FYS-STK4155 – Project 3[†]

Sajjad Ahmadigoltapeh[1], Jan Fredrik Kismul[2], and Silje Helene Løvbrekke[3]

[1]sajjadah@uio.no
[2]j.f.kismul@fys.uio.no
[3]siljehlo@matnat.uio.no

December 16, 2019

Abstract

In this study three methods i.e. Convolution Neural Network (CNN), XGBoost and Support Vector Machines (SVM) are used to study the MNIST data set, the Taiwan Credit card data, and data generated by the Franke function. The results indicate that CNN are especially good for picture recognition with the MNIST set (99% accuracy), while a regular NN handles regression fitting to noisy data better than the other methods (99% accuracy). For data lacking some information, such as the Credit Card data set, XGBoost outperforms other methods tested (0.566 Area Ratio), with NN coming close (0.549 Area Ratio). Furthermore, we find that SVM and XGBoost seem rather susceptible to noise in the data set. Over all, neural networks seem to be the most versatile method, performing well with any data set we've subjected it to.

Contents

[†]Project folder: `https://github.com/jkismul/MachineLearning/tree/master/Project3`

1 Introduction

Autonomous driving, healthcare or retail are just some of the areas where Computer Vision has allowed us to achieve things that, until recently, were considered impossible. Today the dream of a self driving car or automated grocery store does not sound so futuristic anymore. In fact, we are using Computer Vision every day — when we unlock the phone with our face or automatically retouch photos before posting them on social media.

2 Theory and methods

Image recognition is one of the initial skills that mankind learns at the beginning of life. By noticing an image, one is able to immediately analyze and give a label to the image. The process of image processing comprises getting a ”input image”, ”analysis”, and deliver a ”class” or ”probability of a class”.

2.1 The structure of digital photo

The structure of a digital image includes a matrix of numbers that each number presents the brightness of single pixel. For a colorful image, we need three matrices for three main colors i.e. red, green, and blue (RGB: Red, Green, and Blue). Each number (number between 0-225) in these three matrices describes the intensity of relevant color. For example, the matrix RGB(0,225,0) is rendered as green color. We can simply realize that for a white-black image one matrix is required.

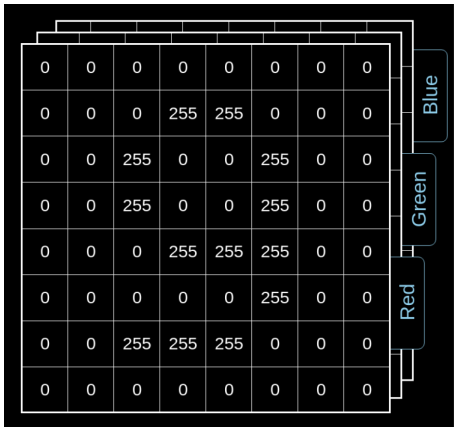


Figure 1: Schematic view of a colorful image that includes three matrices corresponding to three color channels Red, Green and Blue [1].

2.2 The data set: MNIST

In this study, we want to set a convolution neural network that is able to recognize the handwritten image after training. We decided to use a well-known data set so-called MNIST which is convenient

for our case study. The Modified National Institute of Standards and Technology database (MNIST database) is a handwritten digits including sixty thousand images for training set and ten thousand images for test set. These images are size-normalized and centered in a fixed-size 28x28 pixels image. Figure (2) illustrates some examples of handwritten numbers with normalized size.

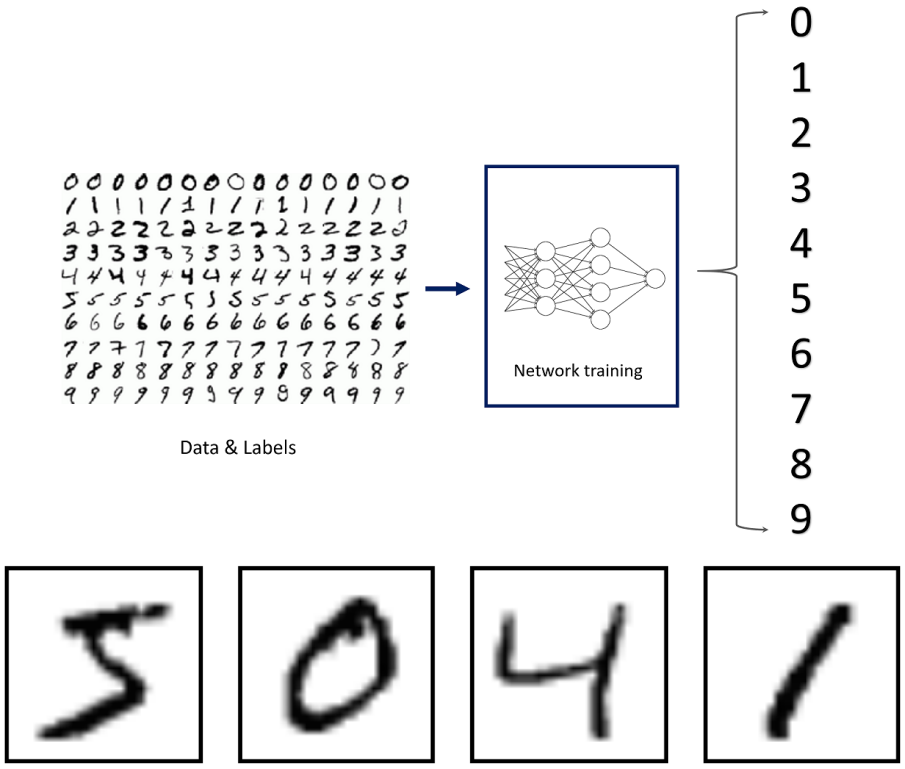


Figure 2: Schematic layout of MNIST data set, network tarining, number of classification, and some sample images from the MNIST data set . These images are size-normalized and centered in a fixed-size 28x28 pixels image.Some sample images from the MNIST data set . These images are size-normalized and centered in a fixed-size 28x28 pixels image. [2].

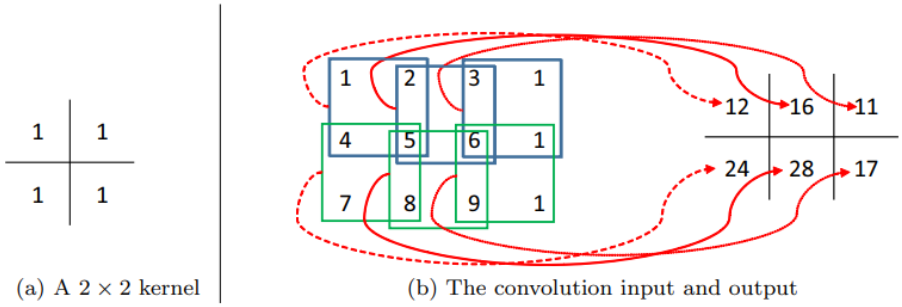
2.3 Convolution neural network (CNN)

Convolution neural network (CNN) is a supervised learning technique that is broadly used for visual imagery. The similarity between CNN and NN is both methods use loss-function, optimizer, and hidden layers (with weights and biases). The difference is, CNN is shaped for analysis of images by using specific architecture such as convolution layers, pooling layers, and fully connected layers. Actually, these layers apply a filter to the input image in order to make a feature map. These feature maps are summarizing the features in the input. Mathematically speaking, convolution is a process that takes a small matrix of numbers so-called kernel or filter, then each filter slides along the input image matrix, calculates the dot product between each part of the image and the filter,

in all dimensions of depth, because CNN arrange neurons in 3-dimention i.e. width, height and depth. Further, this is passed through first convolution layer that includes "ReLu" as activation function. Then, this is passed through "pooling layer" to reduce dimensionality. Finally, everything is summed up and the results are put in the output feature map. It is worth to note that small parts of images are strongly correlated with its neighbors. Therefore, instead of connecting every single pixel to a neuron in the first hidden layer, it is possible to connect each neuron to a small part of the image [3].

2.3.1 Convolution layers

The most important advantage of convolution layers is known as the ability to reduce the size of image. In the other word, convolution layers filter the image to the smaller one but relationship between pixels are preserved. Figure (3) shows an example of convolving a matrix with one single convolution kernel. Assume there is a 3x4 input image and a 2x2 convolution kernel. Multiplying the kernel to top of the image gives a single number. For example, multiplication of kernel to the first box in top gives: $1 \times 1 + 1 \times 4 + 1 \times 2 + 1 \times 5 = 12$ and multiplication to the first box in bottom gives: $1 \times 4 + 1 \times 7 + 1 \times 5 + 1 \times 8 = 24$. The convolution operation is repeated until reaching to the bottom right corner of the input image. For an input tensor with size of $H^l \times W^l \times D^l$, definition of convolution operation is similar. Namely, multiplying the kernel by top of the input tensor at the spatial location (0, 0, 0), gives the products of corresponding elements in all the D^l channels and sum of the products to get the convolution result at relevant spatial location.



(a) A 2 × 2 kernel (b) The convolution input and output

Figure 3: Illustration of the convolution operation.

Figure 3: This example illustrates convolving a matrix with one single convolution kernel [4].

In precise mathematics, by having a third order tensor $\hat{x}^l \in R^{H^l \times W^l \times D^l}$ such that each element of x is presented with indexes of (i^l, j^l, d^l) convolution process is defined as:

$$\hat{y}^l = \sum_{i=0}^{H} \sum_{j=0}^{W} \sum_{d=0}^{D} \hat{f} \times \hat{x}^l$$

where l shows the layer number, $\hat{f}$ is set of convolution kernels and bias $\hat{b}^l$ can be added to $\hat{y}^l$.

2.3.2 Pooling layer

Generally, pooling layers approach is reducing the amount of parameters for the image. The pooling layers provide dimension reduction with keeping important information. To be more specific, they allow us to down sample the feature maps. There are two approaches for pooling method i.e. *average pooling* and *max pooling*. The most used one is max pooling that takes the most activated (largest element) of features. For an input image of $H^l \times W^l$ with spatial extension of D^l, the elements of pooling layer vector is defined as:

$$\text{max is defined as:} \quad y_{i,j,d} = \text{max:(diag)} x^l_{(i,j,d)}$$

$$\text{average is defined as:} \quad y_{i,j,d} = \frac{1}{HW} \sum_{i=0}^{H} \sum_{j=0}^{W} x^l_{(i,j,d)}$$

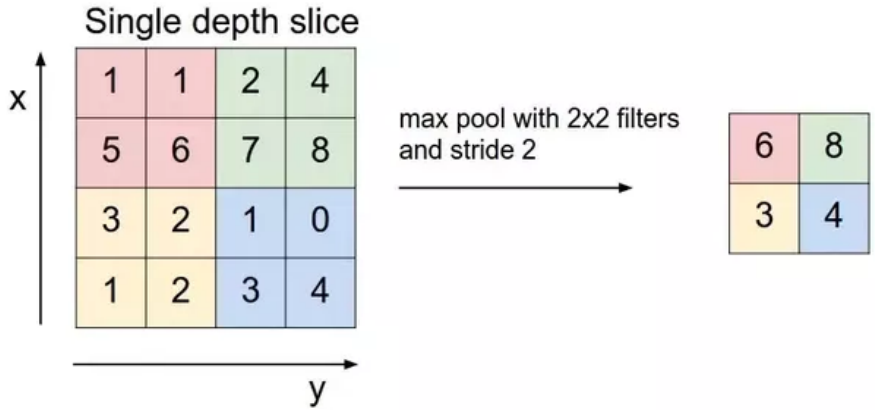


Figure 4: Schematic illustration of Maxpooling. As it is observed, application of Maxpooling ends up to a 2x2 matrix including largest element on diagonal[5].

