FYS-STK4155 Project 2

Bendik Steinsvåg Dalen & Gabriel Sigurd Cabrera

November 5, 2019

Abstract

Introduction

Data

Credit Card Data

Our first dataset contains real credit card metadata for 30,000 people, in the form of a `.xls` file; each given datapoint (or person) has 23 features and one *binary output* denoting whether or not they've defaulted on their credit card debt. These features can be summarized as follows:

Feature No.	Description	Data Type
1	Total Credit Given	Continuous
2	Gender	Categorical
3	Education	Categorical
4	Marital Status	Categorical
5	Age	Continuous
6-11	Month-Wise Repayment Status	Categorical
12-17	Month-Wise Bill Statement	Continuous
18-23	Month-Wise Amount Paid	Continuous

For more detailed information regarding this dataset, and the file itself, visit https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients

The Franke Function

The second dataset will be given by the *Franke function*, which is defined as follows:

$$\begin{aligned} f(x,y) &= \frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) \\ &+ \frac{3}{4}\exp\left(-\frac{9x+1}{49} - \frac{9y+1}{10}\right) \\ &+ \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) \\ &- \frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right) \end{aligned}$$

We will be solving the Franke function for 100 x-values and 100 y-values in the range $[0,1]$, leaving us with a grid containing a total of 10000 xy coordinate pairs. This leaves us with the values plotted in Figure 1.

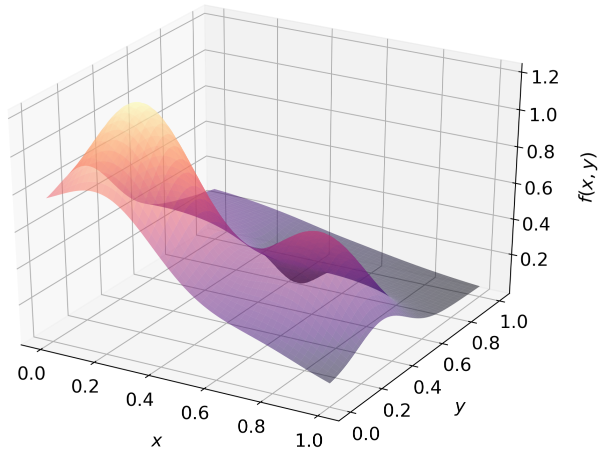

Figure 1: The *Franke function* for x and y values ranging from zero to one.

In addition, we will also be adding *Gaussian noise* to each value $f(x,y)$, such that we are left with values as seen in Figure 2.

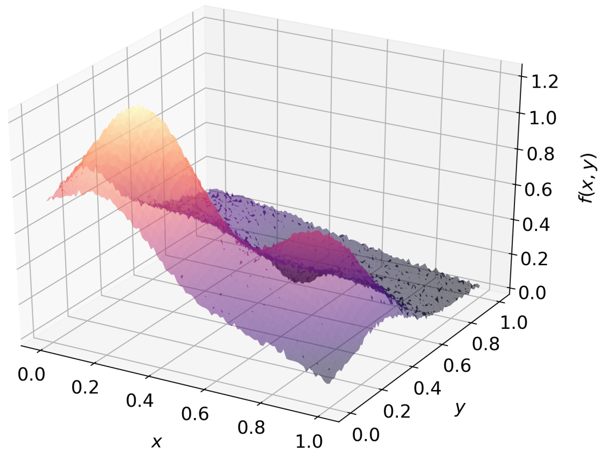

Figure 2: The *Franke function* for x and y values ranging from zero to one, with a Gaussian noise $N(0, 0.01)$

Method

Logistic Regression

The first technique we will be using is called *logistic regression* – we will be using the *sigmoid function* as our *activation function*[1]:

