FYS3150 Machine Learning - Project 3
Classifying Pulsar Candidates With Machine Learning Methods

Nils Johannes Mikkelsen
(Dated: December 17, 2018)

The project deals with the classification of pulsar candidates using statistical measures of the pulsar candidates' integrated pulse profile and the dispersion measure vs. signal-to-noise ratio curve. The classification is performed using both neural networks and support vector machines. Both methods are shown to produce excellent results with a highest accuracy score of 98%, provided reasonable hyperparameters.

All material for project 3 may be found at:
https://github.com/njmikkelsen/machinelearning2018/tree/master/Project3

I. INTRODUCTION

In the field of radio astronomy, pulsars are among the most studied phenomena in nature. But despite astronomers' long history with puslars, little is actually known with certainty. However, much of the uncertainty likely boils down to the difficulty of confirming pulsar observations. While pulsars radiate unmistakable radio signals, they are often lost in the sheer number of radio signals observed by radio telescopes every day. Furthermore, due to the uniqueness of pulsar radio signals, classifying pulsars in large data sets of radio observations have historically been very difficult as human supervision has been a necessity. However, recent advancements in machine learning and data mining has made this task much simpler by introducing incredibly fast, in comparison to humans that is, classification methods. The aim of these machine learning methods is not to remove the role of the astronomer, but rather reduce large amounts of data to smaller, manageable data sets. If done effectively, such methods can be used a primary tool for extracting likely pulsar-candidates in order to survey large regions of the night sky in a fraction of the time.

II. THEORY

A. Radio Analysis of Pulsars

The following theory is mostly based on F.G. Smith's 1977 *Pulsars* [1], with additional information taken from "Chapter 6: Pulsars" of Jim Condon's online book *Essential Radio Astronomy* [2].

1. The basics of pulsars

Pulsars are heavily magnetised rotating astronomical objects that emit regular pulses of electromagnetic (EM) radiation. Initially discovered in 1967 by Antony Hewish and Jocelyn Bell Burnell, it is generally believed by most modern astronomers that pulsars are neutron stars, the product of some supernovae. There are several mechanims that result in the electromagnetic radiation (although still not entirely understood), astronomers have thus devised three major categories of pulsars according to the source of radiation: rotation-powered pulsars, accretion-powered pulsars and magnetars. Rotation-powered pulsars use the loss of rotational energy, accretion-powered pulsars exploit the gravitational potential energy of accreted matter, while magnetars are believed to power its radiation pulses by the decay of an extremely strong magnetic field ($\sim 10^{10}$ T). Other than regular pulses of radiation, the properties of the radiation from the different categories are mostly different. Rotation-powered pulsars radiate mostly in the Radio spectrum, while the accretion-powered pulsars radiate mostly in the X-Ray spectrum. Magnetars have been observed to radiate both in the X-Ray and gamma-ray spectra.

The data studied in this project stem from radio-telescope observations, thus only rotation-powered pulsars are considered here.

2. ISM dispersion

As the name implies, radio pulsars are particular pulsars that radiate in the radio spectrum. When astronomers first began studying radio pulsars they noticed that radiation from the same pulsar with different frequencies arrived on Earth at different times. It was quickly pointed out that this EM dispersion coincided with the expected dispersion of EM waves travelling through the cold plasma of the interstellar medium (ISM). Travelling through an ionised gas, the group and phase velocities of radio waves, v_g and v_p respectively, are related by

$$v_g v_p = c^2 \tag{1}$$

where c is the speed of light in a vacuum. The fractive index of the cold plasma is

$$\mu = \sqrt{1 - \left(\frac{\nu_\mathrm{p}}{\nu}\right)^2} \tag{2}$$

where ν is the radio wave frequency and $\nu_{\rm p}$ is the *plasma frequency* given by:

$$\nu_{\rm p} = \sqrt{\frac{e^2 n_e}{\pi m_e}} \tag{3}$$

Here, n_e and m_e are the electron number density and m_e is the electron mass. Now, radio waves with $\nu \leq \nu_{\rm p}$ are unable to propegate through the ISM. On the other hand, waves with $\nu > \nu_{\rm p}$ do propegate through the ISM with a group velocity of

$$v_g = \mu c \cong \left(1 - \frac{\nu_{\rm p}^2}{2\nu^2}\right) c \tag{4}$$

Hence, the travel time T spent by a radio pulse from a pulsar a distance L away is

$$\begin{aligned} T &= \int_0^L \frac{1}{v_g}\,{\rm d}l = \frac{1}{c}\int_0^L \left(1 + \frac{\nu_{\rm p}^2}{2\nu}\right){\rm d}l \\ &= \frac{d}{c} + \left(\frac{e^2}{2\pi m_e c}\right)\nu^{-2}\int_0^L n_e\,{\rm d}l \end{aligned} \tag{5}$$

The integral in (5) is commonly called the *Dispersion Measure*, and is commonly given in units of pc cm^{-3}:

$$\mathrm{DM} = \int_0^L n_e\,{\rm d}l \tag{6}$$

The Dispersion Measure is not a trivial quantity to estimate because it depends on the constituents and their arrangements of the ISM, which in general is not uniform. Despite accurate theoretical predictions of DM, it continues to be an essential variable parameter in radio astronomy analysis, more or this below. In astronomy-convenient units, the dispersion delay $t = T - d/c$ due to the ISM is

$$t \cong 4.149 \cdot 10^3 \nu^{-2}\,\mathrm{DM\ seconds} \tag{7}$$

where ν in MHz and DM in pc cm^{-3}.

3. *The integrated pulse profile*

Due to difficulties with time-resolution, limited bandwidth, etc., studying a single radio pulse is considerably more difficult than to study the "average pulse" across several frequencies. Taking into account the time delay due to dispersion in the ISM (equation (7)), the radio pulses from radio pulsars may be *folded* to yield an "*integrated pulse profile*" (IPP). Seeing that the DM is essential in the computation of the time delay, astronomers often study the IPP as a function of DM. An example of the folding process is shown in figure 1, note that the delay folds across several pulse periods. Moreover, while the time-signal is fairly chaotic, the integrated pulse profile is very focused.

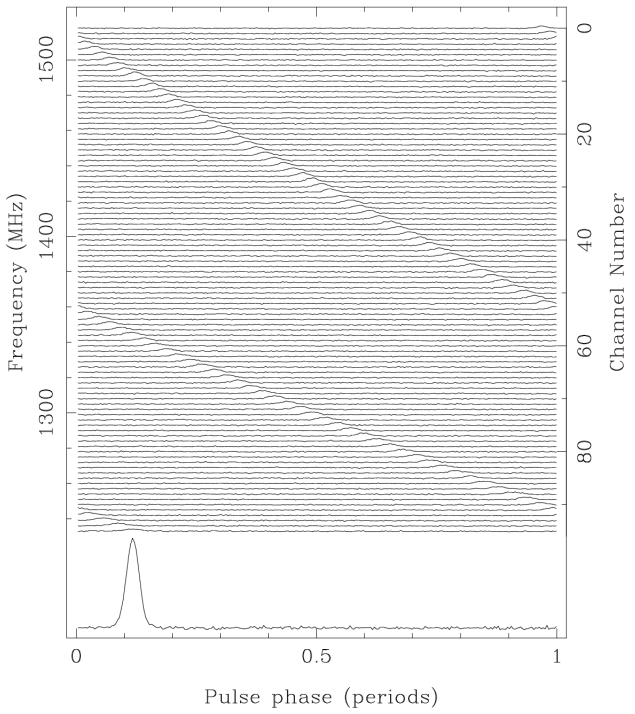


Figure 1: The pulse profile of a radio wave experiences a time delay due to dispersion in the ISM. The integrated pulse profile is a folded profile that finds an average pulse profile by taking into account the time delay. Source: [3]

While all pulsars share the property of a stable pulse profile, it turns out that each pulsar exhibits a particular profile that is more or less unique. This unique profile, often nicknamed the "fingerprint" of the pulsar, can have several peaks and be less or more spread out. Nonetheless, some statistical properties such as a single peak, a double peak, various minor peaks, etc., do occur quite frequently.

4. *Signal-to-noise ratio*

An important quantity in observational astronomy is the Signal-to-Noise Ratio (SNR), which in its simplest form is defined as the ratio of the power of the signal to the power of the noise:

$$\mathrm{SNR} = \frac{P_{\rm signal}}{P_{\rm noise}} \tag{8}$$

SNR is a measure of the quality of observation, or "amount of meaningful information" in an observation. Astronomers actually use the SNR to study the DM parameter space, as selecting the optimal DM is essential for conducting a proper analysis of the IPP. Enter the the DM-SNR curve: the SNR score of an integrated pulse

profile as a function of DM. The DM-SNR curve is usually flat for non-pulsating signals, however peaks about the optimal DM if the signal does pulsate. Consequently, the DM-SNR curve is commonly used as a preliminary test for identifying signals from pulsars in large data sets of radio signals.

B. Moments In Statistics

Given some continuous function $f : \mathbb{R} \to \mathbb{R}$, its n^{th} *moment* $\mu_n(c)$ about the real value c is defined as

$$\mu_n(c) = \int_{-\infty}^{\infty} (x-c)^n f(x)\,\mathrm{d}x \tag{9}$$

Moments are used in several areas of science and mathematics, most prominently in mechanics and probability theory. If f is a probability density (or strictly non-negative and normalizable),[1] there exists two major notation conventions:

$$\mu_n(c) = E\big[(X-c)^n\big] \quad \text{and} \quad \mu_n(c) = \langle (x-c)^n \rangle$$

The former is favoured by statisticians, while the latter is favoured by physicists, this project will employ the physicists' convention.

The moments most often studied in probability theory are the so-called *central* moments μ_n:

$$\mu_n = \mu_n(\mu) = \langle (x-\mu)^n \rangle = \int_{-\infty}^{\infty} (x-\mu)^n f(x)\,\mathrm{d}x \tag{10}$$

where $\mu = \langle x \rangle$ is the mean value of the distribution and $\mu_n = \mu_n(\mu)$ is introduced to simplify notation. The 0^{th} and 1^{st} central moments are 1 and 0 respectively as $f(x)$ is normalized and $(x-\mu)f(x)$ is centered about μ. The 2^{nd} central moment is the variance:

$$\mu_2 = \big\langle (x-\mu)^2 \big\rangle = \sigma^2 \tag{11}$$

where σ is the standard deviation.

Furthermore, the central moments are used to define the *normalized* moments $\tilde{\mu}_n$:

$$\tilde{\mu}_n = \frac{\mu_n}{\sigma^n} = \frac{\langle (x-\mu)^n \rangle}{\langle (x-\mu)^2 \rangle^{n/2}} \tag{12}$$

Normalized moments are dimensionless quantities that describe the underlying distribution independently of the scale of the data. The 3^{rd} normalized moment is known as the *skewness* of a distribution and quantifies the extent to which a distribution is *skewed* to "left or right" of the mean. Positive skewness implies the distribution is skewed towards x values greater than μ and vice versa. The 4^{th} normalized moment is called *kurtosis* and is a measure of how quickly the tails of a distribution tends to zero, i.e. a measure of "*peakness*". Kurtosis is usually reported as *excess kurtosis*, which is the kurtosis of a distribution relative to the normal distribution (whose kurotsis is 3). Hence the definition:

$$\text{Excess Kurtosis} \;=\; \text{Kurtosis} - 3 \tag{13}$$

C. Machine Learning - Artificial Neural Networks

Artificial Nerual Networks (ANN or simply NN) is a Supervised Machine Learning (SML) technique that is inspired by biological neural networks in animal brains. Rather than a specific technique or algorithm, ANN is an umbrella term for several NN structures and algorithms whose complexity, computational difficulty, etc., varies greatly between different techniques.

