ICS5510 Assignment

Carmel Gafa

***Abstract*—TODO: abstract here**

***Index Terms*—TODO: keywords**

I. BACKGROUND

A. Machine learning technique - Neural network

Neural networks are machine learning models inspired by biological neural networks. They comprise layers of interconnected nodes (neurons) that transform input data into meaningful outputs through weights and activation functions. Neural networks are widely used for classification, regression, and generative modelling tasks. This section will look at the important aspects of neural networks.

1) The perceptron: A perceptron is a basic architecture that consists of a layer of input nodes that is fully connected with a layer of output nodes and, therefore, does not have any hidden nodes. Perceptrons can only represent linearly separable functions and are used only for binary classification.

The building blocks of a perceptron are the following:

- The perceptron takes several input features that each represent an aspect of the input data.
- Each input feature differently influences the output through a weight parameter. Weights are optimized during the training process.
- A bias is utilized to make input-independent adjustments to the weighted inputs. Bias values are also optimized during training.
- The weighted inputs are summed together, including the bias; this can be expressed as:

$$z = \sum_{i=1}^{n} w_i x_i + b \tag{1}$$

- The weighted sum is passed through a Heaviside step activation function that maps the weighted sum to a binary output, such that:

$$f(z) = \begin{cases} 0 & \text{if } z < \theta \\ 1 & \text{if } z \geq \theta \end{cases}$$

- The output of the perceptron that results from the activation function can then be used to classify the inputs.

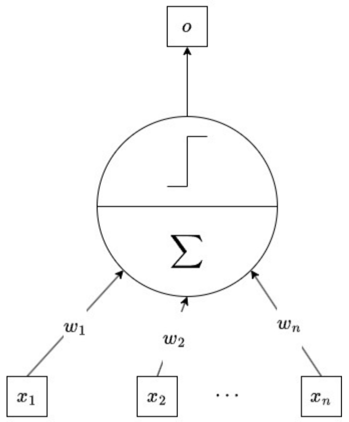


Fig. 1: The perceptron

A simple learning algorithm fits a perceptron to any linearly separable set.

2) Multilayer perceptron: A multilayer perceptron is made up of fully connected layers. It contains an input layer, a number of hidden layers, and an output layer. This structure allows us to model more complex relationships when compared to the simpler perceptron.

The building blocks of a perceptron are the following:

- Each input feature is connected directly to an input neuron to forward the features into the network.
- One, or more than one, hidden layers, each having a designated number of nodes, follow. These layers transform the feature information at each level.
- Finally, an output layer consisting of one node (if the task is regression or binary classification) or more nodes (if the task is multi-class classification) gives us the result of the network.

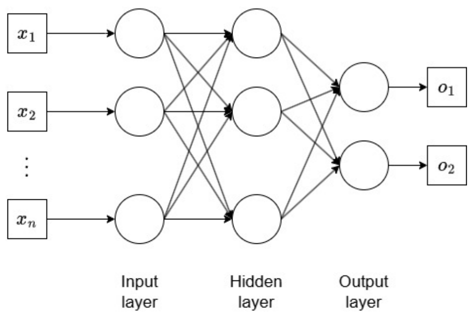


Fig. 2: A multilayer perceptron with a single hidden layer

1) Forward propagation involves feeding feature information to the input layer, passing it through all hidden layers, and finally to the output layer to generate a predicted output. A linear transformation followed by a non-linear activation generates the output for every node at each layer. Forward propagation is comprised of the following steps:

a) For each neuron in layer j, the weighted sum of its inputs is calculated as:

$$z_j = \sum_{i=1}^{n_j} w_{ij}^{(j)} x_i^{(j-1)} + b_j \tag{2}$$

Where:

$x_i^{(j-1)}$	Input from the i^{th} neuron of the previous layer $j-1$.
$w_{ij}^{(j)}$	Weight connecting the i^{th} neuron of layer $j-1$ to the current neuron in layer j.
b_j	Bias term for the neuron in layer j.
z_j	Result of the linear transformation at the current neuron in layer j.
n_j	Number of inputs (or neurons) in the previous layer $j-1$.

b) The result of the weighted sum, z_j, is passed through a non-linear activation function:

$$a_j = f(z_j)$$

This introduces non-linearity to the model. A list of some common activation functions can be seen in the Table I.

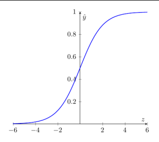
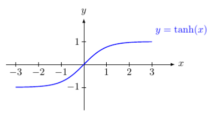
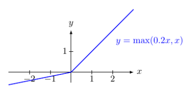
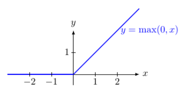
Function	Equation	Diagram
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	
Tanh	$tanh(x)$	
Leaky Relu	$max(0.2x, x)$	
Relu	$max(0, x)$	

TABLE I: Activation Functions

2) The predicted output is compared to the actual output using a loss function to compute the loss. A loss function measures the difference between the actual target values and the model's predicted results. For a single prediction:

$$\text{Loss}(\hat{y}, y) = f(\hat{y}, y)$$

Where:

$\hat{y}$	Predicted value.
y	Actual value.
f	Specific loss function formula.

The cost function aggregates the loss over the entire dataset:

$$\frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}, y)$$

Two examples of cost functions are the following:

- Mean Squared Error (MSE):

$$\text{MSE} = \frac{\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2}{n} \tag{3}$$

- Binary Cross-Entropy: Commonly used for binary classification problems:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right] \tag{4}$$

Where:

L	The average loss (cost) across all N samples in the dataset.
y_i	The true label for the i^{th} sample, where $y_i \in \{0, 1\}$.
$\hat{y}_i$	Predicted probability for $y_i = 1$.
$1-\hat{y}_i$	Predicted probability for $y_i = 0$.
log	The natural logarithm.

