APPLIED ROOTS

AAIC Technologies Pvt Ltd

UNIVERSITY OF HYDERABAD

सा विद्या या विमुक्तये

Machine Learning

Table of Contents

1. FUNDAMENTALS OF MACHINE LEARNING 2

(CLASSIFICATION)

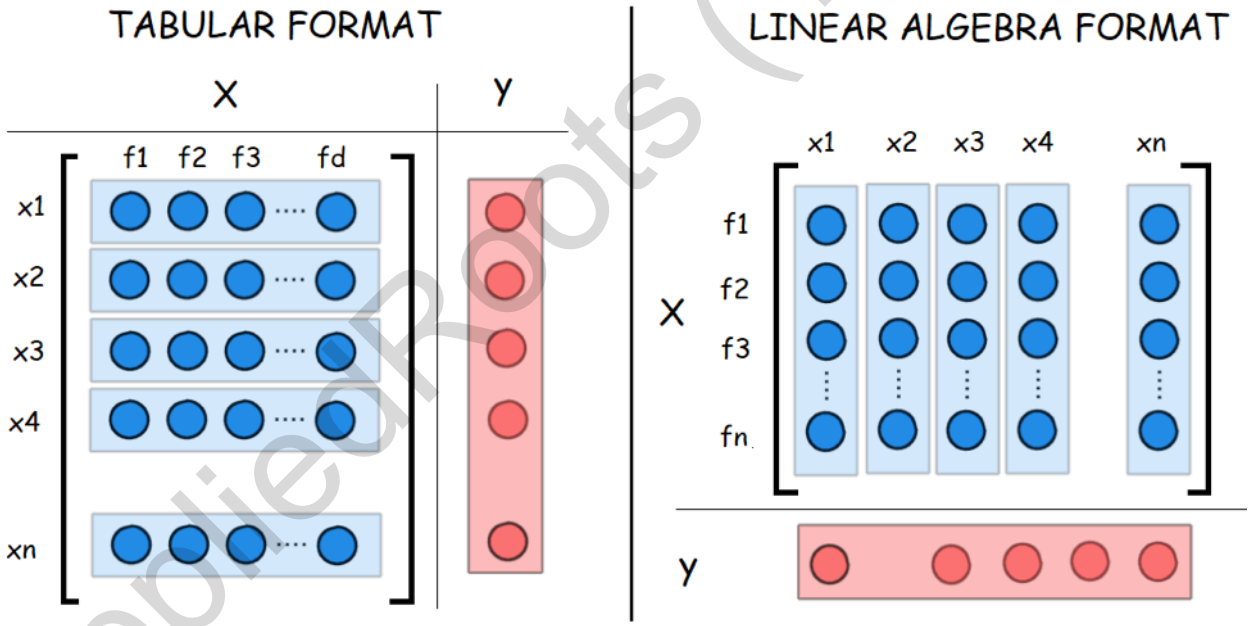
FUNDAMENTALS OF ML 2 (CLASSIFICATION)

DATA REPRESENTATION CONVENTIONS FOR SUPERVISED LEARNING:

The general convention in machine learning for representing data sets is as follows:

1. The data matrix containing all the data point vectors is usually represented by X (ie: capital x)
2. The generic representation for any data point vector in the dataset is xi (ie: small x with subscript i)
3. The vector containing all the corresponding labels/classes for each xi is usually represented by y (small y).
4. Label contained within y corresponding to xi is represented by yi (ie: small y with subscript i)

The images below further clarifies the above explanation:



As can be seen above the Tabular format is much easier to interpret because it allows us to relate each data point to its corresponding label in a simple manner. Also note that, in the tabular representation, the transposed matrix containing the data point vectors is represented by **X** instead of $\mathbf{X}^T$ as one would, if one were to follow the formal Linear Algebraic convention. This is solely for the purposes of convenience.

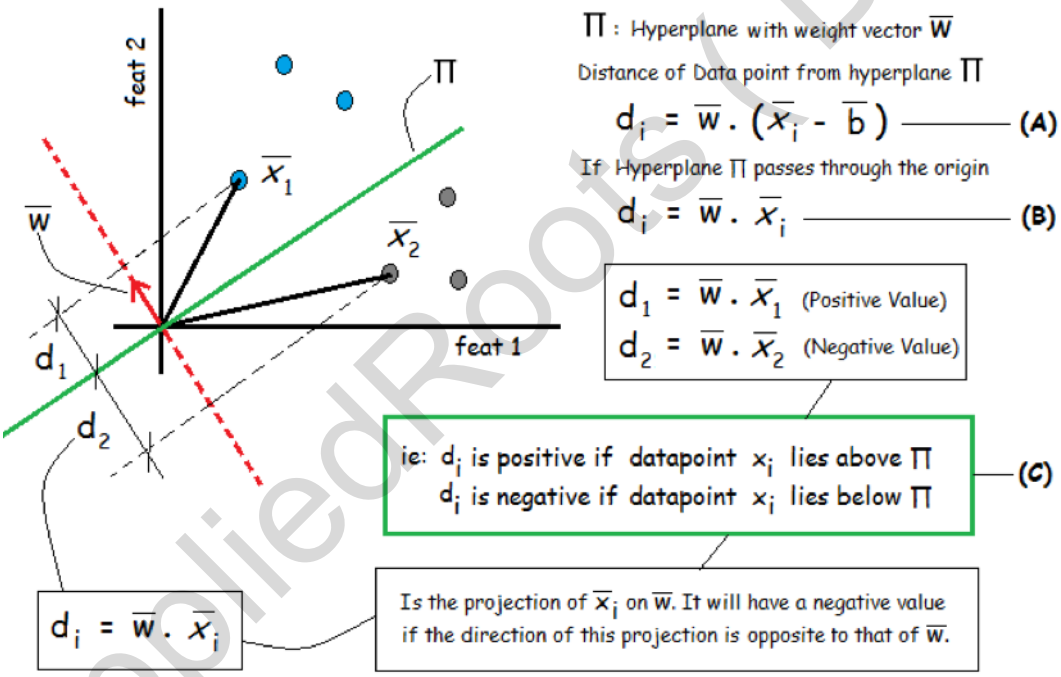
This book will be using the formal Linear Algebraic (Matrix) format in situations where we are dealing with mathematical derivations. In other situations, while describing machine learning pipelines/processes, the tabular format will be used where we will represent $\mathbf{X}^T$ as $\mathbf{X}$ to represent the data matrix. The reader is advised to bear in mind these conventions while reading the book further on.

LOGISTIC REGRESSION:

Logistic Regression is a Supervised Machine Learning algorithm used for data classification. Data Classification tasks are broadly classified into 2 categories, based on the number of labels the data contains:

1. Binary classification - 2 labels or classes.
2. Multiclass Classification - More than 2 classes.

LOGISTIC REGRESSION FOR BINARY CLASSIFICATION:



Consider the image shown above. The dataset contains 2 classes represented by blue and grey points as shown above(ie: **y** contains 2 types of labels). Classification is about finding that hyperplane Π represented by a weight vector **w** which best separates data based on their labels (blue points & grey points).

The distance $\mathbf{d}_i$ of any data point (vector) $\mathbf{x}_i$ from hyperplane $\boldsymbol{\Pi}$ is obtained by using equation **(A)** shown in the image above. In the derivations that follow, for the sake of simplicity, it will be assumed that the separating hyperplane passes through the origin and so this distance will be defined by equation **(B)**. If one wants to avoid this assumption, then all

that needs to be done is to include an intercept vector **b** in the derivations. In other words simply replace xi with $(\mathbf{x_i - b})$ in the derivation.

The hyperplane **Π** will be effective at separating the data based on their classes if it is positioned based on the property **(C)** of distance **ŷ** described in the image above. That is, the hyperplane should:

1. Maximize the sum of all $\mathbf{d_i}$ for all $\mathbf{x_i}$ belonging to one class. This has the effect of positioning the hyperplane such that all data points belonging to this class comes "above" it.
2. It minimizes the sum of all $\mathbf{d_i}$ for all $\mathbf{x_i}$ belonging to the other class. This has the effect of positioning the hyperplane such that all data points belonging to this class comes "below" it.

Fulfilling conditions 1 & 2 results in a hyperplane which will have maximum average distance from both classes.

The Logistic regression algorithm strategically uses to its advantage 2 monotonously increasing functions to formulate an objective function that fulfills the above 2 conditions. These functions are:

1. 1. The Sigmoid function **(σ)**.
1. The Logarithm function.

Monotonously increasing functions are functions whose output increases monotonously (ie: continuously without any change in direction) as the values of its inputs are increased continuously.

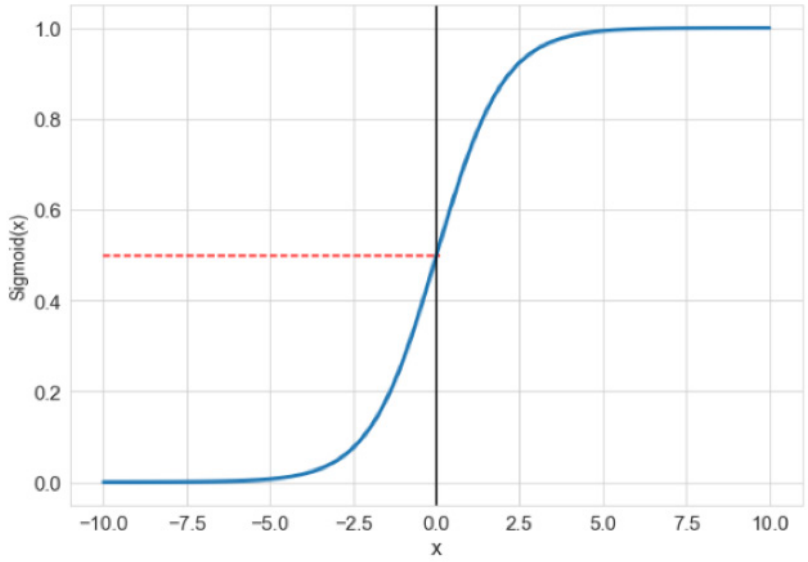
THE SIGMOID FUNCTION (σ):

The Sigmoid function 'squashes" all values between the interval negative infinity & positive infinity, to exist within the interval of zero & one such that:

1. As the value approaches negative infinity, the output of the sigmoid function tends to zero.
2. As the value approaches positive infinity, the output of the sigmoid function tends to one. It is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The function has a characteristic 'S' shaped curve as shown below:

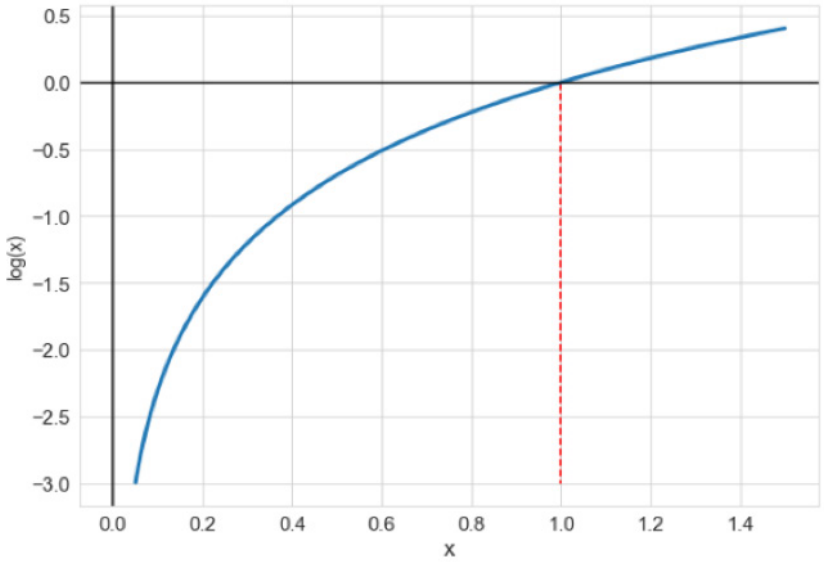


The above mentioned property of sigmoid functions coupled with the fact that they are monotonously increasing, makes the sigmoid function an ideal function to use when one wants to constrain another function's output to exist within zero and one while retaining as much information as possible. Constraining another function's output to lie between zero & one also allows one to interpret the function's outputs probabilistically which is very advantageous as shall be described further on. Sigmoid functions also have the advantage of being easily differentiable, which is imperative for computing step gradients during optimization