1. *The Multi-Layered Perceptron*

This project will employ the so-called Multi-Layered Perceptron (MLP) model, which is a particular ANN that is based around the idea of organizing the network's neurons, or nodes, in layers. A node is loosely based on the concept of biological neuron, which is capable of recieving an electrical signal, process it, and transmit a new signal to other neurons. In the MLP model the neurons are arranged in L consecutive layers with N_l number of nodes per layer (unqeual number of nodes per layer is alloweed). There are three types of layers: input, hidden and output layers. The input and output layers manage the network's inputs and outputs, while the hidden layers serve as the network's data-processing backbone. Particular to the MLP, every node in each hidden layer is connected to every node in the previous and the next layers, but nodes within the same layer are *not* connected. The MLP structure is illustrated in figure 2, note the inclusion of "bias": Bias is an additional property of all layers except the input layer whose purpose is essentially to center the data about 0. Conceptually, one may think of bias as the role of the intercept in Linear Regression analysis.

So far the network has only been described using fluid terms such as "transfer of information" and "node connections", this will be expanded upon now. The number of nodes in the input layer N_1 must exactly match the dimensionality of the input data, thus there is a one-to-one correspondence between the nodes and the indices of the input data (usually a vector quantity). Similarly, the number of nodes in output layer N_L must exactly match the output data of the network, again with a one-to-one correspondence between nodes and data indices.

[1] A probability density $f : D \to \mathbb{R}$ satifies

$$0 \le f(x),\ \forall x \in D \quad \text{and} \quad \int_D f(x)\,\mathrm{d}x = 1.$$

Note that D is usually the entire real number line.

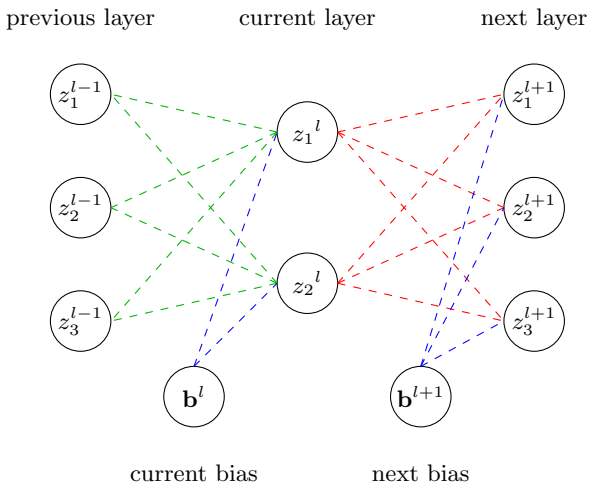


Figure 2: An illustration of the structure of an MLP ANN: Each node is connected to every node in the previous layer and the next layer. The inputs to the current layer consists of the data from the previous layer (green connections) and the current layer's bias (left blue connections). The outputs of the current layer (red connections) together with the next layer's bias (right blue connections) become the inputs of the next layer, and so on.

It follows that in practice it is really only the hidden layers that may have a variable number of nodes N_l, $l \in \{2, \dots, L-1\}$. The network processes input data layer-by-layer, meaning each hidden layer processes the data in sequence. Data from the input layer is sent to the first hidden layer, processed, then transfered to the next layer, and so on.

Mathematically speaking, the connection between two nodes i in layer l and j in layer $l+1$ is a scaling operation. The connection's scaling factor is usually called the weight and is commonly denoted by w_{ij}^{l+1}, note the convention of labeling a connection weight between layers l and $l+1$ by $l+1$. Each node is connected to every node in the previous layer, thus the purpose of weighting the data is to regulate the importance of each node in the previous layer. However, note that unless some other operation is conducted on the data the network's overall operation on the input data is only linear (meaning methods such as Linear Regression are more appropriate). This is circumvented by the addition of a non-linear operation on the every node's input, commonly called the node's *activation* function. Common activation functions include:

$$\text{logistic}(x) = \frac{1}{1+e^{-x}} \tag{14a}$$

$$\text{sigmoid}(x) = \tanh(x) \tag{14b}$$

$$\text{rectifier}(x) = \max(0, x) \tag{14c}$$

Other activation functions are also used. Technically, a network may have a different activation for each node, but this is rarely done in practice. On the other hand, layer-wide activation functions are often implemented, so much so that the following theory will base the mathematics on the presumption of layer-wide activation.

The data in node i in layer l is denoted by y_i^l. The connection weight between node i in layer l and node j in layer $l+1$ is denoted by w_{ij}^{l+1}. The accumulated data input from all nodes in layer l received by node j in layer $l+1$, adjusted for bias b_j, is given by

$$z_j^{l+1} = \sum_{i=1}^{N_l} w_{ij}^{l+1} y_i^l + b_i^l \tag{15}$$

The input data received by the node is $f_{l+1}(z_j^{l+1})$, where f_{l+1} is the activation function of layer $l+1$. Hence, the data in node j in layer $l+1$ is given by

$$y_j^{l+1} = f_{l+1}(z_j^{l+1}) = f_{l+1}\left(\sum_{i=1}^{N_l} w_{ij}^{l+1} y_i^l + b_j^{l+1}\right) \tag{16}$$

Arranging the node data in vectors $\mathbf{y}^l$, the node inputs in vectors $\mathbf{z}^{l+1}$, the connection weights in a weight matrix $\hat{W}^{l+1}$ and the bias in bias vectors $\mathbf{b}^{l+1}$, then (16) can be reformulated as a layer-wide (matrix) operation:

$$\mathbf{y}^{l+1} = f_{l+1}(\mathbf{z}^{l+1}) = f_{l+1}\left(\hat{W}^{l+1}\mathbf{z}^l + \mathbf{b}^{l+1}\right) \tag{17}$$

To conclude this section: The answer to the question *"What is a Multi-Layered Perceptron Neural Network?"* is the following function:

$$\mathbf{y} = f_L\left(\hat{W}^L f_{L-1}\left(\cdots\left(\hat{W}^1\mathbf{x} + \mathbf{b}^1\right)\cdots\right) + \mathbf{b}^L\right) \tag{18}$$

where $\mathbf{x} = \mathbf{y}^1$ and $\mathbf{y} = \mathbf{y}^L$ are the network's inputs and outputs.

2. *Training an MLP*

The basis of the MLP, and many similar network structures, is the so-called *back-propagation* algorithm (BPA). The MLP is an example of a *supervised* machine learning technique, meaning the desired outputs are known. In case of the MLP, it is the BPA that introduces supervision.

At its core, the BPA is essentially a variant of gradient descent techniques. The basic problem of gradient descent methods is to find the input $\mathbf{x}_{\min} \in D$ that minimises some function $F : D \to \mathbb{R}$. The algorithm is based on producing sucessive approximations of $\mathbf{x}_{\min}$, each closer to the true minium. Seeing that the goal is to minimise F, the direction of each step is chosen opposite of the gradient of F. The fundamental algorithm is summarised below:

Algorithm 1 Gradient Descent

1: Select a trial input $\mathbf{x}_0$.
2: **for** $i = 1, \ldots, N$:
3: Compute: $\mathbf{x}_i = \mathbf{x}_{i-1} - \gamma_i \dfrac{\partial F}{\partial \mathbf{x}}\Big|_{\mathbf{x}_{i-1}}$
4: **end for**

Note that the derivative is not scalar. The step length γ_n may be variable or constant depending on the implementation, although proper convergence is impossible with too large steps.

The F function to minimise in the BPA is the so-called *cost* function, denoted C. Similar to activation functions, different cost functions are used in different problems. The cost function most used in classification analysis is the so-called cross-entropy:

$$C(\mathbf{y}|\mathbf{t}) = -\Big[\mathbf{t}^T \ln(\mathbf{y}) + (1-\mathbf{t})^T \ln(1-\mathbf{y})\Big] \tag{19}$$

where $\mathbf{y}$ is a vector of the network's outputs and $\mathbf{t}$ is a vector of the targets, i.e. the desired network output.[2] Note that $C(\mathbf{y}|\mathbf{t})$ is univariate: it is a function of outputs $\mathbf{y}$ provided the targets $\mathbf{t}$ (that is, $\mathbf{t}$ is not a variable quantity). The derivative of $C(\mathbf{y}|\mathbf{t})$ is

$$\frac{\partial C}{\partial \mathbf{y}} = (\mathbf{y}-\mathbf{t}) \oslash \Big(\mathbf{y} \circ (1-\mathbf{y})\Big) \tag{20}$$

where $\oslash$ and $\circ$ denotes the Hadamard division and product (element-wise operations). Now, if this was a problem for gradient descent methods, then $\mathbf{y}$ would be updated directly using Algorithm 1 and equation (20). However, $\mathbf{y}$ is produced by the network according to equation (18), which is dependent on the weights and biases of each layer. Hence, in order for the BPA to improve the network it must study $C(\mathbf{y}|\mathbf{t})$ with respect to all weights and biases.

Although seemingly difficult, the problem is actually somewhat straight-forward, it all comes down to clever use of the chain rule and limiting the number of necessary computations. Observe that the derivative of $C(\mathbf{y}|\mathbf{t})$ with respect to layer l can be written as

$$\frac{\partial C}{\partial \hat{W}^l} = \frac{\partial C}{\partial \mathbf{y}^l}\frac{\partial \mathbf{y}^l}{\partial \mathbf{z}^l}\frac{\partial \mathbf{z}^l}{\partial \hat{W}^l} \tag{21a}$$

$$\frac{\partial C}{\partial \mathbf{b}^l} = \frac{\partial C}{\partial \mathbf{y}^l}\frac{\partial \mathbf{y}^l}{\partial \mathbf{z}^l}\frac{\partial \mathbf{z}^l}{\partial \mathbf{b}^l} \tag{21b}$$

Now define the *signal error* of layer l as

$$\hat{\delta}^l \equiv \frac{\partial C}{\partial \mathbf{z}^l} = \frac{\partial C}{\partial \mathbf{y}^l}\frac{\partial \mathbf{y}^l}{\partial \mathbf{z}^l} = \frac{\partial C}{\partial \mathbf{y}^l} \circ f_l'(\mathbf{z}^l) \tag{22}$$

where $f_l'(\mathbf{z}^l)$ is the derivative of the activation function of layer l. Clearly then equations (21) may be written as

$$\frac{\partial C}{\partial \hat{W}^l} = \hat{\delta}^l \mathbf{y}^{l-1} \tag{23a}$$

$$\frac{\partial C}{\partial \mathbf{b}^l} = \hat{\delta}^l \tag{23b}$$

where $\mathbf{z}^l = \hat{W}^l \mathbf{y}^{l-1} + \mathbf{b}^l$ was used to evalute $\partial \mathbf{z}^l / \partial \hat{W}^l$ and $\partial \mathbf{z}^l / \partial \mathbf{b}^l$. The next step is to relate the signal error of layer l to $l+1$, the idea to rewrite $\partial C / \partial \mathbf{y}^l$ in (22) using a reverse chain rule:[3]

$$\frac{\partial C}{\partial \mathbf{y}^l} = \frac{\partial C}{\partial \mathbf{z}^{l+1}}\frac{\partial \mathbf{z}^{l+1}}{\partial \mathbf{y}^l} = \Big[\big(\hat{\delta}^{l+1}\big)^T \hat{W}^{l+1}\Big]^T = \big(\hat{W}^{l+1}\big)^T \hat{\delta}^{l+1}$$

That is, the signal error of layer l is related to the signal error of layer $l+1$ via

$$\hat{\delta}^l = \big(\hat{W}^{l+1}\big)^T \hat{\delta}^{l+1} \circ f_l'(\mathbf{z}^l) \tag{24}$$

Equations (23), (22) and (24) are what constitutes the *four equations of the back-propagation algorithm*. The equations essentially boil down to the equivalent of step 3 in Algorithm 1 (in particular (23)). Moreover, note that the "back-propagated quantity" in the BPA is the signal error, i.e., the network's output error is propagated backwards through the network (thereby expressing the dependency of the cost function on the signal error of the different layers).