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

We will also need its derivative:

$$\sigma'(x) = \sigma(x)(1-\sigma(x)) \tag{2}$$

Logistic regression is an iterative two-part process consisting of repeated *feed-forward* and *backpropagation* loops between an *input layer* and *output layer*. Assume that we have an input matrix $\mathbf{X} \in \mathbb{R}^{p\times n}$ and output matrix $\mathbf{y} \in \mathbb{R}^{q\times n}$, where n is the number of *datapoints*, p is the number of *features*, and q is the number of *labels*. Knowing this, we can generate a matrix of weights $\mathbf{W} \in \mathbb{R}^{q\times p}$ and vector of biases $\mathbf{b} \in \mathbb{R}^q$ such that:

$$w_{ij} = N(0, 0.5); \qquad b_i = N(0, 0.5)$$

Where $N(\text{mean}, \text{std})$ is the *normal distribution*. To implement the feed-forward portion of the algorithm, we use matrix multiplication to predict an output $\hat{\mathbf{y}}$ with the given weights:

$$\hat{\mathbf{y}} = \sigma(\mathbf{wX} + \mathbf{b})$$

For simplicity, we will define $\mathbf{z} \equiv \mathbf{wX} + \mathbf{b}$.

Next, we want to optimize our weights via backpropagation, which requires the use of gradient descent. Specifically, we want to find $dw_{ij} = \frac{\partial C}{\partial w_{ij}}$, where C is a *cost function*, and $w_{ij} - dw_{ij}$ reduces C; to find such a value, we use the multivariable *chain rule* to rewrite dw_{ij}:

$$dw_{ij} = \frac{\partial C}{\partial w_{ij}} = \frac{\partial C}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial z_i}\frac{\partial z_i}{\partial w_{ij}} = \delta_i \frac{\partial z_i}{\partial w_{ij}}$$

The intent of repeating this process is to reduce our cost function until a local or global minimum is reached and an optimal set of $\mathbf{W}$ and $\mathbf{b}$ are computed. If we use the *mean squared error* as our cost function, this can be rewritten in matrix form as follows:

$$d\mathbf{w} = [2(\hat{\mathbf{y}} - \mathbf{y}) \odot \sigma'(\hat{\mathbf{y}})] \otimes \mathbf{X} \tag{3}$$

One full feedforward–backpropagation iteration is known as a *batch*; if N datapoints are calculated in a single batch, we say that the *batch size* is N, and $d\mathbf{w}$ is averaged over each input.

Once a batch is complete, the updated weights and biases are used in the next batch, and so on for each subsequent batch. When all the batches have been evaluated, we say that an *epoch* has been completed; many epochs are often required to properly optimize a set of weights and biases.

[1] A function that restricts the *range* of predicted outputs.

[2] Those between the final hidden layer and output layer.

Neural Networks

The second technique we will explore is the usage of *multilayer neural networks*; as with logistic regression, there exists an input layer and output layer and a set of weights and biases. Now, however, we also have a variable number of *hidden layers* $L-1$, each consisting of a variable number of *nodes*.

As with logistic regression, we must randomly initialize a set of weights – this time, however, we will need several weight matrices and bias vectors; specifically, we need an amount equal to the number of hidden layers, plus one. Each weight matrix w^ℓ must have as many rows as there are nodes in the subsequent layer, and as many columns as there are in the preceeding layer; each bias vector must be as large as the subsequent layer.

The feed-forward portion of the algorithm is unchanged, with the exception that the process must be repeated layer-by-layer, for each set of weights accompanying the product in question; the backpropagation, however, is more complex.

To update all of our weights, we begin by using (3) on the *last* set of weights[2] $\mathbf{w}^{\mathrm{L}}$. Once this has been accomplished, we can update these weights and use the following equation to find the remaining $d\mathbf{w}^\ell$:

$$d\mathbf{w}^\ell = (\mathbf{W}^{\ell+1})^{\mathrm{T}} \delta_i^{\ell+1} \odot \sigma'(\mathbf{z}^\ell)$$

Mean Squared Error

To get a measure of success with respect to the implemented method and parameters, we can calculate the mean difference in the squares of each measured output y_i and their respective predicted outputs $\hat{y}_i$:

$$MSE(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 = \mathbb{E}\left[(\mathbf{y} - \hat{\mathbf{y}})^2\right]$$

The lower the MSE, the closer the polynomial approximation is to the original dataset. If it is too low, however, we run the risk of overfitting our dataset, which is not desireable either – fortunately, this not an issue within the scope of this report.