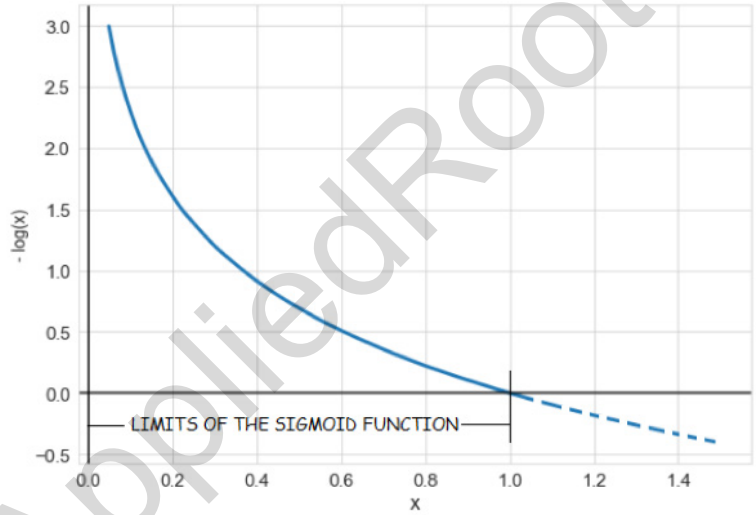
Since the Sigmoid function squashes all values it is fed to exist within the interval [0,1], its outputs are resilient to anomalously large or small values that might exist on the "wrong" side of the hyperplane, within the dataset it transforms. The Sigmoid function thus serves a dual purpose within the Logistic regression algorithm (ie: creating probabilistic interpretability & outlier resilience).

THE LOGARITHM FUNCTION:

The Logarithm function is a monotonously increasing function with the characteristics as shown in the plot below. The log function is basically used to modify the outputs of the Sigmoid function within the Logistic Regression algorithm's pipeline. Since the Sigmoid function's outputs all lie within the interval zero & one, passing it through a log function will transform all its values to lie between negative infinity & zero. Like the Sigmoid function, the log function is also easily differentiable.



The log function is mainly used to provide numerical stability. When we add up the sigmoid values over a large number of data points (as shall be seen while minimizing/maximizing the logistic regression objective function), we may run into numerical stability issues, since the outcomes of the Sigmoid function could be very small decimal numbers. Modifying the output of the Sigmoid function by passing it through a log function takes care of the numerical computation issues that arises, without actually affecting the goal of the objective function



The negative of the log function has the attribute that it transforms all values of the sigmoid function to lie between positive infinity & zero as shown above, which is a very useful attribute when trying to define loss functions as shall be seen further on.

THE LOGISTIC REGRESSION OBJECTIVE FUNCTION:

As explained earlier, in Logistic Regression we want to maximize all **ŷ** for one class and minimize all **ŷ** for the other class, where ŷ is the dot product of any data point vector **x** and the weight vector **w** representing hyperplane **Π**. In other words, **ŷ** is the perpendicular distance of any datapoint **x** from the hyperplane **Π**. These 2 conditions are used as the basis for formulating the objective function.

The objective function for Logistic Regression function is formulated as follows:

1. All data points belonging to one class are labelled as **y = 1** and all data points belonging to the other class are labelled **y = 0**. These values are strategically chosen for reasons that will become evident as we frame the objective function.
2. All values of **ŷ** are passed through the Sigmoid function to get the resulting value z as shown below:.

$$z_i = \sigma(d_i) = \frac{1}{1 + e^{-d_i}} \quad \text{Where } d_i = \overline{w} \cdot \overline{x}_i$$

   a. All values of **z** lie within the interval [0, 1].
   b. For those data points that lie above the hyperplane, the values of **z** are greater than or equal to 0.5. This corresponds to all positive values of **d**.
   c. For those data points that lie below the hyperplane, the values of **z** are lesser than 0.5. This corresponds to all negative values of **d**.
3. The values of z lie within [0, 1], and hence they can be interpreted probabilistically as follows :
   a. The values of **z** can be interpreted as the probability of the data point lying above the hyperplane. The greater the value of **z**, the higher the probability that the datapoint **x** corresponding to it lies above the hyperplane.ie:
   P (x being being above hyperplane) = z
   b. The values of **(1 - z)** can be interpreted as the probability of the data point lying below the hyperplane. The greater the value of **(1 - z)**, the higher the probability that the datapoint **x** corresponding to it lies below the hyperplane.
   ie: **P (x being below the hyperplane) = 1 - z**
4. Therefore the best hyperplane would be that which:
   a. Maximizes values of **z** that belong to class **y = 1.**
   b. Maximizes the values of **(1- z)** for those data points that belong to class **y = 0**.
   c. The above 2 requirements can be formalized into an objective function as shown below:

$$\bar{w} = \underset{\bar{w}}{\operatorname{argmax}} \sum_{i=1}^{n} \underbrace{y_i z_i}_{A} + \underbrace{(1 - y_i)(1 - z_i)}_{B} \quad \text{——— (i)}$$

Only A gets "activated" when $y_i = 1$
Only B gets "activated" when $y_i = 0$

The above objective function can be interpreted as:

$$\bar{w} = \underset{\bar{w}}{\operatorname{argmax}} \sum_{i=1}^{n} y_i P(y_i = 1 \mid \bar{x}_i) + (1 - y_i) P(y_i = 0 \mid \bar{x}_i) \quad \text{——— (ii)}$$

or

$$\bar{w} = \underset{\bar{w}}{\operatorname{argmin}} \sum_{i=1}^{n} - y_i P(y_i = 1 \mid \bar{x}_i) - (1 - y_i) P(y_i = 0 \mid \bar{x}_i)$$

5. Lastly all probability values are passed through the Logarithm function and converted into Log probabilities to arrive at the final form of the objective function:

$$\bar{w} = \underset{\bar{w}}{\operatorname{argmin}} \sum_{i=1}^{n} - y_i \log z_i - (1 - y_i) \log (1 - z_i) \quad \text{——— (iii)}$$

Where:

$z_i = \sigma(d_i)$ & $d_i = \bar{w} \cdot (\bar{x}_i - \bar{b})$ OR $\bar{w} \cdot \bar{x}_i$ (if hyperplane passes through the origin)

ERROR & LOSS:

Error in Machine Learning, is a measure of how much a prediction deviates from the actual value that a predicting system was aiming to emulate. In other words given an input vector $\mathbf{x}_i$ belonging to a set of input vectors $\mathbf{X}$ (dataset) and its corresponding label $\mathbf{y}_i$, error is a measure of how much the prediction $\hat{\mathbf{y}}_i$ made by the prediction mechanism deviates from $\mathbf{y}_i$.

Loss in Machine Learning, is the average error produced by a prediction system. It is a measure of the expected deviation that the prediction system's predictions will have, with respect to the actual values/classes it is trying to predict.

REGRESSION ERROR & CLASSIFICATION ERROR:

All parametric prediction algorithms initially output values (**ŷi**) that are some function of the distance of data points from a hyperplane (ie: They are continuous/quantitative values). In case of regression where the labels are also continuous values, the definition of error (ie: the error function) is usually quite simple because the predicted value is directly comparable with the actual value trying to be predicted. That is, the error is some function of the difference between the actual value **y** and the predicted value **ŷ**. Shown below are a couple of examples of regression errors:

$$\text{Squared error} = (y - \hat{y})^2$$
$$\text{Absolute error} = |y - \hat{y}|$$

In case of classification, where the labels are categorical, the definition of error is not as simple. This is because the predicted value (which is quantitative/numerical) is not directly comparable with the actual value trying to be predicted (labels are categorical). To overcome this lack of comparability, suitable loss functions are "engineered", which take in as their inputs, the numeric values predicted by the predictive mechanism and modifies them such that they now represent values, the minimization of which results in better classification. The description of log loss that follows will clarify this concept further.

LOG LOSS:

The **log-loss-error** is the negative log of the probability of a datapoint belonging to the class as denoted by its label. Log loss is the average log error computed over a set of data points. For binary classification, it is defined by the equation shown below:

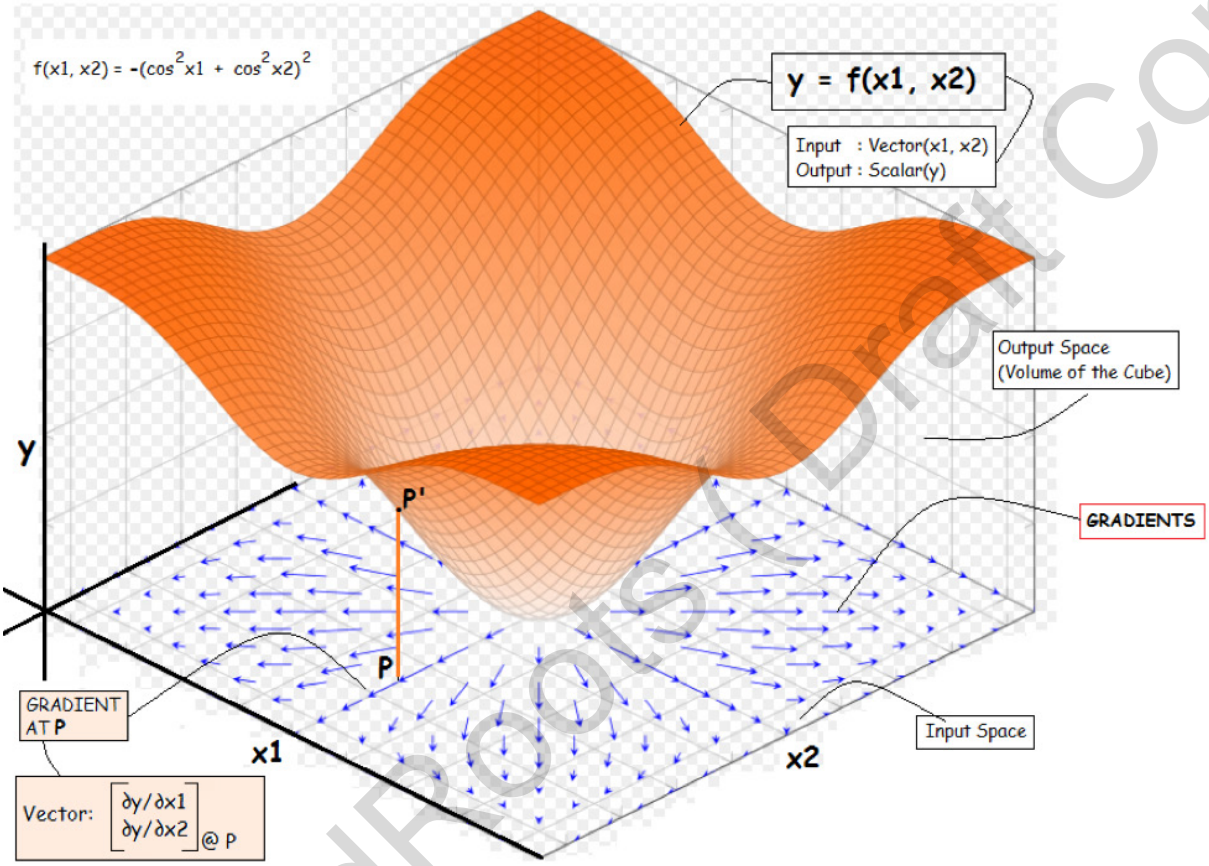
$$\text{Binary Log Loss} = \frac{1}{n}\sum_{i=1}^{n} -y_i \underbrace{\text{Log } P(y_i = 1 \mid \overline{x}_i)}_{A} - (1 - y_i) \underbrace{\text{Log } P(y_i = 0 \mid \overline{x}_i)}_{B}$$

In the equation above, the more the probability values (A or B as the case maybe) move away from one, the lesser the confidence the predictive mechanism (ie: the separating hyperplane Π) will have, about the data points belonging to the classes denoted by their corresponding labels and hence the larger the Log Loss becomes(see plot of negative log shown earlier). Therefore Log Loss is a measure of the "lack of confidence" the predictive mechanism has about its predictions. Minimizing this value implicitly means improving the classification performance.