The pooling layers are added after adding convolution layers. Although, down sampling is achievable via changing the stride in convolution layers, but it is more robust to use pooling layers dedicatedly.

2.3.3 Fully connected layer

The task of this layer is taking an input from whether ReLu or convolution layer and build an N dimensional vector. Here N is the number of classes that should be recognized by model. In our case that the aim of model is classification of handwritten numbers, N equals to 10 since there are 10 digits between 0-9.

2.4 Support Vector Machines

A support vector machine (SVM) is a powerful tool and versatile machine learning model capable of performing linear and non linear classification, regression and outlier detection. It is a popular tool, well suited for classification of complex small or medium sized data sets. [6] SVMs rely on the definition of hyperplanes and the definition of a margin which separates classes (classification problem) of variables. SVMs distinguish between hard and soft margins. The latter introduces a so-called softening parameter to be discussed later. We can also distinguish between linear and non linear approaches. As we rarely can separate classes by a straight line, the nonlinear approach is most frequently used [3].

2.4.1 SVM for Classification

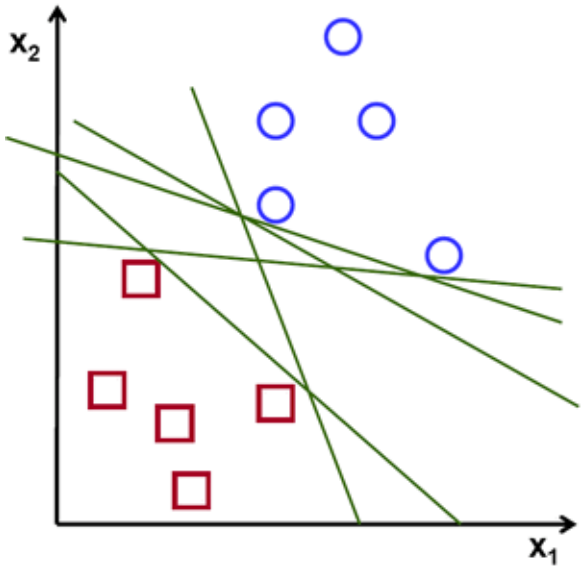


Figure 5: Illustration showing how two classes can be seperated by hyperplanes. Picture borrowed from: `https://docs.opencv.org/2.4/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html`

Let's begin with defining a hyperplane: Let w_1, w_2, ... , $w_N \in \mathbb{R}$ be non zero. Then the set S consisting of all vectors

$$\text{X} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix} \text{ in } \mathbb{R}^N \text{ such that}$$

$$w_1 x_1 + w_2 x_2 + \ldots + w_N x_n = 0$$

is a subspace of $\mathbb{R}^N$ called a hyperplane. More generally, a hyperplane is any codimension - 1 vector subspace of a vector space [7].

To motivate SVM for classification one can take a brief look at a toy example of a binary classification problem. With two classes one can attempt to separate them with a hyperplane $b + w_1x_1 + w_2x_2 = 0$, where b is the intercept, w $\in \mathbb{R}^p$ and x $\in \mathbb{R}^{n\times p}$(for illustration, see figure (5)). There are infinitely many possible separating hyperplanes. The goal with these hyperplanes, or lines in this example, is to pick the optimal one and use it as a threshold or decision boundary for the classification. With two classes of outputs y_i we assign the classes values $y_i \in \{-1, 1\}$. If the above condition is not met by a given vector x_i we have

$$b + w_1x_1 + w_2x_2 > 0$$

if $y_i = 1$. Which means x_i lies on one side of the hyperplane. On the other hand if

$$b + w_1x_1 + w_2x_2 < 0$$

$y_i = -1$ and x_i lies on the other side of the hyperplane. From this we can define the margin as the shortest distance between the observations and the threshold[3].

With these details in place one can find the optimal separating hyperplane that separates the two classes and maximizes the distance to the closest point from either class. This provides a unique solution to the separating hyperplane problem [8]. Let's dig into the details. We wish to find a margin M with **w** normalized to $\|\mathbf{w}\| = 1$ subject to the condition

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq M \; \forall \; i = 1, 2, ..., p$$

Thus all points are a signed distance from the decision boundary defined by the line L defined by parameters b, and **w**. We are looking for the largest value M defined by

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq M \, \|\mathbf{w}\| \; \forall \; i = 1, 2, ..., p$$

By scaling the equation such that $\|\mathbf{w}\|$ = 1/M, we need to find the minimum of $\mathbf{w}^T\mathbf{w} = \|\mathbf{w}\|$ subject to the condition

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1 \; \forall \; i = 1, 2, ..., p$$

The margin is now defined as the inverse of the norm of **w**. We need to minimize the norm in order to find the greatest margin M. Seeing how this is a constrained optimization problem we use Lagrangian multipliers for this problem. Therefore by adding the Lagrange multiplier can define the following function to be minimized

$$\mathcal{L}(\lambda, b, \mathbf{w}) = \frac{1}{2}\mathbf{w}^T\mathbf{w} - \sum_{i=1}^{n} \lambda_i[y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1]$$

λ is the Lagrange multiplier subject to the constraints $\lambda_i \geq 0$. By computing the partial derivatives $\mathcal{L}$ with respect to the intercept and $\mathbf{w}$ we can find expressions for $\mathbf{w}$ that can be inserted into the equation for $\mathcal{L}$.

$$\frac{\partial \mathcal{L}}{\partial b} = -\sum_i \lambda_i = 0 \text{ and } \frac{\partial \mathcal{L}}{\partial \mathbf{w}} = 0 = \mathbf{w} - \sum_i \lambda_i y_i \mathbf{x}_i$$

Inserting these into the equation for $\mathcal{L}$

$$\mathcal{L}(\lambda, b, \mathbf{w}) = \sum_i \lambda_i - \frac{1}{2}\sum_{ij}^{n} \lambda_i \lambda_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \tag{1}$$

Subject to the constraints $\lambda_i \geq 0$ and $\sum_i \lambda_i y_i = 0$. It must additionally satisfy the Karush-Kuhn-Tucker condition.

$$\lambda_i [y_i(\mathbf{w}^T \mathbf{x}_i + b) - 1] \, \forall \, i$$

x_i is said to be on the boundary if $\lambda_i > 0$ then $(\mathbf{w}^T \mathbf{x}_i + b) = 1$, and $\mathbf{x}_i$ are said to be support vectors. They are the vectors closest to the hyperplane and define the margin M. This defines a hard margin. If $(\mathbf{w}^T \mathbf{x}_i + b) > 1$, x_i is not on the boundary and we set $\lambda_i = 0$ [3].
Equation 1 can also be rewritten on matrix form. When we solve the problem in equation 1 we get the values of λ_i and from there we can get the coefficients $\mathbf{w}$ that we can use to calculate our hyperplanes, which also yield the intercept b.

Soft Optimization problem

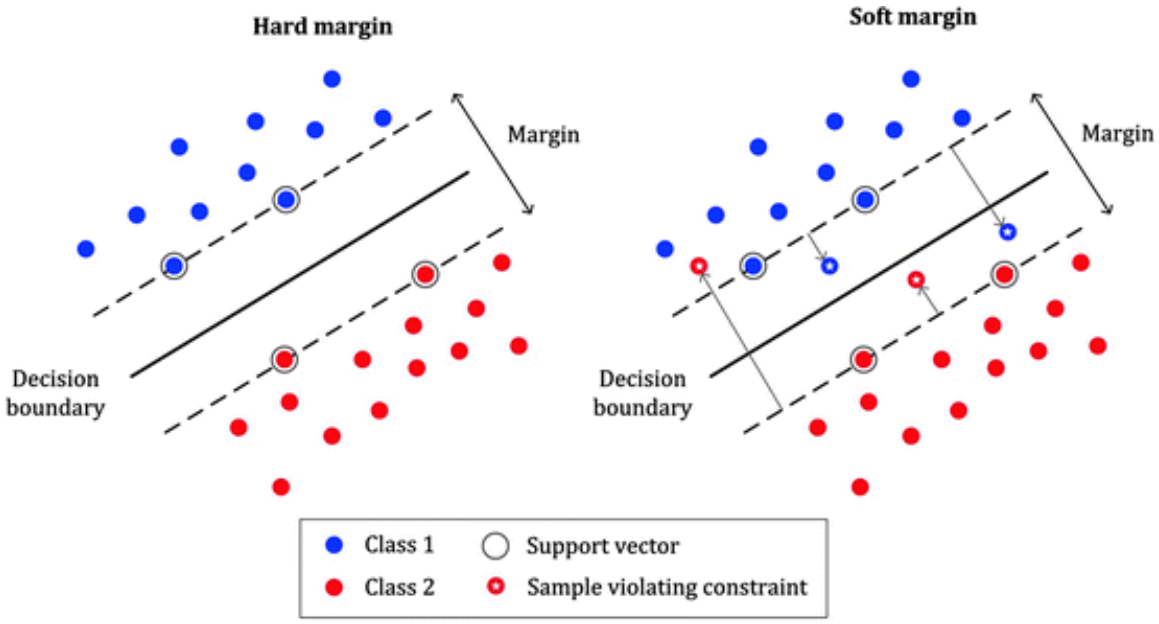


Figure 6: Illustration of the difference between a hard margin and a soft margin. Picture borrowed from `https://mc.ai/math-behind-svmsupport-vector-machine/`

What if the classes overlap in feature space?
We can allow some slack in the sense that some points can be on the wrong side of the margin. Now we can introduce the so-called slack variable ξ. And we can modify the equation $y_i(\mathbf{w}^T\mathbf{x}_i + b) = 1$ to

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) = 1 - \xi_i$$

With $\xi_i \geq 0$. The total violation of the margin is now $\sum_i \xi_i$. One value ξ_i corresponds to how far a prediction is on the wrong side of its margin. Then by bounding $\sum_i \xi_i$ we can bound the total violation of the margin. Misclassification occurs when $\xi_i > 1$. Bounding $\sum_i \xi_i$ by some C value bounds the total number of misclassifications. This leads to a change in our optimization problem to finding the minimum of

$$\mathcal{L} = \frac{1}{2}\mathbf{w}^T\mathbf{x} - \sum_i^n \lambda_i[y_i(\mathbf{w}^T\mathbf{x}_i + b) - (1 - \xi)] + C\sum_{i=1}^n \xi_i - \sum_{i=1}^n \gamma_i\xi_i \tag{2}$$

where λ_i and γ_i are Lagrange multipliers. Subject to

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) - 1 - \xi_i \; \forall(i|\xi_i \geq 0)$$

Taking partial derivatives leads to finding expressions for b and **w** that we can insert into the equation for $\mathcal{L}$ again. This leads to the same equation as before, but subject to new constraints.

$$\mathcal{L}(\lambda, b, \mathbf{w}) = \sum_i \lambda_i - \frac{1}{2}\sum_{ij}^n \lambda_i\lambda_j y_i y_j \mathbf{x}_i^T\mathbf{x}_j \tag{3}$$

Subject to constraints $\lambda_i \geq 0, \sum_i \lambda_i y_i = 0$ and $0 \leq \lambda_i \leq C$. $\mathcal{L}$ must in addition satisfy the Karush-Kuhn-Tuck condition which now is

$$\lambda_i[y_i(\mathbf{w}^T\mathbf{x}_i + b) - (1 - \xi_i)] = 0 \; \forall i$$

$$\gamma_i\xi_i = 0,$$

and

$$y_i(\mathbf{w}^T\mathbf{x}_i + b) - (1 - \xi_i) \geq 0 \; \forall i$$

Kernel trick

When facing a non linear problem we need more tools. The ideal would be that via some transformation we achieved a more separable feature space. It is possible to achieve this by changing our basis function from x $\rightarrow$ z = $\phi(x_i)$. However via the kernel trick our algorithm can be modified by replacing inner products on the form $\langle \mathbf{x}, \mathbf{x}' \rangle$ with a call to a kernel function k(x,x'). This leads to the same results as if we went through performing a transformation $\phi(x_i)^T\phi(x_i)$ during the SVM calculations. By chosing a Mercer kernel, Mercer's theorem makes sure that we can work with kernels to transform x without knowing the basis function [3][9].