The back-propagation algorithm is composed of 3 steps: the feed forward cycle, the back-propagation and the network update. The feed forward step refers to the computation of the network's output; the input data $\mathbf{x}$ is "fed forward" to the output. The back-propagation step involves the computation of the signal errors and the final network update is the updating of the network's weights and biases using the derivatives of the cost function. The full algorithm is shown below.

Algorithm 2 The Back-Propagation Algorithm

1: Initiate the network weights and biases.
2: **for** $i = 1, \ldots, N$:
3: Feed forward input $\mathbf{x}$.
4: Compute $\hat{\delta}^L$.
5: **for** $l = L-1, \ldots, 1$:
6: Compute $\hat{\delta}^l$.
7: **end for**
8: Update the network weights and biases according to

$$\hat{W}^l_{\text{next}} = \hat{W}^l_{\text{prev}} - \gamma_i \frac{\partial C}{\partial \hat{W}^l_{\text{prev}}}$$

$$\mathbf{b}^l_{\text{next}} = \mathbf{b}^l_{\text{prev}} - \gamma_i \frac{\partial C}{\partial \mathbf{b}^l_{\text{prev}}}$$

9: **end for**

[2] The introduction of $\mathbf{t}$ in the cost function is the actual introduction of machine learning supervision in MLP ANNs. Also, note that naming $\mathbf{t}$ the "targets" refers to the goal of neural networks: namely to construct the network such that it reproduces the target outputs.

[3] The awkward double-transpose notation is used to preserve the correct dimensions of $\partial C / \partial \mathbf{y}^l$.

3. MLP Classifiers

The MLP network described up until now technically does not perform a classification, but a regression. This is not a problem however as adjusting the network to function as a classifier is not particularly difficult. The only meaningful difference is in the output layer, in particular the output layer activation function.

The most common choice of ouput layer activation for classification problems is the softmax function:

$$\text{softmax}(\mathbf{x}, i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \tag{25}$$

where $\mathbf{x}$ is a vector of N components. The reason why the softmax is so popular is that it is normalized: $\sum_i^N \text{softmax}(\mathbf{x}, i) = 1$, which is an important property with regards to probability. The idea to use the MLP network as a regression model to produce probabilities, and in turn classify the input data according to the most likely classes.

4. Improving BPA convergence

First and foremost, it is very difficult to predict how the BPA behaves with respect to a contant learning rate. Hence, a guarantee of convergence is much more likely if the learning rate is adjusted during the iteration. For the time being, no particular updating scheme seems to be universally optimal. The updating scheme used in this project is the so-called "*adaptive scheme*", which updates the learning rate if the cost function is not improved after n iterations. This project will update the learning rate if 10 consecutive iterations fail to improve the cost via:

$$\gamma_{n+1} = \frac{\gamma_n}{5} \tag{26}$$

As it stands, notice that Algorithm 2 makes no mention of the possibility of multiple data inputs x. A simple way to accomodate several data samples would be to iterate over the available samples $\mathbf{x}_i$ within the BPA loop. However, this is very error-prone as it is likely to favour the first samples (especially with the adaptive learning rate). This can be slightly improved by using a stochastic approach, meaning the samples are picked at random. Nevertheless, this would only choose "the first samples" stochastically, not fix the underlying problem. A better approach is to use the average of several samples chosen at random, which means the mathematics must be somewhat adjusted (although not much): The input vector $\mathbf{x}$ is replaced by an input matrix $\mathbf{X} = \begin{bmatrix}\mathbf{x}_1 & \cdots & \mathbf{x}_M\end{bmatrix}$, which implies that the vector quantities $\hat{\delta}^l$, $\mathbf{z}^l$ and $\mathbf{y}^l$ now become equivalent matrix quantities. Both $\hat{W}^l$ and $\mathbf{b}^l$ remain the same. Finally $\hat{\delta}^l$ is reduced from a matrix of column vectors to the average column vector before being used to compute $\partial C/\partial \hat{W}^l_{\text{prev}}$ and $\partial C/\partial \mathbf{b}^l_{\text{prev}}$. What remains now is to select the set of input vectors to use in $\mathbf{X}$. Including all vectors usually leads to too large matrices, and is also sensitive to outliers. The so-called *Stochastic Gradient Descent* (SGD) method solves this by dividing the training set (all samples) into K "mini-batches" of size $B = N/K$. The SGD update is:

$$\mathbf{x}_i = \mathbf{x}_{i-1} - \gamma_i \sum_{j=1}^{B} \frac{\partial F}{\partial \mathbf{x}}\bigg|_{\mathbf{x}=\mathbf{x}_j} \tag{27}$$

Yet another improvement of the standard gradient descent method is the so-called "*Momentum Gradient Descent*" (MGD). The idea is to include an additional "momentum" term whose purpose is to guide the descent in regions where the gradients are much larger in some dimensions than others (which is common in regions close to local minima). Because of MGDs innate relationship with mechanics, common practice is to denote the gradient by $\mathbf{v}_i$. The MGD update is then written as

$$\mathbf{x}_i = \mathbf{x}_{i-1} - \eta_i \mathbf{v}_{i-1} - \gamma_i \mathbf{v}_i \tag{28}$$

where $\eta_i \mathbf{v}_{i-1}$ is the "momentum".

D. Machine Learning - Support Vector Machines

One of the most versatile and diverse approaches to SML is Support Vector Machines (SVM). SVMs are very popular for both regression and classification problems, this project however will only perform a binary classification. The mathematics are therefore presented with a binary labelling in mind.[4]

1. The basic principles behind SVMs

The basic idea of an SVM classifiers is to divide a data set into discrete classes using a *decision boundary*, whose mathematical description is that of a *hyperplane*. Technically speaking, hyperplanes are affine subspaces that represent the direct generalisation of the concept of two-dimensional planes located in three-dimensional spaces. It happens that any decision boundary of a p-dimensional predictor space must be a $(p-1)$-dimensional hyperplane.

Hyperplanes may be parametrised via a linear equation using an intercept β_0 and a parameter vector $\hat{\beta}^T = \begin{pmatrix}\beta_1 & \cdots & \beta_p\end{pmatrix}$. Written in set notation, the definition of a hyperplane in a real p-dimensional space is

$$\text{hyperplane} = \{\mathbf{x} \mid \mathbf{x}^T \hat{\beta} + \beta_0 = 0\} \tag{29}$$

[4] SVM classification with several classes is actually just a recursive use of binary classifications. E.g. Dividing the real number line into three classes $(-\infty, 0)$, $[0, 10]$ and $(10, \infty)$ is the same as first dividing the real number line into $(-\infty, 0)$ and $[0, \infty)$, then subdividing $[0, \infty)$ into $[0, 10]$ and $(10, \infty)$.

This definition of a hyperplane is particularly useful, because it accounts for the two regions separated by the hyperplane by changing the equality in $\mathbf{x}^T\hat{\beta} + \beta_0$ to either $<$ or $>$. Hence, a natural binary classification based on (29) is to simply use the sign:[5]

$$\text{class label} = \text{sign}(\mathbf{x}^T\hat{\beta} + \beta_0) \tag{30}$$

where the binary label (i.e. ± 1) is an arbitrary choice that simplifies notation. Note that the description of decision boundaries as hyperplanes necessarily restricts the possible boundaries to linear boundaries. It turns out that this problem is easily circumvented by introducing a benefitial transformation of the predictor space. This is done in later sections.

Suppose a particular linear decision boundary is parametrised by β_0 and $\hat{\beta}$. By definition, any point $\mathbf{x}_i$ on the hyperplane satisfies $\hat{\beta}^T\mathbf{x}_i = -\beta_0$, and consequently any two points on the hyperplane $\mathbf{x}_1$ and $\mathbf{x}_2$ must therefore satisfy $\hat{\beta}^T(\mathbf{x}_2 - \mathbf{x}_1) = 0$. It follows that $\hat{\beta}/||\hat{\beta}||$ is a unit vector normal to the surface of the hyperplane. Now consider any point in the predictor space $\mathbf{x}$: because the hyperplane is infinite in reach, there must exist some point on the hyperplane $\mathbf{x}_0$ such that the distance from $\mathbf{x}$ to $\mathbf{x}_0$ is minimal. Because $\hat{\beta}/||\hat{\beta}||$ is necessarily the unit vector pointing along the line defined by $\mathbf{x}$ and $\mathbf{x}_0$,[6] the signed distance from $\mathbf{x}_0$ to $\mathbf{x}$ must satisfy

$$\frac{\hat{\beta}^T}{||\hat{\beta}||}(\mathbf{x} - \mathbf{x}_0) = \frac{1}{||\hat{\beta}||}(\hat{\beta}^T\mathbf{x} + \beta_0) \tag{31}$$

That is, the signed distance from any data point $\mathbf{x}$ to $\mathbf{x}_0$ on the decision boundary is proportional to $\hat{\beta}^T\mathbf{x} + \beta_0$. The smallest distance between any data point and the decision boundary is called the decision boundary's *margin* M. The mission statement of SVMs is to find a hyperplane (i.e. determine $\hat{\beta}$ and β_0) that maximises the margin. Using the class labelling

$$y_i = \begin{cases} +1, & \mathbf{x}_i^T\hat{\beta} + \beta_0 \geq 0 \\ -1, & \mathbf{x}_i^T\hat{\beta} + \beta_0 < 0 \end{cases} \tag{32}$$

the SVM mission statement may be expressed as an optimization problem:

$$\frac{y_i}{||\hat{\beta}||}(\mathbf{x}_i^T\hat{\beta} + \beta_0) \geq M, \quad i \in \{1, \ldots, N\} \tag{33}$$

or alternatively if $||\hat{\beta}|| = 1/M$:

$$\min_{\hat{\beta},\beta_0} \frac{1}{2}||\hat{\beta}|| \quad \text{subject to} \quad y_i(\mathbf{x}_i^T\hat{\beta} + \beta_0) \geq 1, \quad \forall i \tag{34}$$

This is an example of a convex optimization problem, and is appropriately studied using the method of Lagrange multipliers.