Minimizing the objective function of the Logistic Regression algorithm (eqn iii shown earlier) has the same effect as minimizing the Log Loss.

THE GRADIENT:

The gradient is a property of (differentiable) functions, that take in multiple input variables (ie: vectors inputs) and produce a scalar output. The gradient of such a function is a vector containing the partial derivatives values of that function with respect to all its input variables. Consider the image shown below:

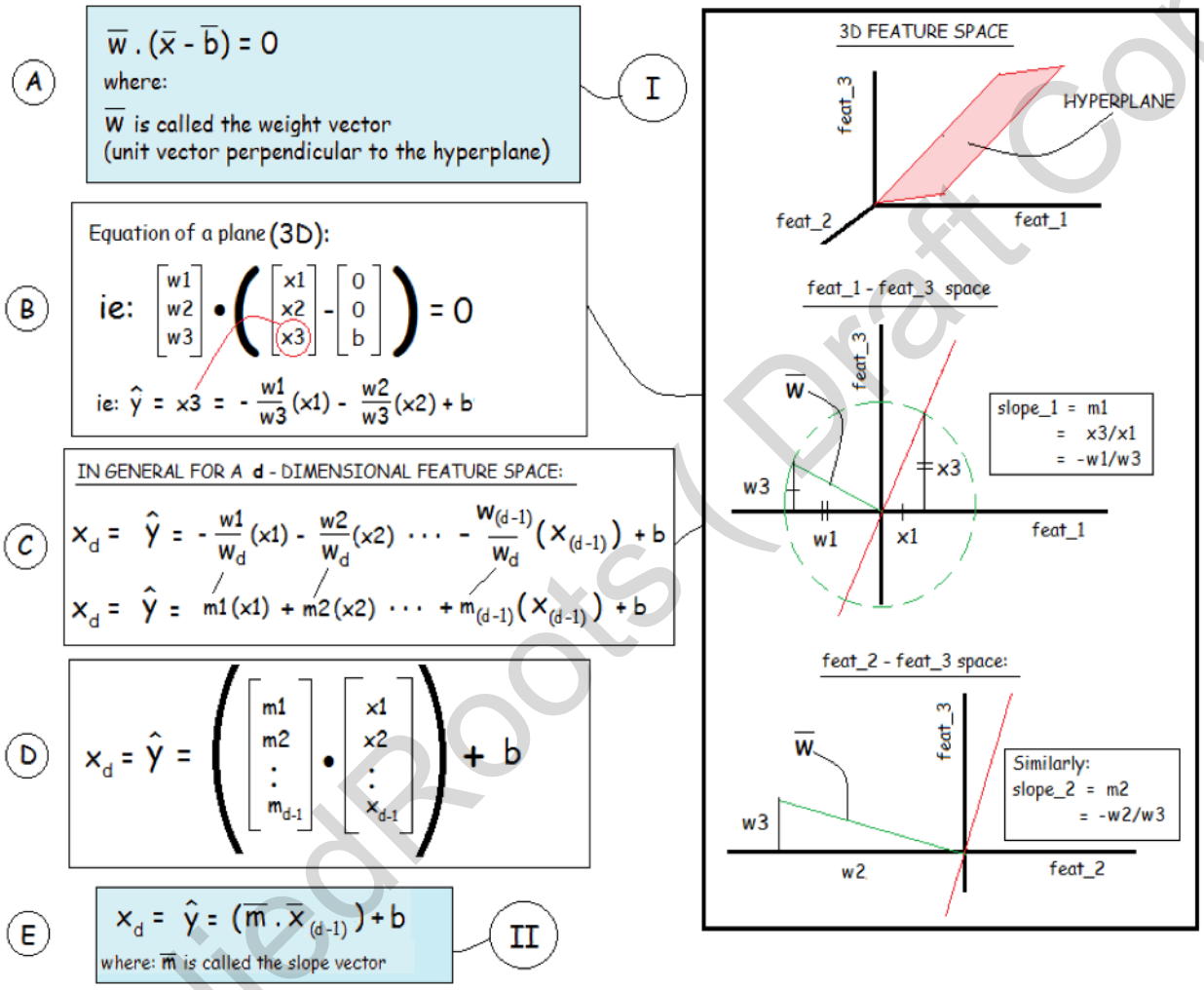


The Gradient at any point **P** in the input space of the function is a vector that has the same direction as the slope of the function's curve corresponding to that point (ie : **P'**) and has a magnitude that is proportional to the magnitude of that slope. In other words the Gradient at any point in the input space of the function is a vector that points in the direction of maximum increase in the value of the function y = f(x1, x2). In the image above, the blue arrows try to approximately represent the gradients at all points in the input space. As can be seen they all point towards maximum increase in the value of **y** from the point being considered. Also the size of these arrows are larger for those that correspond to points on the function's curve where the slope is steeper.

Given the above mentioned facts, if we were to randomly select a point in the input space of the function and then consecutively follow the opposite direction of the gradient vectors from that point on, we will eventually reach that point in the input space that produces the minimum value of y (ie: The minima of the function y = f(x1, x2)). It is this principle that is used in the Gradient Descent algorithm which will be introduced further on.

EQUATION OF A PLANE REVISITED (WEIGHT VECTOR FORM & SLOPE VECTOR FORM):

Consider the image shown below. A hyperplane can be expressed in terms of its weight vector **w (eqn I)** or in terms of the vector containing the slopes of the hyperplane (slope vector **m**) - **(eqn II)** . As seen in the geometrical diagrams on the right side of this image, the slope vector can be derived from the weight vector and vice-versa.



For a 3D dataset, the relationship between the weight vector & the slope vector can be expressed as follows:

$$\bar{m} = \begin{bmatrix} m1 \\ m2 \end{bmatrix} = \begin{bmatrix} -w1 / w3 \\ -w2 / w3 \end{bmatrix} \quad \& \quad \bar{w} = \begin{bmatrix} w1 \\ w2 \\ w3 \end{bmatrix} = \begin{bmatrix} -m1\, w3 \\ -m2\, w3 \\ w3 \end{bmatrix}$$

For a d- dimensional feature space:

$\bar{w}$ will be d-dimensional

$\bar{m}$ will be (d-1) dimensional

OPTIMIZATION:

Once an objective function is defined, the next step is to find some solution that satisfies it. The process of finding an adequate solution to the objective function is called Optimization. In other words, optimization refers to the mathematical task of finding the input variables that produce the minimum value (or maximum) of a function that is subjected to a set of constraints. The theory of Optimization is a vast subject by itself, which consists of many optimization techniques/algorithms. One of the most simple and widely used optimization algorithms is the **Gradient Descent algorithm.**

Consider the Loss Function for Logistic Regression (ie: Log Loss) in a 3D feature space:

$$\text{Loss, } L = \frac{1}{n}\sum_{i=1}^{n} - y_i \log z_i - (1 - y_i) \log (1 - z_i)$$

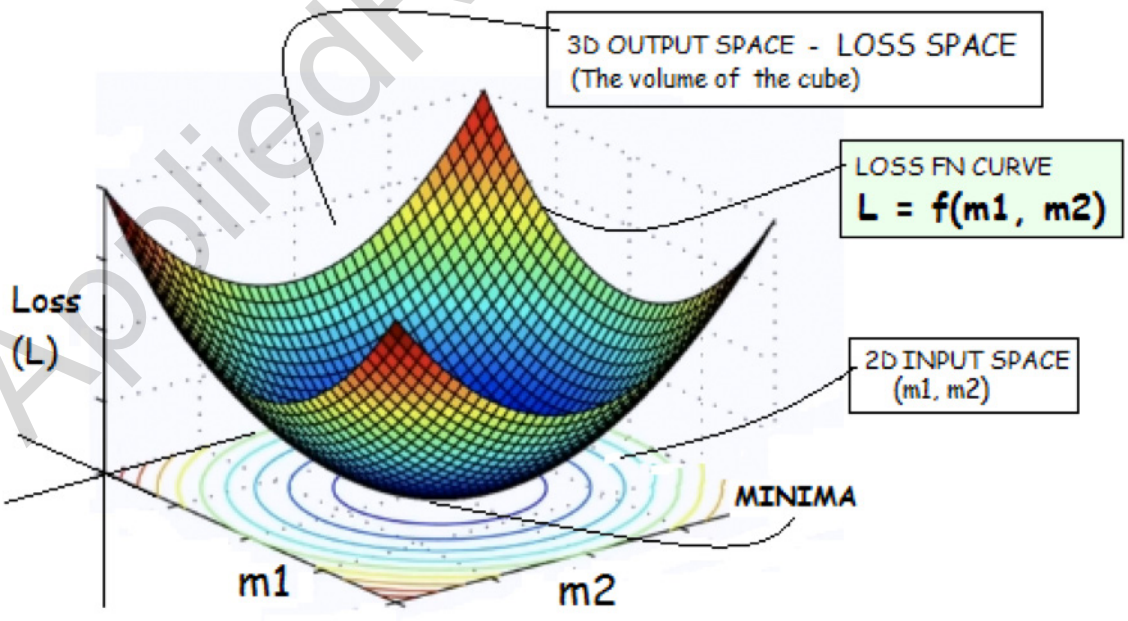
Where:

$$z_i = \sigma(d_i) \;\&\; d_i = \overline{w} \cdot (\overline{x_i} - \overline{b})$$

Given data **X** and corresponding labels **y**, Loss **L** can be expressed as a function of the slope vector $\overline{m}$ and intercept **b**, since the weight vector $\overline{w}$ can be derived from the slope vector.

