ML STUDY GROUP NOTES

Ferrara Cosmology Group

Author
Patrick Stengel
INFN Sezione di Ferrara
July 13, 2023

Contents

1 Machine Learning Basics

1.1 Learning Algorithms

Definition 1.1. A computer program is said to **learn** from **experience** with respect to some class of **tasks** and **performance measure**, if its performance at the tasks improves with experience.

Following Chapter 5 of Ref. [1], machine learning tasks are usually described in terms of how **examples** are processed. An examples is a collection of **features** that have been measured from some object or event within a set of data. For example, pixel values are typically the features of image data. Common machine learning tasks include:

- **Classification**: Specify which of a given set of categories some input belongs. Examples include pattern recognition, spread of infectious disease, collider physics...
- **Regression**: Predict a numerical value given some input. Examples include predicting the weather, algorithmic trading, parameter estimation in cosmology...
- **Anomaly detection**: Sift through a set of objects and flag some of them as unusual. Examples include credit card fraud, manufacturing faults, faint astrophysical sources...

Performance measures generally depend on the specific task. For classification tasks, we can measure the **accuracy** or **error rate** of an algorithm based on the number of examples for which the output is correct or incorrect. As an example, consider a model to predict that a tumor is malignant (positive) or benign (negative) with 1 TP, 1 FP, 8 FN and 90 TN.

- Accuracy $= (\mathrm{TP} + \mathrm{TN})/(\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}) = 0.91$, fraction correct
- **Precision** $= \mathrm{TP}/(\mathrm{TP} + \mathrm{FP}) = 0.50$, fraction of correct positive identifications
- **Recall** $= \mathrm{TP}/(\mathrm{TP} + \mathrm{FN}) = 0.11$, fraction of true positives identified correctly

Machine learning algorithms can be broadly characterized by what kind of experience they are allowed to have during the learning process. We focus on algorithms allowed to experience a complete collection of many examples, or **dataset**, and each example can be associated with a **label** in addition to a set features.

- **Supervised learning** algorithms experience labeled datasets. For a dataset containing examples of $\boldsymbol{x}$ associated with a label $\boldsymbol{y}$, these algorithms learn to predict $\boldsymbol{y}$ from $\boldsymbol{x}$, usually by estimating the conditional probability distribution $p(\boldsymbol{y}|\boldsymbol{x})$. Typical examples tasks involving supervised learning include classification and regression.
- **Unsupervised learning** algorithms are designed to find patterns and structures in unlabeled datasets. Typical examples are anomaly detection and the more general estimation of the probability distribution $p(\boldsymbol{x})$ for an arbitrary point in the feature space $\boldsymbol{x}$. Note the distinction from supervised approaches is not always clear.
- **Reinforcement learning** algorithms interact with an environment and, thus, do not experience a fixed dataset. Fro example, a robot can be trained to navigate in a complex environment by assigning a high reward to actions that help the robot reach a desired destination.

1.2 Interactive examples

In this section we primarily want to work through a few examples which demonstrate some of the basic principles of machine learning. We will briefly summarize some of the background and key takeaways here, but all of the specific results and plots referenced are explained in the reasonably well annotated Jupyter notebooks.

1.2.1 Unsupervised vs. supervised learning

The objective with these two examples is to demonstrate the basics of unsupervised and supervised learning using the simplest possible realizations, histograms and linear regression. As discussed in the previous section, unsupervised learning is often associated with density estimation, or trying to reconstruct some probability distribution from scratch.

- Following the discussion in Section 2.5 of Ref. [2], we focus on partitioning a single continuous variable x into distinct bins of constant width Δ and then count the number n_i of observations of x falling in bin i.
- In order to turn this count into a normalized probability density, we simply divide by the total number N of observations and by the width Δ of the bins to obtain probability values for each bin given by $p_i = n_i/(N\Delta)$.
- We learn to estimate the probability density at a particular location, we should consider the data points that lie within some **local neighborhood** of that point. Note that the concept of locality requires that we assume some form of distance measure.
- This is defined by the bins for a histogram and there is a natural **smoothing parameter** describing the spatial extent of the local region, in this case the bin width, should be neither too large nor too small in order to obtain good results.

To demonstrate some of the basic features of supervised learning, we consider a simple example of linear regression. The estimate of the conditional probability distribution $p(\boldsymbol{y}|\boldsymbol{x})$ described in the previous section in the case of linear regression corresponds to fitting a function $y(x)$ to training data in order to predict y for arbitrary values of x.

- Following Sections 5.1.3 and 5.1.4 of Ref. [1], we can use the features observed in a generic dataset define a **design matrix**, $\boldsymbol{\Phi} \in \mathbb{R}^{M\times n}$, where M is the number of examples and n is the number of features.
- A common performance measure for regression is the **mean squared error**, $MSE = \sum_m (\hat{\boldsymbol{y}} - \boldsymbol{y})^2_m/M$, where $\hat{\boldsymbol{y}}$ is the prediction of the model for the ith example of the dataset described by the associated feature vector $\boldsymbol{x} \in \mathbb{R}^n$ from the design matrix.
- Specifically for linear regression, we define the scalar $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$ for a vector of **weights**, $\boldsymbol{w} \in \mathbb{R}^n$, which determines how each feature affects the prediction. The weights are determined by minimizing MSE for the **training dataset** $(\boldsymbol{\Phi}^{(\text{train})}, \boldsymbol{y}^{(\text{train})})$.
- In the notebook, we draw random points from a line with unit slope and add Gaussian random noise for each point to define the training dataset. The weights which minimize the corresponding MSE are given by $\boldsymbol{w} = (\boldsymbol{\Phi}^{(\text{train})\top}\boldsymbol{\Phi}^{(\text{train})})^{-1}\,\boldsymbol{\Phi}^{(\text{train})\top}\boldsymbol{y}^{(\text{train})}$.

1.2.2 Why is Machine Learning Difficult?

In the the previous example, we used linear regression to find the best fit for a training dataset. Following the discussion in Section 5.2 of Ref. [1] and borrowing the notebook from Section II of Ref. [3], we use examples from polynomial regression to show the difficulty in **generalizing** a given model to perform well on previously unobserved input.