The Lagrangian primal to be minimized is

$$\mathcal{L}(\hat{\beta}, \beta_0, \hat{\lambda}) = \frac{1}{2}||\hat{\beta}||^2 - \sum_{i=1}^{N} \lambda_i\big(y_i(\mathbf{x}_i^T\hat{\beta} + \beta_0) - 1\big) \tag{35}$$

Its derivatives are

$$\frac{\partial\mathcal{L}}{\partial\hat{\beta}} = \hat{\beta} - \sum_{i=1}^{N} \lambda_i y_i \mathbf{x}_i \tag{36a}$$

$$\frac{\partial\mathcal{L}}{\partial\beta_0} = -\sum_{i=1}^{N} \lambda_i y_i \tag{36b}$$

Setting $\partial\mathcal{L}/\partial\hat{\beta} = 0$ and $\partial\mathcal{L}/\partial\beta_0 = 0$, then inserting the results back into (35) one finds the Wolfe dual (a maximization problem):

$$\mathcal{L}(\hat{\lambda}) = \sum_{i=1}^{N} \lambda_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \lambda_i\lambda_j y_i y_j \mathbf{x}_i^T\mathbf{x}_j \tag{37a}$$

$$\text{subject to } \lambda_i \geq 0 \quad \text{and} \quad \sum_{i=1}^{N} \lambda_i y_i = 0 \tag{37b}$$

In addition to the constraints of the Wolfe dual, the solution must also satisfy the so-called KKT (Karush-Kuhn-Tucker) conditions:

$$\lambda_i\big(y_i(\mathbf{x}_i^T\hat{\beta} + \beta_0) - 1\big) = 0, \quad \forall i \tag{38}$$

It follows from (38) that either of two events is possible: If $\lambda_i > 0$, then it must be so that $y_i(\mathbf{x}^T\hat{\beta} + \beta_0) = 1$, in which case $\mathbf{x}_i$ is on the boundary. Else, if $y_i(\mathbf{x}^T\hat{\beta} + \beta_0) > 1$, then $\lambda_i = 0$ and $\mathbf{x}_i$ is not on the boundary. Due to the particular role of the $\mathbf{x}_i$ vectors in the Wolfe dual, they are often called the *support vectors* of the hyperplane (hence the naming of SVM). Computationally the Wolfe dual optimization problem is actually much simpler than the Lagrangian primal optimization problem. Efficient algorithms have been developed and are implemented in most standard optimization software.

Having found the Lagrange multipliers λ_i, the decision boundary hyperplane is found via

$$\hat{\beta} = \sum_{i=1}^{N} \lambda_i y_i \mathbf{x}_i \quad \text{and} \quad \beta_0 = \frac{1}{y_i} - \hat{\beta}^T\mathbf{x}_i, \ \forall i$$

The final SVM classifier is given by equation (30).

---

[5] This classification rule actually does not include points on the decision boundary, thus common practice is to let these points belong to the positive sign class by default.

[6] This follows from the fact that the dimensionality of the hyperplane is one less than the dimensionality of the predictor space. That is, the only degree of freedom that separates $\mathbf{x}$ from $\mathbf{x}_0$ is the direction of the unit vector normal to the hyperplane surface, which is $\hat{\beta}/||\hat{\beta}||$.

2. Soft classifiers

The classifier described in the previous section is what is known as a *hard* linear classifier, i.e. a linear classifier that does not allow any cross-over between the two classes. Consequently, hard linear classifiers assume that at least one hyperplane that successfully separates the data set does exist, but this is easily not the case for all data sets. For example, any presence of significant noise could easily produce some data points that "fall on the wrong side" of the decision boundary. Because of its shortcomings, most modern SVM implementations does not use a hard classifier, but instead a so-called *soft* classifier. Soft classifiers are similar to hard classifiers in their structure and purpose, however allows some *slack* with regards to the data set.

Define the slack variables $\hat{\xi} = \begin{pmatrix}\xi_1 & \cdots & \xi_N\end{pmatrix}$, where $\xi_i \geq 0$, and modify equation (33) to "allow some slack" on the form:

$$\frac{y_i}{||\hat{\beta}||}(\mathbf{x}_i^T\hat{\beta} + \beta_0) \geq M(1-\xi_i) \tag{39}$$

or alternatively if $||\hat{\beta}|| = 1/M$:

$$\min_{\hat{\beta},\beta_0} \frac{1}{2}||\hat{\beta}|| \quad \text{subject to} \quad y_i(\mathbf{x}_i^T\hat{\beta}+\beta_0) \geq 1-\xi_i,\ \forall i \tag{40}$$

The total *violation* of (33) is now given by $\sum_{i=1}^N \xi_i$, hence bounding the sum from above by some constant C allows for ease of control of the total slack. Equation (40) can be re-expressed in the equivalent form:

$$\min_{\hat{\beta},\beta_0}\left\{\frac{1}{2}||\hat{\beta}||^2 + C\sum_{i=1}^N \xi_i\right\} \tag{41a}$$

$$\text{subject to } \xi_i \geq 0,\ y_i(\mathbf{x}_i^T\hat{\beta}+\beta_0) \geq 1-\xi_i,\ \forall i \tag{41b}$$

The Lagrangian primal to minimize is now:

$$\mathcal{L}(\hat{\beta},\beta_0,\hat{\lambda},\hat{\gamma}) = \frac{1}{2}||\hat{\beta}||^2 + C\sum_{i=1}^N \xi_i - \sum_{i=1}^N \gamma_i\xi_i s \tag{42a}$$

$$-\sum_{i=1}^N \lambda_i\big(y_i(\mathbf{x}_i^T\hat{\beta}+\beta_0) - (1-\xi_i)\big) \tag{42b}$$

From here the derivation follows the same outline as the non-slacked problem, the resulting Wolfe dual to maximize is given by (37a) with the constraints

$$\max_{\hat{\lambda}} \mathcal{L} \text{ subject to } 0 \leq \lambda_i \leq C \quad \text{and} \quad \sum_{i=1}^N \lambda_i y_i = 0 \tag{43}$$

The corresponding KKT conditions are

$$\lambda_i\big(y_i(\mathbf{x}_i^T\hat{\beta}+\beta_0) - (1-\xi_i)\big) = 0 \tag{44a}$$

$$\gamma_i\xi_i = 0 \tag{44b}$$

$$y_i(\mathbf{x}_i^T\hat{\beta}+\beta_0) - (1-\xi_i) \geq 0 \tag{44c}$$

for all i. Resorting back to the classifier is identical to the non-slacked case.

Table I: Commonly used classes of kernels in SVM classification.

Class of Kernels	$K(\mathbf{x},\mathbf{y})$
Linear functions	$\mathbf{x}^T\mathbf{y}$
n^{th} degree polynomials	$(\mathbf{x}^T\mathbf{y}+r)^n$
Radial Gaussians	$e^{-\gamma\|\|\mathbf{x}-\mathbf{y}\|\|^2}$
Sigmoids	$\tanh\big(\mathbf{x}^T\mathbf{y}+r\big)$

3. Non-linear classifiers

So far, only linear classifiers have been discussed. It turns out that a simple "kernel trick" can be exploited to produce non-linear classifiers. Consider the following basis transformation of the predictor space $\mathbf{x}$ to $h(\mathbf{x})$:

$$h(\mathbf{x}) = \begin{pmatrix}h_1(\mathbf{x}) & \cdots & h_H(\mathbf{x})\end{pmatrix}^T \tag{45}$$

where H is a positive integer (which can be larger than N). The so-called kernel of $h(\mathbf{x})$ is given by

$$K(\mathbf{x}_i,\mathbf{x}_j) = h(\mathbf{x}_i)^T h(\mathbf{x}_j) \tag{46}$$

Proceding through the linear classification as outlined in previous sections one arrives on the Wolfe dual

$$\mathcal{L}(\hat{\lambda}) = \sum_{i=1}^N \lambda_i - \frac{1}{2}\sum_{i=1}^N\sum_{j=1}^N \lambda_i\lambda_j y_i y_j K(\mathbf{x}_i,\mathbf{x}_j) \tag{47}$$

Note that (47) is not expressed in terms of the basis expansion $h(\mathbf{x})$, but instead of the kernel function $K(\mathbf{x}_i,\mathbf{x}_j)$. This is implies that a non-linear classification requires only knowledge of the desired kernel, and not the underlying basis transformation. Common kernels in the SVM literature are shown in table I.

To revert back to the original "hyperplane" (which is now possibly a curved surface) one would necessarily require an expression for the basis expansion $h(\mathbf{x})$. However, this can be avoided by introducing the function

$$f(\mathbf{x}) = h(\mathbf{x})^T\hat{\beta} + \beta_0 = \sum_{i=1}^N \lambda_i y_i K(\mathbf{x},\mathbf{x}_i) + \beta_0 \tag{48}$$

Hence, instead of (30), the binary classification of new data points is given by

$$\text{class label} = \text{sign}\big(f(\mathbf{x})\big) \tag{49}$$

III. METHOD

The data set used in this project can be downloaded from the following Kaggle post:

https://www.kaggle.com/pavanraj159/predicting-a-pulsar-star

The data set contains just under 18,000 samples of pulsar condidates from the High Time Resolution Universe Survey, all post-verified as either a known pulsar or a known non-pulsar. The data set originates from the 2012 article [4] by S.D. Bathes et. al. in which an MLP classifier was successfully used to *"blindly detect 85% of pulsars"*.

A. Preparation

The data set contains of 8 real-valued predictors and 1 target label. The predictors include the mean value, standard deviation, skewness and excess kurtosis of the pulsar candidates' IPP and DM-SNR curve. The target label is a binary label ± 1 that identifies the candidate as a pulsar (+1) or a non-pulsar (-1). A histogram for each predictor has been included in appendix A, the pulsars and non-pulsars have been separated to highlight the difference.

Despite some minor overlap in the predictor space, there are clear differences between the pulsars and non-pulsars, and consequently an appropriate neural network or SVM should not have much trouble separating the two. With this in mind, this project deals not with *whether* the two classes are separable, in fact Bathes et. al. proved as much in their original article, but instead to what extent are neural networks and SVMs applicable.

The machine learning in this project is implemented using `scikit-learn` [5], in particular the classes `neural_network.MLPClassifier` and `svm.SVC`, in addition to the functionality provided by `train_test_split` and `StandardScaler`. The train-test-split function randomly divides a data set into a training set and test set. The scaler transforms the predictors space by standardizing the predictors, i.e. remove the mean and scale the variance:

$$Z = \frac{X - \langle x \rangle}{\sigma_X} \tag{50}$$

The neural networks and SVMs are compared using a so-called accuracy score. This score is a measure of the average correct identification of a pulsar or non-pulsar:

$$\text{accuracy} = \frac{1}{N}\sum_{i=1}^{N} I_i, \quad I_i = \begin{cases} 1, & \text{correct label} \\ 0, & \text{incorrect label} \end{cases} \tag{51}$$

where i traverses the test set. The accuracy score is number between 0 and 1.

B. NN Hyperparameter Analysis

The MLP network is very robust, converging on a minimum for almost all reasonable choices of hyperparameters. Ideally one would be able to vary all hyperparameters simultaneously, but this is simply not feasible on a standard computer, and in fact, probably not necessary. Instead certain "types" of hyperparameters, e.g. "learning parameters" and "structure parameters", will be studied separately.