Therefore for a 3D feature space:
$L = f(\overline{m}, b) = f(m1, m2, b)$

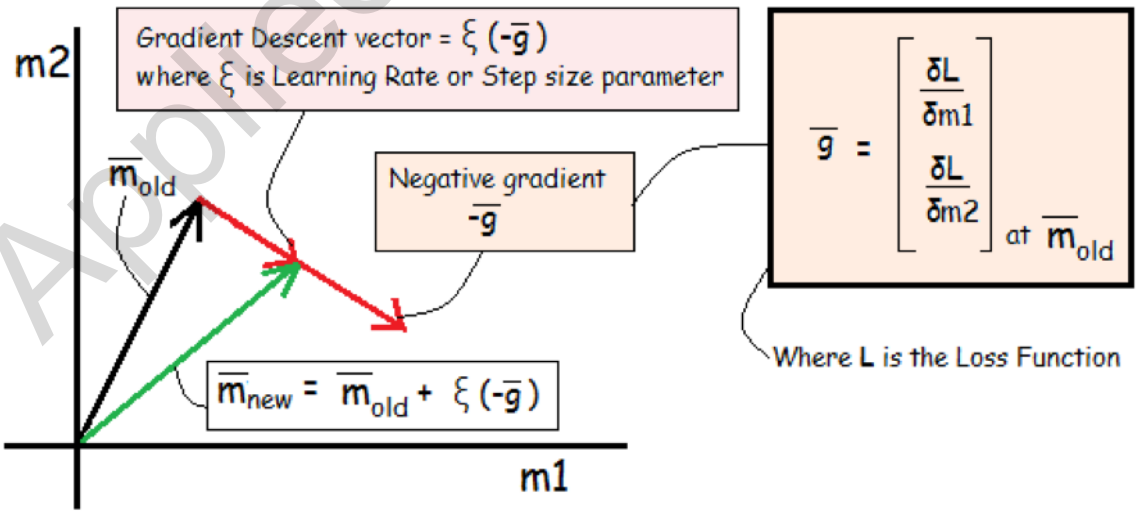
For the sake of simplicity let as assume that the dataset is best separable by a hyperplane that passes through the origin and hence we will not consider the intercept term in the equation above. Then Loss function described above becomes a function of only the slope vector. Such a loss function could be visualized as shown below:



In the image above, every point (Loss) on the Loss Curve corresponds to a slope vector which further corresponds to a hyperplane. The aim of the objective function as mentioned before, is to find that slope vector which corresponds to the minima of the Loss Curve or in other words find the hyperplane which results in the least loss. The gradient descent algorithm satisfies this aim by implementing the following steps:

1. Randomly select a slope vector.
2. This slope vector will correspond to a hyperplane, whose loss can be computed using the loss function. Compute loss corresponding to this hyperplane using all the data points in the dataset.
3. Compute the gradient of the Loss function wrt the slope vector just selected.
4. Take a minuscule "step" in the direction opposite to the gradient just computed. This will result in a new slope vector which will correspond to a new hyperplane that produces less loss than the previous hyperplane.
5. Repeat steps 2, 3 & 4 till we arrive at the slope vector corresponding to the minima of the loss curve (Stop when further iterations result in an increase in loss).
6. This final hyperplane corresponding to the slope vector at the minima to the loss curve will be the hyperplane that produces the best classification.

In step 4 described above the "minuscule" step is achieved by scaling the negative gradient with a very small positive scalar value called the step size parameter thus resulting in a vector (**"gradient descent vector**") that has the opposite direction as the gradient, but having a minuscule magnitude. Step 4 can then be interpreted in terms of the vector addition of the "old" slope vector with the "gradient descent vector" that corresponds to this old slope vector, thus resulting in the "new" slope vector. The image below further illustrates this point:



THE LEARNING RATE OR STEP SIZE PARAMETER:

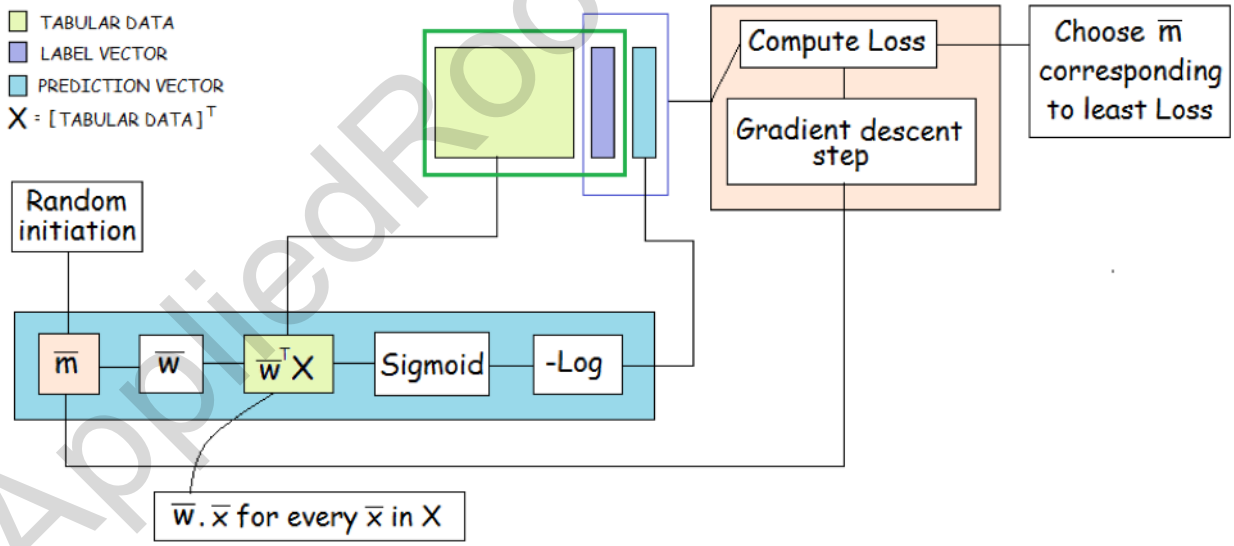
The step size parameter gives us the ability to control the magnitude of the Gradient Descent Vector. This is important because the shape/curvature of the loss curve could be complex if the Loss Space is high dimensional. Hence to determine the optimal long term path towards least loss (ie: the minima) one iteratively does the following:

1. Take a minuscule step in the direction of maximum decrease in loss (move along the gradient descent vector).
2. Recompute the direction of maximum decrease in loss wrt the new position (recompute new gradient descent vector).

The smaller the step size parameter, the more the ability of the gradient descent algorithm to handle any complexity that might exist in the loss curve.

The optimal magnitude of the step size parameter is dependent on the nature of the particular loss curve being considered. Generally a smaller step size means smaller granularity and hence more accurate directions at each step of the gradient descent, but this results in an increase in the time required to reach convergence (ie: reach the minima of the loss curve). Larger step sizes come with the danger of "overstepping" the minima or the algorithm being led astray by the complexity of the loss curve

OVERVIEW OF THE LOGISTIC REGRESSION ALGORITHM:



COMPUTING GRADIENTS IN PYTHON USING TENSORFLOW GRADIENT TAPE:

Gradients for any differentiable, vector input - scalar output, function can be easily computed using the tensorflow library as shown below:

```
def fn_compute_grads(x1, x2):

    with tf.GradientTape(persistent=True) as g:

        g.watch([x1, x2])
        y = 7*x1**2 + x2**2

    gradient  = g.gradient(y, [x1, x2]) #---GRADIENT of y wrt x1, x2
    del g  #----Drop the reference to the tape

    return  np.array([i.numpy() for i in gradient])
```

```
# GRADIENT OF FUNCTION:
# y = 7*x1**2 + x2**2 @ x1, x2 = 4.0, 7.7

x1 = tf.Variable(4.0)
x2 = tf.Variable(7.7)

gradient = fn_compute_grads(x1, x2)
gradient
```

```
array([56. , 15.4], dtype=float32)
```

IMPLEMENTATION OF GRADIENT DESCENT FOR LOGISTIC REGRESSION:

Shown below are some of the main functions used to perform Logistic Regression:

```
@tf.function # tf decorator indicating fn for which derivatives have to be found
def fn_Loss(y, m1, m2, b, x1, x2, x3):
    '''
    The Loss fn we want to differentiate and find gradient
    Gradient: vector(Derivatives of a fn wrt all its input variables)
    '''
    w3 = (1/(m1**2 + m2**2 + 1))*(1/2) #---- since w is a unit vector
    d = m1*w3*x1 + m2*w3*x2 + w3*(x3-b) #--- d = w.(x-b)

    Loss1 = -y*tf.math.log(tf.math.sigmoid(d))
    Loss2 = -(1-y)*(1-tf.math.log(tf.math.sigmoid(d)))

    Loss = Loss1 + Loss2

    return Loss
```

```python
def fn_grads(x, y, loss_fn_input_vec):
    '''
    Fn to find Loss gradient given x, y & a Hyperplane(m1, m2, b)
    x: a data pt
    y: Label corresponding to x &
    loss_fn_input_vec: array([slope_1, slope_2, intercept])
    '''
    m1, m2, b = [tf.Variable(var) for var in [*loss_fn_input_vec]]
    x1, x2, x3, y = [tf.Variable(var) for var in [*x, y]]

    wrt =  [m1, m2, b] #---------------- Differentiate Loss fn wrt these vars
    with tf.GradientTape(persistent=True) as g:

        g.watch(wrt)
        Loss = fn_Loss(y, m1, m2, b, x1, x2, x3)

    loss_fn_gradient  = g.gradient(Loss, wrt) #---Gradient of loss fn
    loss_fn_gradient = np.array([i.numpy() for i in loss_fn_gradient], dtype = np.float32)
    del g  #----Drop the reference to the tape

    return  loss_fn_gradient
```

```python
def fn_gradient_vec(loss_fn_input_vec, Xy):
    '''
    Fn to compute gradient for the whole dataset X given Hyperplane(m1, m2, b)
    loss_fn_input_vec: array([slope_1, slope_2, intercept])
    Xy : concatenation of X_tab & y
    '''
    gradient_vec = [fn_grads(row[:-1], row[-1], loss_fn_input_vec) for row in Xy]
    gradient_vec= np.array(gradient_vec).mean(axis = 0)

    return gradient_vec
```

```python
def fn_grad_descent(Xy, X_tab, y, lr = 0.01, n_iters = 300):
    '''
    X_tab : Tabular feature_set
    Xy : concatenation of X_tab & y
    '''
    label_vec = y
    loss_fn_input_vec = np.array(np.random.randn(3),  dtype = np.float32)
    list0_loss_fn_input_vec, list0_losses = [], []
    pbar = ProgressBar()
    for iter in pbar(range(n_iters)):

        log_loss = fn_compute_log_loss(X_tab, loss_fn_input_vec, label_vec)
        gradient_vec = fn_gradient_vec(loss_fn_input_vec, Xy)
        loss_fn_input_vec = loss_fn_input_vec - (lr * gradient_vec)

        list0_losses.append(log_loss)
        list0_loss_fn_input_vec.append(loss_fn_input_vec)

    return list0_loss_fn_input_vec, list0_losses
```

Consider the following 3D dataset:

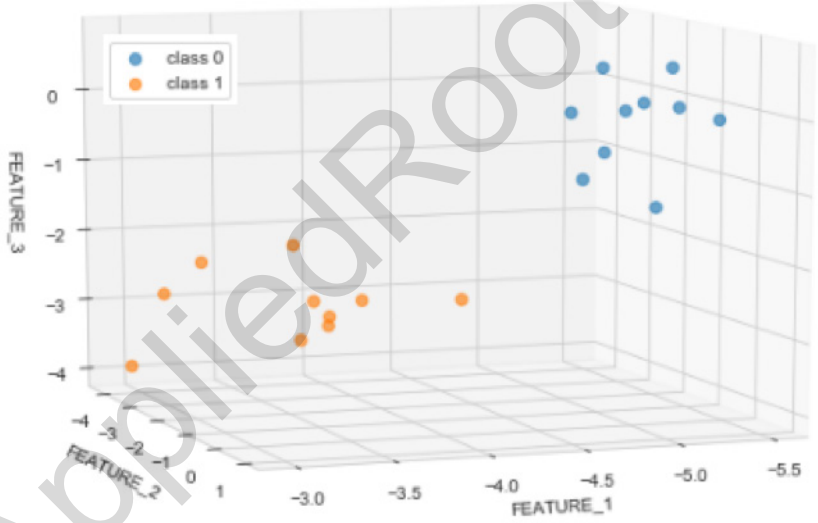
```
df_3D_classifn_dataset = pd.read_csv('df_3D_classifn_dataset.csv')
display(df_3D_classifn_dataset.head(), df_3D_classifn_dataset.describe())
```

	feat_1	feat_2	feat_3	labels
0	-4.883475	0.386004	0.109937	0
1	-4.951108	0.537778	0.227841	0
2	-5.261570	-0.072180	0.065459	0
3	-3.752165	-3.419918	-2.150532	1
4	-4.764894	-0.180544	-0.878978	0

	feat_1	feat_2	feat_3	labels
count	20.000000	20.000000	20.000000	20.000000
mean	-4.350767	-1.614879	-1.547560	0.500000
std	0.745383	1.962025	1.607767	0.512989
min	-5.503191	-4.257823	-3.931761	0.000000
25%	-4.894930	-3.408220	-2.992776	0.000000
50%	-4.706831	-1.496267	-1.630821	0.500000
75%	-3.862717	0.280264	0.047298	1.000000
max	-2.984250	1.000819	0.686859	1.000000