- It is important to distinguish between **training error** associated with the performance of the model on the training dataset (e.g. the MSE from the previous example) and **generaliztion error**, which is the expected error on new input, the **test dataset**.
- The central challenges in machine learning are obtaining a low training error, **underfitting**, and keeping the gap between the training and generalization error small, **overfitting**. The **capacity** of a model is its ability to fit a wide variety of functions.

Figure 1: From Section 5.2 of Ref. [1]. At the left end of the graph, training error and generalization error are both high. This is the underfitting regime. As we increase capacity, training error decreases, but the gap between training and generalization error increases. Eventually, the size of this gap outweighs the decrease in training error, and we enter the overfitting regime, where capacity is too large, above the optimal capacity.

- If we first look at noiseless examples in the notebook, even using a small training set with 10 examples, we find that the model class that generated the data also provides the best fit and the most accurate out-of-sample predictions.
- With the noise turned on and 100 examples, the tenth order model now provides the best fit to the training data and the worst out-of-sample predictions. At small sample sizes, noise can create fluctuations in the data that look like genuine patterns.
- When we increase the size of the training data set by two more orders of magnitude, the tenth order polynomial clearly gives both the best fits and the most predictive power over the entire training range and even slightly beyond.
- Even with ten thousand data points and a model class that generated the data, the performance quickly degrades beyond the original training data range. This demonstrates the difficulty of predicting beyond the training data or generalizing.

2 Linear Models for Regression

For linear and polynomial regression, we assumed that models were linear combinations of the input variables or higher powers thereof. In these models, the **basis functions** $\phi_j(\boldsymbol{x})$ were polynomials of the input variables $\boldsymbol{x}$ such that we had $\phi_j(\boldsymbol{x}) = x^j$ and

$$\hat{y} = \sum_{j=0}^{n-1} w_j x^j \tag{1}$$

for a single input variable x. Models defined by n parameters, corresponding to the weights w_j, were polynomials of degree $n-1$. However, we can generalize the idea of basis functions

to include arbitrary non-linear functions of the input variables,

$$\hat{y} = \sum_{j=0}^{n-1} w_j \phi_j(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}) \,, \tag{2}$$

fixing $\phi_0(\boldsymbol{x}) = 1$ such that the **bias parameter** is given by w_0. In addition to polynomial basis functions, we can also consider Gaussian and logistic sigmoid functions specified in one input space dimension by a position μ_j and spatial extent s

$$\phi_j(x) = \exp\left\{-\frac{(x-\mu_j)^2}{2s^2}\right\} \,, \quad \phi_j(x) = \left(1 + \exp\left\{-\frac{x-\mu_j}{s}\right\}\right)^{-1} \tag{3}$$

Figure 2: From Section 3.1 of Ref. [2]. Examples of basis functions, showing polynomials on the left, Gaussians in the center, and logistic sigmoids on the right.

2.1 Maximum likelihood and least squares

Following from the previous section, we can connect the least squares approach to the error function. First, we define the target variable $t = y + \epsilon$ where the noise ϵ is a zero mean Gaussian random variable with precision β such that the conditional probability

$$p(t|\boldsymbol{x}, \boldsymbol{w}, \beta) = \mathcal{N}(t|y, \beta^{-1}) \tag{4}$$

for a normal distribution $\mathcal{N}$ with average y and variance β^{-1}. For a set of inputs $\boldsymbol{X} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M\}$ mapped to corresponding target values $\boldsymbol{t} = t_1, \ldots, t_M$ and assuming these data points are drawn independently from the distribution, the **likelihood function** is

$$p(\boldsymbol{t}|\boldsymbol{X}, \boldsymbol{w}, \beta) = \prod_{m=1}^{M} \mathcal{N}(t_m|\boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}_m), \beta^{-1}) \,. \tag{5}$$

Taking the logarithm of the likelihood function, we find the sum-of-squares **error function**

$$\ln p(\boldsymbol{t}|\boldsymbol{X}, \boldsymbol{w}, \beta) = \frac{M}{2} \ln \beta - \frac{M}{2} \ln(2\pi) - \beta E_D(\boldsymbol{w}) \,, \quad E_D(\boldsymbol{w}) = \frac{1}{2} \sum_{m=1}^{M} \{t_m - \boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}_m)\}^2 \,. \tag{6}$$

Thus we see that maximizing the likelihood function is equivalent to minimizing the sum-of-squares error function for regression under the assumption of a Gaussian noise model. We can explicitly maximize the the log likelihood w.r.t. $\boldsymbol{w}$ and β

$$\boldsymbol{w}_{\mathrm{ML}} = (\boldsymbol{\Phi}^\top \boldsymbol{\Phi})^{-1} \boldsymbol{\Phi}^\top \boldsymbol{t} \,, \quad \frac{1}{\beta_{\mathrm{ML}}} = \frac{1}{M} \sum_{m=1}^{M} \{t_m - \boldsymbol{w}_{\mathrm{ML}}^\top \boldsymbol{\phi}(\boldsymbol{x}_m)\}^2 \,. \tag{7}$$

The former are know as the **normal equations**, in which we can recognize the design matrix from the last section, with elements $\Phi_{mj} = \phi_j(\boldsymbol{x}_m)$. The latter demonstrates that the noise variance which maximizes the likelihood is given by the MSE from the last section.

2.2 Sequential learning and regularization

While linear regression models allow for closed solutions to the maximization of the likelihood, this can be costly for large datasets and not possible in general. **Sequential** learning algorithms consider one data point at a time and update model parameters. For example,

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} - \eta \nabla E_m \tag{8}$$

updates the weights for a given model by **stochastic gradient descent**, in which τ denotes the iteration number after the presentation of the mth data point with contribution E_m to the error function and η is the **learning rate**. Plugging in the sum-of-squares error function,

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} + \eta\{t_m - \boldsymbol{w}^{(\tau)\top}\boldsymbol{\phi}(\boldsymbol{x}_m)\}\boldsymbol{\phi}(\boldsymbol{x}_m)\,, \tag{9}$$

Sequential learning is associated with the concept of **regularization** through the particular example of **weight decay**. We can add a quadratic function of $\boldsymbol{w}$ to the sum-of-squares error function to get the total error function

$$\frac{1}{2}\sum_{m=1}^{M}\{t_m - \boldsymbol{w}^\top\boldsymbol{\phi}(\boldsymbol{x}_m)\}^2 + \frac{\lambda}{2}\boldsymbol{w}^\top\boldsymbol{w}\,, \tag{10}$$

where $\lambda > 0$ is the **regularization coefficient** that controls the relative importance of the data-dependent error $E_D(\boldsymbol{w})$ and the regularization term. Because the regularizer is quadratic, we can calculate the closed-form normal equations by minimizing w.r.t. $\boldsymbol{w}$

$$\boldsymbol{w} = (\lambda\boldsymbol{I} + \boldsymbol{\Phi}^\top\boldsymbol{\Phi})^{-1}\,\boldsymbol{\Phi}^\top\boldsymbol{t}\,. \tag{11}$$

For an orthogonal design matrix, we see that this equation reduces to $\boldsymbol{w} = \boldsymbol{w}_{\mathrm{ML}}/(1+\lambda)$. Thus, in sequential learning algorithms trained with a quadratic regularizer, weight values are encouraged to decay towards zero, unless supported by the data.