3) Backpropagation is the process by which the network's weights and biases are adjusted to minimize loss. The loss is propagated backwards from the output layer to the input layer. At each step, the gradient of the loss with respect to the parameters determines the adjustment. The following steps make up the backpropagation phase:

a) The gradient of the loss with respect to each weight and bias in every node is calculated, starting from the output layer and moving backwards to the input layer.
b) The gradient for any layer builds on the gradient of the subsequent (later) layers.
c) The network's parameters for any node are adjusted in the direction opposite to the respective gradient using optimization algorithms like stochastic gradient descent (SGD) or variants such as the Adam optimizer.

3) Stochastic gradient descent: Stochastic gradient descent is an optimization algorithm that optimizes the parameters Θ of the network, to minimize the loss

$$\Theta = \arg\min_{\Theta} \frac{1}{N} \sum_{i=1}^{n} l(x_i, \Theta)$$

Where:

$x_{1,2,\ldots,N}$	The training set.
$l(x_i, \Theta)$	The loss for a given data point x_i.
N	The number of examples in the dataset.

As Wu noted[2], the loss, z is a supervision signal that guides any modifications that should take place on the parameters. The stochastic gradient descent method of such modification is as follows:

$$\theta^i \leftarrow \theta^i - \eta \frac{\partial z}{\partial \theta^i}$$

where i represents the layer index η is the learning rate $\frac{\partial z}{\partial \theta^i}$ is the gradient of the loss z with respect to parameter θ^i

The parameters are updated iteratively, as follows:

$$\theta^i_{t+1} = \theta^i_t - \eta \frac{\partial z}{\partial \theta^i_t}$$

There are a few considerations that are important to note about stochastic gradient descent.

The learning rate affects the step size for each update, that is how much the parameters are adjusted in the direction of the gradient. A large learning rate might result in overshooting and even divergence, whereas a small learning rate leads to more stable convergence, albeit slower

The parameters are updated for every sample or for a small batch of samples, unlike traditional gradient descent that computes that gradients for the entire dataset. This approach leads to faster updates but is also prone to more noise in the optimization process.

4) Momentum: Momentum is a technique used in training neural networks to accelerate optimization. This accelerated optimization is achieved by considering also the values of previous gradients for each parameter. Momentum can, in this way, mitigate oscillations in the optimization path and speed up learning.

Momentum acts on the update rule of gradient descent by adding a portion of the update from the previous step to the current step. This approach allows the optimizer to maintain a directional memory, or velocity, effectively smoothing out updates and reducing oscillations.

Momentum is commonly integrated with optimization algorithms like SGD and is a key component of advanced optimizers like Adam optimizer in Section I-A6.

The momentum update rule is given by:

$$v^i_t = \beta v^i_{t-1} + (1-\beta) \frac{\partial z}{\partial \theta^i_t}$$

$$\theta^i_{t+1} = \theta^i_t - \eta v^i_t$$

Where:

v^i_t	The velocity term at step t for the parameters at index i.
β	The momentum coefficient (e.g., 0.9), controlling how much of the previous velocity to retain.
$\frac{\partial z}{\partial \theta^i_t}$	The gradient of the loss function z with respect to parameters θ^i at step t.
η	The learning rate.
θ^i_t	The parameters at layer index i at step t.

A typical value is $\beta = 0.9$, which retains 90% of the previous velocity. Smaller values result in less "memory" of past updates.

5) RMSProp: Root Mean Square Propagation (RMSProp) is an adaptive learning rate optimization algorithm designed to improve the training of neural networks. Its main function is to address challenges such as inconsistent or large gradient magnitudes. RMSProp helps stabilize convergence during optimization due to its effectiveness in dealing with noisy gradients.

RMSProp adjusts the learning rate for each parameter based on the average of recent squared gradients. In this way, the optimizer adapts to the scale of the gradients, preventing large updates that could destabilize training.

The algorithm is similar to [[neural-network-momentum]], but here it introduces a running average of squared gradients to normalize parameter updates.

The RMSProp update rule is given by:

$$S^i_t = \beta S^i_{t-1} + (1-\beta) \left(\frac{\partial z}{\partial \theta^i_t} \right)^2 \tag{5}$$

Where:

S^i_t	Exponentially weighted moving average of squared gradients at step t for the parameters at layer index i.
β	Decay rate for the moving average (commonly set to 0.9).
$\left(\frac{\partial z}{\partial \theta^i_t} \right)^2$	Gradient of the loss function z with respect to parameters θ^i_t at step t and layer index i.

The parameters are updated using a learning rate scaled by the square root of S^i_t (with a small value ϵ added for numerical stability):

$$\theta^i_{t+1} = \theta^i_t - \frac{\eta}{\sqrt{S_t + \epsilon}} \left(\frac{\partial z}{\partial \theta^i_t} \right) \tag{6}$$

Where:

η	Global learning rate, which is set to 0.001 in most frameworks.
ϵ	Small constant (e.g., 10^{-8}) to prevent division by zero.

RMSProp is a widely used optimizer that forms the foundation for more advanced methods such as adam optimizer that is discussed in detail in Section I-A6. It is especially effective in training deep neural networks where gradient magnitudes can vary significantly.