Shown below is the example 3D data presented above, represented in its feature space:

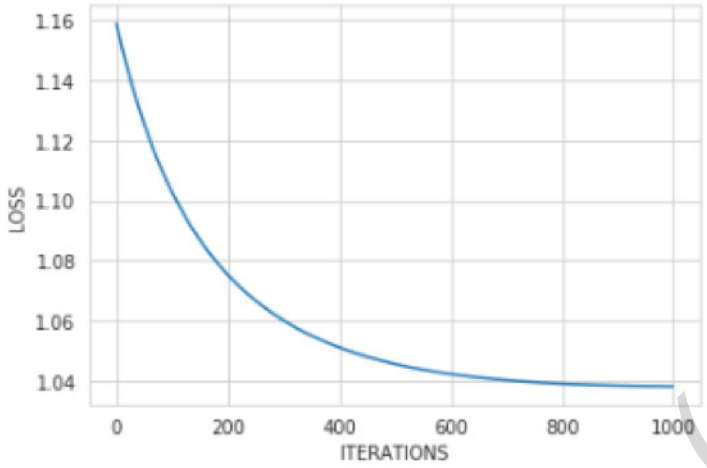
```
fn_plot_3D_scatter(df_3D_classifn_dataset)
```



Shown below is the decrease in Log Loss resulting from applying the Logistic Regression algorithm to the above dataset using a step size parameter of 0.05, for 1000 iterations:

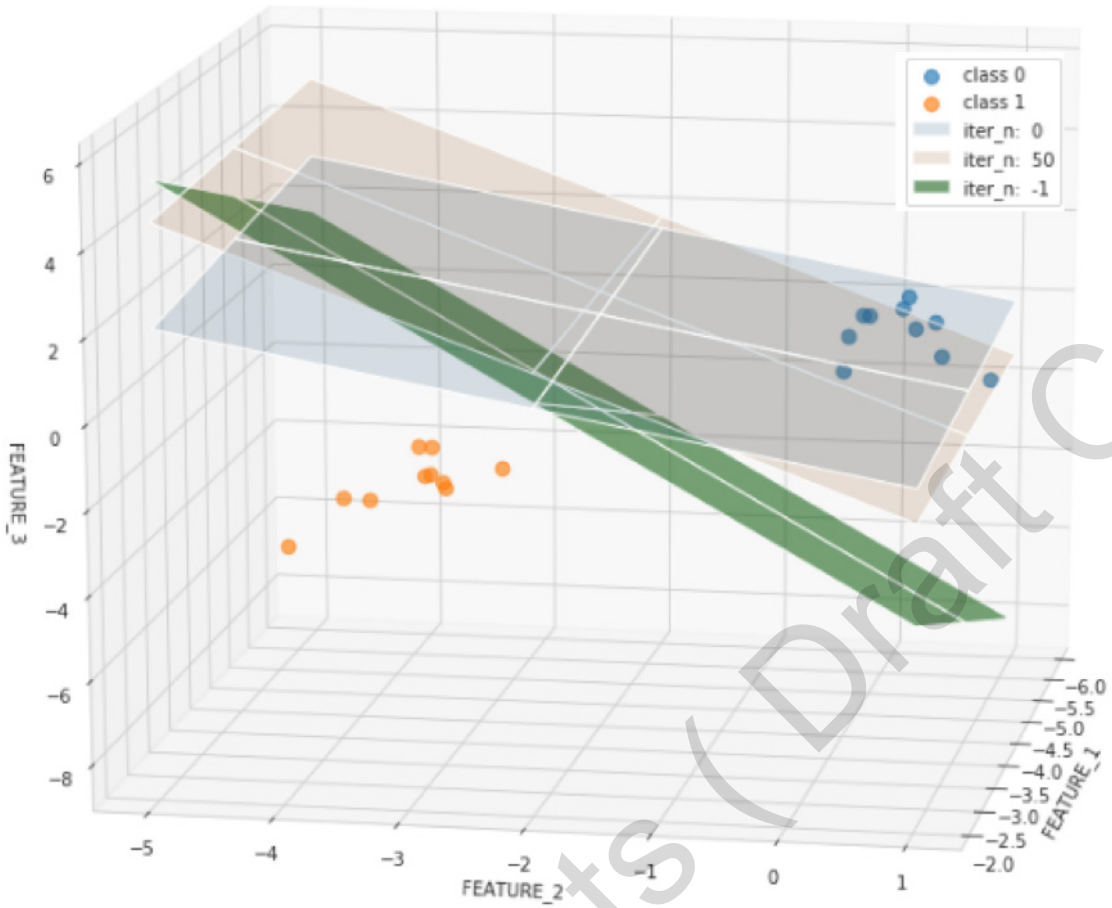
```
Xy = df_3D_classifn_dataset.values
X  = df_3D_classifn_dataset.values[:, :-1]
y  = df_3D_classifn_dataset.values[:, -1]

list0_loss_fn_input_vec, list0_losses = fn_grad_descent(Xy, X, y,
                                                        lr = 0.05, n_iters = 1000)
fn_plot_loss(list0_losses)
```

```
100% (1000 of 1000) |####################| Elapsed Time: 0:01:08 Time:  0:01:08
```



Shown below is the visualization of the hyperplanes at different stages of the gradient descent:

```
x1_limits, x2_limits = (-6, -2), (-5, 1)

list0_m_b_vecs = list0_loss_fn_input_vec
list0_iter_n = [0, 50, -1]

fn_plot_hyperplane(df_3D_classifn_dataset, list0_m_b_vecs, list0_iter_n,
                   x1_limits, x2_limits, view = (15, 10))
```



2. FUNDAMENTALS OF ML – 3 (REGRESSION)

LINEAR REGRESSION:

As defined earlier, Regression is a Supervised Machine Learning task where the labels that need to be predicted are continuous in nature. Consider the data shown below:

```
1  df_regression_3D = pd.read_csv('df_regression_3D.csv')
2
3  display(df_regression_3D.shape, df_regression_3D.head(), df_regression_3D.describe())
```

(50, 3)

	feat_1	feat_2	labels
0	0.775786	0.762097	0.807087
1	0.148123	0.792339	0.346457
2	0.055800	0.925403	0.303150
3	0.509976	0.832661	0.665354
4	0.609063	0.217742	0.444882

	feat_1	feat_2	labels
count	50.000000	50.000000	50.000000
mean	0.510497	0.473710	0.493543
std	0.305880	0.284985	0.198208
min	0.026716	0.028226	0.125984
25%	0.228103	0.240423	0.337598
50%	0.549882	0.467742	0.444882
75%	0.769445	0.699093	0.622047
max	0.990531	0.995968	0.937008

Given a set of random variables: feat_1, feat_2, etc (ie: X) and their corresponding Labels (y) we wish to create a predictive mechanism, which can learn the relationships between them (ie: X & y), such that it becomes capable of estimating the label values of new 'X' data of the same kind.

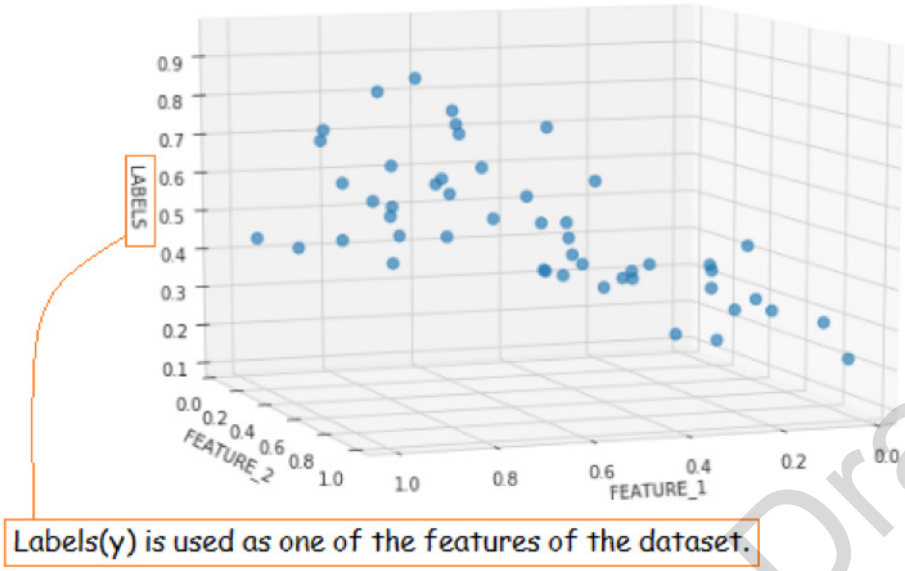
Using Linear Algebra we could frame the Regression task as follows:

1. Since the Labels for regression data are continuous in nature, they could be interpreted as a random variable representing another feature of the data-object in consideration.
2. The data is then represented in a feature space that expresses the above perspective, ie: We create a new axes, which is orthogonal to all the other axes to represent the labels. (see image below).
3. We then define the Learning as : " Finding the hyperplane in this modified feature space, which has the least average distance from the data-objects represented in it."

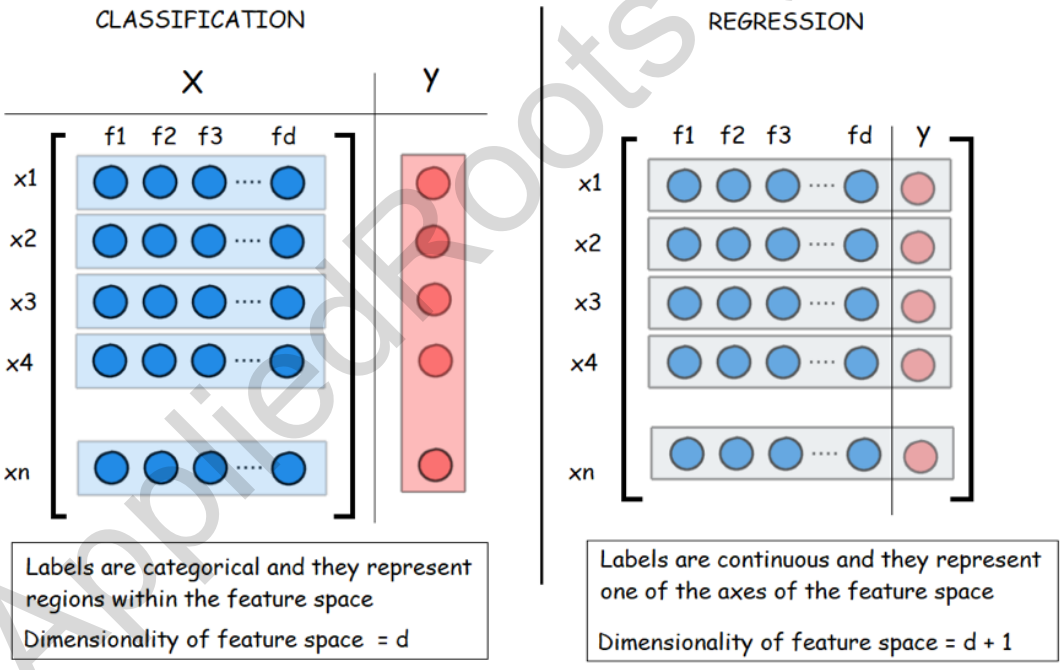
So in essence, the regression model learns from data containing the complete feature set Xy (ie: X concatenated with y) and then becomes capable of predicting the values of "missing" feature (y) given new 'X' data.