2.3 Interactive examples

2.3.1 Linear Modeling with Gaussians

First we look at a simple example of trying to fit noisy data generated around an underlying sinusoidal function. In generating the data, we can select the number of (x, y) points, whether the inputs $\boldsymbol{x} \in$ are uniformly or randomly sampled between 0 and 1, and the width of the Gaussian noise added to the corresponding $\boldsymbol{y} = \sin(2\pi\boldsymbol{x})$.

- We set up the basis functions with the bias parameter $\phi_0 = 1$ and Gaussians $\phi_{1\ldots M}$ of the form in Eq. 2, setting $\mu_j = x_m$ (i.e. one basis function centered around every input point) and fixing the width s to be uniform for each.

- For a linear combination of basis functions, we calculate the weights for fitting the data using the total error function from Eq. 10. In particular we solve the normal equations with Eq. 11, allowing for a quadratic regularization term with parameter λ.

- Even without any noise added to the data, we see that the un-regularized weighted sum of Gaussians will not predict the underlying function very well without making

the widths relatively large. Random sampling of $\boldsymbol{x}$ either leads to bias or large weights.

- With a relatively small regularization parameter, the noiseless data can be fit to predict the underlying function. Adding noise to the data leads to overfitting, but this can be mitigated by increasing the regularization (i.e. bias-variance tradeoff).

2.3.2 Gradient Descent

In this notebook from section IV of Ref. [3], we will visualize what different gradient descent methods are doing using some simple surfaces. From the onset, we emphasize that doing gradient descent on the surfaces is different from performing gradient descent on an error function. The reason is that we want to find good minima that generalize well to new data.

- Let us look at the dependence of stochastic gradient descent from Eq. 8 on the learning rate η for a simple quadratic minima of the form $z = ax^2 + by^2 - 1$. We can also study the effects of adding random noise to the gradients.

- The trajectory converges to the global minima in multiple steps for small learning rates. Increasing the learning rate further causes the trajectory to oscillate around the global minima before converging or to diverge from the minima.

- One problem with gradient descent is that it has no memory of where the "ball rolling down the hill" comes from. This can be an issue when there are many shallow minima in our landscape. The lack of memory is equivalent to having no inertia or momentum.

- We can add a memory or momentum term to the stochastic gradient descent, such that $\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} - \boldsymbol{v}^{(\tau)}$ for $\boldsymbol{v}^{(\tau)} = \gamma \boldsymbol{v}^{(\tau-1)} + \eta \nabla E_m$. The momentum parameter $\gamma < 1$ controls the typical memory lifetimes of the gradient, $(1-\gamma)^{-1}$.

3 Linear Models for Classification

The goal in **classification** is to take an input vector $\boldsymbol{x}$ and to assign it to one of K discrete classes C_k where $k = 1, \ldots, K$. The input space is divided into **decision regions** whose boundaries are called **decision surfaces**. Linear models for classification have decision surfaces defined by $(D-1)$-dimensional hyperplanes within the D-dimensional input space.

- In contrast to regression, the target variable $\boldsymbol{t}$ is used to represent class labels. For probabilistic models, binary classification problems use a single target variable which takes a value $t = 1$ to represent class $\mathcal{C}_1$ and $t = 0$ to represent class $\mathcal{C}_2$.

- The simplest approach to classification involves a **discriminant function** which directly assigns each vector $\boldsymbol{x}$ to a class. Alternatively, **probabilistic models** first infer the conditional probabilities $p(\mathcal{C}_k|\boldsymbol{x})$ before making optimal classification decisions.

- In order to predict discrete class labels or posterior probabilities, we need to transform the linear functions of the weights $\boldsymbol{w}$ using a generally non-linear **activation function** $f(\cdot)$, such that model predictions are $\hat{y} = f(\boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}))$ for basis functions $\boldsymbol{\phi}(\boldsymbol{x})$.

- For binary classification with a trivial activation function, $\hat{y} = \boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x})$, the simplest linear discriminant assigns $\boldsymbol{x}$ to $\mathcal{C}_1$ for $\hat{y} \geq 0$ and $\mathcal{C}_2$ otherwise. The bias w_0 defines the normal distance from the origin to the decision surface and $\boldsymbol{w}_{j>0}$ gives the orientation.

3.1 The perceptron algorithm

The perceptron is a linear discriminant model for binary classification with a nonlinear activation function which is a step function $f(a)$ such that $f(a) = \pm 1$ for $a \geq (<) 0$. The corresponding error function is the **perceptron criterion** given by

$$E_P(\boldsymbol{w}) = -\sum_{m \in \mathcal{M}} \boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}_m) t_m \,, \tag{12}$$

where $\mathcal{M}$ denotes the set of all misclassified patterns and the target values correspond to $t = \pm 1$ for class $\mathcal{C}_{1,2}$. The contributions to the error are linear functions of $\boldsymbol{w}$ in regions of $\boldsymbol{w}$ space where the pattern is misclassified. We apply stochastic gradient descent such that

$$\boldsymbol{w}^{(\tau+1)} = \boldsymbol{w}^{(\tau)} + \eta \boldsymbol{\phi}(\boldsymbol{x}_m) t_m \,. \tag{13}$$

Because the perceptron function $\hat{y}$ is unchanged if we multiply $\boldsymbol{w}$ by a constant, we can set the learning rate $\eta = 1$. If we consider the effect of a single update, we see that the contribution to the error from a misclassified pattern will be reduced

$$-\boldsymbol{w}^{(\tau+1)\top} \boldsymbol{\phi}(\boldsymbol{x}_m) t_m = -\boldsymbol{w}^{(\tau)\top} \boldsymbol{\phi}(\boldsymbol{x}_m) t_m - (\boldsymbol{\phi}(\boldsymbol{x}_m) t_m)^\top \boldsymbol{\phi}(\boldsymbol{x}_m) t_m < -\boldsymbol{w}^{(\tau)\top} \boldsymbol{\phi}(\boldsymbol{x}_m) t_m \,, \tag{14}$$

where we have used the Euclidean norm $||\boldsymbol{\phi}(\boldsymbol{x}_m) t_m||^2 > 0$. This does not imply that the contributions from the other misclassified patterns will have been reduced and also note that some previously correctly classified patterns can become misclassified.