There are countless possible variations of even the simplest of networks. Therefore, the following restrictions are introduced in order to reduce the number of combinations of paramaters:

1. All layers in a network have the same number of nodes.
2. All layers in a network use the same activation function.
3. With the exception of section III B 3, the pulsar data set is divided 50-50 into training and test sets.

Having trained a network, the performance of the network is measured according to three metrics: its accuracy score, the number of required training epochs and the time spent training. The accuracy score is interesting for obvious reasons, the two latter metrics are interesting because they reveal the network's training efficiency.

1. *Network complexity*

The first stage of the analysis takes a look at whether simple or complex networks work best with the pulsar data set. Recall that L is the number of layers and N is the number of nodes per layer. To span a large range of complexity levels, 900 networks structures are considered: The networks use 30 linearly spaced L values between 1 and 50, and 30 linearly spaced N values between 1 and 100.[7] Each network is trained using logistic, sigmoid and rectifier activation. Hence, a total of $3 \times 30 \times 30 = 2700$ unique networks are trained. The network designs used in the following sections will be largely based on the results of this complexity analysis.

2. *Network learning scheme*

Having looked at the network designs, this analysis will study the network's learning phase. The hyperparameters of interest are the initial learning rate γ_0 and the strength of the learning momentum η. Note that equation (28) allows for a varying momentum strength, but this will not be explored here. This analysis will look at 17×15 networks using 17 logarithmically spaced γ_0 values between 10^{-4} and 10^{0}, and 15 linearly spaced η values between 0.01 and 0.99. Each network is trained using a logistic, sigmoid and rectifier activation, making a total number of 765 unique learning schemes per network structure.

[7] The "linearly spaced" L and N values are created using `np.linspace(...).astype(int)`. The type switch from `float` to `int` may slightly interfer with the linear spacing.

3. Overfitting the training data

The previous two analyses have studied hyperparameters concerning the network itself, this analysis will instead look at whether the networks are sensitive to different divisions between the training and test sets. Each network is trained against 100 training sets with training set sizes between 10% and 90%.[8]

C. SVM Hyperparameter Analysis

While much harder to train than neural networks, SVMs are very flexible. Unfortunately, this flexibility comes at a price of large hyperparameter spaces, which in general are very difficult to fully explore in depth. Like with the neural networks, hyperparameters will instead be studied in groups of similar types of parameters.

This project will use the following assumptions on the behaviour of the SVMs:

1. While different kernels may behave differently with respect to the same violations strengths, all kernels are assumed to be similarly with respect to the concept of violations. In particular, different kernels may disagree what value constitutes a "large violation", but necessarily agree on what a large violation is.

2. The kernel hyperparameters are assumed to be more or less invariant of the violation strength, meaning the kernel hyperparameters and the violation strength can be explored separately.

3. Similar to the violation strength, the kernel hyperparameters are also assumed to be more or less invariant of a particular the training set size.

4. SVMs are particularly sensitive to uneven predictor spaces, thus all predictors are standardized in order to avoid long runtimes (this does not affect the results).

The assumptions presented above are true in general, but are introduced in order to limit the score of the project.

Unless otherwise specified, references to "SVMs" should be interpreted as 4 different support vector machines: one with a linear kernel, one with a 3^{rd} degree polynomial kernel, one with a radial Gaussian kernel and one with a sigmoid kernel. The parameters of the kernels, unless otherwise specified, are set to the default parameters of `sciki-learn`. More details on the default values can be found here: https://scikit-learn.org/stable/modules/svm.html

[8] By "training set size", what is meant is the portion of the original data set that is selected as the training set.

Having trained an SVM, the performance of the SVM is measured according to two metrics: its accuracy score and training time. Measuring both the accuracy and the training time may yield a compromise between accuracy and efficiency.

1. Violation control

It is very important to correctly control an SVMs violation in order to optimize results. The goal of this analysis is to perform a preliminary check of how different kernels behave with different violation strengths C. To this extent, 100 logarithmically spaced C values between 10^{-5} and 10^{2} are used to train the SVMs.

2. Overfitting the training data

Mirroring section III B 3, this analysis is concerned with whether the SVMs are sensitive to different divisions between the training and test sets. Again, the hyperparameters concerning the kernels are also not explored here. The violation strength for the different kernels are chosen based on the results from the previous analysis. Each SVM is trained using 30 training sets with training set sizes between 5% and 95%.

3. Kernels

The aim of this section is to study the hyperparameters of the polynomial, radial Gaussian and sigmoid kernels (the linear kernel does not have a hyperparameter, and thus is not included). The violation strength for the different kernels are chosen based on the results from the previous analysis. For a description of the hyperparameters, see table I. There are no absolute guidelines for determining the hyperparameters, the values presented here are based on trail and error.

For the polynomial kernel, the hyperparameters are the polynomial degree n and the center shift r. The polynomial degrees considered include 2,3,4,5 and 6. For each polynomial degree, 50 logarithmically spaced r values between 10^{-4} and 10^{2} are used.

In case of the radial Gaussian kernel, the hyperparameters include only the scaling factor γ. 200 logarithmically spaced γ values between 10^{-6} and 10^{2} are used.

The hyperparameters of the sigmoid kernel consists only of an argument shift r. As tanh is an odd function, it is uneccessary to explore negative r values. Consequently 50 r values between 10^{-6} and 10^{6} are used.

IV. RESULTS

A. Neural Networks

1. Network complexity

The main results of the complexity analysis are shown in figure 3. Additional results including similar figures have been included in the GitHub repository.

First and foremost, all networks score 90% or higher, something which is surprising considering Bathes et. al.'s score of 85%. Second, there are clear differences between networks with different activations, most notably the logistic networks compared to the other two. Third, regardless of the activation function, the networks exhibit a considerable boundary in the training times images, ca. along the 30 nodes per layer vertical. Interestingly however, this boundary is not present in the number of epochs image, meaning this boundary does not represent a mathematical turning point within the BPA algorithm. It follows that this boundary most likely actually stems from a change of numerical algorithm within either `NumPy` itself or the BPA implementation in `scikit-learn`.

Another surprising result is that the networks, regardless of activation, seem to exhibit a boundary between high and low accuracy. The boundary is particularly interesting in the logistic case where upon closer inspection, the boundary has been shown to exclusively include networks with a single layer or 2 layers.

As expected, there is a general trend for more complex networks to require more training, both in number of epochs and time. Exempt from this rule however is the rectifier networks, which require particular attention along the boundary between the high and low accuracy regions.

For the logistic networks, using 1 or 2 layers with about 10-20 nodes is recommended. Such a network is trained fast and scores optimally. Sigmoid networks on the other hand are a little more flexible as it can accept as many as about 25 layers without suffering from long training. However, the layers should have more than about 20 nodes per layer in order to be optimally accurate. Finally, the rectifier networks are optimized using a small number of layers, say less than 10, with ca. 10 nodes or more. Overall, the logistic networks train much faster than the sigmoid and rectifier networks. On the other hand, the high accuracy regions of the logistic networks is much smaller than for the other two, making them prone to large drops in accuracy with only small adjustments to the networks.

In order to standardize the neural networks for the next analyses, two different structures are implemented: one with a single layer and 8 nodes and one with 3 layers and 30 nodes per layer. For lack of better terminology, the former will be referred to as the "simple" structure, while the latter is referred to as the "complex" structure.

2. Network training scheme

The main results of the learning scheme analysis are shown in figure 4. Additional results have been included in the GitHub repository.

A striking difference between the simple and complex networks is the boundary between the low and high accuracy regions. Whereas the simple networks exhibit a more gradual boundary, the complex networks show a much more intense boundary reminiscent of the boundaries in figure 3.

Furthemore, the simple and complex networks also do not agree on the optimal learning parameters. The logistic networks exhibit the most stark contrast: while the simple networks are optimized for a whole range of parameters, the complex networks require more attention to detail. The center of the high accuracy region for the simple networks is just on the boundary the low and high accuracy regions. Meanwhile, the center of the high accuracy region for the complex networks is displaced down in the parameter space. Similar to the logistic networks, the rectifier networks also limit its range of optimal learning parameters, albeit to a lesser degree and the high accuracy center remains intact. The sigmoid networks on the other hand exhibit the most minimal difference between the simple and complex structures.

Seeing that the networks learn faster in the lower right regions (generally speaking) of the images, it is no surprise that the training times are generally very small in this region. There are some notable exceptions however, in particular for the logistic networks. The simple logistc networks behave exactly as predicted with respect to training times and the number of epochs spent training, however the complex networks form a centered band of learning parameters (with $\gamma_0 \approx 10^{-2}$) in which training is very slow. The band is most likely linked to the hard boundary between the low and high accuracy regions as it seems to trace the boundary quite closely.

To strike a compromise between learning efficiency and accuracy, most networks should be trained using a set of learning parameters along the diagonal from the lower left to the upper right. There are, of course, exceptions to this rule. In case the complex logistic networks are desired, it is important to first survey for the approximate location of the low-to-high accuracy boundary. To use rectifier networks, it is adviced to a slightly smaller initial learning rate.

For the next analyses, the following three learning parameter sets will be used: one with $(\gamma_0, \eta) = (10^{-1}, 0.3)$, one with $(\gamma_0, \eta) = (10^{-2}, 0.5)$ and the final with $(\gamma_0, \eta) = (10^{-3}, 0.7)$.

