. A. Harville, *Matrix Algebra From a Statistician's Perspective*. Springer Publishing Company, Incorporated, 1997.

[33] E. Fix and J. Hodges Jr, "Discriminatory analysis-nonparametric discrimination: Consistency properties," CALIFORNIA UNIV BERKELEY, Tech. Rep., 1951.

[34] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015.

[35] C. M. Bishop, *Pattern Recognition and Machine Learning*, 1st ed. Springer New York, NY, 2006.

[36] R. Rojas, *The Backpropagation Algorithm*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996. [Online]. Available: https://doi.org/10.1007/978-3-642-61068-4_7

[37] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016. [Online]. Available: http://arxiv.org/abs/1609.04747

[38] F. Chollet, *Deep Learning with Python*. Manning, Nov. 2017.

[39] S. Sharma, S. Sharma, and A. Athaiya, "Activation functions in neural networks," *International Journal of Engineering Applied Sciences and Technology*, vol. 04, pp. 310–316, 05 2020.

[40] A. Olgac and B. Karlik, "Performance analysis of various activation functions in generalized mlp architectures of neural networks," *International Journal of Artificial Intelligence And Expert Systems*, vol. 1, pp. 111–122, 02 2011.

[41] L. Noriega, "Multilayer perceptron tutorial," 01 2005.

[42] M. I. Jordan, "Why the logistic function? a tutorial discussion on probabilities and neural networks," 1995.

[43] Stinchcombe and White, "Universal approximation using feedforward networks with non-sigmoid hidden layer activation functions," in *International 1989 Joint Conference on Neural Networks*, 1989, pp. 613–617 vol.1.

[44] C.-F. Wang, "The vanishing gradient problem," 2019. [Online]. Available: https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484

[45] J. Schmidt-Hieber, "Nonparametric regression using deep neural networks with ReLU activation function," *The Annals of Statistics*, vol. 48, no. 4, aug 2020. [Online]. Available: https://doi.org/10.1214%2F19-aos1875

[46] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *CoRR*, vol. abs/1502.01852, 2015. [Online]. Available: http://arxiv.org/abs/1502.01852

[47] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

[48] D. Arpit and Y. Bengio, "The benefits of over-parameterization at initialization in deep relu networks," 2019. [Online]. Available: https://arxiv.org/abs/1901.03611

[49] L. Lu, "Dying ReLU and initialization: Theory and numerical examples," *Communications in Computational Physics*, vol. 28, no. 5, pp. 1671–1706, jun 2020. [Online]. Available: https://doi.org/10.4208%2Fcicp.oa-2020-0165

[50] S. Fortmann-Roe, "Understanding the bias-variance tradeoff," 2012.

[51] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[52] A. Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, 2nd ed. O'Reilly, 2017.

[53] B. Neal, S. Mittal, A. Baratin, V. Tantia, M. Scicluna, S. Lacoste-Julien, and I. Mitliagkas, "A modern take on the bias-variance tradeoff in neural networks," *CoRR*, vol. abs/1810.08591, 2018. [Online]. Available: http://arxiv.org/abs/1810.08591