Figure 3: From Section 4.1 of Ref. [2]. Illustration of the convergence of the perceptron learning algorithm, showing data points from two classes in a two-dimensional feature space. The top left plot shows the initial parameter vector $\boldsymbol{w}$ shown as a black arrow together with the corresponding decision boundary, in which the arrow points towards the decision region associated with the red class. The circled data point is misclassified and so its feature vector is added to the weight vector, giving the new decision boundary shown in the top right plot.

3.2 Logistic regression

If we want to consider a probabilistic approach to binary classification using linear models, we can start with the posterior probability defined by by Bayes' theorem for class $\mathcal{C}_1$ given input data $\boldsymbol{x}$

$$p(\mathcal{C}_1|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\boldsymbol{x}|\mathcal{C}_1)p(\mathcal{C}_1) + p(\boldsymbol{x}|\mathcal{C}_2)p(\mathcal{C}_2)} = \sigma(a) \,, \tag{15}$$

where the **logistic sigmoid** function is defined as

$$\sigma(a) = \frac{1}{1 + \exp(-a)} \,, \quad \text{with} \quad a = \frac{p(\boldsymbol{x}|\mathcal{C}_1)p(\mathcal{C}_1)}{p(\boldsymbol{x}|\mathcal{C}_2)p(\mathcal{C}_2)} \,. \tag{16}$$

For class-conditional probabilities $p(\boldsymbol{\phi}(\boldsymbol{x})|\mathcal{C}_k)$ which are Gaussian in the feature space defined by the basis functions $\boldsymbol{\phi}$ with the corresponding distributions sharing the same covariance matrix, we can write the posterior probability for $\mathcal{C}_1$ as

$$p(\mathcal{C}_1|\boldsymbol{\phi}(\boldsymbol{x})) = \hat{y} = \sigma(\boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x})) \,. \tag{17}$$

We now use maximum likelihood to determine the parameters of the logistic regression model. For a dataset $\{\boldsymbol{\phi}(\boldsymbol{x}_m), t_m\}$ with $t_m \in \{0, 1\}$ and $m = 1, \ldots, M$, the likelihood function can be written

$$p(\boldsymbol{t}|\boldsymbol{X}, \boldsymbol{w}) = \prod_{m=1}^{M} \hat{y}_m^{t_m} \{1 - \hat{y}_m\}^{1-t_m} \,, \tag{18}$$

where $\boldsymbol{t} = (t_1, \ldots, t_M)^\top$, $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M)^\top$ and $\hat{y}_m = p(\mathcal{C}_1|\boldsymbol{\phi}(\boldsymbol{x}_m))$. We can then define an error function by taking the negative logarithm of the likelihood, which gives the **cross-entropy** error function in the form

$$E(\boldsymbol{w}) = -\ln p(\boldsymbol{t}|\boldsymbol{X}, \boldsymbol{w}) = -\sum_{m=1}^{M} \{t_m \ln \hat{y}_m + (1 - t_m) \ln(1 - \hat{y}_m)\} \,, \tag{19}$$

where $\hat{y}_m = \sigma(a_m)$ and $a_m = \boldsymbol{w}^\top \boldsymbol{\phi}(\boldsymbol{x}_m)$. Taking the gradient of the error function w.r.t $\boldsymbol{w}$,

$$\nabla E(\boldsymbol{w}) = \sum_{m=1}^{M} (\hat{y}_m - t_m) \boldsymbol{\phi}(\boldsymbol{x}_m) \,, \tag{20}$$

the same form as the gradient of the sum-of-squares error function for linear regression. There is no longer a closed-form solution for $\boldsymbol{w}$ due to the nonlinearity of the logistic sigmoid function. We thus consider gradient descent using the **Newton-Raphson** method,

$$\boldsymbol{w}^{(\text{new})} = \boldsymbol{w}^{(\text{old})} - \mathbf{H}^{-1} \nabla E(\boldsymbol{w}) \,, \tag{21}$$

where $\mathbf{H}$ is the Hessian matrix whose elements comprise the second derivatives of $E(\boldsymbol{w})$ w.r.t. the components of $\boldsymbol{w}$. Note that, unlike logistic regression, the sum-of-squares error function is quadratic and hence the method gives the exact solution for $\boldsymbol{w}$ in one step.

3.3 Interactive examples

3.3.1 Perceptron and non-linear basis functions

In order to get a sense of how linear models for classification work in practice, we will first look at a simple example with synthetic data before moving to a more "realistic" case. In particular, we want to show a relatively straight-forward implementation of the perceptron algorithm and demonstrate how non-linear basis functions can yield better performance.

- First we set up data for two classes centered around Gaussians in 2D coordinate space. We also arbitrarily initialize a 1D decision boundary defined by the weight vector. If the classes are **linearly separable** then the perceptron algorithm should converge.

- We see that as the weight vector is updated whenever a misclassified data point is found, a correct decision boundary is found in a finite, perhaps large, number of steps. If the classes are not linearly separable then the algorithm never converges.

- Now we set up data almost guaranteed to not be linearly separable by putting the Gaussian generating the second class between the two Gaussians generating the first. We use non-linear basis functions to get separable data in the feature space.

- We then project the decision region for the second class from the feature space back into the original coordinates. We can see that well-chosen non-linear basis functions allow for effectively non-linear decision boundaries between classes in the input space.