Some popular kernels are

1. Linear: $K(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T\mathbf{y}$
2. Polynomial: $K(\mathbf{x}, \mathbf{y}) = (\mathbf{x}^T\mathbf{y} + \gamma)^d$
3. Gaussian Radial Basis Function (RBF): $K(\mathbf{x}, \mathbf{y}) = \exp(-\gamma \|\mathbf{x} - \mathbf{y}\|^2)$
4. Tanh: $\tanh(\mathbf{x}^T\mathbf{y} + \gamma)$

2.4.2 SVM for Regression

Now we are going to reverse the objective from SVM for classification; instead of trying to fit the largest possible margin between the two classes while limiting margin violations, SVM for regression tries to fit as many instances as possible on the hyperplane and its surrounding margin (which looks like a street) while limiting margin violations. The width of the "street" is controlled by a hyperparameter ϵ. The points inside the ϵ tube are ignored, but the points that fall outside are penalized and used as support vectors to determine the placement of the hyperplane [6].

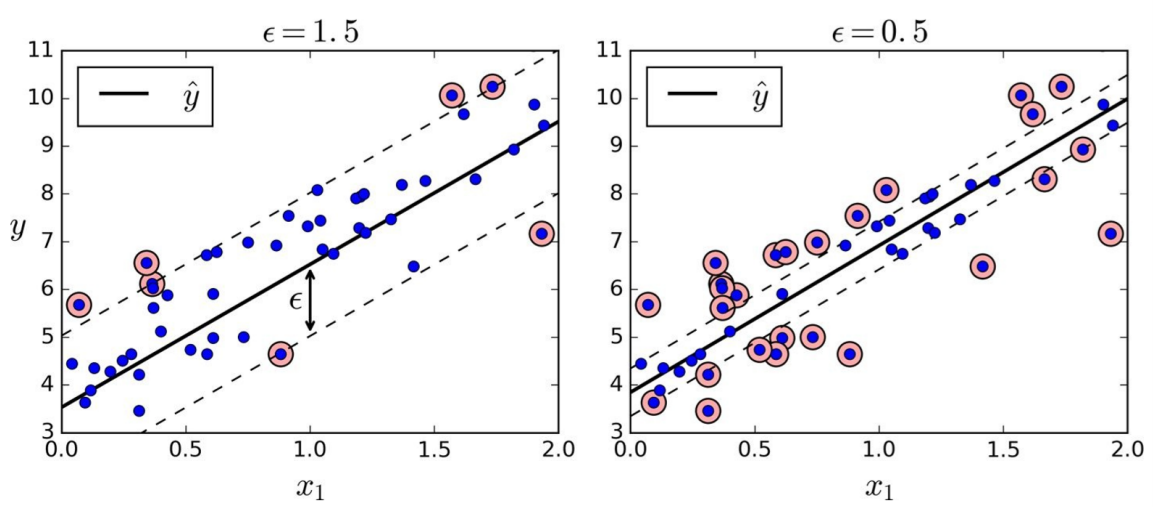


Figure 5-10. SVM Regression

Figure 7: Illustration of SVM regression and how the width of the "street" varies with $\epsilon = 1.5$ and $\epsilon = 0.5$. Picture borrowed from [6] chapter 5.

This gives the optimization problem to be solved

$$\mathcal{L} = C\sum_{i=1}^{n}(\xi_i^+ + \xi_i^-) + \frac{1}{2}\|\mathbf{w}\|^2 \tag{4}$$

With constraints

$$y_i \leq \mathbf{w}^T\mathbf{x}_i + b + \epsilon + \xi_i^+$$
$$y_i \geq \mathbf{w}^T\mathbf{x}_i + b - \epsilon - \xi_i^-$$

as well as $\xi_i^+ \geq 0$ and $\xi_i^- \geq 0$. [9]

Where $\xi_i^{\pm}$ represents the degree to which each point lies outside the epsilon tube, C = 1/λ is a regularization constant working in a similar way to how the λ in Ridge regression shrinks the β's for the ridge regression problem. We can find the gradients of $\mathcal{L}$ w.r.t **w** and b to find expressions for **w** and b to put back into $\mathcal{L}$. When the model has been trained we can make predictions using

$$\hat{y}_i(\mathbf{x}) = \mathbf{w}^T \mathbf{x}_i + b$$

this also has the flexibility of using kernels[9].

2.5 Decision trees and boosting

Decision trees are intuitive algorithms that can be used for both classification and regression, and can be used for both supervised and unsupervised learning.

The algorithm is essentially the game "Twenty questions", where you ask a series of yes/no questions to find out what your opponent was thinking of, with the goal of finding the correct answer in as few questions as possible. There is a multitude of different tree algorithms, so the following is very superficial.

The Decision Tree algorithm gets its name from its tree-like structure, consisting of a root node, interior nodes and leaf nodes, connected by branches. The root node is characterized by having no incoming branches, the interior node by having one incoming branch and two out (for binary trees), while a leaf has one incoming branch and zero branches going out. See Figure (9) for an example with 1 root node, 2 interior nodes and 4 leaves.

The algorithm goes as follows:
I. Split the data set by the feature that gives the highest information gain ratio (or lowest impurity), to be defined later. II. Repeat step I for each new node (child node), until a child node either consists of only one class, or there is no further information gain from splitting it again, or some predefined criteria is reached.

The idea of information gain can be understood easily from the example of a game of 20 questions. You would typically start with a question such as "is it a living thing?". The answer will then exclude a slew of options.
You could ask something like "is it an F-16?". Unless you're miraculously correct, you have only gained a very small bit of information. The word bit is fitting here, as one of the measures of information gain uses Shannon entropy, as seen below. Had you asked "is it a plane?", you would then have gained one bit of information for each plane in existence, including the one from an F-16. In other words, it would have been a better question to start with.

Information gain can be expressed mathematically as [10]

$$\text{GAIN}(\mathcal{D}, x_j) = \text{H}(\mathcal{D}) - \sum_{v \in \text{Values}(x_j)} \frac{|\mathcal{D}_v|}{|\mathcal{D}|} \text{H}(\mathcal{D}_v), \tag{5}$$

where $\mathcal{D}$ is the data set in the parent node and $\mathcal{D}_v$ is the data set at a child node after splitting. Note that this definition will tend to favor splitting along features containing many classes, and Information Gain Ratio is a better measure [11]. H is a function measuring impurity, such as Shannon entropy, or the Gini Impurity defined below. The variable x_j is the feature the data set is being split by. Note that information gain is the impurity of the node before being split and the weighted average of the impurities after splitting.

Shannon entropy is given by

$$\text{H}(p) = -\sum_i p_i \log_2(p_i), \tag{6}$$

and Gini Impurity by

$$\text{Gini}(p) = \sum_i p_i(1 - p_i) \tag{7}$$

For categorical features, the possible splits to calculate information gain from are obvious, but for numerical features it is less so. The way to handle these is to first sort the feature from lowest to highest, then split between each numerical value, for example on the mean between value i and i+1 (unless they are the same value).

As the data set gets split in two each time, the growth of the tree is exponential.

We created a toy example to more easily explain decision trees and boosting, where we look at the Body Mass Index (BMI) of randomly generated "people". The BMI is a function of height (in [m]eters) and weight (in [kg]),

$$BMI(h, w) = \frac{w}{h^2} \tag{8}$$

To illustrate decision trees' tendency to overfit, we also included an irrelevant column of data, namely if the person is Swedish or not. The randomly generated data can be seen in table 1

H [cm]	W[kg]	Swedish	BMI
184.96	68.04	No	19.9
178.61	68.01	No	21.3
186.47	78.62	Yes	22.6
195.23	46.30	No	12.1
177.65	49.12	Yes	15.6
177.65	66.56	No	21.1
195.79	59.80	Yes	15.6
187.67	79.71	No	22.6
175.30	61.37	No	20.0
185.42	53.81	Yes	15.7

Table 1: Table of (H)eight, (W)eight, nationality(Swedish or not Swedish) and BMI for randomly generated test data.

Allowing a decision tree to run to a depth of 3 will lead to overfitting, where the algorithm starts fitting to noise or useless data, as seen in Figure (8)

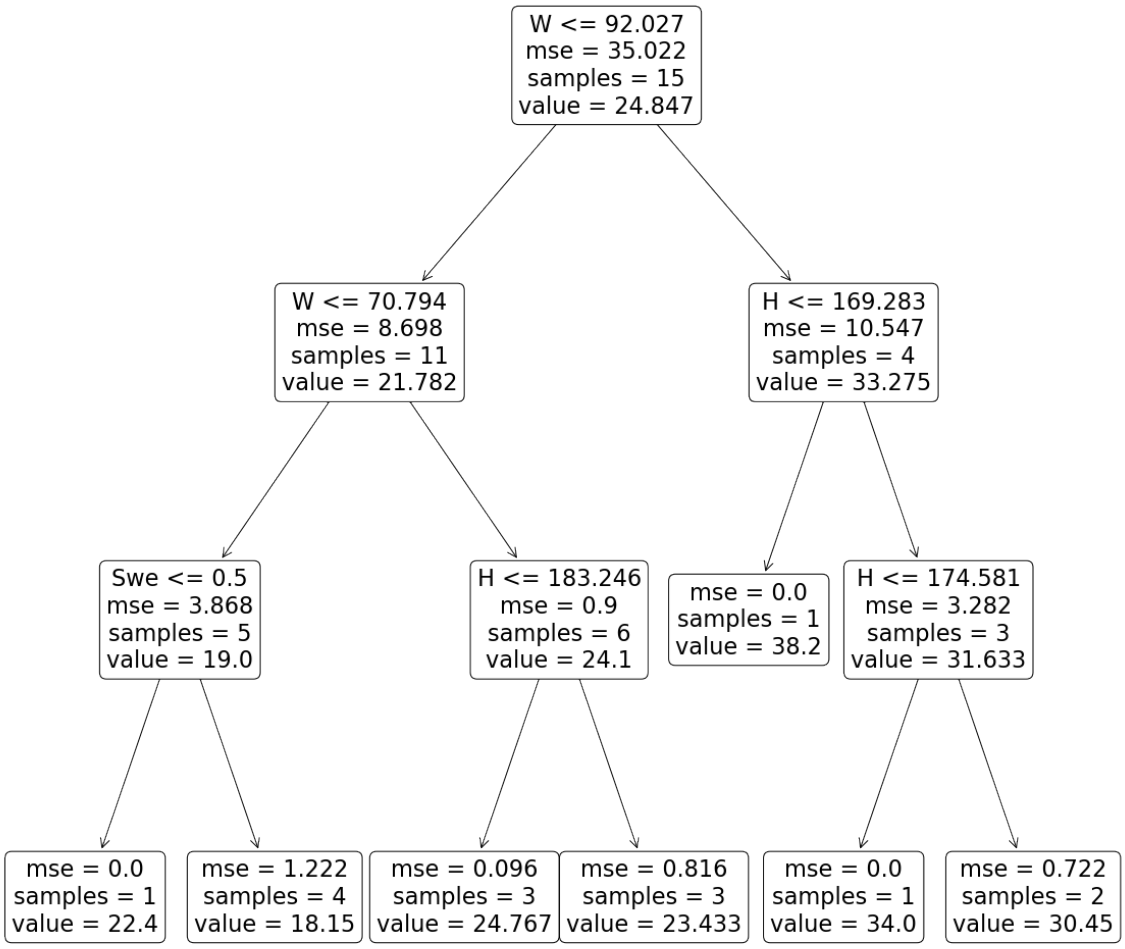


Figure 8: A decision tree fitted to the BMI toy example. As seen on the far left, a split has been made along the irrelevant feature "Swedish", illustrating the decision tree algorithms penchant to overfit.

