A first encounter with Machine Learning

Abstract:

October 28, 2024

Contents

1 Introduction

1.1 Generalities

In the first sections of these notes we review some basic Machine Learning techniques, mainly based on [1]. The aforementioned review is based on Python and comes with jupyter notebooks. The simplest Python package for Machine Learning is Scikit-Learn, which we will also use at times, throughout these notes. Since version 11.0, `Mathematica` also comes with an integrated package (`Neuronetworks`) adapted to our purposes. Similarly to Scikit-learn, this is relatively straightforward to use as well, having a rather simple interface for building neural networks. For an early review of the use of machine learning in physics, see [2]. For more recent uses of `Mathematica` in building neural networks, see [3, 4], for instance.

Problems in Machine Learning start with a data set $\mathcal{D}$, and a model f, which is a function that predicts the output from the input variables using a set of parameters. Additionally, there is a *cost function* that keeps track of how good the model is, being minimized by changing the parameters accordingly. The first step in a ML algorithm is to divide the data set into two randomly selected mutually exclusive sets $\mathcal{D} = \mathcal{D}_{train} \oplus \mathcal{D}_{test}$. This provides an unbiased estimate for the predictive performance of the model, which is known as *cross-validation*. The training set usually amounts for $80 - 90\%$ of the whole dataset. The model is then fit by minimizing the chosen cost function only over the training set and its performance evaluated by computing the cost function on the test set. The value of the cost function for the best fit model

on the training set is called the *in-sample error* E_{in}, while the value on the test set is usually referred to as the *out-of-sample error* E_{out}.

It is important to stress out that most of the time one has $E_{out} \geq E_{in}$. Multiple candidate models are typically compared using E_{out} and, perhaps surprisingly at first, the model having the lowest E_{in} does not have the lowest E_{out} in most cases. At small sample sizes, noise can create fluctuations in the data that look like genuine patterns. Simple models such as linear functions cannot represent complicated patterns in the data, so they are forced to ignore the fluctuations and to focus on the larger trends. Complex models with many parameters, on the other hand, can capture both the global trends and noise-generated patterns at the same time; thus, the model can be tricked into thinking that the noise encodes real information, which is a typical problem called *overfitting*. This problem can be avoided by either using models with less parameters, or using larger data sets, which decrease the likelihood of patterns in the noise. This is what is known as *bias-variance tradeoff*, i.e. limited data can be often better predicted by models with less parameters, thus having more 'bias', but since it might not be a close to the 'real' model it has less 'variance'.

However, even increasing the data set size, there is a fundamental issue that consists. That is, predicting the behaviour beyond the training data domain is a very difficult task. It should thus be emphasized that *fitting is not predicting* - i.e. fitting existing data well is fundamentally different from making predictions about new data. Thus, one might wonder how, in principle, can an algorithm learn the behaviour of the data set outside the training set. The answer is that learning is possible in the restricted sense that the fitted model will probably perform approximately as well on new data as it did on the training data. This is, of course, a statement about the relation between E_{in} and E_{out}, which is the domain of Statistical learning theory. We summarise the basic intuitions of statistical learning in a few statements.

First, the amount of training data is correlated to the two errors as follows. As the training data increases, E_{in} typically increases as well, as the models are not powerful enough to learn the true function of the model. Additionally, the E_{out} error will decrease since the sampling noise decreases, with the training data set becoming more representative of the true distribution from which the data is drawn. In the limit where the sample size is infinite, the two errors tend to be equal, which is called the 'bias' of the model. However, since the sample size is typically finite, one tries to minimise E_{out}, rather then the bias. Note that *models with a large difference between the in-sample and out-of-sample errors are said to "overfit" the data.* The other statement involves the relation to the complexity of the model, i.e. the number of parameters used. As stated before, increasing the complexity of the model will always reduce the bias, but, at some point, the model becomes too complex for the amount of training data, leading to a high variance due to finite size sampling effects (i.e. noise becomes a real pattern) and thus a high E_{out} error. This goes back to the

bias-variance tradeoff mentioned before. Schematically, one has:

$$E_{out} = Bias^2 + Var + Noise \ . \tag{1.1}$$

1.2 Stochastic Gradient Descent

One of the most used methods for minimising the cost function is called the 'gradient descent'. The idea is to adjust the *hyperparameters* of the model at every step in the direction where the gradient of the cost function is large and negative, such that the parameters flow to local minima of the cost function. The simplest Gradient descent

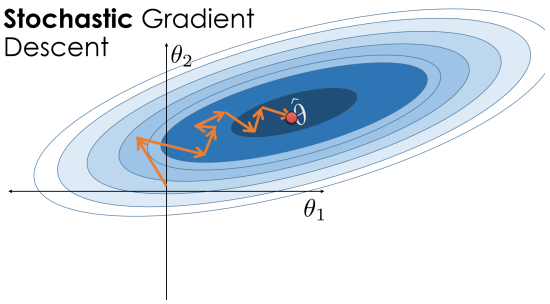


Figure 1: Stochastic Gradient Descent

algorithm updates the parameters $\boldsymbol{\theta}$ at any step t according to:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \nabla_\theta E(\boldsymbol{\theta}_t) \ , \tag{1.2}$$

where $E(\boldsymbol{\theta})$ is the function to be minimised. Here η_t is called the learning rate and controls the size of the step. Of course, a small learning rate will ensure that the method converges to a local minimum, but this might come with a larger computational time. If the learning rate is too large, then the algorithm eventually diverges instead. Note that in order to reach a global minimum instead, the above algorithm needs to be modified, by introducing *stochasticity*. Additionally, calculating gradients over the whole data set is computationally ineffective, which is why 'mini-batches' are preferred.