The example data above represented in its modified feature space is shown below:

```
1  fn_plot_3D_scatter(df_regression_3D)
```



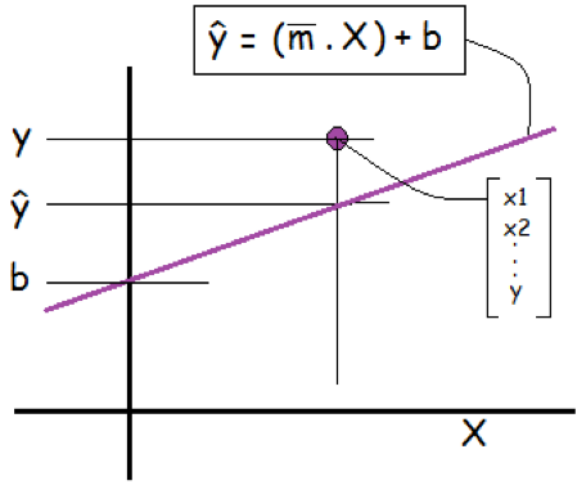
CLASSIFICATION & REGRESSION - DIFFERENCE IN FEATURE SPACE:



THE LINEAR REGRESSION OBJECTIVE FUNCTION:

The objective function for Linear Regression could be expressed as: " Find that hyperplane which minimizes the average squared error". In the case of regression the objective function is defined using the intercept form of the equation of a plane. This is because, as described earlier, in regression we are in effect predicting the values of a "missing" feature and the intercept form of the equation neatly isolates "y" (ie: the missing feature):

The image below describes this objective function in relation to its variables:

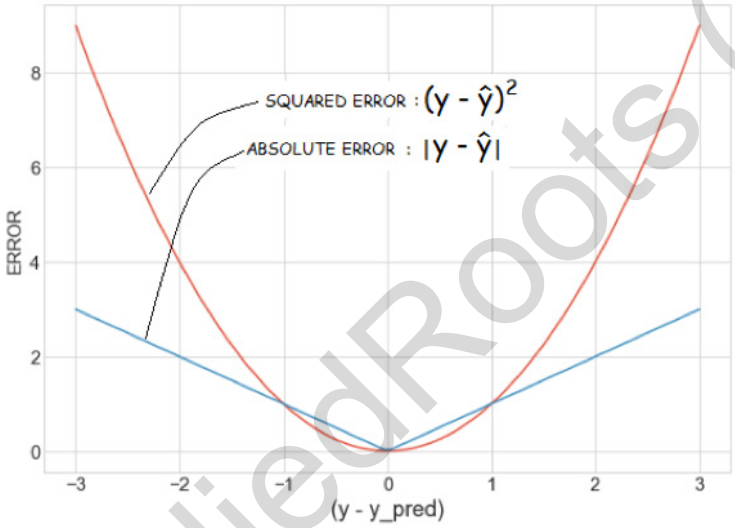


OBJECTIVE FUNCTION:

$$\bar{m}, b = \text{argmin} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

CHARACTERISTICS OF SQUARED ERROR:

Shown in the image below is a comparison of the curves of squared and absolute errors:



As seen in the plot above, the squared error increases exponentially as the difference between the predicted and actual label values increases. This means that when it is used as the loss function during gradient descent optimization, it causes the algorithm to respond more drastically to data points that are further away from the hyperplane that those close to it. This is advantageous if the data is free of outliers. Furthermore squared error is easily differentiable and hence well suited for computing step gradients.

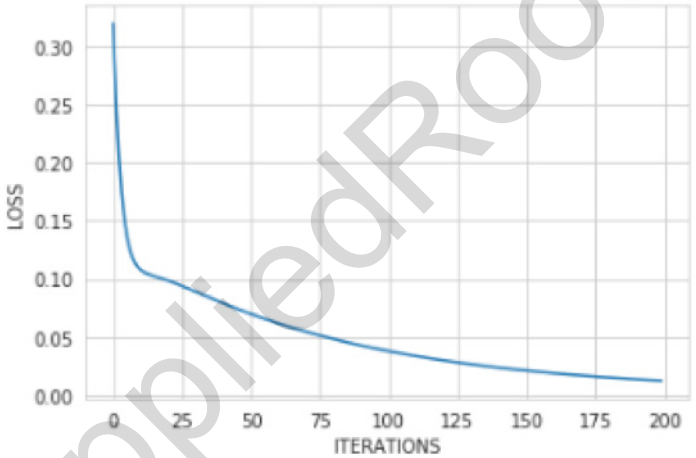
IMPLEMENTATION OF GRADIENT DESCENT FOR LINEAR REGRESSION:

The same code (with some minor tweaks) used for Logistic regression can be used for Linear regression, only the loss function needs to be redefined as:

```
@tf.function
def fn_Loss(y, m1, m2, b, x1, x2):
    '''
    Loss fn we want to differentiate and find gradient
    Gradient: [list of derivatives of a fn wrt all its input variables]
    '''
    y_pred = m1*x1 + m2*x2 + b
    Loss = (y - y_pred)**2

    return Loss
```

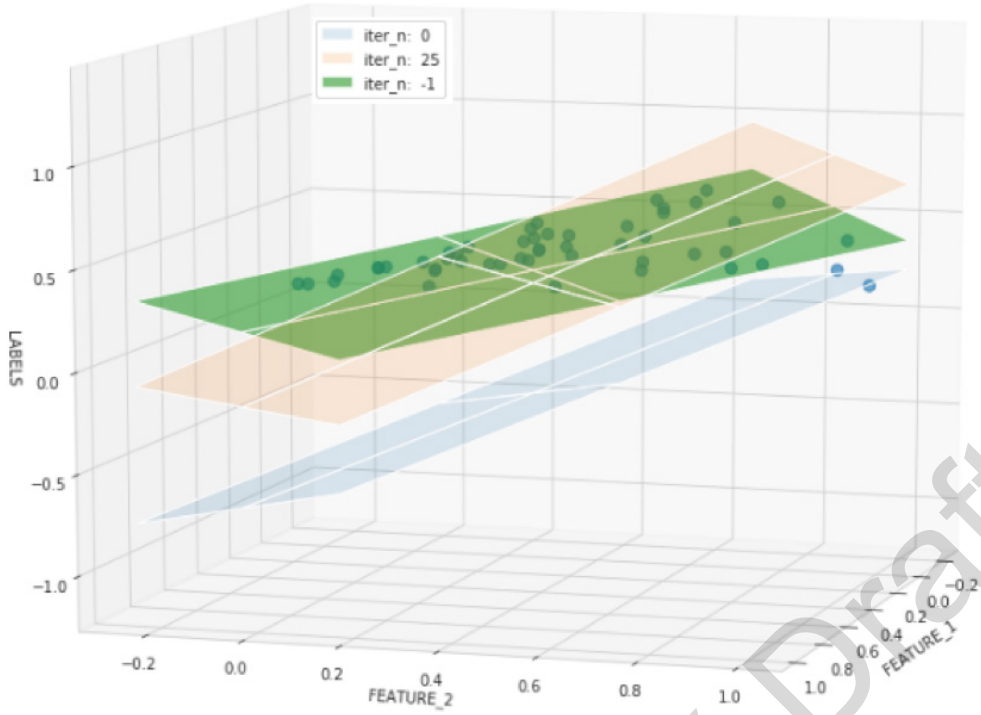
Shown below is the decrease in Log Loss resulting from applying the Logistic Regression algorithm to the above dataset using a step size parameter of 0.05, for 200 iterations:

```
df_regression_3D = df_regression_3D.astype(np.float32)
Xy = df_regression_3D.values
X_tab  = df_regression_3D.values[:, :-1]
y  = df_regression_3D.values[:, -1]

list0_loss_fn_input_vec, list0_losses = fn_grad_descent(Xy, X_tab, y,
                                                        lr = 0.05, n_iters = 200)
fn_plot_loss(list0_losses)
```



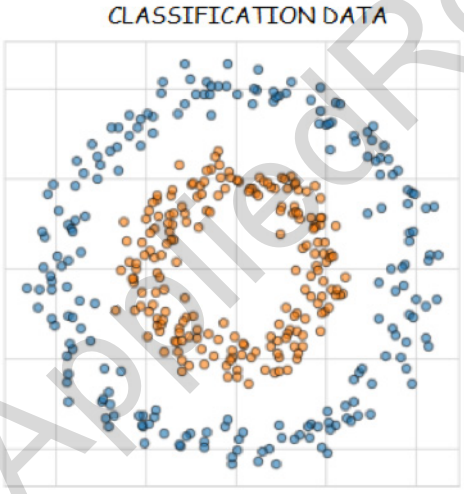
Shown below is the visualization of the hyperplanes at different stages of the gradient descent:

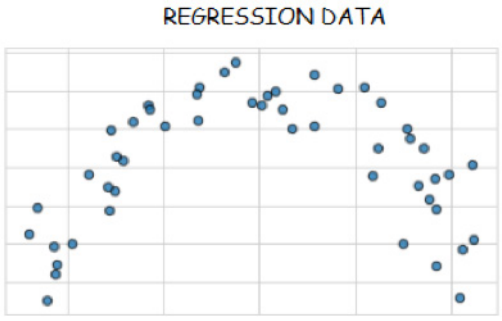
```
x1_limits, x2_limits = (-0.25, 1), (-0.25, 1)

list0_m_b_vecs = list0_loss_fn_input_vec
list0_iter_n = [0, 25, -1]

fn_plot_hyperplane(df_regression_3D, list0_m_b_vecs, list0_iter_n,
                   x1_limits, x2_limits, view = (10, 17))
```



SUMMARY - LOGISTIC REGRESSION & LINEAR REGRESSION:

Both Logistic Regression & Linear Regression are "Linear Models", in the sense that they use linear (or uncurved) hyperplanes to perform their tasks. This means that these models would be ineffective when dealing with "Non-Linear" data (examples of which are shown below):





Though it seems from the above examples that Linear Models generally cannot be used for classification/regression of nonlinear data, this limitation generally does not apply as the dimensionality of the feature space increases. In other words, Linear Models become quite powerful when working with high dimensional data. This is because linear surfaces in high dimensional feature spaces acquire non linear attributes, due to the inherent nonlinear nature of high dimensional feature spaces.

3. FUNDAMENTALS OF ML – 4

(REGULARIZATION & HYPERPARAMETER TUNING)

FUNDAMENTALS OF ML – 4

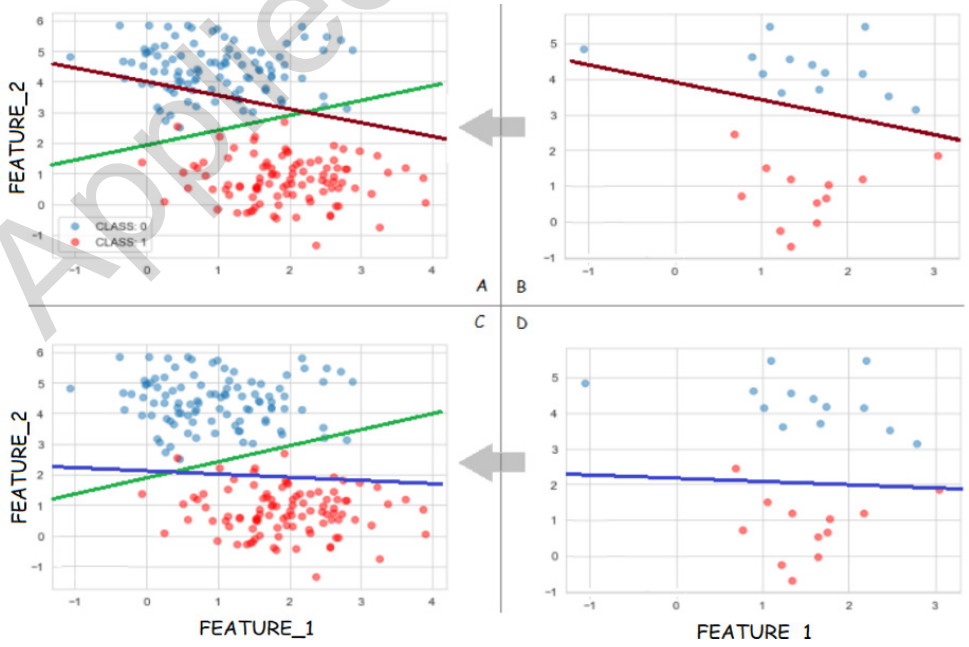
(REGULARIZATION & HYPERPARAMETER TUNING)

REGULARIZATION – BASIC CONCEPT:

A sample is a carefully chosen subset of a population on which we want to automate a particular decision making process (Classification or Regression). Ideally the techniques employed to collect & curate such a sample, should be such that they ensure that the distribution of the parent population is captured by the sampled data as much as possible.