6) Adam optimizer: The Adam Optimizer (Adaptive Moment Estimation) is a widely used optimization algorithm for training deep learning models. It combines the benefits of momentum and rmsprop to provide an efficient, robust, and adaptable optimizer. Adam is particularly effective for problems with sparse gradients and non-stationary objectives.

Adam optimizer had a number of hyper parameters:

- Learning Rate α: `0.001`
- First Moment Decay Rate β_1: `0.9`
- Second Moment Decay Rate β_2: `0.999`
- Numerical Stability Term ϵ: `1e-8`

The Adam-optimizer update rule is given by the following steps:

1) First, calculate the first moment estimate, or the Momentum that we discussed in Section I-A4, which is the exponentially weighted moving average of the gradients. This is done to smooth gradient updates over time by including past gradients and thus maintaining a directional memory:

$$m_t^i = \beta_1 m_{t-1}^i + (1-\beta_1)\frac{\partial z}{\partial \theta_t^i}$$

2) Calculate the second moment estimate, which is the basic idea of RMSProp that we discussed in Section I-A5, to track the exponentially weighted variance of gradients:

$$v_t^i = \beta_2 v_{t-1}^i + (1-\beta_2)\left(\frac{\partial z}{\partial \theta_t^i}\right)^2$$

3) Correct these results for initialization bias:

$$\hat{m}_t^i = \frac{m_t^i}{1-\beta_1^t}$$

$$\hat{v}_t^i = \frac{v_t^i}{1-\beta_2^t}$$

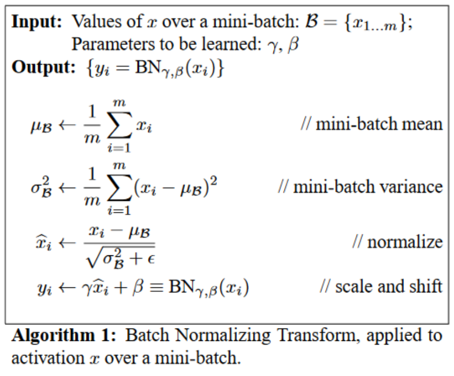
4) Update the parameters using the corrected estimates:

$$\theta_{t+1}^i = \theta_t^i - \frac{\alpha \hat{m}_t^i}{\sqrt{\hat{v}_t^i}+\epsilon}$$

7) Batch normalization: Batch normalization is used to control the statistics of the activations in the neural networks. It is commonly used in neural network architectures, and it is usually found after layers that have multiplications, like linear or convolutional layers

Let us consider a hidden state neural net node

We have basically a pre-activation value, that is the result of



```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
       Parameters to be learned: $\gamma, \beta$
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                       // mini-batch mean
$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$     // mini-batch variance
$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$                 // normalize
$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$          // scale and shift
```

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Fig. 3: Batch normalization algorithm, from [1]

$$z = \left(\sum_{i=1}^{n} x_i w_i\right) + b$$

That is fed into a non-linear element such as RELu or tanh.

- We do not want the pre-activation to be very small as the tanh activation will not effectively activate.
- We also do not want these values to be too large, as the tanh will become saturated.
- Furthermore, we want these values to be roughly Gaussian, that is, with a mean of 0 and a standard deviation of 1.

Sergey Ioffe and Christian Szegedy propose batch-normalization [1] to take the hidden states and normalize them to be Gaussian.

There are two ways whereby batch normalization can be implemented:

- Pre-activation normalization, where batch-normalization is carried out before the activation function is applied.
- Post-activation normalization, where batch-normalization is carried out after the activation function is applied.

The original paper specified pre-activation normalization, however studies have shown that post-activation batch normalization will give superior results.

8) Regularization: Regularization are techniques used to prevent overfitting in machine learning. This is achieved by adding a penalty term to the [[loss-function]], that discourage the model from assigning larger weights to features.

L1 regularization or Lasso Regularization adds the sum of the absolute values of the weights to the [[loss-function]]:

$$\mathcal{L}_{L1} = \mathcal{L} + \lambda \sum_{i=1}^{n} |w_i| \tag{7}$$

Where:

$\mathcal{L}$	The original loss function (e.g., cost-function-mean-square-error, cost-function-cross-entropy).
λ	The regularization strength, a hyperparameter.

w_i	The weight of the i^{th} node.

L1 regularization drives some weights to zero, thus encouraging sparsity in the network. Since L1 penalizes large weights, it promotes simpler networks with improved generalization.

L2 or Ridge regularization adds the sum of the squared values of the weights to the [[loss-function]]:

$$\mathcal{L}_{L2} = \mathcal{L} + \lambda \sum_{i=1}^{n} w_i^2 \tag{8}$$

Where:

$\mathcal{L}$	The original loss function (e.g., cost-function-mean-square-error, cost-function-cross-entropy).
λ	The regularization strength, a hyperparameter.
w_i	The weight of the i^{th} node.

L2 penalizes large weights but does not drive them to zero but shrinks them closer to zero. It hence promotes smoother solutions and avoids sharp changes in parameter values.

B. *Machine learning technique - Logistic regression*

Logistic regression is a machine learning algorithm commonly used for binary classification tasks.

Given a feature vector $X \in \mathbb{R}^{n_x}$, the goal of logistic regression is to predict the probability $\hat{y}$ that a binary output variable y takes the value 1, given X, that is $\hat{y} = P(y = 1|X)$, $0 \le y \le 1$. For example, in the case of image classification, logistic regression can be used to predict the probability that an image contains a cat.