By using a weaker learner, in this case a decision tree of depth 2, the result is a tree that is not overfit, see Figure(9). This is of importance in ensemble methods such as boosting.

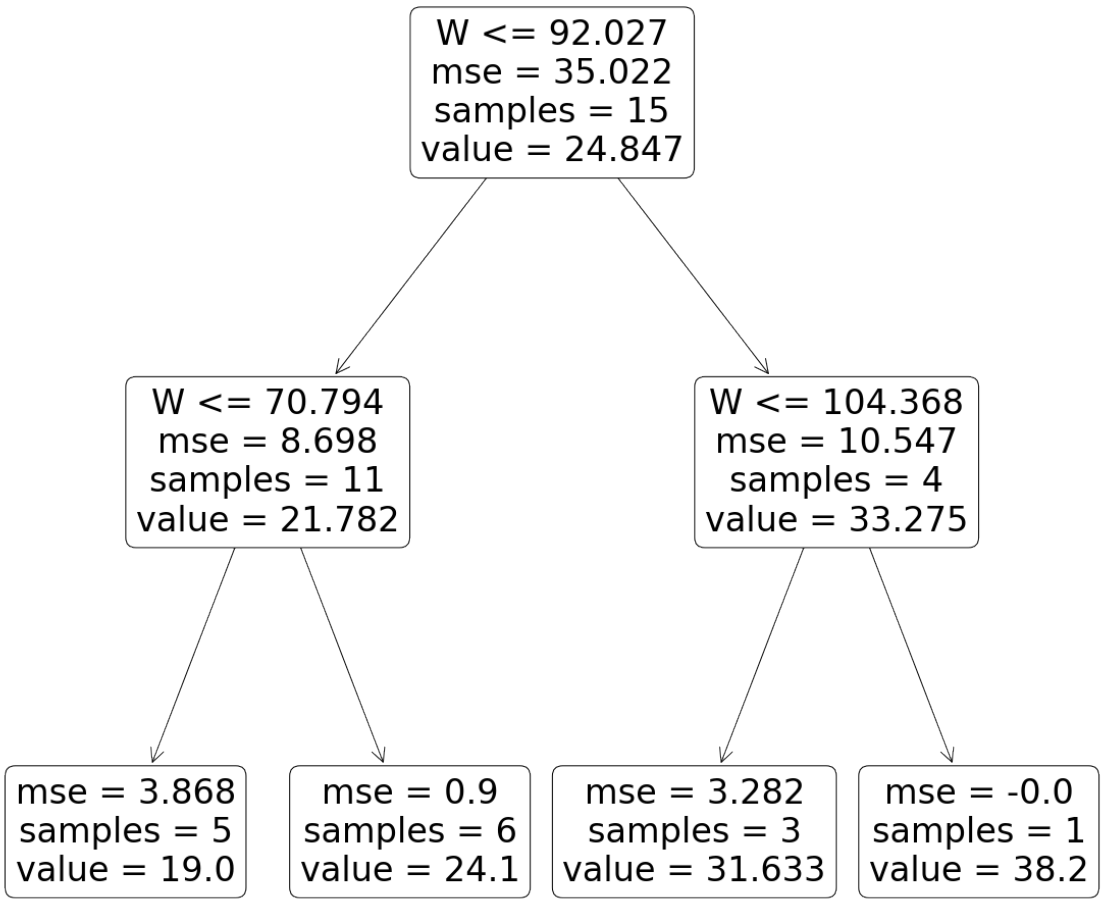
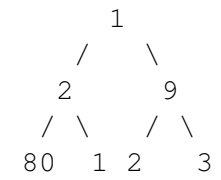


Figure 9: A decision tree fitted to the BMI toy example. Setting the maximum depth to 2, overfitting is hindered.

It is worth noting that the decision tree algorithm described above is a so-called greedy algorithm, meaning it will choose the split that gives the highest information gain in that specific node, which may not be the one that would give the highest information gain if you looked several steps ahead.

An easy to follow example of a greedy algorithm getting a poor result is if you consider the triangle of numbers below. Assume your goal is to find the highest sum possible while going from the top to the bottom. We can easily see that the value 80 is so huge that the best result is found by following

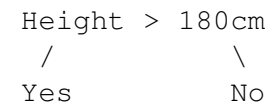
the leftmost path, yielding a sum of 83.
A greedy algorithm on the other hand will start at the top, and select the right path, as $9 > 2$, then $3 > 2$, giving a final sum of 13, much lower than the optimal result.



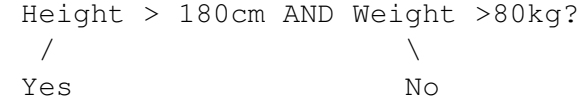
With the above general introduction to decision trees, it's time to describe one specific algorithm, Classification and Regression Trees(CART).

2.5.1 CART

In the CART algorithm the splits made are univariate, meaning the splitting criterion only looks at the impurity-loss of splitting along one feature, such as



while one could imagine splitting like



In the univariate splitting scheme, the latter example would be done in two steps, first split by height, then split the YES child by weight. CART splits only split in two, Yes or No (True or False), and continues splitting children nodes until:

- The child node is pure
- The Gini index of all possible splits are equal
- The improvement of the impurity is lower than a predefined value
- The tree reaches a predefined depth
- The node size (number of samples contained in the node) reaches a predefined size

To decide which feature (k) and what threshold value (t_k), such as height >180cm above, the CART algorithm has two cost functions to minimize. One for categorical features, one for numerical features.

The cost function for classification is [3]:

$$\mathcal{C}(k, t_k) = \frac{m_{left}}{m}\mathrm{G}_{left} + \frac{m_{right}}{m}\mathrm{G}_{right}, \tag{9}$$

where $\mathrm{G}_{left/right}$ is the Gini impurity of the left/right or yes/no, true/false subset, the $m_{left/right}$ are the number of samples in the left/right subsets, and m is the number of samples in node k.

The cost function for regression is similar, but instead of minimizing Gini impurity, we want to minimize Mean Squared Error (MSE)[3].

$$\mathcal{C}(k, t_k) = \frac{m_{left}}{m}\mathrm{MSE}_{left} + \frac{m_{right}}{m}\mathrm{MSE}_{right}, \tag{10}$$

where the MSE for a specific node is defined as

$$\mathrm{MSE}_{node} = \frac{1}{m_{node}} \sum_{i \in node} (\bar{y}_{node} - y_i)^2, \tag{11}$$

with

$$\bar{y}_{node} = \frac{1}{m_{node}} \sum_{i \in node} y_i. \tag{12}$$

After the tree has finished splitting, each leaf should make a prediction of the target value. For pure nodes, this is obvious, while for impure nodes the target with the highest representation in the leaf "wins".
Finally, the CART algorithm uses a technique called Cost-Complexity Pruning to avoid overfitting. It is quite similar to L1 regularization, where the total impurity is given by

$$I = I_{\text{Gini}} + \alpha|N|, \tag{13}$$

where $\alpha > 0$ is a tuning parameter and $|N|$ is the total number of nodes in the tree.

2.5.2 Pros and cons of Decision Trees

Some of the pros of Decision trees are

- They are easy to interpret and to explain
- They are independent of feature scaling
- They can model nonlinear relationships
- They are easy to display (unless they become large)

Some of the cons of Decision Trees are

- They are easy to overfit

- They may need clever pruning
- They may become computationally expensive for even simple fits
- the output range is bounded by the training examples, leading to poor predictive accuracy
- small changes in training data may lead to completely different trees
- trees may easily become biased by a biased training data set
- using impurity or information gain as cost may cause the tree to prefer splitting along features containing many categories. Can be rectified by using ratios [11].

2.5.3 Ensemble methods: Boosting

In the boosting algorithm one uses several weak learners together, in order to slowly converge towards the real value. A weak learner is a learner that does slightly better than pure chance.

The boosting algorithm is described in algorithm 1, and requires a training data set $\{(x_i, y_i)\}_{i=1}^n$, a differentiable loss function (such as MSE) $\mathcal{L}(y, \phi(x))$, where $\phi(x)$ is a weak learner, and a number of iterations M.

Algorithm 1: Gradient Boosting (Algorithm 16.4 in Murphy [9])

Initialize $f_0(\mathbf{x}) = \text{argmin}_\gamma \sum_{i=1}^N \mathcal{L}(y_i, \phi(\mathbf{x}_i; \gamma))$;
for *m=1:M* **do**
  Compute the gradient residual using $r_{im} = -\left[\frac{\partial \mathcal{L}(y_i, f(\mathbf{x}_i))}{\partial f(\mathbf{x}_i}\right]_{f(\mathbf{x}_i)=f_{m-1}(\mathbf{x}_i)}$
  Use the weak learner to compute γ_m which minimizes $\sum_{i=1}^N (r_{im} - \phi(\mathbf{x}_i; \gamma))^2$;
  [optional] Do a line search to compute ν_m, else use a static value ν.
  Update $f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \nu\phi(\mathbf{x}_i; \gamma_m)$;
Return $f(\mathbf{x}) = f_M(\mathbf{x})$

The first step is to make an initial guess at the target variable. For the BMI example from table 1, one would typically start with the mean BMI as the guess, as seen in the column $F0$ in table 2. Using Squared Error ($\mathcal{L}(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$) as the loss function, the gradient residual is simply $y - \hat{y}$.Taking the pseudo-residuals of BMI and F0 gives the values seen in column PR0. The next step is to create a weak learner, often a tree that is not deep, (a maximum of 8-32 leaves is a value one often will see suggested), using the pseudo residuals as the target values. In the BMI example, the first tree, $\phi_0(\mathbf{x}; \gamma)$ is shown on the left in Figure (10). It has been limited to a depth of 1 for readability. The fifth, optional, step calculates a step length γ_0 via a line search. Updating f_0 to f_1 as seen in the algorithm yields the values shown in the F1 column.
Taking one more iteration in the same way gives the next set of columns in table 2, where the second weak learner, again restricted to a maximum depth of 1, is shown on the right in Figure (10). Had

we allowed slightly deeper trees, and more iteration, one would more easily see that the predicted values converge towards the real values.

BMI	F0	PR0	ϕ_0	gamma0	F1	PR1	ϕ_1	gamma1	F2
19.9	18.65	1.25	2.6	1.14	21.614	-1.714	-0.22	1.04	21.3852
21.3	18.65	2.65	2.6	1.14	21.614	-0.314	-0.22	1.04	21.3852
22.6	18.65	3.95	2.6	1.14	21.614	0.986	-0.22	1.04	21.3852
12.1	18.65	-6.55	-3.9	0.79	15.569	-3.469	-3.47	0.65	13.3135
15.6	18.65	-3.05	-3.9	0.79	15.569	0.031	-0.22	1.04	15.3402
21.1	18.65	2.45	2.6	1.14	21.614	-0.514	-0.22	1.04	21.3852
15.6	18.65	-3.05	-3.9	0.79	15.569	0.031	-0.22	1.04	15.3402
22.6	18.65	3.95	2.6	1.14	21.614	0.986	-0.22	1.04	21.3852
20.0	18.65	1.35	2.6	1.14	21.614	-1.614	-0.22	1.04	21.3852
15.7	18.65	-2.95	-3.9	0.79	15.569	0.131	-0.22	1.04	15.3402

Table 2: Development of predicted values for two steps of boosting. BMI is the target variable, F0,F1 and F2 are predicted values for each boosting step, PR0,PR1 are the pseduo-residuals for each new predicted value. ϕ_0,ϕ_1 are the steps calculated by each learner, and gamma0,gamma1 are the learning rates for each learner.