R² Score

Another measure of success is the *coefficient of determination*, colloquially known as the R^2 score, is given by the following expression:

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y}_i)^2}$$

The closer R^2 is to one, the closer the polynomial approximation is to the input/output dataset, although a perfect score can once again arise due to overfitting just as in the case of the MSE.

ROC

The *ROC* (short for *receiver operating characteristic*) of a binary classifier is a measure of how well a model performs, by accumulating and comparing the rate of true positives[3] (*TPR*) and rate of false positives[4] (*FPR*). These can be calculated as follows:

$$TPR = \frac{TP}{TP + FN}; \qquad FPR = \frac{FP}{FP + TN} \tag{4}$$

Where TP, FN, FP, and TN are the total number of *true positives*, *false negatives*, *false positives*, and *true negatives*, respectively.

Note that the equations in (4) are *scalar values* – to construct our curve, we must therefore calculate TPR and FPR for *every datapoint*. Once this is accomplished, the values are ordered such that predicted outputs furthest from 0.5 are prioritized. Next, the values are cumulatively summed up and the resulting series are normalized to 1: normalization is important, since we are interested in *rates* of prediction.

AUC

The *AUC* (or *area under ROC*) is a measure of model performance; an AUC closer to 1 suggests that a model is good at correctly making predictions, while an AUC closer to 0 implies that a model is bad at binary classification. The AUC of a model is calculated by finding the *area* under the ROC curve of the given model; since the ROC is assumed to be normalized in the range 0 to 1, so will the AUC.

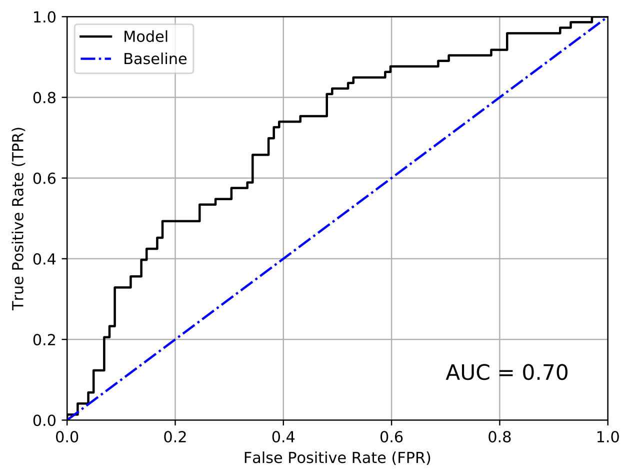

Figure 3: An example *ROC* curve with its *AUC*.

The project text also mentions theese things that we may or may not need to wtire about in this section:

Accuracy score?

Or is that basically the ROC?

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} I\left(t_i = y_i\right)}{n}$$

Cross validation

Do we use *k-fold* cross validation? If not, we probally should.

Bootstrap?

The introduction mentions the bootstrap method as a thing we could use (I think it was as a alternative to *k-fold*). We didn't use it for project 1, so I don't think we need to use it here either.

Other cost funnctions

This is basically part d, I guess. The project text says we should change the cost function for our neural network code in order to perform a regression analysis. I'm guessing we already do that, but we should probally say something about that.

We could also test different cost functions to find which gives the best results, and say a bit about each of them.

learning rates and regularization parameters

The text also mentions that we should discuss learning rates and regularization parameters. We should probally talk about which values we test.

Scikit-Learn/tensorflow/keras?

The text also says we could compare our nn with Scikit-Learn or tensorflow/keras, but I'm not sure how necessary that is, or if we need to write about that.

Results

Discussion

Conclusion

Part e would basically be the conclusion rigth?

Appendix?

Code explenation?

We could have an explenation of the structure of the code, like which files does what, and stuff like that. Would set this as low priority though, only do it if we have time.

[3]How often 1 is predicted when 1 is expected.
[4]How often 1 is predicted when 0 is expected.

References

2