One algorithm that solves the above mentioned issues is the stochastic gradient descent. This approximates the gradient on a subset of the data, called a 'mini-batch', whose size is much smaller than the total number of data points:

$$\nabla_\theta E(\boldsymbol{\theta}) \longrightarrow \sum_{i \in B_k} \nabla_\theta e_i(\boldsymbol{x}_i, \boldsymbol{\theta}) \ , \tag{1.3}$$

where for linear regression e_i is just the mean square-error for the data point i. One then cycles over the mini-batches, updating $\boldsymbol{\theta}$ at every step k. The full iteration over all data points is usually referred to as an *epoch*. This method has two important benefits. First, it introduces stochasticity and decreases the chance that our fitting

algorithm gets stuck in isolated local minima. Second, it significantly speeds up the calculation as one does not have to use all n data points to approximate the gradient. Additionally, stochasticity prevents overfitting in those isolated minima.

The SGD algorithm is typically also used with a 'momentum' term that serves as a memory of the direction we are moving in parameter space, implemented as:

$$\boldsymbol{v}_t = \gamma \boldsymbol{v}_{t-1} + \eta_t \nabla_\theta E(\boldsymbol{\theta}_t) \ , \boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{v}_t \ , \tag{1.4}$$

with $\gamma \in [0, 1]$, leading to:

$$\Delta \boldsymbol{\theta}_{t+1} = \gamma \Delta \boldsymbol{\theta}_t - \eta_t \nabla_\theta E(\boldsymbol{\theta}_t) \ . \tag{1.5}$$

This usually helps the SGD algorithm gain speed in directions with persistent but small gradients even in the presence of stochasticity. A slight modification of this algorithm is the Nesterov Accelerated Gradient (NAG), which instead calculates the gradient at the expected value of the parameters given the current momentum, allowing thus for the use of larger learning rates.

A further improvement to these algorithms is given by adaptive changes of the learning rate; this can be accomplished by keeping track of the second moment of the gradient as well, and some examples are AdaGrad, AdaDelta, RMSprop or ADAM. The ADAM optimizer, for instance, also performs an additional bias correction to account for the fact that we are estimating the first two moments the gradient using a running average.

2 Supervised Machine Learning

2.1 Linear Regression

For the rest of this section consider a labelled dataset with n-samples $\mathcal{D} = \{(y_i, \boldsymbol{x}_i)\}$, where $\boldsymbol{x}_i \in \mathbb{R}^p$ are the observables while y_i are the corresponding response and let f be the true model that generated the responses:

$$y_i = f(\boldsymbol{x}_i; \boldsymbol{w}_{true}) + \epsilon_i \ . \tag{2.1}$$

with $\boldsymbol{w}_{true} \in \mathbb{R}^p$ a parameter vector and ϵ_i some white noise. We are seeking a function g with parameters $\boldsymbol{w}$ to best approximate f.

Least-square regression . Ordinary least squares linear regression is defined as the minimization of the norm of the difference between the response y_i and the predictor $g(\boldsymbol{x}_i, \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{x}_i$, namely:

$$min_{\boldsymbol{w} \in \mathbb{R}^p} ||\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}||^2 = min_{\boldsymbol{w} \in \mathbb{R}^p} \sum_{i=1}^{n} \left(\boldsymbol{w}^T \boldsymbol{x}_i - y_i \right)^2 \ . \tag{2.2}$$

Here $\boldsymbol{X} \in \mathbb{R}^{n\times p}$ is the *design matrix*, with the rows being the $\boldsymbol{x}_{(i)}$ observables. The predictor function $g(\boldsymbol{x}_i, \boldsymbol{w})$ defines a hyperplane in $\mathbb{R}^p$, with the minimization procedure being equivalent to minimizing the sum of all projections of the points $\boldsymbol{x}_i \in \mathbb{R}^p$ to this hyperplane. The optimal solution for $n \geq p$ is unique and we denote by $\boldsymbol{w}_{LS}$. One can show that the average in-sample and out-of-sample errors are related by:

$$|\mathbf{E}_{in} - \mathbf{E}_{out}| = 2\sigma^2 \frac{p}{n} \ , \tag{2.3}$$

with σ the deviation from the fit. Thus, we note that even if $p \sim n$, the noise in σ^2 has an important influence on the prediction, while when p is much larger then n, the outcome of the least squares regression cannot be trusted. This issue can be ameliorated using *regularization*. We will see that we the regularized regression problem can be viewed as an un-regularized problem but with a constrained set of parameters.

Ridge regression . The Ridge regression adds to the least squares loss function a regularizer defined as the norm of the parameter vector we wish to optimize over, that is we wish to solve:

$$min_{\boldsymbol{w}\in\mathbb{R}^p} \left(\sum_{i=1}^{n} \left(\boldsymbol{w}^T \boldsymbol{x}_i - y_i \right)^2 + \lambda \boldsymbol{\omega}^T \boldsymbol{\omega} \right) \ . \tag{2.4}$$

This turns out to be equivalent to a constrained optimization problem, i.e. a least square regression which constraints the magnitude of the parameter vector $\boldsymbol{w}$. The result of the Ridge problem is the equivalent least squares solution, but scaled by a factor of $(1+\lambda)$. In particular, for the case when there is a unique solution, one has:

$$\boldsymbol{w}(\lambda) = \left(\boldsymbol{X}^T \boldsymbol{X} + \lambda \right)^{-1} \boldsymbol{X}^T \boldsymbol{y} \ , \tag{2.5}$$

which reduces to the least squares solution for $\lambda \to 0$.