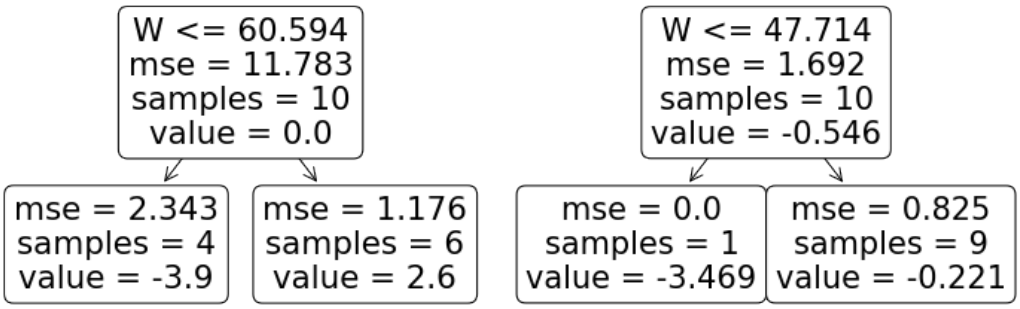


Figure 10: Left: The first learner fit to the BMI example. Right: The second learner fit to the BMI example.

2.5.4 XGBoost

XGBoost, or Extreme Gradient Boosting, is a gradient boosting algorithm optimized for speed and performance, with support for parallelization and distributed computing (and more). Describing the theory in detail will take many pages, and probably cannot be done better than in Tianqi Chens article [12].

3 Implementation

3.1 Using PCA on MNIST

After importing the 8x8 MNIST data set, we plotted a heat map of the feature matrix, see Figure (11), to visualize the data, and see if we noticed any interesting characteristics. What we see is a banded structure, where especially the black bands can be seen to indicate features of little to no importance in predicting a digit. We then use a second method of visualizing the data, see Figure (12), both for the 8x8 and the 28x28 MNIST data sets, where we plot the average digit of a subset of 80% of the data next to a sample digit from the respective set.

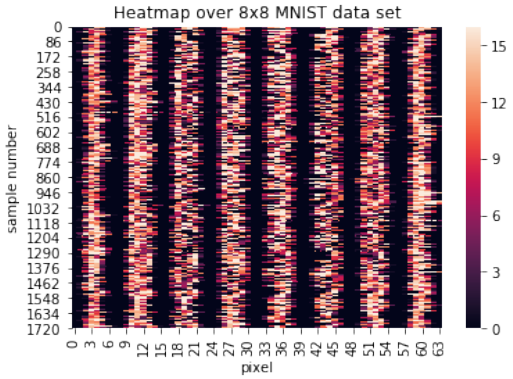


Figure 11: Visual representation of the 8x8 MNIST feature matrix.

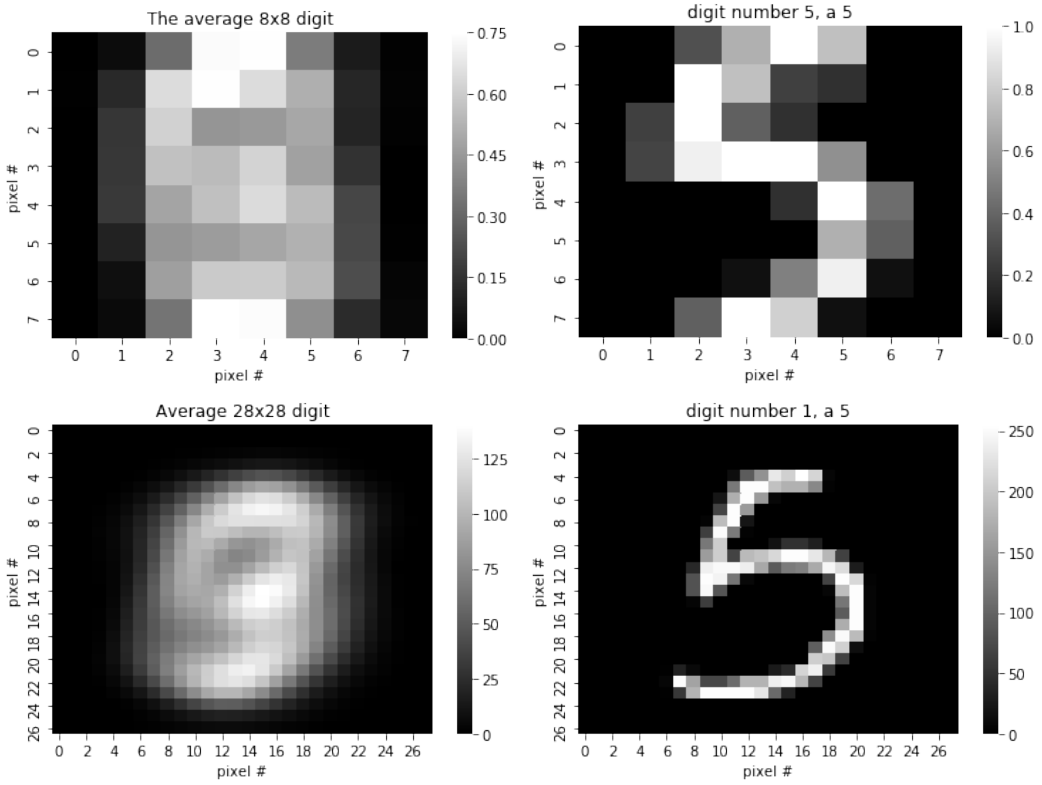


Figure 12: The average digits of the 8x8 and 28x28 MNIST data set respectively in the left column, and a sample digit from each set in the right column. Colours are inverted to more clearly see the pixels of lowest value.

Both methods of visualization indicate that there are many features of negligible importance, so that this data set is a good candidate for dimensionality reduction.
Performing a principal component analysis (PCA) on both data sets confirm our suspicions, as seen in Figure (13). For the 28x28 MNIST set, it seems we can use less than 4% of the features and obtain close to the same results as with the full set.
For time consuming methods, such as boosting and SVM, the time saved allows for parameter tuning using far more parameter values than when using the full set without having to wait days, or even weeks, for results.

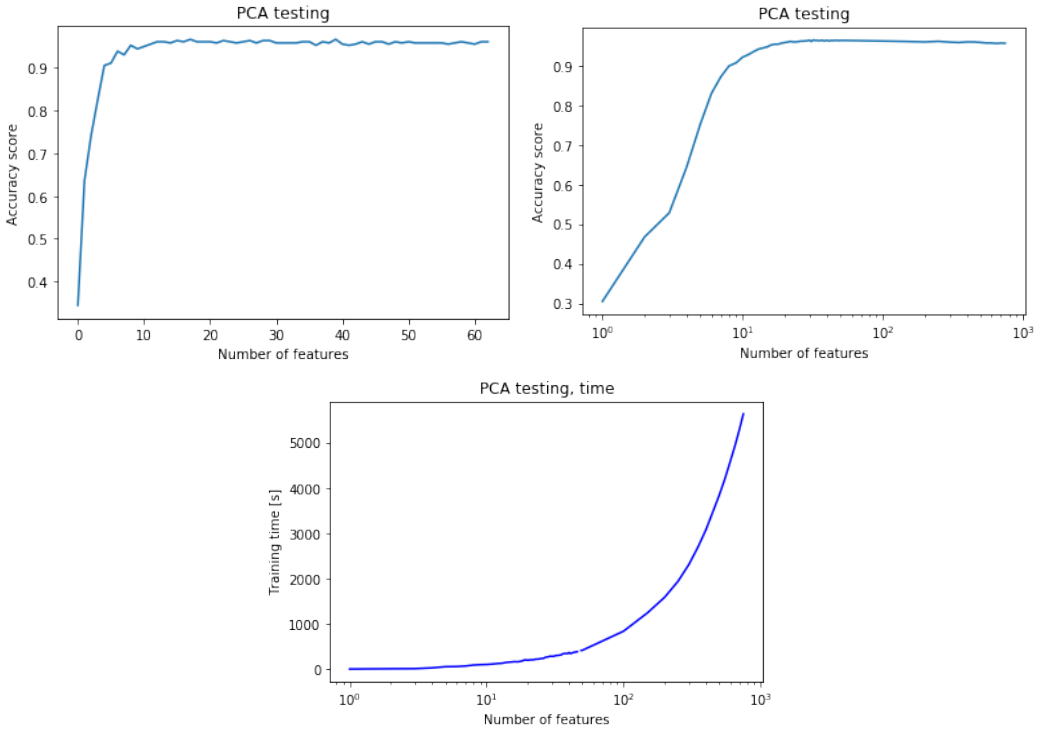


Figure 13: The top row shows the accuracy score obtained using XGBoost on the 8x8 and 28x28 MNIST data sets respectively. The bottom row shows the increase in training time using XGBoost as a function of number of PCA features. Note the logarithmic x-axis for the 28x28 plots.

3.2 Building CNN with Keras

First, all necessary libraries are called as below:

```
import numpy as np
import keras
from keras.datasets import mnist
from keras.models import Model, Sequential
from keras.layers import Dense, Input, Conv2D, MaxPooling2D, Dropout, Flatten
from keras import backend as k
from keras.utils import to_categorical
import matplotlib.pyplot as plt
```

Then, MNIST data set is loaded:

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

The MNIST data set including 70000 images is divided to (x_train, y_train) and (x_test, y_test) so that the training and test data include 60000 and 10000 images respectively. Then, format of `x_train` and `x_test` are checked and they are reshaped it if it was required as below:

```
img_row, img_col= (28, 28)

"""
    img_row and img_col are image dimensions and in MNIST dataset it is 28 x 28
"""


# The ordering of the dimensions in input

if k.image_data_format() == 'channels_first':

"""  channel_first : corrspond to inputs with shapes (batch, channels, height,
    width)  """

    x_train = x_train.reshape(x_train.shape[0], 1, img_row, img_col)
    x_test  = x_test.reshape(x_test.shape[0], 1, img_row, img_col)
    np = (1, img_row, img_col)


else:

"""  channel_last :  corrspond to inputs with shapes (batch, height, width,
    channels) """

    x_train = x_train.reshape(x_train.shape[0], img_row, img_col, 1)
    x_test = x_test.reshape(x_test.shape[0], img_row, img_col, 1)
    np = (img_row, img_col, 1)
```

In fact, the aim of above code is to set `x_train=x_train.reshape(60000,28,28,1)` and `x_test=x_test.reshape(10000,28,28,1)`. Here last digit " 1 " indicates the gray scale image.
Since it is usually a challenge for neural networks to work with categorical data directly, we have chosen to convert categorical class vector to a binary class matrix. In other words, for each categorical data we make a column including a set of zeros and a single one. The function `keras.utils.to_categorical` is used for making the binary class matrix.

```
from keras.utils import to_categorical
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

It is worth to note that functionality of this function is similar to application of `one hot encoding` technique.
Then, architecture of the layers are coded as following:

```
np     = Input(shape = np)
layer1 = Conv2D(32, kernel_size=(3, 3), activation='relu')(np)
"""
layer1: Conv2d layer which convolves
the image using 32 filters each of size (3*3)
"""
layer2 = Conv2D(64, (3, 3), activation='relu')(layer1)
"""
```

```
layer2: Conv2D layer which convolve
the image and is using 64 filters each of size (3*3)
"""

layer3 = MaxPooling2D(pool_size=(3, 3))(layer2)
"""
layer3: MaxPooling2D layer which picks
the max value out of a matrix of size (3*3)
"""
layer4 = Dropout(0.5)(layer3)
"""
layer4: shows Dropout at a rate of 0.5
"""
layer5 = Flatten()(layer4)
"""
layer5: flatten the output obtained from layer4
and this flatten output is passed to layer6
"""
layer6 = Dense(250, activation = 'sigmoid')(layer5)
"""
layer6: shows a hidden layer of neural network containng 250 neurons.
"""
layer7 = Dense(10, activation = 'softmax')(layer6)
"""
layer7: shows output layer having 10 neurons for 10
classes of output that is using the softmax function
the final Dense layer must have 10 neurons since we
have 10 number classes (0, 1, 2, ..., 9)
```

The first layer is Conv2D, which makes a convolution kernel. The first parameter in Conc2D indicates number of neurons. The kernel_size in Conv2D presents the height and width of the filter and ReLU is used as an activation function. The last parameter `(np)` in Conv2D indicates size (dimension) of the input image. The second layer is set same as first layer and the only difference is number of neurons. The third layer is Maxpooling and it is described in details in section (2.3.2). The fourth layer is Dropout which randomly turns off some neurons and stops memorizing of training data by network. This refers to this fact that not all the neurons will be active at the same time and when neuron is inactive cannot learn anything. The Dropout layer helps to avoid overfitting. The fifth layer i.e. Flatten layer is described in details in section (2.3.3) respectively. The sixth layer (Dense layer with 250 neurons) is hidden layer of neural network and the last one (Dense layer with 10 neurons) is another hidden layer which is created to classify the numbers to ten classes between 0-9. The relevant results are presented in table (3), figure (14), and figure (15) which are interpreted and discussed in section (4.1).

3.3 Support Vector Machines(SVM)

MNIST

First of all, neccessary libraries called below:

```
import matplotlib.pyplot as plt
```

```
import numpy as np
from sklearn import datasets, svm, metrics
from sklearn.datasets import fetch_openml, load_digits
from sklearn.model_selection import train_test_split, GridSearchCV
import pandas as pd
import seaborn as sn
from sklearn.compose import ColumnTransformer
import scikitplot as skplt
from PIL import Image
```

Loading the MNIST dataset, and splitting in training and test data sets

```
#full 28x28 pixel data set
mnist = fetch_openml('mnist_784')
X = mnist.data
y = mnist.target

#Setting division size
trainingShare = 0.8
seed  = 1234
#splitting in train and test data sets
trainingShare = 0.8
seed  = 1234
XTrain, XTest, yTrain, yTest=train_test_split(X, y,train_size=trainingShare,
                                              random_state=seed, stratify = y)
```

Fitting the model took a long time using the full 28x28 pixel data set using a small laptop with 4 cores. As discussed in section 3.1 we found good reason to reduce amount of components with PCA, and we kept the 30 most significant ones for SVM on MNIST. For the following PCA will already have been implemented.

```
#making a model
svc = svm.SVC(C = 3, gamma = 'scale', kernel = 'rbf')
#fitting the model using 30 most significant componants from PCA on the data set
svc.fit(X2D, yTrain)
predicted = svc.predict(X2Dt)
```

There are a number of parameters to be tuned in both the classification, and the regression implementation of SVM in scikit learn.
Some of them are as follows:

- C: Is a parameter that regularizes the slack parameters. Is dependent on λ and γ, (which both are Lagrange multipliers). For large values of C focuses more on the correctly classified points near the decition boundary, smaller C values focuses on data further away. [8]. In classification the C parameter influence where the margin falls.
- Kernel: If the data set is not easily separated into classes, then the kernel trick can be applied to increase the dimension of the data set to explore whether the classes are more easily separable there. The different kernels available are: Linear, polynomial(this kernel has an additional parameter, degree, which can be altered by the user), radial basis function, and the sigmoid function.

- Degree: Relevant for the polynomial kernel to set which polynomial degree one wants the kernel to be.

- Gamma: Used in the kernels, different from the Lagrange multiplier used above. Usually set to 1/n_features but sklearn uses "scale" which performed best for all the data sets we tried. scale = 1 / (n_features * x.var()).

For tuning different hyper parameters it is useful to use k-fold cross validation. Scikit learn has a tool for this called GridSearchCV where you can test several different values of as many hyper parameters as you want together to find the optimal fit. Seeing how time consuming it was to run the 28x28pixel dataset, we applied GridSearchCV for SVM only on the smaller MNIST data set which contains 8x8 pixel sized pictures. This gave a starting point to tune and test out which parameters gave the best fit.

Credit Card Data

Implementing SVM on the credit card data set used in Project 2 was straight forward. We tried using PCA, but it did not yield any interesting results for SVM. The implementation is as follows

```
#making model
classifier = svm.SVC(C = 1, kernel = 'rbf', gamma = 'scale',
    decision_function_shape= 'ovo', probability = True)
classifier.fit(XTrain_CD, np.squeeze(yTrain_CD))
#using predict_proba to get probabilities to be used in computing area under
    curve and for gains plot.
predicted_CD = classifier.predict_proba(XTest_CD)
predicted_ = classifier.predict(XTest_CD)
```

Tuning the parameters was trial and error based, and it ended up being the pre-assigned variables that performed best.

Franke Function

As for the other data, implementing SVM with regression for Franke function was simple. For this instance as for the MNIST set we tried GridSearchCV, but after two days we gave up on getting results. Therefore the parameters have been hand tuned to try to find the optimal fit.
Implementation:

```
#making model
svr = svm.SVR(C=1.5, gamma = 'scale',kernel='rbf', verbose=True, epsilon = 0.01)
fitR = svr.fit(xtr,ytr)
#predicting using test set
pred = svr.predict(xte)
```

Different from SVM for classification, for the regression case epsilon defines the area around the hyperplane where observations are not penalized, points falling outside the tube are used as support vectors. The C parameter in the regression case is used to minimize the distance of the points outside the tube to the hyperplane.

3.4 Boosting

MNIST

Boosting is implemented via the XGBoost library in python. There is a plethora of parameters to de/activate and tune to avoid overfitting. Some parameters controls the model complexity, while others add stochasticity to reduce the impact of noise.
Parameters to curb model complexity include:
max_depth: limits the depth of each tree, the lower the value, the more conservative the algorithm is, i.e. the less it is prone to overfit. The default value is 6.
min_child_weight: essentially a parameter reflecting a minimum number of instances in a node. If a proposed split would result in nodes with a lower weight than this cutoff, the split is not made.
$gamma$: Minimum loss reduction required to perform a split. If a proposed split does not improve the loss more than gamma, the split is not made.

Parameters to increase stochasticity include:
$subsample$: This parameter makes the algorithm sample a fraction of the feature matrix, where a new subsample is drawn for each new tree.
$colsample\_bytree$: Draws a fraction of features (columns) from the feature matrix. As we have performed a PCA and reduced the dimensionality already, be careful with this parameter.

Other important parameters:
$learning\_rate$: Reducing learning rate reduces step length, decreasing your chance of overstepping a minima. This parameter should be adjusted together with num_round.
num_round: This parameter increases the number of iterations. Should be increased it $learning\_rate$ is reduced.

The PCA performed in chapter 3.1 shows that we can use very few features and still reach the same accuracy. In the following, the 30 most significant PCA features have been used.
While one will often use the scikit-learn like booster model XGBClassifier, we find it easier to tune a model made simply using the $train$ method, as seen in the example below. The feature matrices have had their dimensionality reduced to 30 here.

```
dX = xgb.DMatrix(X_train_pca,y_train)
dXt = xgb.DMatrix(X_test_pca,y_test)
dXv = xgb.DMatrix(X_val_pca,y_val)
classifier_params = {
    'objective':'multi:softmax',
    'num_class':10,
    'max_depth':4,
    'subsample':0.5,
}
evallist=[(dX,'train'),(dXt,'test')]

num_round=1000
early_stopping=100
```

```

bst=xgb.train(classifier_params,dX,num_round,evallist,early_stopping_rounds=
    early_stopping,verbose_eval=True)

y_pred_val=bst.predict(dXv)
print(accuracy_score(y_pred_val,y_val)
```

Parameter tuning can be done by hand, or by running a GridSearchCV (scikit-learn function).

Credit Card Data

Implementing XGBoost on the credit card data set used in Project 2 was straight forward. Using PCA to reduce the dimensionality to 18, the implementation (with the optimal parameters found) is as follows

```
dtrainpca= xgb.DMatrix(x_train_pca,y_train_cc)
dtestpca= xgb.DMatrix(x_test_pca,y_test_cc)

classifier_params = {
    'eta':0.05,
    'max_depth':60,
    'objective':'binary:logistic',
    'min_child_weight':5,
    'subsample':0.8,
    'colsample_bytree':0.8,
    'lambda':10.,
    'alpha':10.,
    'gamma':10.,
    'colsample_bylevel':0.8,
    'eval_metric':'error',
}
evallist=[(dtrainpca,'train'),(dtestpca,'test')]

num_round=200
early_stopping=20

bst=xgb.train(classifier_params, dtrainpca, num_round, evallist, \
              early_stopping_rounds = early_stopping, verbose_eval=False
             )

y_pred_cc = bst.predict(dtestpca,ntree_limit=num_round,output_margin=True)
y_pred_cc_tr = bst.predict(dtrainpca,ntree_limit=num_round)
```

Franke Function

As for the other data, implementation for the Franke function is simple. With hand-tuned optimal parameters, the implementation goes as follows.