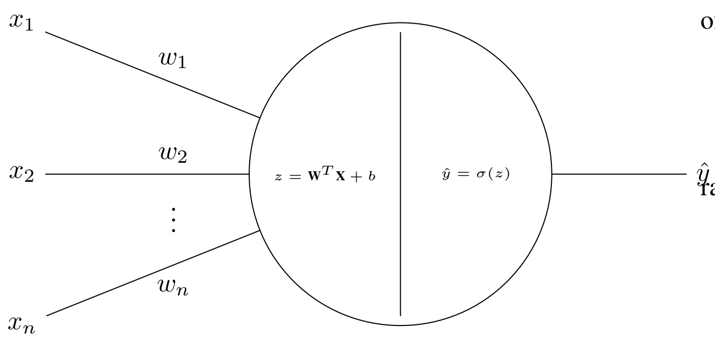


Fig. 4: A logistic regression model

Logistic regression can be visualized as the model shown in Figure 4. It consists of several main components:

Inputs. The input vector to the model $\mathbf{X} \in \mathbb{R}^{n_x}$.
Parameters. A weight vector $\mathbf{W} \in \mathbb{R}^{n_x}$ and a bias term $b \in \mathbb{R}$. These will form the coefficients of a linear equation that gives the log odds ratio.

Pre-activation result: The result is obtained by multiplying the transpose of the weights with the inputs and then adding the bias.

$$z = \mathbf{W}^T\mathbf{X} + b = \begin{bmatrix} w_1 & w_2 & \dots & w_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + b \tag{9}$$

Sigmoid function. A function as shown in Figure 5, $\sigma(z) = \frac{1}{1+e^{-z}}$, which maps any real number z to the range $(0, 1)$. This function is used to ensure that the predicted probability $\hat{y}$ is always between 0 and 1.

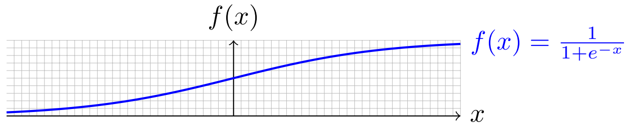


Fig. 5: Sigmoid activation function

Output. The predicted probability $\hat{y}$ is computed as $\hat{y} = \sigma(z) = \sigma(\mathbf{W}^T\mathbf{X} + b)$.

A term that is often encountered in this scenario is the log-odds ratio or logit. Logistic regression models the probability $P(y = 1 \mid X)$, of the binary dependent variable Y given the predictor variables $\mathbf{X} = \{x_1, x_2, \dots, x_n\}$. The goal is to find a relationship between $P(y = 1)$, and the predictors $\mathbf{X}$.

The probability is modelled using the sigmoid function:

$$P(y = 1 \mid X) = \frac{1}{1 + e^{-\eta}}$$

Where: $\eta = b + w_1x_1 + w_2x_2 + \dots + w_nx_n$ is the linear regression function

The odds of $y = 1$ are defined as the ratio of the probability of success to the probability of failure:

$$\text{Odds} = \frac{P(y = 1)}{1 - P(y = 1)}$$

Taking the natural logarithm of the odds gives the [log-odds-ratio]] or logit:

$$\text{Log-Odds} = \ln\left(\frac{P(y = 1)}{1 - P(y = 1)}\right)$$

From the sigmoid function, we can derive the relationship between the probability and the log-odds-ratio:

$$P(y = 1) = \frac{1}{1 + e^{-\eta}}$$

$$1 - P(y = 1) = 1 - \frac{1}{1 + e^{-\eta}} = \frac{e^{-\eta}}{1 + e^{-\eta}}$$

The odds, therefore, are

$$\text{Odds} = \frac{P(y = 1)}{1 - P(y = 1)} = \frac{\frac{1}{1+e^{-\eta}}}{\frac{e^{-\eta}}{1+e^{-\eta}}} = e^{\eta}$$

Taking the natural logarithm of both sides gives the log-odds:

$$ln\left(\frac{P(y=1)}{1-P(y=1)}\right)=\eta$$

Substituting $\eta = b + w_1x_1 + w_2x_2 + \cdots + w_nx_n$

$$\ln\left(\frac{P(y=1)}{1-P(y=1)}\right)=\eta=b+w_1x_1+w_2x_2+\cdots+w_nx_n \tag{10}$$

We conclude this overview of logistic regression by looking at how they work and learn. The weight vector **W** and the bias term b are learned from a labelled training set by minimizing a suitable loss function using techniques such as gradient descent or its variants. Once trained, the logistic regression model can be used to predict the probability of the binary output variable for new input examples.

The feedforward process for logistic regression can be described as follows:

Compute z as the dot product of the weight vector **W** and the input features, plus the bias term b, transforming the input features info a single scalar z that represents the log-odds of the output being $y = 1$:

$$z = \mathbf{W}^T\mathbf{X} + b \tag{11}$$

Pass z through the sigmoid function to map the log-odds z to a probability $\hat{y} = P(y = 1 \mid X)$, ensuring the output is between 0 and 1:

$$\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}} \tag{12}$$

During training, we define the loss function $\mathcal{L}$ as the negative log-likelihood of the predicted output given the true label:

$$\mathcal{L} = -\left(y\ln(\hat{y}) + (1-y)\ln(1-\hat{y})\right) \tag{13}$$

For a trained system, we compare $\hat{y}$ to a threshold to convert the probabilistic output into the final binary classification.

We now look at **feedback process** for logistic regression. To optimize the weight vector **W**, we compute the derivatives of the loss function with respect to each weight and the bias term and use these derivatives to update the weights in the opposite direction of the gradient. This is known as gradient descent.