LASSO regression . The least absolute shrinkage and selection operator does, on the other hand, introduce a different regularizer:

$$min_{\boldsymbol{w}\in\mathbb{R}^p} \left(\sum_{i=1}^{n} \left(\boldsymbol{w}^T \boldsymbol{x}_i - y_i \right)^2 + \lambda \sqrt{\boldsymbol{\omega}^T \boldsymbol{\omega}} \right) \ . \tag{2.6}$$

Note that this regularizer is often referred to as an $L1$ regularizer, while the one appearing in the Ridge regression is sometimes called an $L2$ regularizer. As in the Ridge regression, this can be interpreted as a constrained optimizer, but unlike the previous two regressions, the solution is more difficult to find analytically, as the gradient of the optimizer might not well defined for any $\boldsymbol{w}$. For orthogonal $\boldsymbol{X}$, one finds:

$$w_j^{LASSO}(\lambda) = sign(w_j^{LS}) \left(|w_j^{LS}| - \lambda \right)_+ \ , \tag{2.7}$$

where we take the positive part of the right hand side. As a result, while Ridge does a 'rescaling', LASSO does a 'soft-thresholding', with many components of $\boldsymbol{w}^{LASSO}$ being zero typically. Thus, Lasso tends to do well if there are a small number of significant parameters, with the others being close to zero. On the other hand, Ridge regularization is better when most predictors impact the response.

Let us finally comment on the new parameter λ. This parameter is not part of the regression, being predetermined, but the learning performance and solution depends strongly on it. Thus it is vital to choose it properly. The best way of choosing it is through some optimization procedure - i.e. check performance for various values.

2.2 Classification Tasks

A wide variety of problems, such as classification, are concerned with outcomes taking the form of discrete variables. In this section, we introduce logistic regression which deals with binary, dichotomous outcomes. As such, we assume that the dependent variables $y_i \in \mathbb{Z}$ are discrete and only take values from $m = 0, \dots, M-1$, with M the number of classes. For future use, we define the linear classifier:

$$s_i = \boldsymbol{x}_i^T \boldsymbol{w} + b_0 = \mathbf{x}_i^T \mathbf{w} \ , \tag{2.8}$$

with $\mathbf{x}_i = (1, \boldsymbol{x}_i)$. While this function takes values in $\mathbb{R}$, one can get a discrete output by $\sigma(s_i) = sign(s_i)$, for instance, which is usually called the *perceptron* in the ML literature.

Logistic regression. Usually, it is preferred to work with a so-called 'soft' classifier, an example of which is the above perceptron. The canonical example of such a classifier is the logistic/sigmoid regression:

$$\sigma(s) = \frac{1}{1+e^{-s}} \ , \qquad 1 - \sigma(s) = \sigma(-s) \ . \tag{2.9}$$

Here, the probability that a data point belongs to a category $y_i = \{0, 1\}$ is:

$$P(y_i = 1) = \sigma(s_i) \ , \qquad P(y_i = 0) = 1 - P(y_i = 1) \ , \tag{2.10}$$

with $\mathbf{w}$ the weights we wish to learn from the data. The cost function for logistic regression is typically defined using Maximum Likelihood Estimation (MLE), which chooses parameters to maximize the probability of seeing the observed data set. This leads to the maximum likelihood estimator:

$$max_{\mathbf{w}} \sum_{i=1}^{n} y_i \ \log \sigma(s_i) + (1 - y_i) \log(1 - \sigma(s_i)) \ , \tag{2.11}$$

and to the cost (error) function:

$$\mathcal{C}(\mathbf{w}) = \sum_{i=i}^{n} -y_i \log(\sigma(s_i)) - (1 - y_i) \log(1 - \sigma(s_i)) \ , \tag{2.12}$$

which is known as the *cross entropy*. Note also that as in linear regression, one supplements the cross-entropy with additional regularization terms. We further note that minimizing the cost function leads to the transcedental equation:

$$\sum_{i=1}^{n} \left(\sigma(s_i) - y_i\right) \mathbf{x}_i = 0 \ . \tag{2.13}$$

One can show that the cross-entropy error function used in logistic regression has a unique minimum. In practice, these algorithms can be easily implemented in python, using Scikit-learn. The default logistic regression of Scikit is *liblinear*, but one can also use Stochastic Gradient Descent based routines. (See notebook 6 for example - use logistic regression for $r = 0$ or not or for smooth/not-smooth.)