3.3.2 Logistic regression for new physics at LHC

This notebook will serve as an introduction to logistic regression as well as the second version of the TensorFlow library for machine learning from Google. We will also learn how to use the versatile Pandas package for handling data. Throughout, we will work with the SUSY dataset, available from the UCI Machine Learning Repository.

- First, we can have a look at the data for the kinematic features associated with the SUSY signal and SM background events simulated at the LHC. Note that all of the kinematic distributions have been rescaled to be dimensionless and $\mathcal{O}(1)$.
- We will run logistic regression on the SUSY data for both the simple features (first 8 features) and the full feature space. We will also investigate the use of weight decay by testing the results as function of the regularization parameter α.
- Using just the kinematic variables of the final state objects there is no need for regularization and in fact as we turn off the relative weights of the variables we lose discrimination power. Thus, variance in the data is low and bias causes underfitting.
- We notice only modest improvement when trained on the full feature space, so our logistic regression model is already doing a good job of capturing the information present in the complex variables using only the low-level variables as input.

4 Neural Networks for Regression

We extend the linear models for regression and classification from previous sections by making the non-linear basis functions dependent on parameters which can be adjusted along with the weights during training. For neural networks, each basis function is itself a nonlinear function of a linear combination of the inputs, where the coefficients are adaptive parameters.

- The **hidden units** which characterize **feed-forward** neural networks are defined by the activation function $z_j = h(a_j)$ with activation, $a_j = \boldsymbol{w}^{(1)\top}\boldsymbol{x}$, which is a linear combinations of input variables or hidden units from a previous layer of the network.
- Similar to how the outputs of linear and logistic regression are functions of linear combinations of the inputs, the outputs of a neural network are functions of linear combinations of hidden units, with $\hat{y}_k = f(a_k)$ for output unit activations $a_k = \boldsymbol{w}^{(2)\top}\boldsymbol{z}$.
- Focusing on the simplest case of a **multilayer perceptron** (MLP) with an input layer, a single hidden layer and an output layer, we can compactly write the outputs

$$\hat{y}_k = f\left(\sum_{j=0}^{n} w_{kj}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)\right) , \tag{22}$$

  for a network with n hidden units and D-dimensional input space. Also note the bias terms of the input and hidden layers are defined such that $x_0 = 1$ and $z_0 = 1$.

- Compared to the perceptron, the neural network uses continuous sigmoidal nonlinearities in the hidden units, making the neural network differentiable w.r.t. the network parameters and allows for the use of gradient descent methods for optimization.

Figure 4: From Chapter 10 of Ref. [4]. Architecture of a multilayer perceptron with two inputs, one hidden layer of four neurons, and three output neurons.

4.1 Network training and backpropagation

There is a natural choice of both output unit activation function and matching error function, according to the type of problem being solved. Following the example of linear regression, we use linear outputs and a sum-of-squares error for the MLP, such that

$$\hat{y}_k = a_k \,, \quad E(\boldsymbol{w}) = \frac{1}{2} \sum_{m=1}^{M} \left[\sum_k \{t_{mk} - \hat{y}_{mk}\}^2 \right] \,, \tag{23}$$

for M examples in the training set with $\hat{y}_{mk} = \hat{y}_k(\boldsymbol{x}_m, \boldsymbol{w})$ and target variables t_{mk}. Since $\hat{y}_{mk}$ is a non-linear function of $\boldsymbol{w}$, we must use **error backpropagation** to calculate the gradients in network parameter space. The partial derivative w.r.t. a weight w_{ji} is

$$\frac{\partial E_m}{\partial w_{ji}} = \frac{\partial E_m}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} = \delta_j z_i \,, \tag{24}$$

where we are assuming a generalized activation $a_j = \sum_i w_{ji} z_i$ and z_i could be an input or a hidden unit associated with an activation function $z_j = h(a_j)$. We can calculate necessary gradients for one training example using the chain rule by starting with the prediction error

$$\delta_k = \hat{y}_k - t_k \,, \tag{25}$$

where we have dropped the example index m . The corresponding errors for the hidden units can be obtained by propagating the errors backwards from units higher up in the network. Another application of the chain rule gives the backpropagation formula

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k \,. \tag{26}$$

Because we already know the errors for the output units, it follows that by recursively applying we can evaluate the errors for all of the hidden units in a feed-forward network, regardless of its topology. The algorithm for training a neural network can be summarized

1. Forward propagation of input points $\boldsymbol{x}_m$ using Eq. 22
2. Calculation of the output errors using Eq. 25
3. Backpropagation of errors using Eq. 26
4. Evaluation of the gradient using Eq. 24
5. Adjust $\boldsymbol{w}$ using gradient and repeat

4.2 Interactive examples

4.2.1 MLP with single hidden layer

We return to our example from linear regression in which we had tried to fit noisy data around an underlying sinusoidal function with Gaussian basis function centered at each point in the training data. We now try to improve this fitting procedure by incorporating a simple feed-forward network optimized by gradient descent.

- We can work through the same variations of the dataset that we had before, with the x-coordinates uniformly or randomly distributed and with noise in the y-coordinates either turned off or turned on for varying noise widths.
- We implement the training algorithm for neural networks outlined above for a MLP with one input, a single hidden layer with tanh activation functions and a linear output. The derivatives for stochastic gradient descent backpropagate from a MSE function.
- In addition to seeing how well the trained neural network fits the data and matches the underlying sinusoidal, we also plot the evolution of the weights as the network trains and the contributions to the output from the hidden units at various iterations.
- We can deduce that the neural network is able to reconstruct the underlying sinusoidal function despite significant noise in the data with a small number of parameters relative to the linear model with Gaussian basis functions.

4.2.2 MLP for housing prices in California

Following Ref. [4], we look at 1990 census data to try and predict the median price of a house across various districts in California. We will take the dataset from the scikit-learn library, explore the data with Pandas and use Keras (included within TensorFlow) to train and test the MLP. Note that this regression problem has also been used for kaggle competitions.

- We can put the data into a Pandas dataframe to look at the statistics, distributions and correlations of the data. Features of the dataset include median income (units \$10.000), median house age etc. The target is the median house value (units \$100.000).
- After splitting the data, we first try the simple MLP architecture used in the previous example. We see that even with only a few hidden units in a single hidden layer, stochastic gradient descent is inefficient and the MLP does not fit the training data.
- Simply switching to **mini batch** optimization both speeds up the training and yields a much better fit to the training data. For further improvement, we can add more hidden layers with more hidden units to increase the capacity of the model.
- At some point increased complexity either makes the number of parameters in the model computationally expensive to train or you wind up overfitting the training data leading to increasing generalization error when evaluating with the test set.