To compute the derivatives, we use the chain rule:

$$\frac{\partial\mathcal{L}}{\partial w_i} = \frac{\partial\mathcal{L}}{\partial\hat{y}}\cdot\frac{\partial\hat{y}}{\partial z}\cdot\frac{\partial z}{\partial w_i}$$

and

$$\frac{\partial\mathcal{L}}{\partial b} = \frac{\partial\mathcal{L}}{\partial\hat{y}}\cdot\frac{\partial\hat{y}}{\partial z}\cdot\frac{\partial z}{\partial b}$$

We can then use these derivatives to update the weights as follows:

$$w_i \leftarrow w_i - \alpha\frac{\partial\mathcal{L}}{\partial w_i} \tag{14}$$

and

$$b \leftarrow b - \alpha\frac{\partial\mathcal{L}}{\partial b} \tag{15}$$

Where α is the learning rate, which controls the step size of the updates. By iteratively performing these updates on a training set, we can find the optimal weight vector **W** that minimizes the loss function on the training set.

To calculate the derivatives, let us begin by computing the derivative of the loss function with respect to the predicted output $\hat{y}$:

$$\frac{\partial\mathcal{L}}{\partial\hat{y}} = \frac{\partial}{\partial\hat{y}}\left(-\left(y\ln(\hat{y}) + (1-y)\ln(1-\hat{y})\right)\right)$$

since

$$\frac{d(ln(x))}{dx} = \frac{1}{x}$$

we get:

$$\frac{\partial\mathcal{L}}{\partial\hat{y}} = -\left(\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}}\right)$$

The derivative of the predicted output $\hat{y}$ with respect to z is solved using the quotient rule, that is

$$\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{f'(x)g(x) - g'(x)f(x)}{g^2(x)}$$

So, if we let

$f(z) = 1$	$f'(z) = 0$
$g(z) = 1 + e^{-z}$	$g'(z) = -e^{-z}$

$$\begin{aligned}
\frac{\partial\hat{y}}{\partial z} &= \frac{\partial}{\partial z}\left(\frac{1}{1+e^{-z}}\right)\\
&= \frac{e^{-z}}{(1+e^{-z})^2}\\
&= \frac{1}{(1+e^{-z})}\frac{e^{-z}}{(1+e^{-z})}\\
&= \frac{1}{(1+e^{-z})}\frac{1+e^{-z}-1}{1+e^{-z}}\\
&= \frac{1}{(1+e^{-z})}\left(\frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}}\right)\\
&= \hat{y}(1-\hat{y})
\end{aligned}$$

The derivative of z with respect to w_i:

$$\begin{aligned}
\frac{\partial z}{\partial w_i} &= \frac{\partial}{\partial w_i}\mathbf{W}^T\mathbf{X} + b\\
&= \frac{\partial}{\partial w_i}(w_1x_1 + \cdots + w_ix_i + \cdots + w_nx_n + b)\\
&= x_i
\end{aligned} \tag{16}$$

Similarly,

$$\begin{aligned}\frac{\partial z}{\partial b} &= \frac{\partial}{\partial b}\mathbf{W}^T\mathbf{X} + b \\ &= \frac{\partial}{\partial b}(w_1x_1 + \cdots + w_ix_i + \cdots + w_nx_n + b) \\ &= 1\end{aligned} \tag{17}$$

Therefore

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_i} &= \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w_i} \\ &= -\left(\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}}\right) \cdot \hat{y}(1-\hat{y}) \cdot x_i \\ &= -\left(\frac{-y(1-\hat{y}) + (1-y)\hat{y}}{\hat{y}(1-\hat{y})}\right) \cdot \hat{y}(1-\hat{y}) \cdot x_i \\ &= [(1-y)\hat{y} - y(1-\hat{y})]\, x_i \\ &= [\hat{y} - y\hat{y} - y + y\hat{y}]\, x_i\end{aligned}$$

$$\frac{\partial \mathcal{L}}{\partial w_i} = (\hat{y} - y)x_i \tag{18}$$

and similarly

$$\frac{\partial \mathcal{L}}{\partial b} = (\hat{y} - y) \tag{19}$$

C. Machine learning technique - K-nearest neighbours algorithm

K-nearest neighbours is simple yet robust for datasets with well-structured local relationships. It is particularly effective for non-linear decision boundaries and finds applications in anomaly detection, pattern recognition, and recommendation systems.

The k-nearest neighbours algorithm is non-parametric, that is, it does not make assumptions about the underlying distribution or functional form of the data; instance-based, that is, it does not explicitly learn a model during a training phase, but it relies on the raw training data to make predictions at query time, unlike parametric models, which "learn" a set of fixed parameters (e.g., coefficients in linear regression) during training, learning method used for classification and regression tasks. It predicts the output of a query point based on the labels or values of its k-nearest neighbours in feature space.

KNN does not assume a specific functional form for the data. Instead, it relies on the local structure of the dataset. For a given query point x, the algorithm uses a distance metric to identify its k-nearest neighbours and infers the prediction by aggregating their outcomes. Typical distance metrics include:

1) Euclidean Distance

$$d(x, x') = \sqrt{\sum_{i=1}^{n}(x_i - x'_i)^2}$$

2) Manhattan Distance

$$d(x, x') = \sum_{i=1}^{n} |x_i - x'_i|$$

3) Minkowski Distance. Note that for $p = 2$, this reduces to the Euclidean distance; for $p = 1$, it becomes the Manhattan distance.

$$d(x, x') = \left(\sum_{i=1}^{n} |x_i - x'_i|^p\right)^{1/p}$$

In classification tasks, the predicted class $\hat{y}$ for a query point x is determined by majority voting among its k-nearest neighbours. If we let $N_k(x)$ represent the set of the k-nearest neighbours,he predicted class is given by:

$$\hat{y} = \text{argmax}_c \sum_{x' \in N_k(x)} \mathbb{I}(y(x') = c)$$

Where:

argmax_c	Identifies the class c with the highest count among the neighbours.
$\mathbb{I}(\cdot)$	The indicator function, which equals 1 if $y(x') = c$, and 0 otherwise.