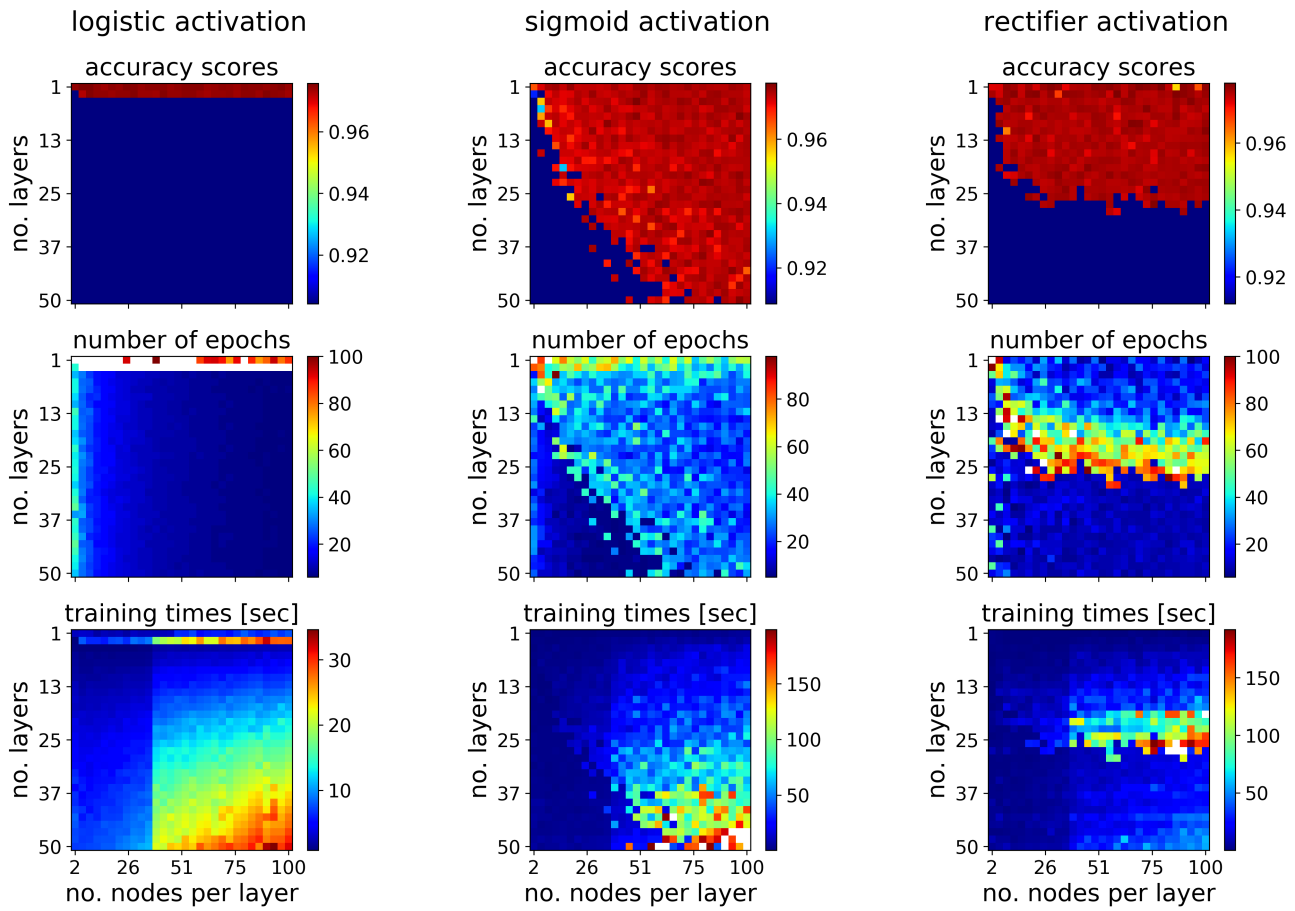
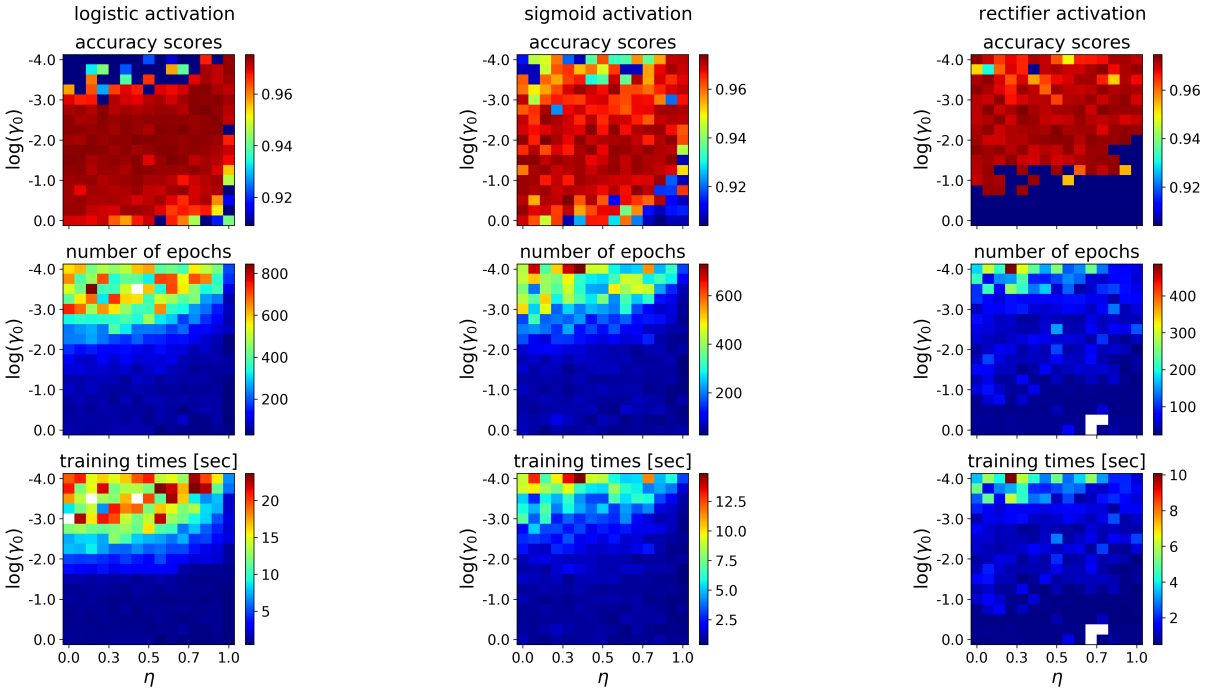
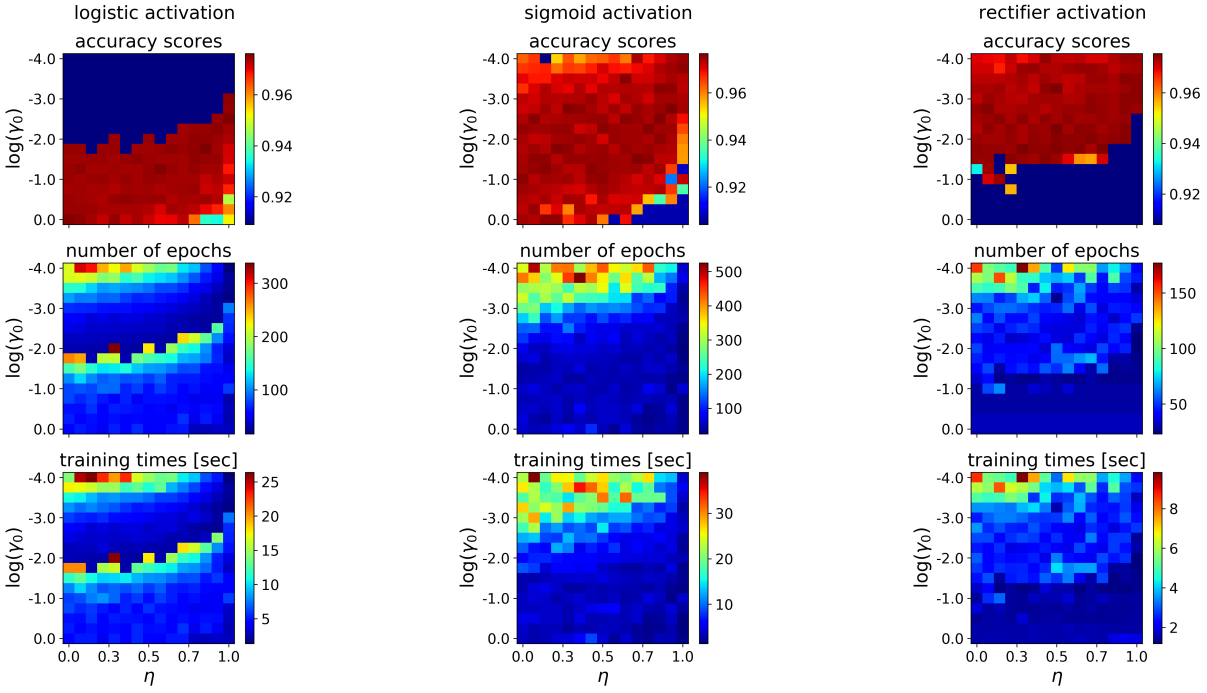


Figure 3: Hyperparameter analysis of the complexity of neural networks. In general, network complexity increases along the diagonal from the upper left to the lower right. An upper threshold of 150 epochs and 200 seconds is used to focus the colormaps, white pixels have passed the threshold.

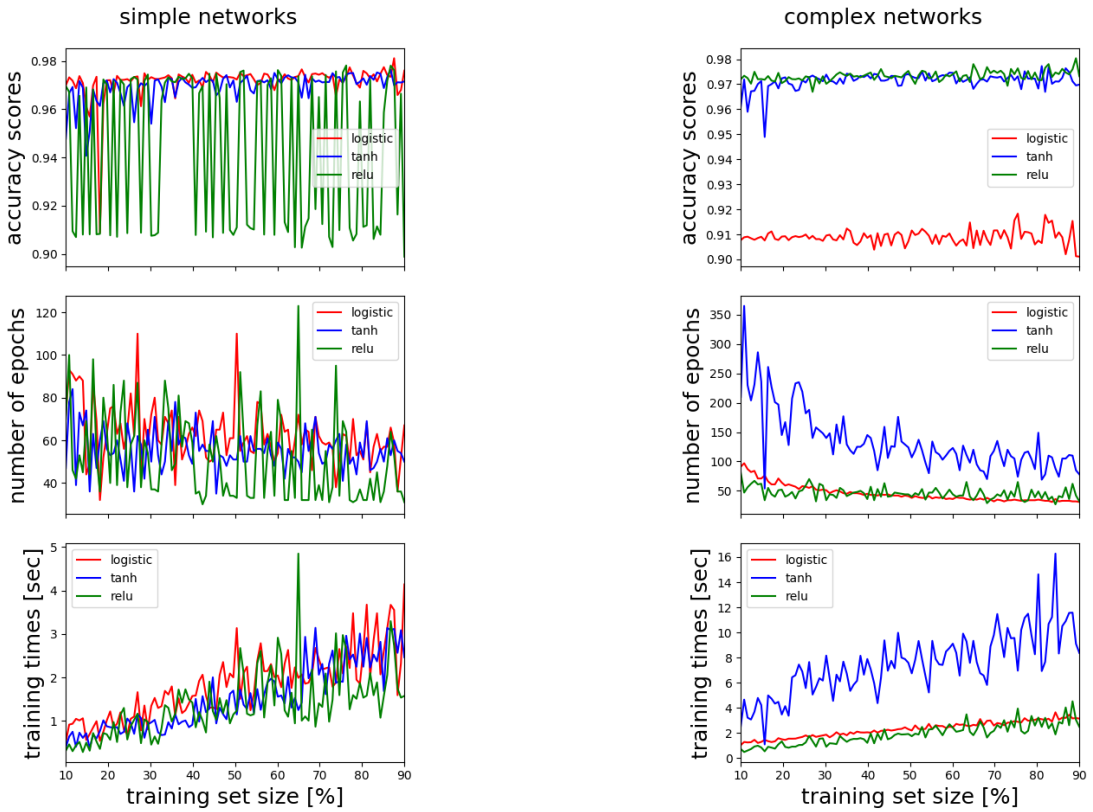


(a) The simple networks. An upper threshold of 1000 epochs and 25 seconds are used to focus the colormap, white pixels have passed the threshold.



(b) The complex networks.

Figure 4: Hyperparameter analysis of the learning scheme of neural networks. As a general rule, the learning rate increases along the diagonal from the upper left to the lower right.



(a) Simple networks, trained with $\gamma_0 = 10^{-1}$ and $\eta = 0.3$.

(b) Complex networks, trained with $\gamma_0 = 10^{-3}$ and $\eta = 0.7$.

Figure 5: Hyperparameter analysis of training set size's impact on neural networks overfitting.

3. *Training set overfitting*

The overfitting analysis was run for all network activations, for both simple and complex network structures, and for each of the three learning parameter sets from the training scheme analysis. Much of the results highlight the same relationships, as well as reiterate the results of the previous analysis. To avoid uneccessary figures, only two plots have been included: the analysis using simple networks, trained with $(\gamma_0, \eta) = (10^{-1}, 0.3)$, and the analysis using complex networks, trained with $(\gamma_0, \eta) = (10^{-3}, 0.7)$. The rest of the material, together with some additional material using different network structures, is included in the GitHub repository. The main results of the analysis are shown in figure 5.