Non-binary classifier. The previously introduced logistic regression can be generalized to a multi-class classification, which we do by treating the label vectors as $\boldsymbol{y}_i \in \mathbb{Z}_2^M$, i.e. binary strings of length M, with only one component $y_{i,j}$ being 1 and the rest 0. Thus, the SoftMax function is defined by the probability of a data point $\boldsymbol{x}_i$ of being in class m':

$$P\left(y_{i,m'} = 1 \,|\, \{\mathbf{w}_k\}_{k=0}^{M-1}\right) = \frac{e^{-\mathbf{x}_i^T \mathbf{w}_{m'}}}{\sum_{m=0}^{M-1} e^{-\mathbf{x}_i^T \mathbf{w}_m}} \ . \tag{2.14}$$

The cost function is then generalized to:

$$\mathcal{C}(\mathbf{w}) = -\sum_{i=1}^{n} \sum_{m=0}^{M-1} y_{i,m} \log P(y_{i,m} = 1) + (1 - y_{i,m}) \log(1 - P(y_{i,m} = 1)) \ . \tag{2.15}$$

This regressor is again implemented in the Scikit-learn package and can be used with the LogisticRegression by setting the 'multi_class' option to 'multinomial'.

2.3 Combining models

One of the most powerful and widely-applied ideas in modern machine learning is the use of ensemble methods that combine predictions from multiple statistical models to improve predictive performance. Of course, this weighted sum of many predictors does not always work; it turns out that the degree of correlation between the models in the ensemble can be used to determine whether the ensemble methods can be succesfully used. We will review the most popular ensemble methods, namely bagging, boosting, random forests and gradient boosted trees (XGBoost).

Bias-Variance tradeoff for Ensembles. Let us briefly review the bias-variance tradeoff in the context of ensemples. As for a single model, the expected generalization error (i.e. out-of-sample error) decomposes schematically as:

$$E_{out} = Bias^2 + Var \ , \tag{2.16}$$

with the Bias measuring the deviation of the expectation value of the estimator from the true value, while the variance measures how much the estimator fluctuates due to finite-sample effects. However, for ensembles, the expression of variance becomes:

$$Var(\mathbf{x}) = \rho(\mathbf{x})\sigma^2 + \frac{1-\rho(\mathbf{x})}{M}\sigma^2 \ , \tag{2.17}$$

where σ is the standard deviation of the ensemble predictor and ρ is the correlation function between the M models. The above expression says that as $M \to \infty$ (large ensemble), the variance can be significantly reduced, while if the models are completely uncorrelated then the variance is maximally supressed. Additionally, this does not come at the expense of a very large bias, which is just the expected bias of a single model, so one can add more models without increasing the bias.

Bagging. One of the most employed ensemble methods is Bootstrap AGGregation, or BAGGing. For this, consider a large dataset $\mathcal{L} = (\mathcal{L}_1, \ldots, \mathcal{L}_M)$, partitioned into M smaller sets. Supposing each subset is large enough to learn a predictor, then we can define an aggregate predictor, which, for continuous predictors like regression, is just the average of all individual predictors:

$$g_{\mathcal{L}}(\mathbf{x}) = \frac{1}{M}\sum_{i=1}^{M} g_{\mathcal{L}_i}(\mathbf{x}) \ , \tag{2.18}$$

while for classification tasks where each classifier predicts a class label $j \in \{1, \ldots, J\}$, this is a maximization problem:

$$max_j \sum_{i=1}^{M} I\left(g_{\mathcal{L}_i}(\mathbf{x}) = j\right) \ , \tag{2.19}$$

which is just a majority vote of all the predictors. If the $\mathcal{L}_i$ subsets are not large enough, one can resort to empirical bootstrapping, where the datasets are allowed to share data points. This will, generally, lead to an increase in the bias compared to bagged estimators.

Boosting. The next ensemble method that we briefly discuss is Boosting. Unlike Bagging, the aggregate classifier is now formed by assigning some weights to each individual classifier/predictor:

$$g(\mathbf{x}) = \sum_{k=1}^{M} \alpha_k g_{\mathcal{L}_k}(\mathbf{x}) \ , \tag{2.20}$$

with $\sum \alpha_k = 1$. A common boosting ensemble method is 'adaptive boosting' (AdaBoost), in which the aggregate classifier is formed in an iterative process by highighting the data points where the aggregate classifier performs poorly. Let $(\boldsymbol{x}_i, y_i)_{i=1}^{N}$ be

the data set, with $y_i \in \{\pm 1\}$. The boosted classifier is constructed by first initialising:

$$w_{t=1}(\boldsymbol{x}_i) = \frac{1}{N} \ , \qquad i = 1, \ldots, N \ . \tag{2.21}$$

Then, for $t = 2, \ldots, T$ (desired termination step), one selects a hypothesis that minimizes the weighted error:

$$\epsilon_t = \sum_i w_t(\boldsymbol{x}_i) I(g_t(\boldsymbol{x}_i) \neq y_i) \ , \tag{2.22}$$

and update the weight for each data point as follows:

$$w_{t+1}(\boldsymbol{x}_i) = \frac{1}{Z_t} w_t(\boldsymbol{x}_i) e^{-\alpha_t y_i g_t(\boldsymbol{x}_i)} \ , \qquad Z_t = \sum_i w_t(\boldsymbol{x}_i) e^{-\alpha_t y_i g_t(\boldsymbol{x}_i)} \ , \tag{2.23}$$

with $\alpha_t = \frac{1}{2} ln \frac{1-\epsilon_t}{\epsilon_t}$.

Random Forests. One of the most widely used and versatile algorithms in data science is the Random Forests method. This is usually used for complex classification tasks, being composed of a family of randomized tree-based classifier decision trees. These are high-variance, weak classifiers that can be easily randomized, thus reducing the variance of the ensemble.