The parameter k determines the number of neighbours considered.

- A small k (e.g., $k = 1$) makes the model sensitive to noise, as predictions rely heavily on individual points.
- A large k smooths the decision boundary but may lead to underfitting.

Finding the optimal value for k is one of the main challenges in the k-nearest neighbour. Cross-validation is typically employed to find a value using the following procedure:

1) Iterate Over Candidate Values of k
   - Choose a range of potential k values, for example, $k = 1, 2, \ldots, 20$.
2) Evaluate Performance for Each k
   - For each k, use K-fold cross-validation:
     - Train the KNN model using $K - 1$ folds.
     - Validate it on the remaining fold.
     - Compute the average performance metric across all K folds.
3) Select k with the Best Performance
   - After evaluating all k values, choose the k that maximizes the performance metric (e.g., accuracy) or minimizes the error.

The k-nearest neighbour algorithm is, therefore, as follows;

1) Input:

- Dataset: $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$, where x_i is the feature vector, and y_i is the corresponding label or value.
- Query point: x_{query}, the data point for which the prediction is required.
- Number of neighbours: k, the number of nearest neighbours to consider.
- Distance metric: $d(x, x')$ (e.g., Euclidean distance).

2) Steps::

1) Calculate Distance. For the query point x_{query}, compute the distance to every point x_i in the dataset D using the chosen distance metric. For Euclidean distance:

$$d(x_{query}, x_i) = \sqrt{\sum_{j=1}^{n} (x_{query_j} - x_{i_j})^2},$$
$$\forall i \in \{1, 2, \ldots, N\}.$$

2) Sort Neighbours. Rank all data points x_i in D by their distance to x_{query} in ascending order. Let this sorted set be D_{sorted}.
3) Select k-Nearest Neighbours. Extract the top k data points from D_{sorted}. Denote this set as $N_k(x_{query})$.
4) Aggregate Neighbours' Outputs.
   - For **classification**: Perform majority voting among the labels y_i of the k-nearest neighbours:

$$\hat{y} = \text{argmax}_c \sum_{x_i \in N_k(x_{query})} \mathbb{I}(y_i = c),$$

   where $\mathbb{I}(y_i = c)$ is an indicator function that equals 1 if $y_i = c$, and 0 otherwise.
5) **Output Prediction:** Return $\hat{y}$ as the predicted label (for classification) or value (for regression) for x_{query}.

3) Output:: The predicted label or value $\hat{y}$ for the query point x_{query}.

D. Feature scaling

Feature scaling is an important data transformation process. It is a very important aspect of many machine learning algorithms including logistic regression, support vector machines and neural networks, as, the performance of such algorithms is adversely impacted when the numerical features have different scales. The two methods to transform features on the same scale are normalization and rescaling.

During normalization or rescaling, values are scaled and shifted so that they are mapped onto the $[0, 1]$ interval. This is achieved by applying the following transformation;

$$x_{normalized} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Normalization is also sometimes referred to as min-max scaling.

Standardization is particularly suited for algorithms that assume Gaussian distributions, like linear regression and logistic regression. It is achieved by first subtracting values from the mean so that the mean of the normalized values is zero, and then dividing by the standard deviation so that the variance of the normalized distribution is one. Thus;

$$x_{standardized} = \frac{x - \mu}{\sigma}$$

Unlike normalization, the range of standardization is not fixed, and this can sometimes be an issue if a value in the $[0, 1]$ interval is expected. Standardization is however more resilient to the effect of outliers.

Tree based machine learning models, like decision trees and random forests do not normally require feature scaling.

E. Cross-validation

Cross-validation is a technique that can be applied to any ML algorithm that is aimed to reduce overfitting by estimating how well each hypothesis generalizes to unseen data. In practice, a portion of the data is reserved for this purpose.

The K-fold cross validation technique splits the dataset into k folds, training that model on $k - 1$ folds and testing on the remaining one. The test fold is rotated and the process is repeated so that each fold will act as the test set once. Metrics like mean square error are averaged across folds, ensuring a robust estimate.

Therefore, if for example, we split the data into 10 folds, so that each fold will have $n/10$ records, we train the model on 9 folds and test on the remaining one. We then rotate the test fold and repeat this process 10 times. The final performance metric is computed as the mean of the metrics across the 10 folds.

In stratified K-fold cross validation, the folds have the same proportion of the classes as in the original dataset.

Other types of cross validation include Leave one out cross-validation, where each data point is used as a test set once.

F. Measurement

Quantitative measurements numerically represent attributes and are fundamental for evaluating machine learning models by providing an objective means to assess the model's performance. We can analyse the model's effectiveness by comparing model predictions to actual outcomes. Appropriate selection and reporting of measurement methods, including their reliability, validity, and potential biases, are essential to ensure accurate interpretation and meaningful results. This section will look at the measurements used in this study.

The function used in machine learning and statistical modelling to quantify the difference between the predicted outputs of a model and the actual target values is called the loss function. It serves as a measure of the model's performance, guiding the optimisation process to improve predictions.