Quite unexpectedly, the training set sizes have almost no impact on the network accuracies. Nevertheless, some data results do show a slight decrease in accuracy with training set sizes less than 20%. In addition, some networks' accuracies become less stable with increased training set sizes, e.g. the logistic networks in figure 5b. This could suggest some minor overfitting, however the resulting variations in the accuracy scores are miniscule compared the difference between the low and high accuracy regions seen in previous analyses.

Seeing that $(\gamma_0, \eta) = (10^{-1}, 0.3)$ is just on the low-to-high accuracy boundary for the rectifier networks (see figure 4), the unstable behaviour of the rectifier networks in figure 5b comes as no surprise. The erratic behaviour seems to be uniformly spread, suggesting whether a nerual network falls within the low or accuracy regions boils down to chance. This hypothesis is further supported by the fact that the neural network's weights and biases have random initial conditions. There does not seem to be any clear mechanisms with which such a strict boundary should appear (as opposed to smooth boundary), other than its a peculiarity of the BPA algorithm itself.

Furthermore, there is a general trend for training time to increase with increasing training set size, but this is expected. Especially in the case of the simple networks, which are more or less the same for all network activations. However, in the case of the complex networks, the sigmoid networks require a much larger numer of epochs, and thus time, to train. Despite this however, the sigmoid networks have no accuracy benefit over the rectifier networks (that is, in figure 5b).

It is clear that the training set size need not be greater than say 20%. As long as the training set is large enough, increasing it yields neither accuracy nor efficiency.

B. Support Vector Machines

1. *Violation control*

The results of the violation control analysis are shown in figure 6. While the SVMs with linear, polynomial and radial Gaussian kernels generally improve with increasing violation, the sigmoid SVM does not. Additionally, the simoid SVM is also the least accurate SVM overall, barely reaching an accuracy of above 91%. Furthermore, whereas the accuracy scores for the linear and polynomial SVMs improve with increasing violation, their corresponding training times explodes. On the other hand, the radial Gaussian SVM is capable of achieving similar accuracy with only a fraction of the training time.

Overall the linear kernel is the fastest SVM to train, in addition to having the highest score for all violations greater than 10^{-4}. It is the clear better alternative in case a fast SVM is needed.

The radial Gaussian SVM are clearly optimized with a violation of about 1: the accuracy is maximal (ca. 98%) as well as the training time is minimal (ca. 0.5 seconds). Interestingly, the training times for the sigmoid SVM does not seem to be affected by the increasing violation, despite its major drop in accuracy. The training times for the linear and polynomial SVMs are optimized for a violation between 10^{-2} and 10^{-1}, with which the SVMs score just below 98%. No particular violation strength seems to optimize the sigmoid SVM's training time, although it is slightly decreasing with increasing violation.

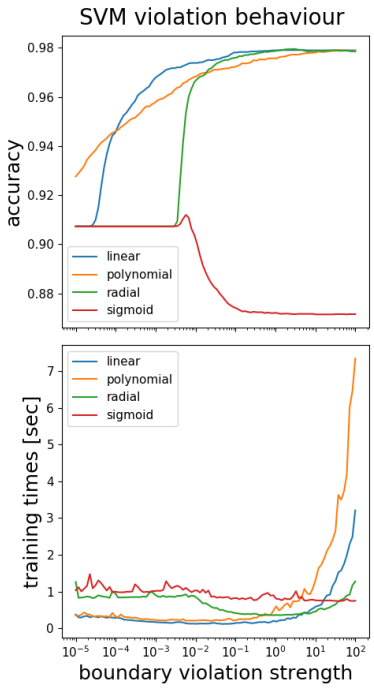


Figure 6: A crude analysis of how the different SVM kernels behave with respect to different violation strengths.

2. *Training set overfitting*

The results of the training set size analysis are shown in figure 7. Interestingly, in accordance with the neural networks, the SVMs also show little variation with the different training set sizes. The training times increases with increasing training set size, as expected, although admittedly much faster for the sigmoid SVMs. The same pattern of slightly larger fluctuations in the accuracy scores with increasing training set size can also be seen here.

Much like the neural networks, no rel benefit is added by incresaing the training set size. Hence, a training set size of more than about 20% is not necessary.

It's important to note that the trend seen in figures 5 and 7 is most likely due to the separability of the pulsar data set, and not the nerual networks or SVM kernels.

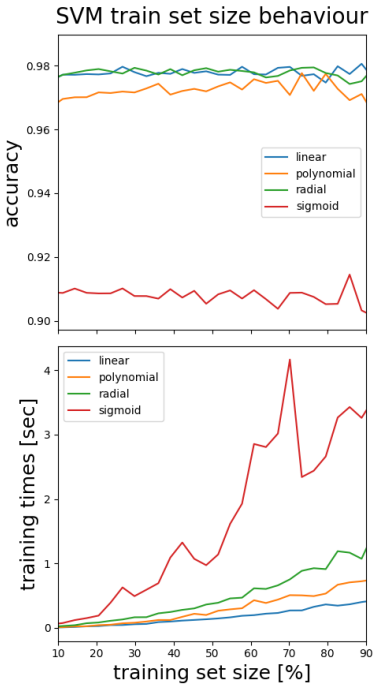


Figure 7: A crude analysis of how the different SVM kernels behave with respect to different training set sizes.

3. *Kernel hyperparameters*

The results for the SVM kernel hyperparameter analyses are shown in figures 8, 9 and 10. The three kernels exhibit quite different behaviour with respect to their respective hyperparameters.

The accuracy of the polynomial kernels generally increase with increasing polynomial shift until about $r \approx 10^1$, where most of the kernels exhibit some form of maximum or peak accuracy (although the 2$^{\text{nd}}$ degree kernel could still be on the rise, the details are somewhat unclear). When it comes to training times, increasing the degree of the polynomial has more or less a blatant negative effect. Out of the different polynomial kernels, the 3$^{\text{rd}}$ degree polynomial stands out as both very effective, yet highly accurate (which is lucky considering the polynomial kernel used in the two previous analyses were both of degree 3).

Out of the different SVM kernels, the radial Gaussian SVM is sepecial in that it features a *very clear* optimal hyperparameter, in fact a scaling factor of just about $\gamma = 10^{-1}$. This scaling both maximizes accuracy and minimizes training time, with little to no cost.

On the other hand, the Sigmoid SVM accuracy is more or less completely invariant of the argument shift r. However, the training times make a hard drop around $r \approx 10^1$. Hence, a large r, say $r = 10^3$, is clearly benefitial.

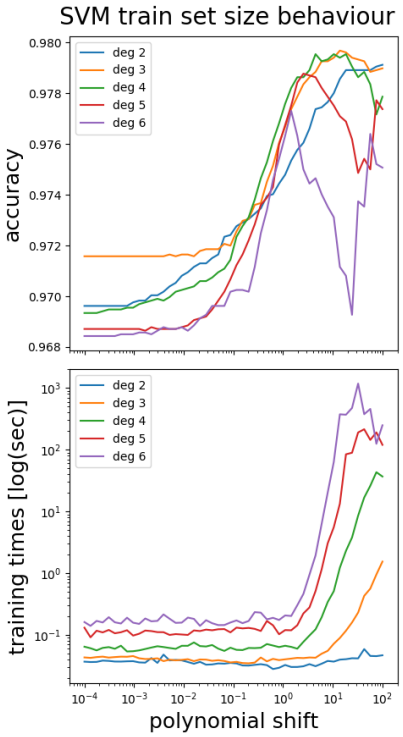


Figure 8: Hyperparameter analysis of the polynomial SVM kernel.

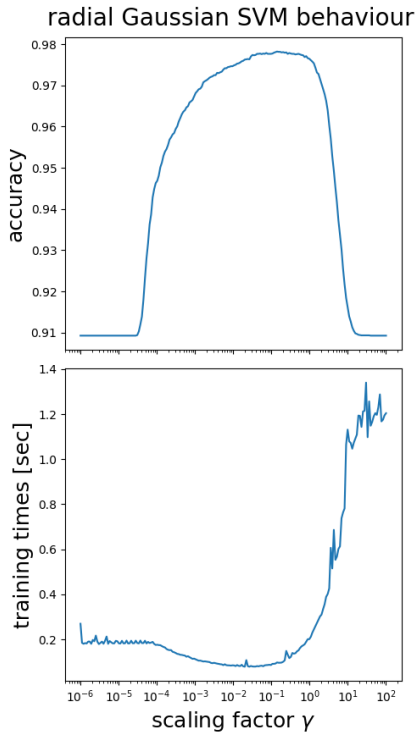


Figure 9: Hyperparameter analysis of the radial Gaussian SVM kernel.

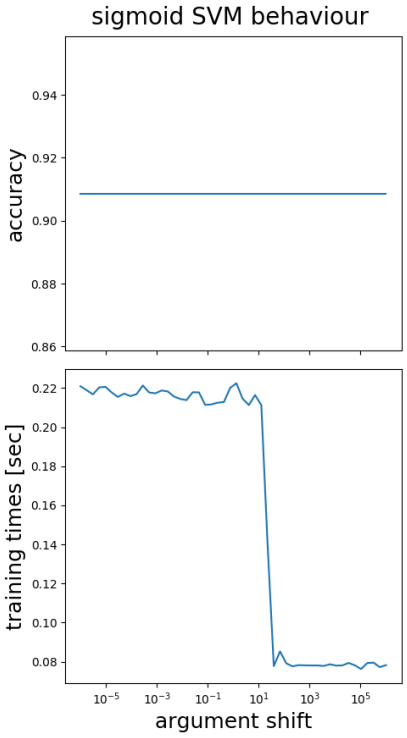


Figure 10: Hyperparameter analysis of the sigmoid SVM kernel.

V. DISCUSSION

The goal of this project is to examine whether neural networks and support vector machines could be introduced as a standard tool for examining large data sets in radio astronomy. In particular whether these methods can be used to extract pulsar candidates from radio signals using their integrated pulse profiles and DM-SNR curves. Bathes et. al.'s 2012 attempt yielded an MLP network that was able to *"blindly"* detect 85% of pulsar candidates. However, as the results of this project has shown, the methods discussed in this project have the potential to improve his accuracy to as high as 98%.

When it comes to the neural networks, it was clear that less complex networks, i.e. networks with less than about 10 layers, generally performed just as well as the more complex networks. In addition, the smaller network structures required much less training in general. An intersting case was the logistic networks, which required either a single layer or 2 layers in order to perform optimally. Overall, all network activation functions yielded good results, if parametrised correctly.

Nonetheless, based on the variety of acceptable hyperparameters that optimized the networks, the sigmoid networks definitely stood out as the most easy to work with. Especially considering that their training times were not particularly larger than the other two network activations. The sigmoid networks also performed very well for almost all choices of learning parameters, something that the logistic and rectifier networks did not. In fact, the sigmoid network is very stable for most combinations of hyperparameters; performing not only adequately, but exceptionally well.