Each branch of the decision tree consists of a question that splits the data into smaller subsets, with the leaves of the tree corresponding to the ultimate partitions of the data. The goal is to construct trees such that the partitions are informative about the class label. It is clear that more complex decision trees lead to finer partitions that give improved performance on the training set, but, generally, this leads to over-fitting, limiting the out-of-sample performance. Thus, it is advised to use some form of regularization (e.g. maximum depth of the tree) to control complexity and reduce over-fitting.

The randomness needed to decrease the variance of the ensemble is usually introduced in the random forests in one of three distinc ways: (1) training each decision tree on a different bootstrapped dataset, (2) use a different random subset of the features at each split in the tree, (3) extreme randomization procedure where splitting is done randomly instead of using optimality criteria.

Gradient Boosted Trees. Finally, let us mention the Gradient-Boosted Trees, which use a mixture of methods from Boosting and gradient descent to construct ensembles of decision trees. Like in boosting, the ensembles are created by iteratively adding new decision trees to the ensemble; at each step one computes the gradient of a cost function and adds trees that move in the direction of the gradient.

3 Neural Networks

In this section we aim to introduce the basic concepts behind Neural Networks and possibly touch upon Deep Learning methods. Neural networks (or neural nets) are nonlinear models for supervised machine learning. They are more powerful extensions of linear and logistic regressions.

Introduction. The basic building block of a neural net is a 'neuron', which takes a vector $\boldsymbol{x} = (x_1, \ldots, x_d)$ as input and produces a scalar output $a(\boldsymbol{x})$. A neural network has many such neurons, stacked into layers, with the middle layers (i.e. those that are not input or output layers) called hidden layers. The function $a(\boldsymbol{x})$ varies depending on the type of non-linearity used in the neural network, but it can be decomposed into a linear operation - that weights the relative importance of the various inputs - and a non-linear transformation which is usually the same for all neurons:

$$a_i(\boldsymbol{x}) = \sigma_i(z^{(i)}) = \sigma_i(\boldsymbol{w}^{(i)}\boldsymbol{x} + b^{(i)}) \ , \tag{3.1}$$

where the index i labels the neuron, $\sigma^{(i)}$ is the non-linear transformation and $\boldsymbol{w}^{(i)}$ are neuron-specific weights. Typical example of non-linear functions are step-functions (perceptrons), sigmoids, hyperbolic-tangent, rectified linear units - ReLU, etc. The neural nets are trained using gradient descent based methods, which involve the derivatives of the neural functions with respect to the weights and bias, leading to different computational and training properties for neurons for different non-linear functions. Note that certain functions have a major drawback - when input weights become large in training, the activation function saturates, leading to 'vanishing gradients'. ReLU functions, however, have finite gradients even for large inputs.

The use of hidden layers greatly expands the representational power of the neural net, when compared with a simple soft-max or linear regression network. Note also that according to the 'universal approximation theorem' a neural network with a single hidden layer can approximate any continuous function with arbitrary accuracy, as long as the number of neurons is not constrained. The idea is that neurons allow networks to generate step functions with arbitrary offsets and heights, which, added together, can approximate any function.

Deep neural networks are neural networks with multiple hidden layers. A general rule of thumb that seems to be emerging is that the number of parameters in the neural net should be large enough to prevent underfitting.

Neural networks can be trained similarly to the linear supervised learning algorithms used before. That is, one constructs a cost/loss function and then uses gradient descent based methods to find the wights and bias by minimising the cost function. Depending on whether one wishes to make continuous or categorical predictions, different cost functions must be used. For continuous data, the common

cost functions are those used in linear regression, such as the mean squared error:

$$E(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{y}_i(\mathbf{w})\right)^2 \ , \tag{3.2}$$

where $(\mathbf{x}_i, y_i)$ are the labelled data points are $\hat{y}_i(\mathbf{w})$ are the neural network predictions, with n the number of points. Alternatively, one can use the mean-absolute error (the $L1$ norm):

$$E(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\left|y_i - \hat{y}_i(\mathbf{w})\right| \ , \tag{3.3}$$

but regularizers can be also included ($L1$, $L2$ regularizers etc.). For classifiers, the cross-entropy is typically used, as the output layer is often taken to be a softmax classifier. As before, this can be supplemented by regularizers. Training is then done by using gradient descent or similar methods that optimize the cost function. That is, the weights are updated to move in the direction of the gradient of the cost function. Note, however, that unlike the previous models, the gradients for a neural network require a specialized algorithm called *Backpropagation*.

Backpropagation algorithm. As stated before, the training procedure involves the derivatives of the cost function with respect to the weights of all neurons in the neural network and thus cannot be done by brute force due to the large computational times. However, the layered structure of the network can be exploited to do this more efficiently. Assume there are L layers and denote by w^l_{jk} the weight for the connection from the k-th neuron in layer $l-1$ to the j-th neuron in layer l. The latter has bias b^l_j. In a *feed-forward* neural network, the activation function a^l_j of the j-th neuron in the l-th layer can be related to the neurons in the layer below by:

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right) = \sigma(z^l_j) \ . \tag{3.4}$$