For a single prediction:

$$\text{Loss}(\hat{y}, y) = f(\hat{y}, y)$$

Where:

$\hat{y}$: Predicted value.

y : Actual value.

f : Specific loss function formula

The aggregation of the loss function across the entire dataset is called the cost function so that;

$$\text{Cost}(\hat{y}, y) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(\widehat{y}_i, y_i)$$

In the problem we are considering, we will try to predict whether a person will recidivate given information about his or her previous criminal history is a binary classifier problem. A binary classifier is a function that can be applied to features X such as $(x_1, x_2, x_3, \ldots, x_n)$ and maps them to an output Y, where $Y \in \{0, 1\}$. It is a supervised learning technique; therefore, a test set is extracted from the available data to validate the model before being deployed in production.

$$f(x_1, x_2, x_3, \ldots, x_n) = Y \in \{0, 1\}$$

The function will return a value between 0 and 1; therefore, a threshold value is operated to classify the result as true or false. The model will subsequently classify predictions as true or false according to the threshold value.

For the classification problem that we have in hand, we will primarily use the log-likelihood cost function defined as:

$$\mathcal{L}_{\text{log-likelihood}} = -\sum_{i=1}^{N} [y_i \ln(\hat{y}_i) + (1 - y_i) \ln(1 - \hat{y}_i)]$$

Negative log-likelihood focuses on the distance between the labels y_i and the predicted probabilities $\hat{y}_i$. If we consider a sample where:

- y_i is one and $\hat{y}_i$ is close to 1: $y_i \ln(\hat{y}_i)$ is close to zero, while $(1-y_i)\ln(1-\hat{y}_i)$ is zero, so that the resultant loss is close to zero.
- y_i is 1 and $\hat{y}_i$ is far from 1: $y_i \ln(\hat{y}_i)$ is large, while $(1 - y_i)\ln(1 - \hat{y}_i)$ is zero so that the resultant loss is large.
- y_i is zero and $\hat{y}_i$ is close to 0: $y_i \ln(\hat{y}_i)$ is zero, while $(1-y_i)\ln(1-\hat{y}_i)$ is also close to zero so that the resultant loss is close to zero.
- y_i is zero and $\hat{y}_i$ is far from 0: $y_i \ln(\hat{y}_i)$ is zero, while $(1 - y_i)\ln(1 - \hat{y}_i)$ is large so that the resultant loss is large.

In this scenario, we can obtain four kinds of results:

The number of samples that are TP, TN, FP or FN can be organised in what is known as a confusion matrix, that is shown in Figure 6. This tool makes it easy to perform calculations that determine the validity of the model at hand.

Prediction	Classif	Outcome	Description
1	1	True Positive (TP)	The model correctly predicted the positive class.
1	0	False Positive (FP)	The model incorrectly predicted the positive class (a false alarm).
0	1	False Negative (FN)	The model incorrectly predicted the negative class (a miss).
0	0	True Negative (TN)	The model correctly predicted the negative class.

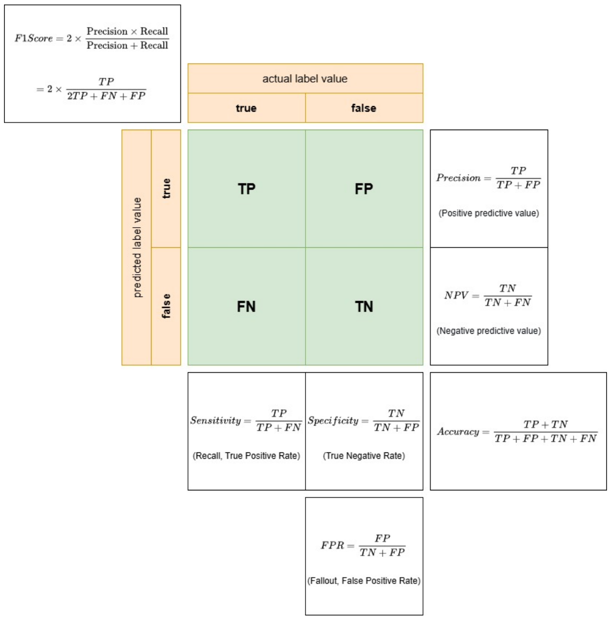

$F1Score = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$
$= 2 \times \frac{TP}{2TP + FN + FP}$
actual label value
true
false
predicted label value
true
false
TP
FP
FN
TN
$Precision = \frac{TP}{TP + FP}$
(Positive predictive value)
$NPV = \frac{TN}{TN + FN}$
(Negative predictive value)
$Sensitivity = \frac{TP}{TP + FN}$
(Recall, True Positive Rate)
$Specificity = \frac{TN}{TN + FP}$
(True Negative Rate)
$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$
$FPR = \frac{FP}{TN + FP}$
(Fallout, False Positive Rate)

Fig. 6: Confusion matrix and associated equations

There are several key performance metrics derived from the confusion matrix, each offering a different perspective on evaluating the effectiveness of a binary classification model.

The **accuracy** of a model is a straightforward measure that evaluates the proportion of correct predictions (both True Positives and True Negatives) out of the total number of predictions. It can be mathematically defined as:

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

for a dataset,

$$Accuracy = \frac{1}{N} \sum_{i=1}^{N} 1(\hat{y}_i = y_i)$$

Precision answers the question: ”Out of all the observations predicted to be positive, how many were positive?” In other words, it measures the model’s ability to avoid false positives.

$$Precision = \frac{TP}{TP + FP}$$

Recall or **sensitivity** measures the model's ability to correctly identify positive cases. It answers the question: "Out of all the actual positive instances, how many did the model correctly identify as positive?"

$$Recall = \frac{TP}{TP + FN}$$

Specificity measures the model's ability to identify negative cases correctly. It answers the question: "Out of all the actual negative instances, how many did the model correctly identify as negative?"