Irrespective of how big the sample size is or how effective the sampling process was at capturing the distribution of the population, the sample will still always be an approximation of the population. This means that the best possible hyperplane with respect to the sample data, may not necessarily be the best hyperplane with respect to the population. Due to this fact, it is necessary that the machine learning model be trained such that it does not "over learn" the characteristics of the sample, but instead learn only those characteristics that it has in common with the population. In other words we do not want our model to "overfit" the sample data, instead we want it to generalize for the population, so that the model could be applied to all other data in the real world. Regularization is a machine learning/ statistical technique that does just that.

Consider the image shown below. For the sake of explainability, imagine that the scatter plots on the left represent a hypothetical population (for which we want to create a classification model) and the scatter plots on the left are those of a sample. The aim of Machine Learning is to find a hyperplane that performs best on the population by using the sample at hand (since in most cases it is impossible to have access to the entire population).

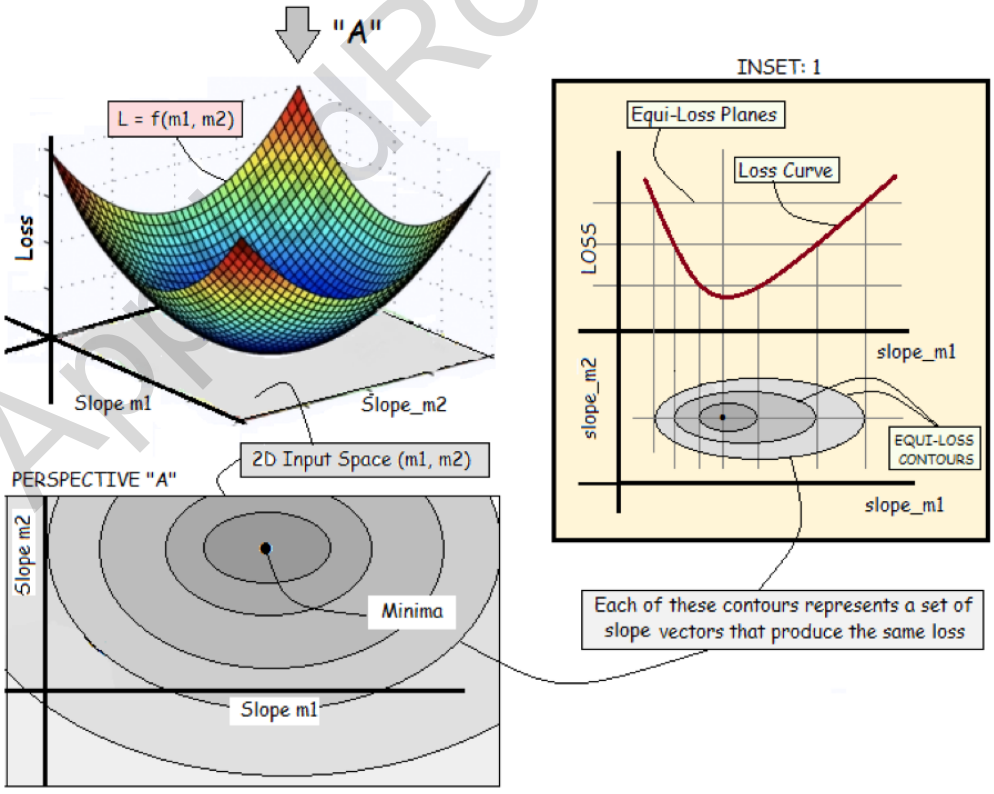


In the image above, the green hyperplane is the best hyperplane with respect to the Population and the brown hyperplane is the best hyperplane with respect to the sample. The brown hyperplane is said to have "**overfit**" to the sample, or in other words - The model has found the hyperplane that corresponds to the minima of this particular sample's loss curve. But since this minima may not be the same as that of the Population, this hyperplane may not be able to "**generalize**" its performance to new data (ie: other samples from the same population).

Regularization in essence is a technique that constrains the "learning capacity" of the model, with the aim of learning a hyperplane that is more general. The Blue hyperplane in the above figure represents a hyperplane that is the outcome of regularization (or constrained optimization). It performs suboptimally with respect to the sample it was trained on, but may perform much better than the brown hyperplane (which is an outcome of unconstrained optimization) when applied to the population in general.

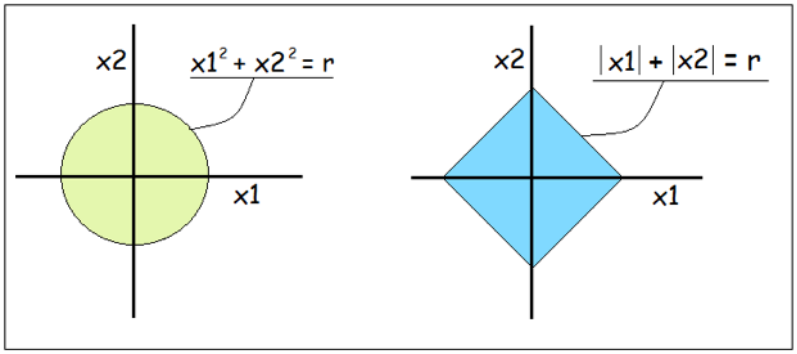
EQUI-LOSS CONTOURS (THE LOSS CURVE REVISITED):

Consider the image shown below. As described before, the loss curve is the outcome of applying the loss function over a range of input values (ie: m1, m2). If we were to slice/intersect the loss curve using multiple equi-loss planes placed at equal intervals (see inset:1), each intersection would produce a contour on the associated equi-loss plane.
The projection of this contour onto the input space produces equi-loss contours. Each of these equi-loss contours thus represent a set of input values (m1, m2) that produce the same loss.



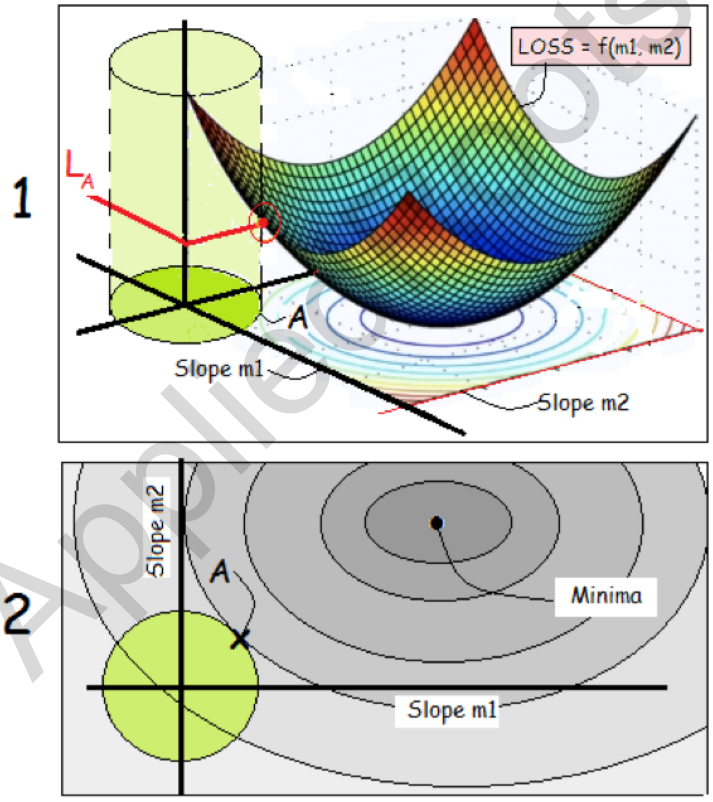
SHAPES ASSOCIATED WITH L2 & L1 NORMS:

Shown below are shapes associated with L2 & L1 norms. By constraining the square of the L2 norm to be equal to some value **r**, we get a circular/spherical shape which represents all vectors that fulfill that constraint. Similarly, by constraining the L1 norm to some value r we get a diamond/rhombus shape.



L2 REGULARIZATION (SPATIAL PERSPECTIVE):

Shown below in images **1** & **2** is a visualization of the effect of applying L2 regularization while optimizing for the best hyperplane, for a 3D dataset (Classification/regression).



REGULARIZED OBJ FUNCTION

$\overline{m} = \text{argmin}\,(\text{Loss} + \lambda r)$

λr is called the Regularization Term

- L2 Regularization:

$r = m_1^2 + m_2^2$

Creates a spherical constraint on the input space.

- L1 Regularization:

$r = |m_1| + |m_2|$

Creates a diamond shaped constraint on the input space.

The resulting effect of adding the regularization term λr to the objective function (as shown above), is that it creates a constraint on the input space of the Loss Function (ie: it creates limits on the range of values **m1, m2** can take).

Image 2 shows the effect of applying L2 regularization - which results in the green spherical shaped constraint on the input space. This means that the model is now constrained to choose only those hyperplanes that correspond to the slope vectors m = [m1, m2] that lie within this constrained space. The size of this constrained space is a function of the data and the hyperparameter λ. (It is called a " hyper" parameter because it is not a parameter that the objective function optimizes, but a parameter that is chosen beforehand, external to the optimization process).

Regularization can also be interpreted using **image 1**. Constraining the input space, in effect constrains the model to find the minima, of only that part of the loss curve, that corresponds to that constrained input space (ie: the model is constrained to find the minima of only that part of the loss curve that lies within the green cylindrical space depicted). Under this condition, the loss LA corresponding to point A on the loss curve is the least possible loss and the hyperplane corresponding to this point is the best hyperplane.

L1 regularization also produces the same effects described above, except that the constrained space will be diamond shaped.

REGULARIZATION (MATHEMATICAL PERSPECTIVE):

Given a particular loss function, the value of r that the regularized objective function finally settles on, depends on the data and value of λ. At λ = 1, the effect of hyperparameter λ will be neutral and the value of r will completely depend on the data. The table below shows the values of Loss, r & the sum of L & r (ie L_r) for different points on the loss curve. The first row in the table represents the minima of the Loss Curve, while the other rows represent other points on the loss curve.

$$\bar{m} = \underset{\bar{m}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \underbrace{\text{Loss} + \lambda r}_{L_r} \qquad \text{where } r = m1^2 + m2^2$$

REGULARIZATION @ λ = 1 :

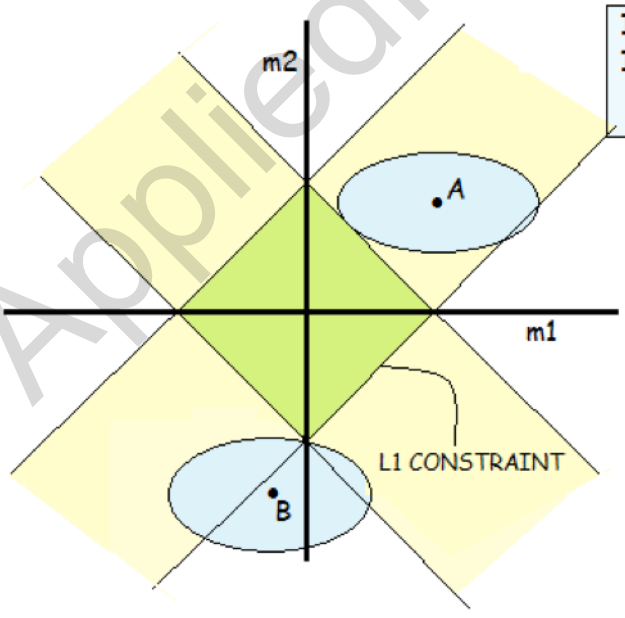
PT ON LOSS CURVE	LOSS	$r = m1^2 + m2^2$	$L_r = \text{LOSS} + (m1^2 + m2^2)$
P_{minima}	L_{minima}	r_{minima}	$L_{r_{minima}}$
P1	L_1	r1	L_{r1}
P2	L_2	r2	L_{r2}
P3	L_3	r3	L_{r3}
⋮	⋮	⋮	⋮

Basically what the constrained model does is, it searches for that point on the loss curve, where the value of **L_r** is the least. Since the minima is the point on the loss curve that corresponds to the least loss, all other points will have a greater loss. But there will be many points on the loss curve where the corresponding value of **L_r (ie: Loss + r)** is lesser than that at the minima (ie: Lesser than **L_r_minima**). The model, by virtue of this modified objective function, will choose that hyperplane (or slope vector **m = [m1, m2]**) which corresponds to the point on the loss curve that produces the least **L_r** value.