5 Neural Networks for Classification

While a more general outline for classification (and regression) using neural networks can be found in the previous section, there are a few odds and ends which we have not discussed in detail that are important for but not limited in relevance to neural network classification. The following summarizes parts of Chapters 4 and 5 from Ref. [2] and Chapter 7 of Ref. [1].

5.1 Multiclass classification

Recall the goal in classification is to take an input vector $\boldsymbol{x}$ and to assign it to one of K discrete classes C_k, where $k = 1, \ldots, K$, such that the target variable $\boldsymbol{t}$ represents class labels. For $K > 2$ classes, it is convenient to use a **1-of-K coding** scheme in which

$$\boldsymbol{t} = (0, 1, 0, 0, 0, \ldots)^\top \tag{27}$$

is a vector of length K such that if the labeled class is C_j, then all elements t_k of $\boldsymbol{t}$ are zero except element t_j, which takes the value 1 (e.g. $j = 2$ above). We can interpret the value of t_k as the probability that $\boldsymbol{x}$ is in the class is C_k, which from Bayes' theorem we define as

$$p(\mathcal{C}_k|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{\sum_j p(\boldsymbol{x}|\mathcal{C}_j)p(\mathcal{C}_j)} = \frac{\exp(a_k)}{\sum_j \exp(a_j)}\,. \tag{28}$$

This distribution is known as the **softmax function** and can be regarded as a multiclass generalization of the logistic sigmoid we had defined for binary classification. The softmax function yields $p(\mathcal{C}_k|\boldsymbol{x}) \simeq 1$ and $p(\mathcal{C}_j|\boldsymbol{x}) \simeq 0$ if $a_k \gg a_j$ for all $j \neq k$ with the activation

$$a_k = \ln p(\mathcal{C}_k|\boldsymbol{x})p(\mathcal{C}_k)\,. \tag{29}$$

For the specific case of neural networks, the activations correspond to the a linear combination of outputs from the final hidden layer. Given a dataset with M examples and an $M \times K$ matrix $\boldsymbol{T}$ of target variables, the corresponding likelihood function is

$$p(\boldsymbol{T}|\boldsymbol{X}, \boldsymbol{w}) = \prod_{m=1}^{M}\prod_{k=1}^{K} p(\mathcal{C}_k|\boldsymbol{x}_m)^{t_{mk}} = \prod_{m=1}^{M}\prod_{k=1}^{K} \hat{y}_{mk}^{t_{mk}}\,, \tag{30}$$

where $\hat{y}_{mk} = \hat{y}_k(\boldsymbol{x}_m, \boldsymbol{w})$ is the output of neural network (e.g. for an MLP with a single hidden layer in Eq. 22) with the activation function given by the softmax function. We define the cross-entropy error function by taking the negative logarithm of the likelihood

$$E(\boldsymbol{w}) = -\ln p(\boldsymbol{T}|\boldsymbol{X}, \boldsymbol{w}) = -\sum_{m=1}^{M}\sum_{k=1}^{K} t_{mk} \ln \hat{y}_{mk}\,, \tag{31}$$

which is similar to the cross-entropy error function for binary classification (e.g. for logistic regression in Eq. 19). Using the technique of backpropagation described in the previous section, minimizing the error function w.r.t. $\boldsymbol{w}$ corresponds to maximizing the likelihood.

5.2 Dropout regularization

In addition to methods such as **parameter regularization** (e.g. weight decay) and **early stopping**, overfitting of neural networks can be mitigated by the random **dropout** of nodes during training. Dropout thus effectively trains the **ensemble** of models consisting of all subnetworks that can be formed by removing non-output units, as shown in Fig. 5.

Figure 5: From Section 7.12 of Ref. [1]. Dropout trains an ensemble consisting of all subnetworks that can be constructed by removing nonoutput units from an underlying base network. Here, we begin with a base network with two visible units and two hidden units. There are sixteen possible subsets of these four units. We show all sixteen subnetworks that may be formed by dropping out different subsets of units from the original network.

- For a minibatch-based learning algorithm that makes small steps, such as stochastic gradient descent, each time we load an example into a minibatch, we randomly sample a different binary mask to apply to all the input and hidden units in the network.
- The mask for each unit is sampled independently from all the others and is not a function of other model parameters or data. The probability of sampling a mask value of zero (i.e. excluding the unit) is the **dropout rate** fixed before training begins.
- Rather than sampling all possible subnetworks, a tiny fraction are each trained for a single step of gradient descent, and the parameter sharing causes the remaining subnetworks to arrive at good settings of the parameters.
- To make sure that the expected input to a unit at test time is roughly the same as the expected input to that unit at train time, **weight scaling** requires the division (multiplication) of the weights by the **keep probability** during (after) training.

5.3 Interactive examples

5.3.1 MLP for noisy and overlapping classes

For synthetic datasets similar to those used for the perceptron classification notebook, we apply both logistic regression and a simple MLP with one hidden layer in order to demonstrate the importance of non-linearity when training on noisy data with classes that overlap in the feature space. We also investigate different classification thresholds.

- First consider logistic regression to classify a dataset with overlapping but somewhat well-separated classes. We see that the misclassification rate is relatively low after a short computation time for the decision boundary which maximizes the likelihood.
- If we make the classes in the dataset overlap more, then the linear decision boundaries possible with logistic regression will have a high misclassification rate regardless of classification threshold. This is the fundamental limitation of linear models.
- Now if we train the simple MLP with a single hidden layer containing only two hidden nodes, the resulting non-linear decision boundary will have a lower misclassification rate. We can also look at how well the neural network generalizes to new data.

- If we increase the capacity of the neural network by increasing the number of hidden nodes, we see that the generalization to new data can become worse. Specifically, the number of false positives can become large depending on the classification threshold.

5.3.2 MLP with dropout for MNIST images

Following the notebook from Chapter 9 of Ref. [3], we will use Keras (included within TensorFlow and also used for an example in the previous section) to train and test MLPs with dropout regularization for classification of the MNIST dataset of handwritten digits. We will also investigate how to tune the hyperparaemters of the MLPs to maximize performance.