The cost function E depends directly on the activities of the output layer a^L_j, by definition. We define the error Δ^L_j of the j-th neuron in the output layer as the change in cost function with respect to the weighted input z^L_j:

$$\Delta^L_j = \frac{\partial E}{\partial z^L_j} \ , \tag{3.5}$$

and analogously, the error of neuron j in layer l as:

$$\begin{aligned}\Delta^l_j &= \frac{\partial E}{\partial z^l_j} = \frac{\partial E}{\partial a^l_j}\sigma'(z^l_j) \\ &= \frac{\partial E}{\partial b^l_j} \ ,\end{aligned} \tag{3.6}$$

where we used the chain rule, together with $\partial b_j^l / \partial z_j^l = 1$. Then, using chain rule further, one shows:

$$\Delta_j^l = \left(\sum_k \Delta_k^{l+1} w_{kj}^{l+1} \right) \sigma'(z_j^l) \ . \tag{3.7}$$

The final backpropagation equation is found by differentiating the cost function and using again chain rule:

$$\frac{\partial E}{\partial w_{jk}^l} = \Delta_j^l a_k^{l-1} \ . \tag{3.8}$$

The above four equations can be combined in a simple algorithm to compute the gradient with respect to all parameters, as follows. (1) **Activation at input layer:** calculate all a_j^1. (2) **Feedforward:** Exploit feed-forward architecture to compute z^l and a^l at each layer. (3) **Error at top layer:** calculate error at top layer from (3.5). (4) **Backpropagate the error:** use (3.7) to calculate Δ_j^l for all layers. (5) **Calculate gradient:** use (3.6) and (3.8) to compute the gradient of the cost function with respect to all parameters.

Neural Networks using Scikit-learn. Let us briefly comment un the use of neural networks using the Scikit-learn package. This is sometimes reffered to as 'multi-layer perceptron', or MLP as well. The module contains the public attributes `coefs_` and `intercepts_`, with the former being a list of weight matrices and the latter a vector of bias vectors (each layer has the same bias).

Let us note that while the MLP is capable of learning non-linear models, its cost function has more local minima and thus differend random weight initializations can lead to different accuracies. Additionally, on needs to tune the number of hyperparameters (layers, number of neurons, iterations a.k.a. epochs) for such an algorithm.

The classification problems can be implemented using `MLPClassifier`, which is an MLP algorithm using backpropagation. In Scikit-learn, the model only supports the Cross-Entropy as the cost function. This supports multi-class classification by applying Softmax as the output function. My understanding is that one cannot use different activation functions for the hidden layers in this case. The regressor can be similarly implemented with `MLPRegressor`.

Neural Networks in Mathematica. For versions 11.0 and above, the `Neuronetworks` package can be used for Machine Learning purposes. For a review, see [2], for instance. A single layer perceptron (SLP) can be defined with `NetChain`, with the argument being a list containing a single elements that specifies the activation function in the layer. The SLP can be initialized with random weights and bias via `NetInitialize`.

Now, an MLP can be set up using `NetChain`, with the list of arguments specifying the activation functions for all layers. The neural network can then be trained using `NetTrain`.

4 IHS Database

In this section we have a first look at the isolated hypersurface singularity (IHS) database generated in [5]. This contains 39094 data points, with the input being the quasi-homogeneous equation in $\mathbb{C}^4$ defining the IHS:

$$\{F(x) = 0 \,|\, x \in \mathbb{C}^4\} \ , \qquad F(\lambda^q x) = \lambda F(x) \ . \tag{4.1}$$

Certain physical data of interest can be obtained straightaway from the scaling weights $q_i \in \mathbb{Q}_+$. However, there is some additional topological data that is, in practice, difficult to obtain, that is r, the rank of the associated 5d SCFT, b_3 the number of 3-cycles in the resolved threefold, as well as the smoothness of the exceptional divisors. Our goal is to implement Machine Learning techniques to investigate these properties associated to a given IHS.

We will have a first look at the dataset generated in [5] in the next subsection and work with a simpler dataset in the ML algorithm. We first attempt some binary classification problems using logistic regression and later generalize this to non-binary classifiers, using the SoftMax regression.

4.1 Rank Binary Classifiers

The dataset of interest contains 39094 datapoints, labelled by their scaling weights $q_i \in \mathbb{Q}_+$, satisfying the constraint:

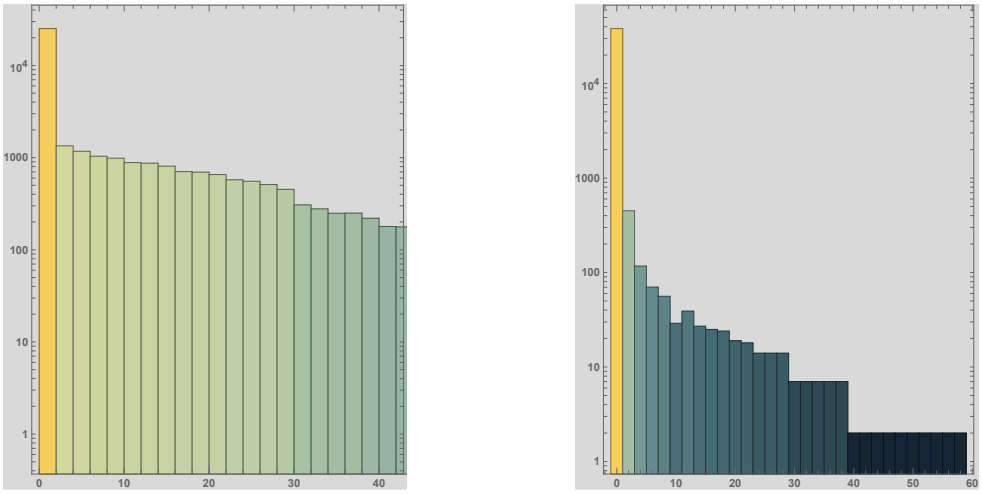
$$\frac{1}{q_i} \leq 60 \ . \tag{4.2}$$

As a first check of the data, we plot the rank r in a histogram, shown in figure 2a. We notice that the vast majority of cases have $r = 0$, namely 25119 of the IHS, with the maximum rank being 240. Thus, given this large number of IHS of rank zero, we will set up a binary classification problem with:

$$\sigma(r) = \begin{cases} 0 & \textit{if} \ \ r = 0 \ , \\ 1 & \textit{if} \ \ r > 0 \ . \end{cases} \tag{4.3}$$

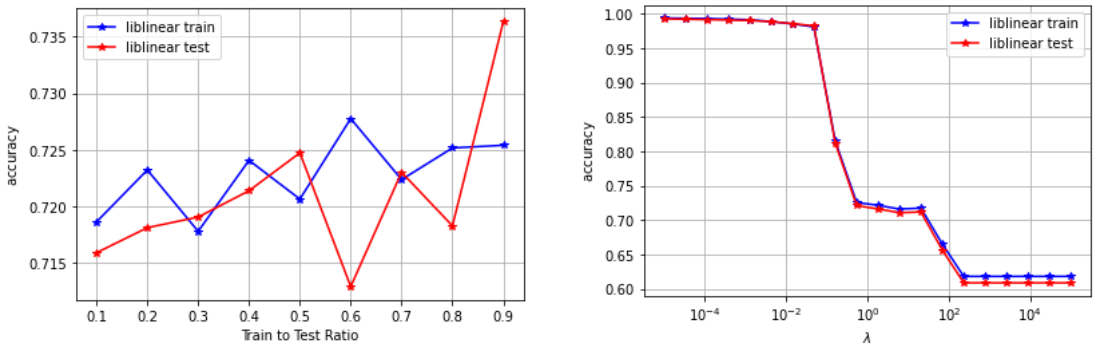
We use Python's Scikit-learn package, with its default logistic regression *liblinear*. For simplicity, we use the *score* function of the logistic regression. We will also use two types of dataset. The first dataset takes as input the scaling weights $(q_i)_{i=1}^4$ for each data point $\boldsymbol{x}_i$, with the output being the rank, for now. The second data set will have some additional data as input, such as:

$$\left((q_i)_{i=1}^4 \, , \, \mu \, , \, \hat{r} \, , \, a \, , c\right) \ . \tag{4.4}$$



(a) (b)

Figure 2: Rank of the 5d SCFT associated to the IHS in our dataset and the number of 3-cycles in the resolved threefold.



(a) (b)

Figure 3: Rank of the 5d SCFT associated to the IHS in our dataset classified using Logistic Regression.

While μ can be directly found from the scaling weights, the rank of the associated 4d SCFT $\hat{r}$ and the central charges contain information about the grading of the Milnor ring.

First Dataset. We run the regression on the simpler dataset, without a regularizer, and check the in-sample and out-of-sample errors for various values of the ratio between the sizes of the train and test datasets, respectively. The results are shown in figure 3a. We note that the difference between the two accuracies does not appear to depend too much on this ratio, which is an indication that the model does not overfit the data. However, the overall performance is not quite as expected, for which reason we introduce a regularizer λ. We run the logistic regression again, with the results shown in figure 3b for a ratio of 0.5. Indeed, as $\lambda \to 1$, we recover the previous

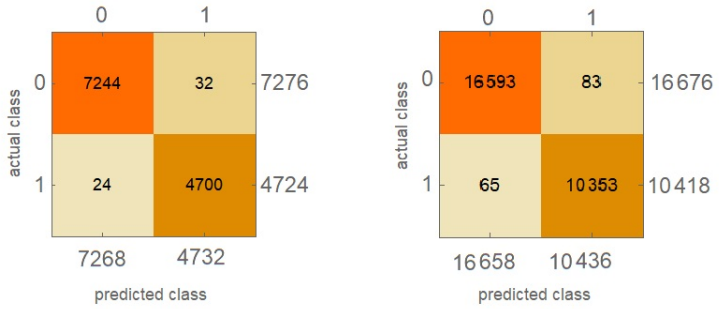


Figure 4: Rank of the 5d SCFT associated to the IHS in our dataset classified using Logistic Regression in Mathematica using Classify. The two confusion matrices correspond to the training and testing sample performances.

result, but for $\lambda \sim 10^{-2}$, the performance improves drastically. Additionally, we also run a regression based on the Stochastic Gradient Descent. As before, the train and test performances are very similar to each other, but, nonetheless, the SGD method performs much worse than logistic regressor.