```
dtrain = xgb.DMatrix(x_train,y_train)
dtest=xgb.DMatrix(x_test,y_test)
dval = xgb.DMatrix(x_val,y_val)
```

```
4
5  params={
6      'max_depth': 6,
7      'learning_rate':0.5,
8      'lambda':5,
9      'objective':'reg:squarederror',
10     'n_estimators':200,
11     'seed':42,
12 }
13 params['eval_metric']='rmse'
14 num_boost_round=14
15 model = xgb.train(params,dtrain,num_boost_round=num_boost_round,
16                   evals=[(dtrain,'Train'),(dtest,'Test')],
17                   early_stopping_rounds=20, verbose_eval=False)
18
19 y_te_pred = model.predict(dtest)
20 y_tr_pred = model.predict(dtrain)
21 y_v_pred = model.predict(dval)
```

Note the inclusion of a validation set, as the above implementation will keep iterating until it has found an optimal value for the test set. This causes a bias towards this test set, so we need a completely unseen set, the validation set, to score the model.

4 Result and discussion

4.1 CNN with Keras

In this section a numerical experiment is performed to study the effect of epochs, neurons, batch_size, and Maxpooling layer on performance of CNN model. Results of this numerical experiment have been summarized in table (3). In this experiment a certain architecture of CNN is trained with train data then, test data is used to estimate the accuracy and loss. In addition, no Dropout layer is implemented. The elapsed time is reported in the last column. Here L1 and L2 present number of neurons in layer number one and two respectively. It is worth to note that by trial and error and using other experiences on CNN, we realized an architecture including L1=32 and L2=64 gives reasonable accuracy, loss and runtime on MNIST data [13]. Furthermore, it seems setting L1 $<$ L2 gives better accuracy and elapsed time. As a reference, one can compare test 1 with test 3 and test 5 with test 7.

Test #	L1	L2	Epochs	Batch_size	Accuracy	Loss	Maxpool	Time(s)
test 1	32	64	10	500	0.9886	0.0331	yes	220
test 2	32	64	10	500	0.9872	0.0403	no	310
test 3	64	32	10	500	0.9869	0.0377	yes	310
test 4	64	32	10	500	0.9842	0.0467	no	380
test 5	16	32	5	500	0.9805	0.0659	yes	45
test 6	16	32	5	500	0.9749	0.0826	no	70
test 7	32	16	5	500	0.9740	0.0863	yes	55
test 8	32	16	5	500	0.9785	0.0709	no	70
test 9	32	64	20	100	0.9916	0.0252	yes	500
test 10	32	64	20	100	0.9886	0.0405	no	1200
test 11	32	64	20	50	0.9915	0.0273	yes	700
test 12	32	64	20	50	0.9890	0.0358	no	2000

Table 3: Illustrates the numerical experiment to study the effect of epochs, neurons, batch_size, Maxpool layer on performance of CNN model. In this experiment a certain architecture of CNN is trained with train data then, test data is used to estimate the accuracy and loss. In addition, no Dropout layer is implemented. The elapsed time is reported in the last column. Here L1 and L2 present number of neurons in layer number one and two respectively.

By comparing test 1, test 9, and test 11 that have similar configuration (L1, L2, and Maxpooling) we notice that reduction of batch_size from 500 to 100 and increasing epochs from 10 to 20 gives higher accuracy and longer runtime. Moreover, reduction of batch_size to 50 results no serious change in efficiency (accuracy and loss), but runtime of 500 (s) rises to 700 (s).
To study the effect of Maxpooling, we executed the similar architectures with and without Maxpooling layer. As it is expected, Maxpooling layer reduces the computation time by decreasing the dimensionality and this fact has been described already in section (2.3.2). Furthermore, Maxpooling improves the efficiency of CNN namely, gives higher accuracy and lower loss. However, comparing test 7 and test 8 shows the former that have Maxpooling layer results accuracy = 0.9740 and loss = 0.0863. Whereas, the latter without Maxpooling layer have accuracy = 0.9785 and loss = 0.0709. It seems the reason of this small difference that is around 10^{-3} order of magnitude, refers to this fact that CNN uses different weights for execution of test 7 and test 8.
The effect of Dropout layer is depicted in figures (14) and (15). In this experiment similar architectures are executed with and without Dropout layer on train and test data. It is worth to note that by trial and error and using other experiences on CNN, we realized Dropout=0.5 gives better accuracy, loss and runtime on MNIST data [13].

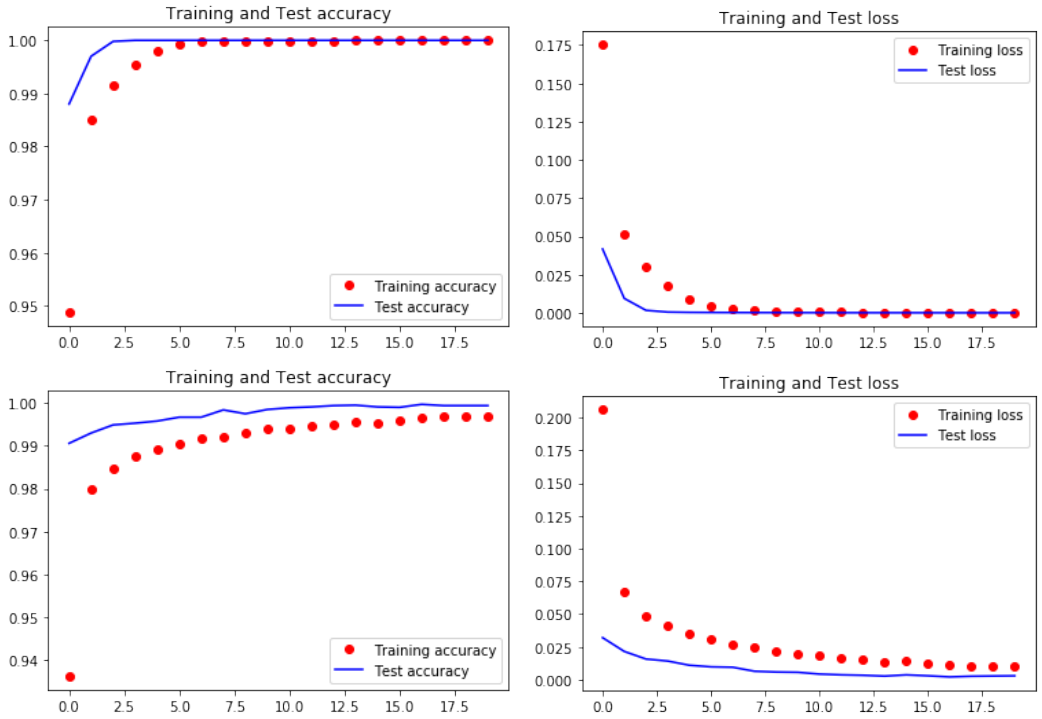


Figure 14: Comparison of accuracy and loss (y-axis) for training and test data with respect to number of epochs (x-axis). Here, the bottom plots include implementation of Dropout$= 0.5$. As it can be seen without Dropout layer in top row plots, train and test plots are saturated at epoch 5 and 2 respectively. However, implementation of Dropout layer in bottom plots allow to avoid overfitting til epoch=20 (Network architecture: L1=32 neurons, L2=64 neurons, epoch=20, batch_size=50).

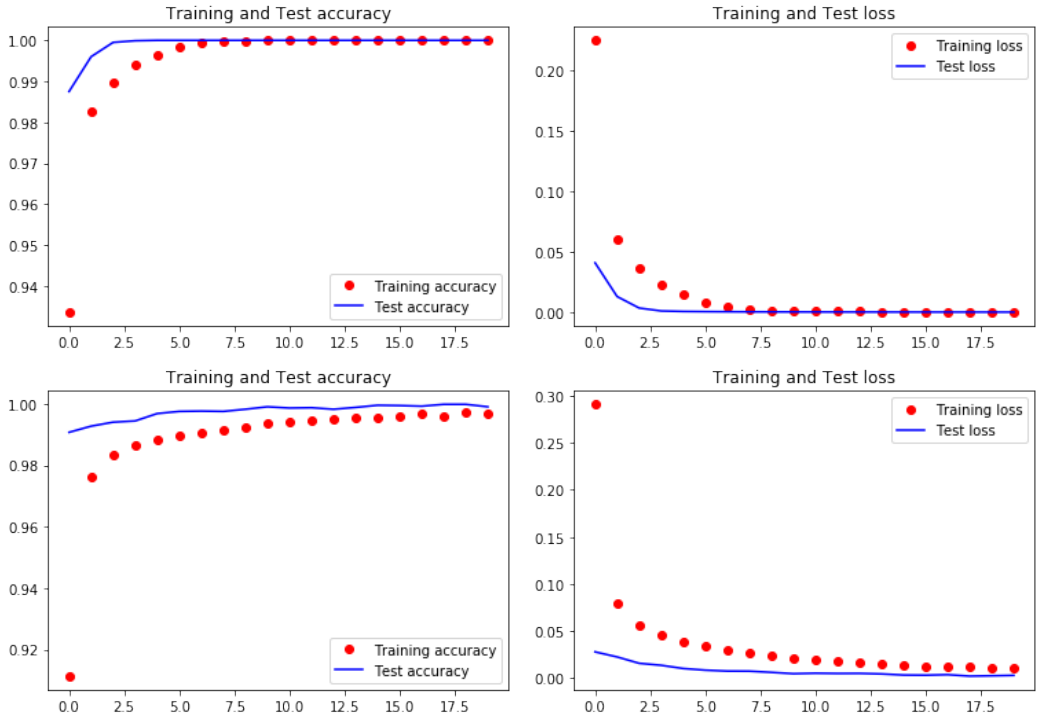


Figure 15: Comparison of accuracy and loss (y-axis) for training and test data with respect to number of epochs (x-axis). Here, the bottom plots include implementation of Dropout$= 0.5$. As it can be seen without Dropout layer in top row plots, train and test plots are saturated at epoch 1 and 4 respectively. However, implementation of Dropout layer in bottom plots allow to avoid overfitting til epoch=20 (Network architecture: L1=32 neurons, L2=64 neurons, epoch=20, batch_size=100).

As it can be observed, upper plots in figure (14) are for CNN model that do not have Dropout layer. In these plots train and test data are stabilized at epoch 5 and 2 respectively. Whereas, overfitting does not happen for bottom plots til epoch=20. The same result observed in figure (14). Namely, train and test data in upper plots without Dropout layer are saturated at epoch 4 and 1 while bottom ones experience no overfitting til epoch=20.

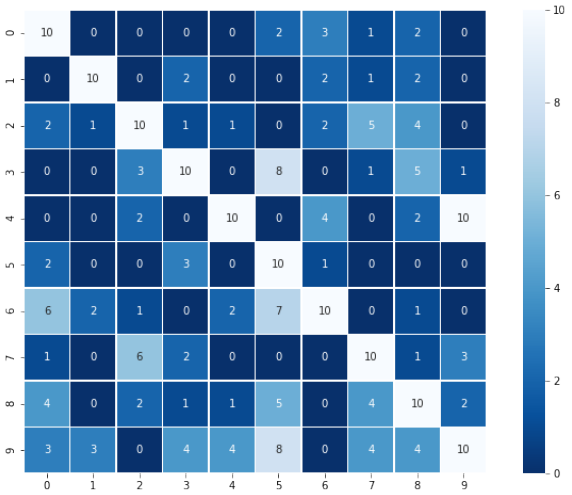


Figure 16: Confusion matrix for CNN on MNIST. This figure shows the performance of CNN algorithm for MNIST data. It is worth to note that the largest values are shown diagonally and marked with white color.

The confusion matrix which is shown in figure (16) shows the performance of CNN algorithm for MNIST data. Each row and column of confusion matrix presents actual class and predicted class respectively. Although, the diagonal values which are marked with white color, illustrate high performance of the model. However for 4, 9 model is confused significantly and also model confusion is observed for 6, 5 and 9, 5.

4.2 SVM

SVM on MNIST

The parameters yielding the best results were found by first using GridSearchCV on the small MNIST set (8x8pixel) then the results from there were used to tune the parameters for the model using the larger dataset (28x28pixel). The only non-default value is C = 3.
A modified confusion matrix of the results is shown in Figure (17), where the modification mentioned was to scale the diagonal values to equal the largest value in any non-diagonal location. This was done as the amount of correct guesses is so large that the misclassifications would be dwarfed in a regular confusion matrix.

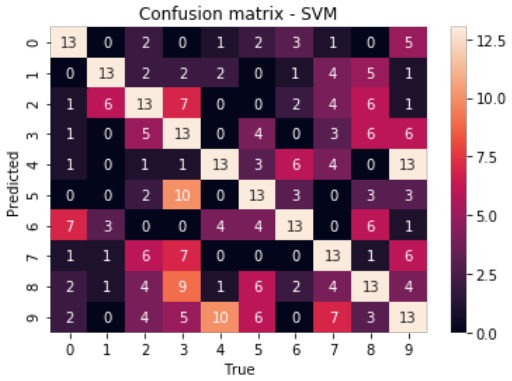


Figure 17: Confusion matrix for SVM on MNIST. Note that the diagonal has been scaled to the maximum non-diagonal value.

4.3 Boosting with XGBoost

Boosting MNIST