- First we load the 2D image pixel data into 1D arrays and unit normalize the pixel values. Then we define the MLP multiclass classifier with two dense hidden layers and dropout normalization with tunable optimizer, learning rate and dropout rate.
- Next we train the model using stochastic gradient descent for optimization and default hyperparameters. The training and validation losses suggests good performance, while evaluating the model on the test data shows the generalization is quite good already.
- To improve performance, first we look at how the metrics on the training data change for different optimizers (e.g. Adam) and learning rates. We see that Adam typically outperforms SGD and the performance is more stable for different learning rates.
- After fixing Adam as the optimizer, we implement a grid search to find the best combination of learning rate and dropout rate. Increasing the dropout rate limits the capacity to fit the training data but can improve the generalization of the model.

6 Decision Trees for Regression

We now consider decision trees and ensembles thereof, largely following the discussion in Chapters 6 and 7 of Ref. [4]. While the ensemble techniques described in this section and the next can in principal be used for any type of algorithm, decision trees are particularly well-suited due to the characteristically high variance associated with individual trees.

- Given a set of input features $\boldsymbol{x}$, a decision tree is trained to predict target values $\boldsymbol{t}$ by recursively slicing the feature space into regions characterized by samples with similar targets. Typically, this slicing will involve setting a threshold θ for a given feature.
- Starting with the **root** which considers all of the training samples, each **split node** in the tree further subdivides the feature space in a way which minimizes the error function for the predictions associated with the reduced dataset at each new node.
- A **leaf node** is defined when further subdivisions of the feature space do not sufficiently reduce the error or regularization conditions are met. For instance, trees can have a maximum depth or a minimum number of samples associated with each node.
- In contrast to algorithms in previous sections, the minimization of the error is not associated with maximization of the likelihood under some assumptions (e.g. Gaussian noise). In principal, decision trees thus tend to have lower bias and higher variance.

Figure 6: From Section 14.4 of Ref. [2]. Example of binary tree (left) partitioning 2D feature space (right) through a recursive application of thresholds starting at the root node (θ_1), continuing through the split nodes ($\theta_{i>1}$) and ending at the leaf nodes (A,B,C,D,E).

6.1 CART algorithm

For simplicity, consider the a binary decision tree for regression with a single target variable t. The prediction and error of the tree at each node are defined by the mean and variance of the target values associated with the subdivision of the dataset of size M_{node} at that node,

$$\hat{y}_{\text{node}} = \frac{\sum_{m\in\text{node}} t_m}{M_{\text{node}}}\,, \quad MSE_{\text{node}} = \frac{\sum_{m\in\text{node}} (\hat{y}_{\text{node}} - t_m)^2}{M_{\text{node}}}\,. \tag{32}$$

The Classification and Regression Tree (CART) algorithm recursively grows decision trees from the root node by selecting the feature $k \in \boldsymbol{x}$ and associated threshold θ_k at each node which minimizes the weighted sum of the errors for the predictions of the new nodes,

$$E_{CART}(k, \theta_k) = \frac{M_{\text{left}}}{M} MSE_{\text{left}} + \frac{M_{\text{right}}}{M} MSE_{\text{right}}\,, \tag{33}$$

where left and right nodes are produced by the split of the current node. Note that CART is an example of a **greedy algorithm**, such that it optimizes for the split at each individual node without considering the error of predictions for nodes at deeper layers of the tree.

6.2 Ensemble models

The predictions from a random ensemble of decision trees will tend to have smaller variance compared to an individual tree without increasing the bias too much. A simple way to generate such an ensemble is to train the same model on random samples of the training data. Using these ensemble techniques, the predictions from each individual tree are aggregated.

- For **bagging**, the random samples of the training data are selected with replacement, such that a single decision tree could be trained multiple times on the same data point. Ensemble sampling performed without replacement is refereed as **pasting**.

- Relative to training an individual decision tree, a single decision tree in the ensemble can be trained much more quickly if the random sample is much smaller than the entire training dataset. The ensemble can also been trained in parallel.

- The aggregated prediction for regression trees is typically just the average of the ensemble. For classification, the aggregation function is typically the statistical mode, although the classification scores from each tree can also be averaged.

- As the number of trees grows, the variance of the ensemble shrinks and the bias does not increase when the decision trees are completely uncorrelated. **Random forests** decrease correlations by splitting each tree node using random subsets of features.

Similar to the algorithms trained by iterative methods such as gradient descent described in previous sections, ensembles of decision trees can also be trained sequentially, with each additional tree improving the performance of the whole ensemble. In this section we focus on the application of **gradient boosting** to decision trees trained for regression.

- Unlike the ensemble methods described above, boosting algorithms typically evaluate the performance of previously grown trees on the training data before making corresponding adjustments to the training of the next tree which is added to the ensemble.
- Gradient boosting is a distillation of this principal, with each successive tree in the ensemble trained on the **residual** error between the predictions of the ensemble and the training data, rather than the training data used only to train the first tree.
- After training the $j = 0$ tree as one would in a model with an individual decision tree, the $j > 0$ trees are trained on datasets with the same features but with targets $t_m^{(n)} = t_m^{(0)} - \sum_{j=0}^{n-1} \hat{y}_j$ for the mth example and nth update of the target variable.
- Gradient boosting can be regularized by reducing the number of trees added to the ensemble $n_{\max}$ or by scaling the contributions to the prediction of the ensemble from $\hat{y}_{j>0}$ by a learning rate $\eta < 1$ such that the aggregated prediction is $\hat{y} = \hat{y}_0 + \eta \sum_{j=1}^{n_{\max}} \hat{y}_j$.

6.3 Interactive examples

6.3.1 Decision trees, bagging and gradient boosting

We return to our example from linear and neural network regression in which we had tried to fit noisy data around an underlying sinusoidal function. We look at three models discussed in this section, individual decision trees, bagged trees and gradient boosted trees. For the ensemble models, we will see how the individual predictions aggregate.

- Note that the data must be sorted by feature in order to efficiently determine the node splitting thresholds. That aside, we have the same set up with Gaussian noise added to an underlying sinusoidal function and features sampled uniformly or randomly.
- Focusing on the case with random feature sampling and significant noise, we see the individual tree will overfit the data without regularization. There is a more reasonable fit when the depth of the tree is limited or the minimum node samples is higher.
- The bagged ensemble works well so long as the individual trees are not too biased and the random data samples are not too correlated. In case of the latter, the bagged model can still work reasonably well with sufficient regularization of the trees.
- The gradient boosted tree will overfit the data without regularization, even if the individual trees are regularized. Sufficiently regularized gradient boosting can work well even with biased individual trees which do not work well for bagging.