Let us note that the training and test sets in this example were chosen arbitrarily from the whole dataset. We do a similar analysis in `Mathematica`, using the in-built classifier `Classify[]`, opting again for a Logistic regression. In this case we split the data into roughly the same sizes as before, with 30% of the data in the training set. Additionally, we also use a Validation set, which is a small subset of the testing set (roughly 6.5% of the whole dataset). As with Python, we use an $L2$ regularizer, which is optimized automatically by `Mathematica`. The results are again rather surprising: we find that for a regularizer of 10^{-5}, the training and test accuracies are 99.53% and 99.45%, respectively. The Confusion matrices for the two sets are shown in figure 4. We can use this classifier to find families of scaling weights (q_i) that are likely to have rank zero. For instance, it appears that:

$$(q_i) = \left(q_1\,,\,\frac{1}{2}\,,\,\frac{1}{2}\,,\,\frac{1}{2}\right)\,, \tag{4.5}$$

will always have rank 0 based on this classifier, for $q_1 \leq \frac{1}{2}$. We can indeed check that in the data set all examples up to $q = \frac{1}{60}$ do correspond to $r = 0$. We can also do more complicated checks, such as:

$$(q_i) = \left(q_1\,,\,q_1\,,\,q_1\,,\,\frac{1}{2}\right)\,, \tag{4.6}$$

which turns out to have rank 0 only if $\frac{1}{2} \geq q_1 \geq 0.287$, with a sharp logistic function for the probability around this value.

We would, however, like to restrict the training set to smaller values of the scaling weights and check then the performance of the model on the testing set. To do this, we first impose:

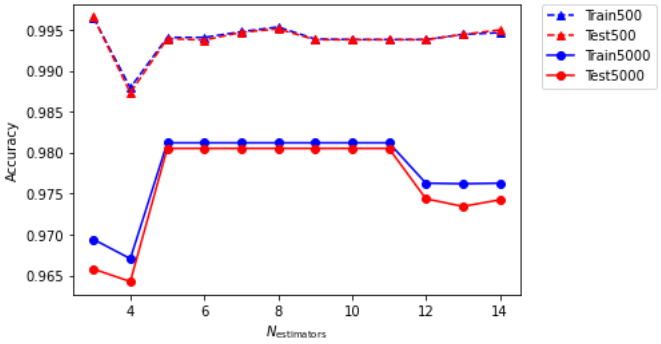
$$q_i \geq \frac{1}{45}\,, \tag{4.7}$$

which leads to two data sets, with 13880 and 25214 samples, respectively. We then use 12000 samples of the 'low' q^{-1} data set (i.e. the first subset) for training, with the remaining samples forming the testing data set. Additionally, we again use a validation set, which is a random subset from the test set, of 2500 sample. We implement the same logistic classifier in `Mathematica` and check the accuracy of the model. While the training accuracy is still very high, at about 99.56%, the performance in the testing set drops significantly, to about 71%. Note that the L2 regularizer set by `Mathematica` is at about 10^{-6}. This, perhaps, should not be a surprise, since the results follow the common lore in machine learning, namely that 'fitting is not predicting'.

We first try to change the strength of the regularizer and find that for $\lambda \sim 10^{-1}$ the testing performance improves to 78%. Looking at the confusion matrix for the model, however, we see that the largest number of missclassifications is represented by theories having $r > 0$ that were predicted to have $r = 0$, and not the other way around. To improve our estimates, we can change the `ClassPriors` option of the classifier, which specifies the explicit prior probabilities to assume for output classes. Indeed, chaning the prior probability for the $r = 0$ class to only 20% already appears to reduce the out-of-sample error, leading to accuracies closer to 85% for the testing set. Additionally, manually changing the value of the $L2$ regularizer for the logistic regression also appears to have a strong impact on the autcome. Setting this value to 1, we obtain accuracies of 98% for both training and testing.

We also try to classify the data using a Random Forests ensemble, which is also implemented in Python's Scikit-learn package as RandomForestClassifier. There are two main hyper-parameters that are important in practice for the performance of the algorithm and the degree to which it overfits/underfits, namely the number of estimators in the ensemble and the depth of the trees used. For the latter, we use the parameter min_samples_split, which encodes the number of data points needed in each node of the classification tree, i.e. a large number means that the trees are less complex. The results are shown in figure 5a. It appears that the number of estimators does not affect the prediction significantly after $n \geq 5$. Moreover, the performance of the classifier is already extremly good for a rather simple tree, with minimal number of data points in each node being 5000.

A similar algorithm can be implemented in `Mathematica`, using `Classify[]`. Training on a random subset of the whole data set, this simple classifier leads to a perfect performance on both the training and testing sets. Thus, we also try training the same classifier on the data set with $q^{-1} < 45$ and, as before, we find that the testing performance drops to 67%, with a similar confusion matrix as before.



(a)

Figure 5: Rank of the 5d SCFT associated to the IHS in our dataset classified using Random Forests.

References

[1] P. Mehta, M. Bukov, C.-H. Wang, A. G. R. Day, C. Richardson, C. K. Fisher et al., *A high-bias, low-variance introduction to Machine Learning for physicists*, **810** (May, 2019) 1–124, [1803.08823].

[2] Y.-H. He, *Deep-Learning the Landscape*, 1706.02714.

[3] Y.-H. He, *Universes as big data*, *Int. J. Mod. Phys. A* **36** (2021) 2130017, [2011.14442].

[4] Y.-H. He and J. M. P. Ipiña, *Machine-Learning the Classification of Spacetimes*, 2201.01644.

[5] C. Closset, S. Schäfer-Nameki and Y.-N. Wang, *Coulomb and Higgs Branches from Canonical Singularities, Part 1: Hypersurfaces with Smooth Calabi-Yau Resolutions*, 2111.13564.