Provided reasonable violation strengths, the support vector machines proved reasonble adversaries for the nerual networks; also scoring as high as 98%. This was not true for every kernel however, the sigmoid kernel was not even able to score 92%, which for comparison was the worst score for the other kernels.

Apart from the linear kernel, which was by far both the fastest and most accurate SVM, both the radial Gaussian and the 3^{rd} degree polynomial kernels achieved an accuracy score of 98% using less than a second to train. The polynomial SVM is trained slightly faster than the radial Gaussian SVM, but this is only measureble for *very* large training sets. Judging by the separability of the pulsar data set (see histograms in appendix A), the data set is most likely linear in its boundary. However, the radial Gaussian and polynomial SVMs are much more flexible than the linear SVM. In case a less linearly separable data set is used, the linear SVM is discouraged.

In general, the SVMs have exhibited slightly longer training times than the neural networks. Thus, neural networks could be adventageous in case training efficiency is important. However, the neural networks tend to have many more hyperparamaters, making the hyperparameter analysis that follows much more demanding than for an SVM. Take for instance the radial Gaussian SVM whose only hyperparameters are the violation strength and the scaling factor, and for comparison, a sigmoid neural network, which requires both a network structure analysis and a learning rate analysis. Assuming a hyperparameter analysis is required, training the SVM is likely much faster than training the neural network. On the other hand, the neural network is very robust, meaning a much smaller size of hyperparameter ranges may be used.

VI. CONCLUSION

In conclusion, the results of this project have shown that both neural networks and SVMs can be efficiently trained to correctly label some 98% pulsars and non-pulsars in a data set consisting of statistical measures radio signals' IPPs and DM-SNR curves. Neither neural networks not SVMs stand out as the better alternative.

Designing a neural network for this purpose, it has been shown that using sigmoid activation, a small number of layers (≈ 2) and few nodes $\approx 20 \pm 5$) is likely to optimize the network for almost any choice of the initial learning rate and learning momentum strength.

Designing an SVM for this purpose, it has been shown that the best results were produced using a linear kernel. However, this finding is most likely due to the linearly separability of the specific pulsar data set analysed in this project. Hence, if more flexibile SVMs are needed, it has been shown that similarly accurate results have been produced using a radial Gaussian with a violation of $C = 1$ and scaling factor $\gamma = 0.1$, and using a 3^{rd} degree polynomial kernel with a violation of $C = 0.1$ and polynomial shift of $r = 10$.

[1] F.Graham Smith, F.R.S., former Director of the Royal Greenwich Observatory (1976-1981). *Pulsars*. Cambridge University Press, 1977.

[2] Jim Condon. Essential radio astronomy, chapter 6: Pulsars. Online resource hosted by National Radio Astronomy Observatory (US), Last modifed 2016. https://www.cv.nrao.edu/~sransom/web/Ch6.html.

[3] Integrated pulse profile of a pulsar. Original work by Lorimer Kramer (Handbook of Pulsar Astronomy), republished by AstroBaki, Last Modified December 6^{th} 2017. https://casper.berkeley.edu/astrobaki/index.php/Dispersion_measure.

[4] S.D. Bathes et. al. The high time resolution universe survey vi: An artificial neural network and timing of 75 pulsars. arXiv:1209.0793v2 [astro-ph.SR], 2012.

[5] sckit-learn: Machine learning in python. Open Source.

Appendix A: Distributions of the Pulsar Data Set Predictors

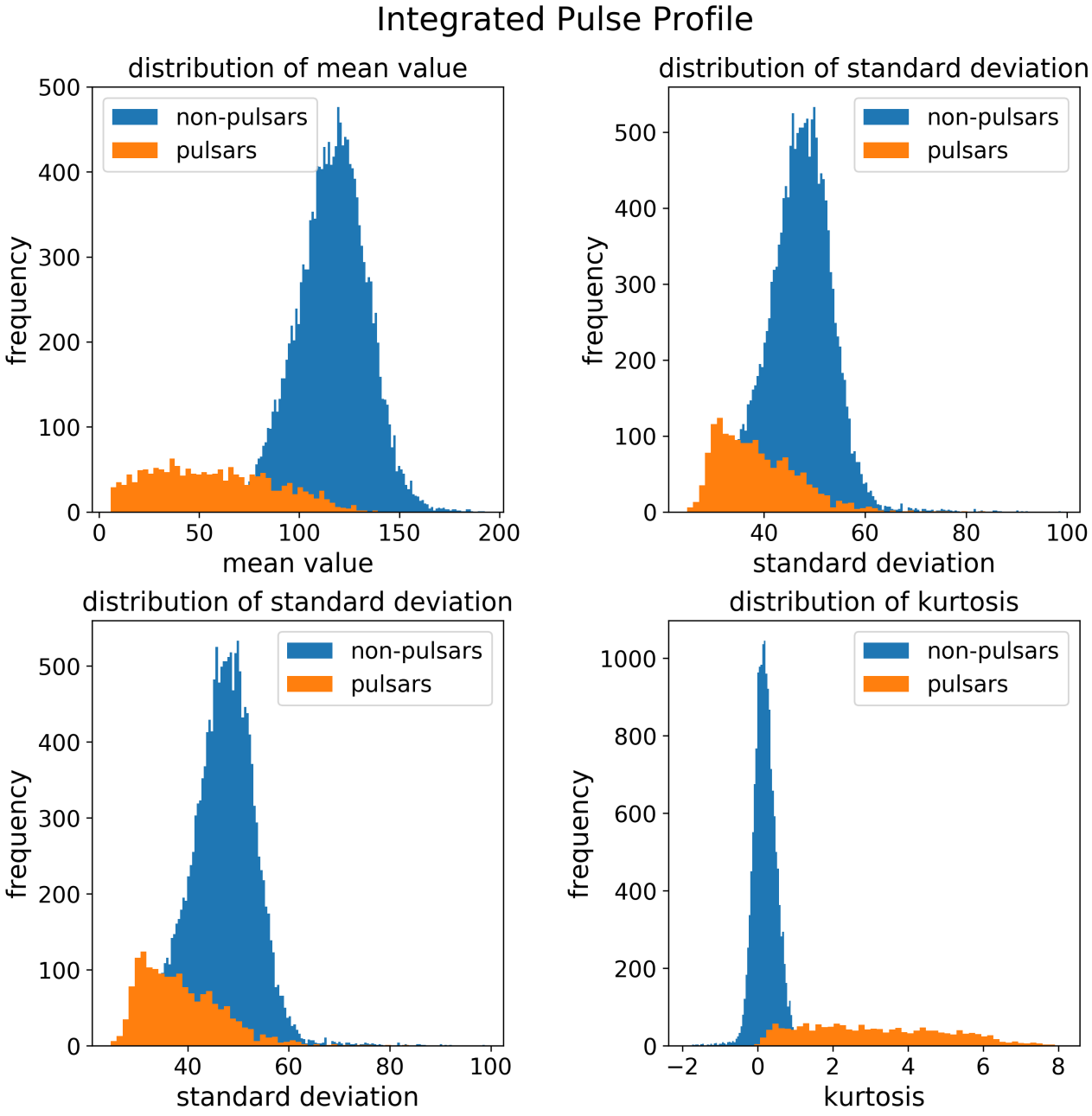


Figure 11: Histograms of the IPP predictors in the pulsar data set. The pulsars have been separeated from the non-pulsars in order to highlight the measurable difference between the two classes.

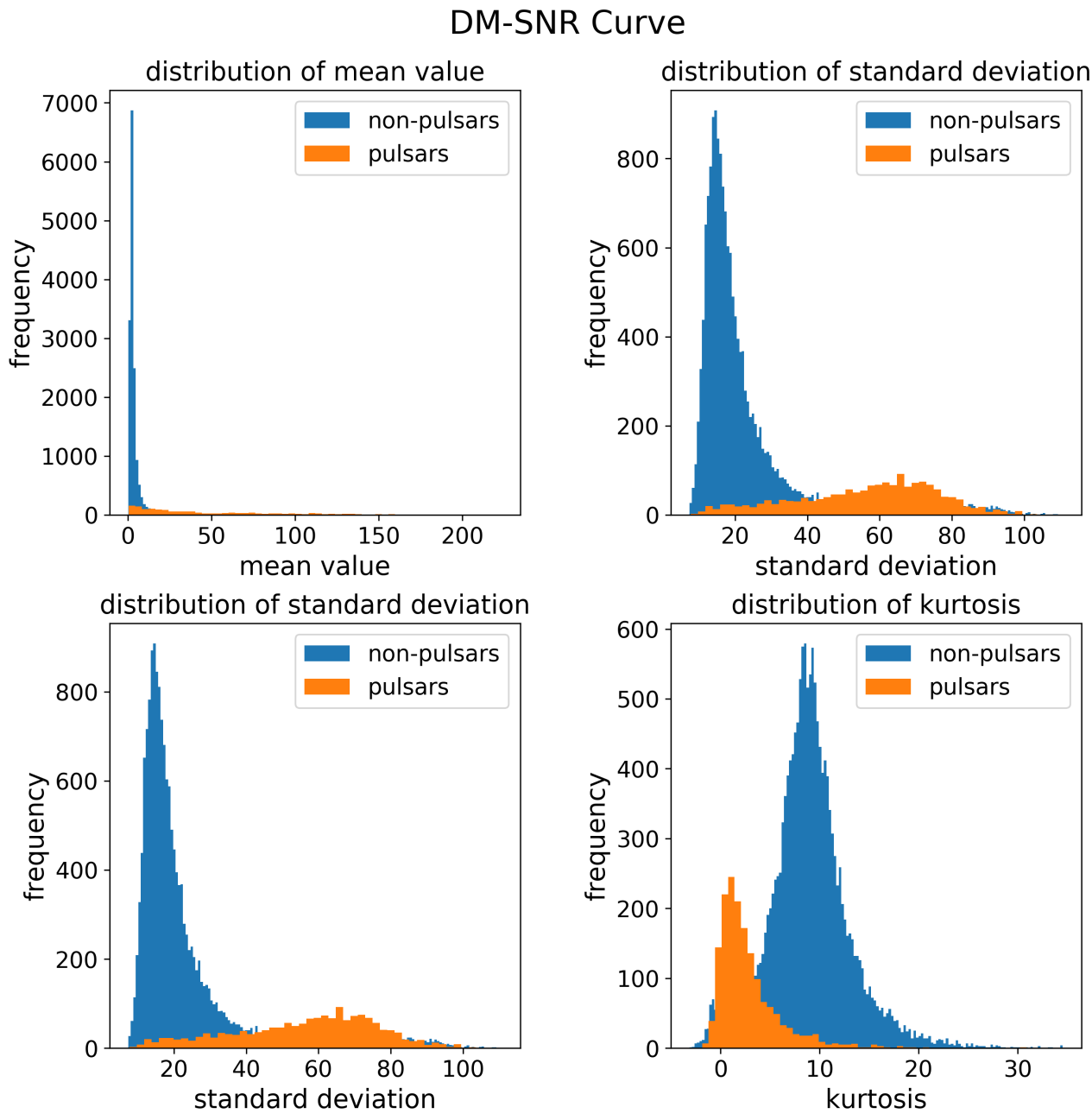


Figure 12: Histograms of the DM-SNR curve predictors in the pulsar data set. The pulsars have been separeated from the non-pulsars in order to highlight the measurable difference between the two classes.