6.3.2 Random forests for housing prices in California

Turning back to the census data on which we had trained neural networks for regression in previous examples, we now follow Ref. [4] to implement both individual decision trees and random forests for regression using scikit-learn. We also show how decision trees and ensembles thereof allow for a relatively transparent measure of feature importance.

- We first train a single decision tree using the full set of 8 input features. We again can see a significant reduction of the error on the predictions in the validation dataset once the tree is sufficiently regularized. We can also see the most important features.

- Feature importance in scikit-learn for a tree or ensemble thereof is given by the sum of the reductions in training error (MSE by default for regression) associated with all nodes split by a given feature, weighted by the number of examples in each split node.
- We can see a slight improvement in performance on the validation training set for a similar decision tree, but only trained on the 4 most important features determined by the training of the first tree. We can also visualize a small version of the tree.
- A random forest with well-regularized individual trees yields performance similar to the most complex neural networks from the previous example. Performance improves with less regularization since the total variance is suppressed for random ensembles.

7 Decision Trees for Classification

We continue the discussion from the previous section, now focused on decision trees and ensembles thereof for classification. Although many of the concepts from regression trees can be directly applied and classification perhaps seems like a more natural application of decision trees, there are several important differences worth pointing out.

- As with other classification problems, the target values for each data point can be defined by a 1-of-K coding scheme for K discrete classes such that e.g. $\boldsymbol{t} = (0, 1, 0)$ would label a data point belonging to the 2nd of 3 classes present in a data set.
- The predicted values of each node in the tree are thus the respective probabilities of an associated data point belonging to each of the classes. Typically, these probabilities are given by the fractions of the training data represented for each class.
- The CART algorithm attempts to minimize the class impurity of additional nodes when evaluating a potential node split. Unlike the regression examples we considered, further splitting does not necessarily minimize the associated error function.
- For probabilities p_k of belonging to certain classes, the class impurity of a node can be defined by the **Gini impurity**, $G = 1 - \sum_k p_k^2$, and the error function for a potential split is the sum of the impurities weighted by the fraction of events in each new node.

Figure 7: From Chapter 6 of Ref. [4]. Example of binary classification tree (top) partitioning a 2D feature space (bottom) of the Iris dataset. The maximum tree depth of the tree (top) is fixed at 1, while the plot (bottom) shows additional partitioning down to a depth of 2.

7.1 Adaptive boosting

In addition to gradient boosting described for regression, adaptive boosting (AdaBoost) is also a commonly used sequential learning technique for ensembles. After the jth tree is grown, the weighted error is calculated as a sum over the misclassified example weights,

$$r_j = \sum_{m \in \mathcal{M}} w_m^{(j)}\,, \tag{34}$$

where $\mathcal{M}$ denotes the misclassified examples and the the initial weights for all data points in the training set are fixed to be equivalent $w_m^{(1)} = 1/M$. The weight of the contribution from the jth tree to the aggregated prediction of the ensemble is then given by

$$\alpha_j = \eta \ln \frac{1 - r_j}{r_j} \,, \tag{35}$$

where η is the learning rate. Before the next tree of the ensemble is grown, the AdaBoost algorithm updates the example weights to focus the subsequent training on misclassified examples, with $w_m^{(j+1)} = w_m^{(j)} \exp(\alpha_j)$ for $m \in \mathcal{M}$ and $w_m^{(j+1)} = w_m^{(j)}$ for $m \notin \mathcal{M}$.

7.2 Interactive examples

7.2.1 Decision trees, bagging and AdaBoost

We return to our example from linear and neural network classification in which we had tried to classify a 2D feature space with examples drawn from overlapping Gaussians associated with two classes. We look at three models discussed in this section, individual decision trees, a bagged ensemble of trees and boosted ensemble of trees implemented with AdaBoost.

- Note the implementation of individual decision trees has been changed relative to the regression tree notebook for clarity but the functions work the same way. Also, an additional set of functions were needed for the weighted datasets used in AdaBoost.

- Focusing on the case with significant overlap between the two classes, we see the individual tree performs relatively well when sufficiently regularized. We can also nicely visualize the structure of the decision tree if it is not too large.

- The bagged ensemble tends to wash out the sharp decision boundaries present for well-regularized individual tree. This tends to generalize poorly for overlapping classes. Well-separated classes generalize better but then an individual tree is sufficient.

- The AdaBoost tree ensemble can perform well with sufficient regularization, even if the individual trees are not well-regularized. If the individual decision trees perform poorly, then more of the ensemble contributes to the aggregated class prediction.

7.2.2 XGBoost for new physics at LHC

We return to the SUSY dataset previously analyzed using logistic regression, but now we try out the package XGBoost for constructing gradient boosted trees to classify signal and background events at LHC in a notebook adapted from Chapter 8 of Ref. [3]. We will also discuss visualizing feature importance as well as techniques using scikit-learn for optimization.

- We define datasets that contain either 8 low-level kinematic features, 10 high-level features or all 18 features combined. We can see that XGBoost predictions are similarly accurate when trained on any of the 3 datasets but the former takes longer to train.

- In XGBoost, we can calculate the feature importance score (Fscore), which measures how many times each feature was split on. The higher this number, the more fine-tuned the partitions in this direction and presumably the more informative it is.

- We see the performances of the boosted trees trained on different feature sets are similar despite the differences in feature importance. This similarity is clear not only in the model accuracy, but also the receiver operating characteristic (ROC) curves.
- Lastly, we demonstrate how to increase the performance of XGBoost a bit more by using scikit-learn to run a grid search over hyperparameters such as maximum tree depth, minimum number of samples associated with each tree node and learning rate.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[2] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006, https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf.

[3] P. Mehta, M. Bukov, C.-H. Wang, A. G. R. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, "A high-bias, low-variance introduction to Machine Learning for physicists," *Phys. Rept.*, vol. 810, pp. 1–124, 2019.

[4] A. Geron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 3rd ed. O'Reilly Media, Inc., 2023.