The formula for specificity is:

$$Specificity = \frac{TN}{TN + FP}$$

The F1-score is the harmonic mean of [precision](model-performance-precision) and [recall](model-performance-recall). It provides a single metric that balances precision and recall, which is useful when there is an uneven class distribution or when false positives and false negatives are important.

The formula for the F1 score is:

$$\begin{aligned} F1 &= 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \\ &= 2 \times \frac{TP}{2TP + FN + FP} \end{aligned}$$

As can be seen in Figure 7, the ROC curve is plotted by varying the decision threshold of the classifier and plotting the corresponding values of FPR (on the x-axis) and sensitivity or TPR (on the y-axis). Each point on the ROC curve represents a (FPR, TPR) pair for a particular threshold value.

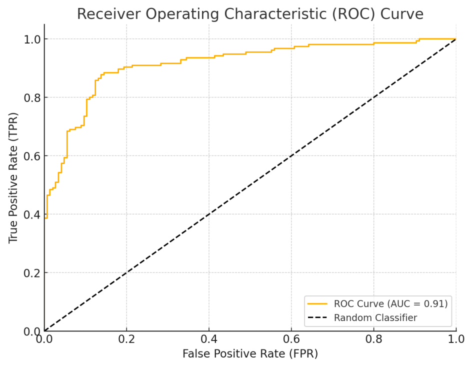


Fig. 7: ROC curve

The Area Under the ROC Curve (AUC) is a key indicator of model performance. The value of the AUC ranges from 0 to 1:

- A perfect classifier has an AUC of 1, indicating it achieves a TPR of 1 while keeping the FPR at 0.
- A model with an AUC of 0.5 performs no better than random guessing.
- Higher AUC values indicate better overall performance.

The goal of a classifier is to maximise the TPR (correctly predicting positive instances) while minimising the FPR (incorrectly classifying negative instances as positive). Ideally, the ROC curve should approach the top-left corner of the plot, indicating a high TPR with a low FPR.

In summary, the ROC curve helps to visualise and compare the trade-offs between true positives and false positives across different thresholds, and the AUC provides a single number summarising the model's ability to discriminate between positive and negative classes.

References

[1] Sergey Ioffe. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[2] Jianxin Wu. "Introduction to convolutional neural networks". In: *National Key Lab for Novel Software Technology. Nanjing University. China* 5.23 (2017), p. 495.

Appendix

Field Name	Description	Type	Options (if Categorical)	Used
id	Unique identifier for each individual.	Numeric	N/A	No
name	Full name of the defendant	Text	N/A	No
first	First name of the defendant (anonymized).	Text	N/A	No
last	Last name of the defendant (anonymized).	Text	N/A	No
compas_screening_date	Date of the COMPAS assessment.	Date	N/A	Yes
sex	Gender of the defendant.	Categorical	Male, Female	Yes
dob	Date of birth of the defendant.	Date	N/A	No
age	Age of the defendant at the time of assessment.	Numeric	N/A	Yes
age_cat	Age category of the defendant.	Categorical	Less than 25 25 - 45 Greater than 45	Yes
race	Race of the defendant.	Categorical	African-American Caucasian Hispanic Asian Native American Other	Yes
juv_fel_count	Number of juvenile felony offenses.	Numeric	N/A	Yes
juv_misd_count	Number of juvenile misdemeanor offenses.	Numeric	N/A	Yes
juv_other_count	Number of other juvenile offenses.	Numeric	N/A	Yes
priors_count	Number of prior offenses (adult and juvenile).	Numeric	N/A	Yes
days_b_screening_arrest	Days between arrest and COMPAS screening.	Numeric	N/A	Yes
c_jail_in	Jail booking date for the charge.	Date	N/A	No
c_jail_out	Jail release date for the charge.	Date	N/A	No
c_case_number	Case number associated with the charge.	Text	N/A	No
c_offense_date	Date of the alleged offense.	Date	N/A	No
c_arrest_date	Arrest date for the charge.	Date	N/A	No
c_charge_degree	Degree of the charge.	Categorical	F (Felony) M (Misdemeanor)	Yes
c_charge_desc	Description of the charge.	Text	Free text	No
is_recid	reoffended after COMPAS screening.	Binary	0 (No), 1 (Yes)	No
r_case_number	Case number for the re-offense.	Text	N/A	No
r_charge_degree	Degree of the re-offense charge.	Categorical	F (Felony) M (Misdemeanor)	No
r_charge_desc	Description of the re-offense charge.	Text	Free text	No
r_jail_in	Jail booking date for the re-offense.	Date	N/A	No
r_jail_out	Jail release date for the re-offense.	Date	N/A	No
two_year_recid	Label: offense within two years.	Binary	0 (No), 1 (Yes)	Yes
decile_score	COMPAS risk score (1-10).	Numeric	1–10	Yes
score_text	Risk category for general recidivism.	Categorical	Low, Medium, High	Yes
v_type_of_assessment	Type of COMPAS assessment conducted.	Text	Risk of Recidivism	No
v_decile_score	Violent recidivism COMPAS score (1-10).	Numeric	1–10	No
v_score_text	Risk category for violent recidivism.	Categorical	Low, Medium, High	No
start	Start date of the two-year recidivism period.	Date	N/A	No
end	End date of the two-year recidivism period.	Date	N/A	No
event	offense during the two-year period.	Binary	0 (No), 1 (Yes)	No