The parameters yielding the best results were tuned by hand (The only non-default values are 'max_depth': 4 and 'subsample': 0.5.)
A modified confusion matrix of the results is shown in Figure (18), where the modification mentioned was to scale the diagonal values to equal the largest value in any non-diagonal location. This was done as the amount of correct guesses is so large that the misclassifications would be dwarfed in a regular confusion matrix.

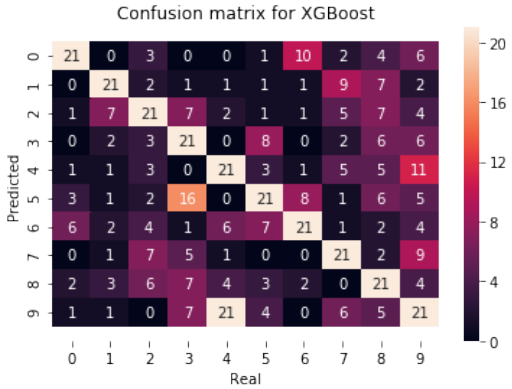


Figure 18: Confusion matrix for MNIST. Note that the diagonal has been scaled to the maximum non-diagonal value. [x and y labels should be Real/Predicted]

The best results of XGBoost on MNIST can be found in chapter 4.4

Boosting Franke's Function

The results of XGBoost on the Franke function data can be found in chapter 4.6. The R^2 score is one of the lowest we've gotten, which is perhaps not surprising. Dietterich [14] found for another popular gradient boosting method (AdaBoost) that it placed more weight on the noisy datapoints than the 'uncorrupted' data points. His article is about classification, not regression, but the point may still stand. Dietterich found that Bagging (Bootstrap Aggregating)got better results on his data, so a quick attempt at using Bagging is performed on the Franke data. See one of several good books for information on the Bagging algorithm [9], [8]. Disappointingly the results of bagging were even worse, with an accuracy score of 0.864.

4.4 MNIST

What we mainly have been focusing on in this project is to study the MNIST data set and to compare performance for various methods, one method known to be optimal for picture recognition CNN, one method known for its strength in classification XGBoost, and one model out of curiosity SVM. We base our comparison on the accuracy score for the given method. (4).

Method	CNN	XGBoost-PCA	SVM-PCA
Accuracy	0.9916	0.9657	0.9821

Table 4: Comparison of CNN with full data set, XGBoost and SVM where we applied PCA keeping the 30 most significant components.

4.5 Credit Card Data

As an extension of project two [15] we applied XGBoost and SVM on the credit card data to see whether we would be able to achieve better results. See table (5).

Method	Error rate		Area Ratio	
	Training	Validation	Training	Validation
GD	0.191	0.192	0.440	0.433
Newton	0.184	0.185	0.473	0.459
SGD	0.184	0.185	0.471	0.457
SGD-mini	0.189	0.190	0.442	0.427
scikitlearn	0.188	0.189	0.465	0.451
NN	0.177	0.180	0.571	0.549
NN-PCA	0.176	0.179	0.566	0.539
NN 'decimated'	0.177	0.181	0.562	0.544
XGBoost	0.173	0.181	0.610	0.566
XGBoost-PCA	0.169	0.183	0.612	0.545
SVM	0.172	0.180	0.574	0.451

Table 5: Accuracy classification for credit card data set which compares the performance of *gradient descent (GD)*, *Newton Raphson*, *stochastic gradient descent (SGD)*, *stochastic gradient descent with mini batches (SGD-mini)*, *scikitlearn*, *NN*, *NN PCA* with 18 features, *NN 'decimated*(Explained in the text), XGBoost and SVM.

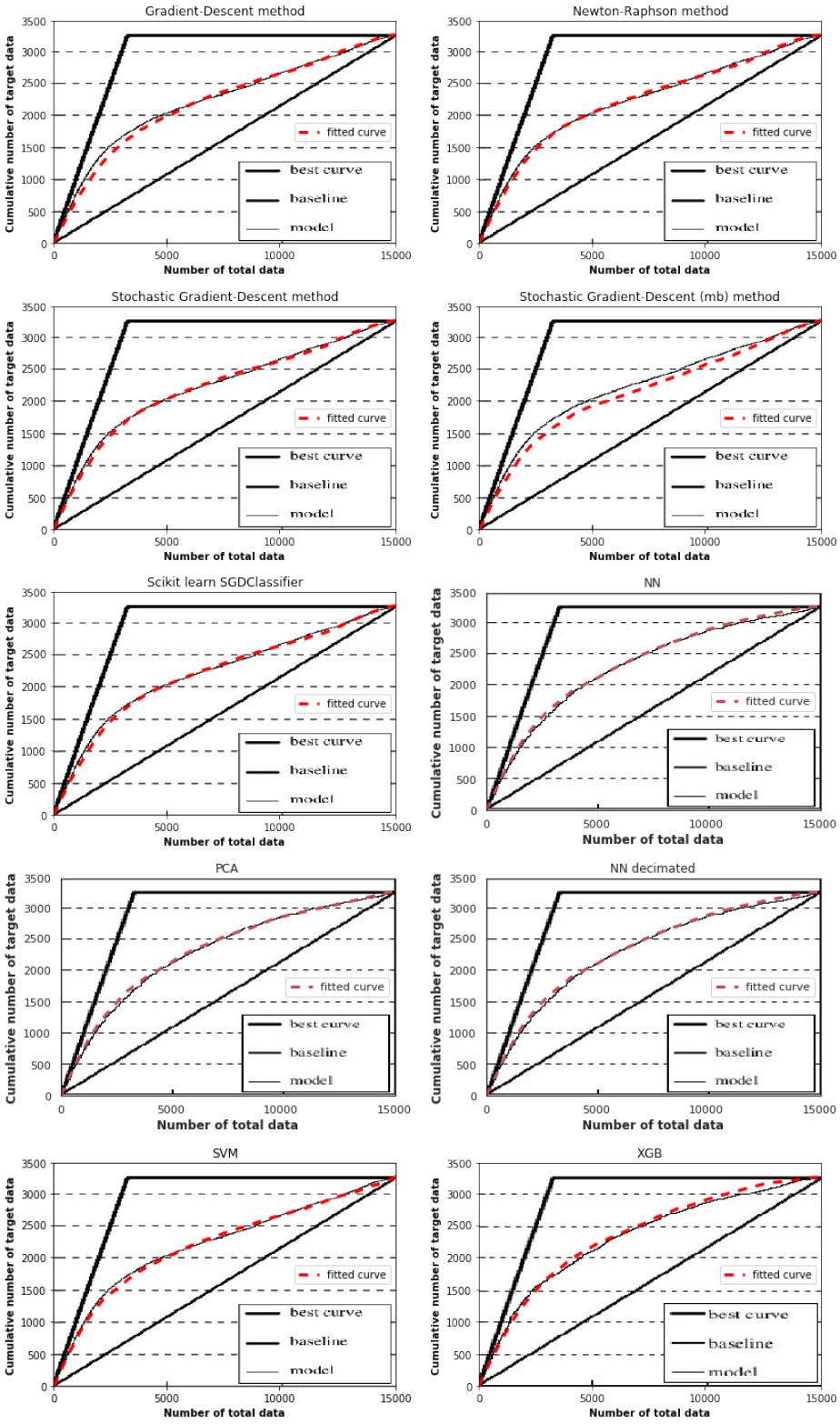


Figure 19: In this figure, as it can be seen the horizontal axis presents number of total data and vertical axis presents cumulative number of target data. The bold black line shows ideal status, the diagonal black line is the baseline, the curved black line is the results from given scientific article and red dashed line is result of our model.

4.6 Franke Function

As an extension of project one and two [15] we applied XGBoost and SVM on the Franke funktion to see wether we were able to achieve better results with these new methods.

Method	MSE(train)	R^2(train)	MSE(test)	R^2(test)
OLS	0.000585	0.9888580	0.0021412	0.9592810
Ridge	0.0002058	0.9960936	0.0011889	0.9778371
Lasso	0.0006745	0.9869749	0.002109	0.9501375
NN	0.0007536	0.9910987	0.0008016	0.9905277
XGBoost	0.0092469	0.9048811	0.0103971	0.9026183
SVM	0.0101733	0.8859237	0.0093828	0.9166053

Table 6: Comparison the performance of OLS, Ridge, Lasso, NN, XGBoost and SVM on Franke's function. Neural network shows the best fit [Note: an update of numpy reduced the performance of regression NN, we did not have time to explore what happened.]

5 Conclusion

5.1 MNIST

Looking at the accuracy scores, we see that CNN performs best, while both XGBoost and SVM get over 95% correct predictions. The main reason CNN outperforms the other two methods is likely that it contains spatial information for each digit, while XGBoost and SVM sees each digit as a row of numbers. Additionaly, The two latter methods employed PCA dimensionality reduction, which may cause a light drop in accuracy.

5.2 Credit Card

Looking at the Area Ratio for test data, we see that XGBoost performs best, with NN as a close contender. SVM performs rather poorly, similar to the Logistic Regression method. One reason for this may be that some data is lacking in the data set, where we set so-called nonsensical data to 0. XGBoost has an advantage over the other methods, in that it handles lacking data well.

5.3 Frankes function

For the Franke data, NN was the clear winner, with gradient descent methods following closely.
SVM and XGBoost both performed rather poorly on this data set. Neither method is very popular for regression, and from this short study it seems they are both sensitive to noise. In fact, fitting to noiseless Franke function data yielded R^2 scores above 0.99.
As NN is essentially a black box, it is difficult to say exactly why it performed as well as it did.

References

[1] P. Skalski, Gentle Dive into Math Behind Convolutional Neural Networks, https://towardsdatascience.com/gentle-dive-into-math-behind-convolutional-neural-networks-79a07dd44cf9, **2019**.

[2] Katacoda, TensorFlow: MNIST for beginners, https://www.katacoda.com/basiafusinska/courses/tensorflow-getting-started/tensorflow-mnist-beginner, **2019**.

[3] M. Hjorth-Jensen, Project 3, Department of Physics, University of Oslo, Norway, **2019**.

[4] J. Wu, *National Key Lab for Novel Software Technology. Nanjing University. China* **2017**, *5*, 23.

[5] R. Prabhu, Understanding of Convolutional Neural Network (CNN) — Deep Learning, https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148, **2019**.

[6] A. Gron, **2017**.

[7] E. W. Weisstein, Hyperplane. From MathWorld - A Wolfram Web Recource, **2019**, http://mathworld.wolfram.com/Hyperplane.html (visited on 11/28/2019).

[8] T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning, 12th printing, ser, **2017**.

[9] K. P. Murphy, *Machine learning: a probabilistic perspective*, MIT press, **2012**.

[10] S. Raschka, STAT 479: Machine Learning Lecture Notes, **2018**, https://sebastianraschka.com/pdf/lecture-notes/stat479fs18/06_trees_notes.pdf (visited on 12/01/2019).

[11] J. Quinlan, *Machine Learning* **1985**, *1*, 81.

[12] T. Chen, C. Guestrin in Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, San Francisco, California, USA, **2016**, pp. 785–794.

[13] N. Gangwar, Applying-convolutional-neural-network-on-mnist-dataset, https://www.geeksforgeeks.org/applying-convolutional-neural-network-on-mnist-dataset/, **2019**.

[14] T. G. Dietterich, *Mach. Learn.* **2000**, *40*, 139–157.

[15] S. Ahmadi, J. Kismul, S. Loevbrekke, Project 2, **2019**, https://github.com/jkismul/MachineLearning/blob/master/Project2/Project_2.pdf (visited on 12/01/2019).