At $\lambda = 0$, there is no regularization (model is unconstrained) and at $\lambda = 1$, the model can be said to be "naturally regularized". The effects of λ at different values are relative to its effect at $\lambda = 1$. For values less than 1, the regularization applied is a fraction of that at natural regularization and for values larger than one, the regularization effect is a multiple of that at natural regularization. In general, the greater the value of λ, the greater the constraint applied (or the smaller the available loss function input space will be).

L1 REGULARIZATION & SPARSITY:

L1 regularization is such that, as the amount of regularization (value of λ is increased), more and more components of the slope vector that the constrained objective function produces, become zero. In other words, as the amount of L1 regularization is increased, the slope vector representing the best hyperplane becomes more and more sparse. This could be considered as a form of **feature selection** (ie:Technique for choosing the most important features from a multidimensional dataset). The reason for this phenomenon is because of the diamond/rhomboid shaped constraint, that L1 regularization imposes on the input space of the loss function (ie: the slope space).



If Loss Curve @ A: Non Sparse slope vector
If Loss Curve @ B: Sparse slope vector (m1 = 0)
where slope vector $\overline{m}$ = [m1, m2, ..., md]

The equi-loss contour of a loss curve, can intersect with the constrained L1 space at:
1. Its vertices
2. Its sides/edges

If the center of the equi-loss curve lies outside the yellow region, then the intersection happens at the vertices (B), else - it happens on the edges (A).

Larger the L1 constrain applied, smaller the green square (constrained space) and larger the probability that the "centre" lies outside the yellow region

Consider the image shown above, The blue colored shapes represent loss curves. Only one equi-loss contour is shown for the sake of simplicity. As understood from earlier discussions, the best slope vector (ie: hyperplane), given a regularization constraint, is obtained at the intersection of the constrained space and the loss curve. In case of L1 regularization, this intersection can happen at the edges of the diamond shape or at the vertices. An equi-loss contour will intersect the edges of the L1 constrained space only if its centre lies within the yellow region shown above. If the centre lies outside the yellow region, the intersection happens at the vertices, which implies that one or more components (depending on the dimensionality of the dataset) of the slope vector becomes zero.

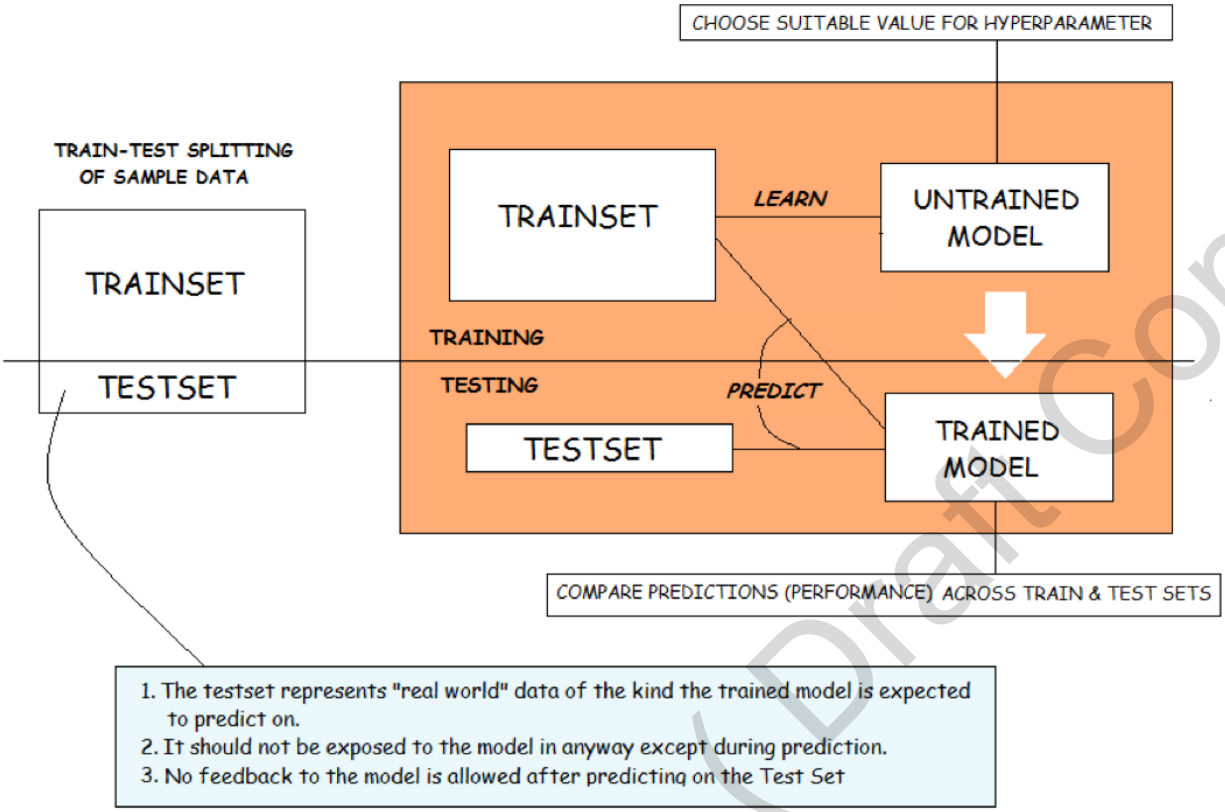
The more the L1 regularization applied, the smaller will be the constrained space (green area) and by association, the smaller will be the yellow region. Smaller the yellow region, more the chances that the centre of the Equi-Loss contours of the loss curve lies outside it and hence more the chances of obtaining a sparse solution.

TRAINING & TESTING:

Most of the discussion till now has been limited to the "Training" part of the machine learning pipeline. Once a model is trained on some sample (ie: the **train set**), it then needs to be tested on another sample belonging to the same population. This "new" sample is called the **test set**.

Testing involves comparing the train set & test set performances of the model and making a judgement call about the generality of the model (ie: whether the model can be expected to generalize its train set performance to new data it was not exposed to).

In actual practice, the train set & the test set are derived from the sample at hand, by splitting it by using a suitable ratio, such that we have more data to train on. Usually a 80% - 20% or a 70% - 30% train set - test set split works fine. The model is then trained on the train set (ie: the best hyperplane is learned using the train set). This trained model is then used to predict on train set & the test set and the performances across these 2 datasets are compared. The image below visually describes the above process.



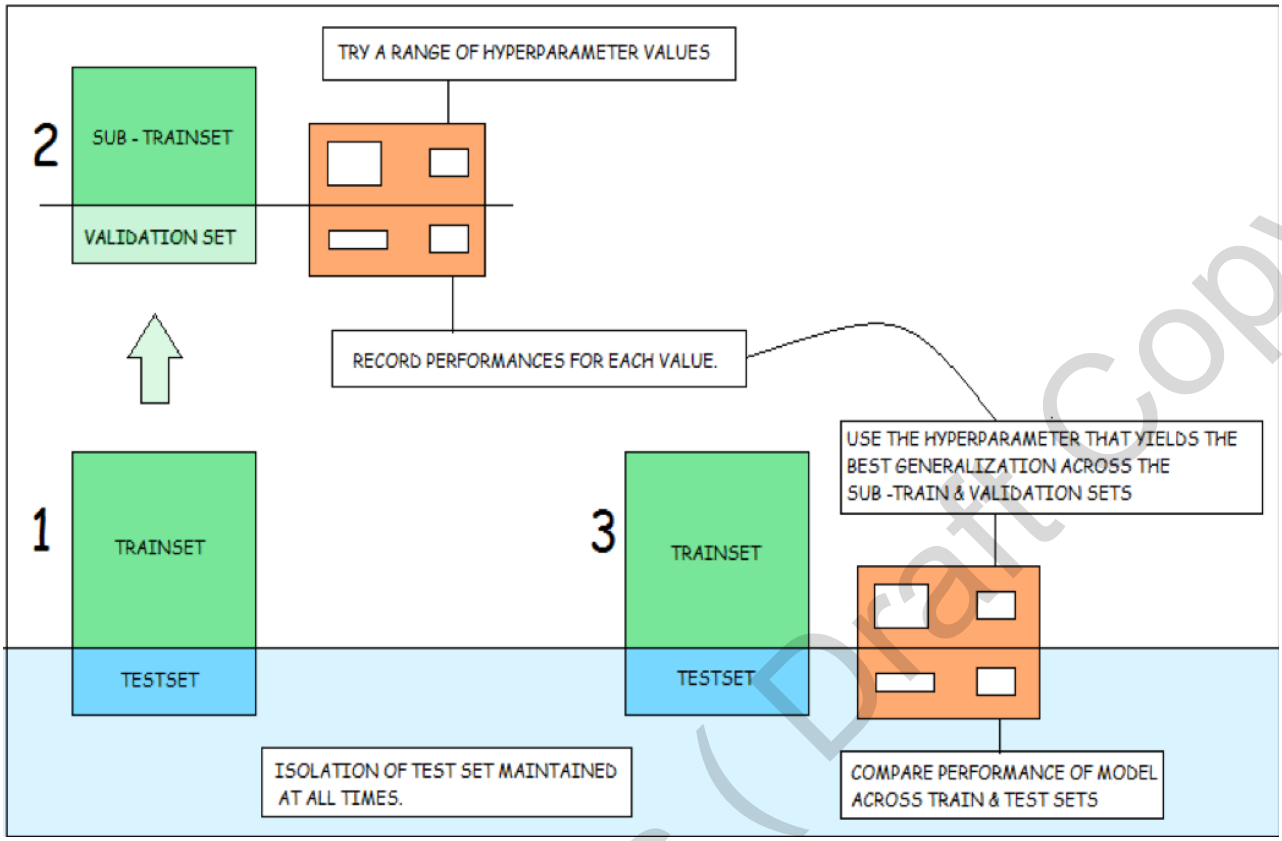
Care should be taken that one does not make the mistake of using the feedback/information received from the model's performance on the test set, to further modify the models parameters, as a means to optimize the model even more. Doing this is akin to the model "learning" from the test set and hence the test set loses its status of being "new" data (ie: It can be considered as a part of the train set after such an action). One can further test such a model only by acquiring more new data and then using this new data as the test set.

Due to the "testing constraints" discussed above, the train-test method is not suitable for **hyperparameter tuning.**

CROSS VALIDATION & HYPERPARAMETER TUNING:

Cross validation is a technique that lets us do hyperparameter tuning (try out a range of hyperparameters) while keeping the test set isolated from the model. It basically consists of further splitting the train set into sub-train and validation sets using a splitting ratio as discussed before.

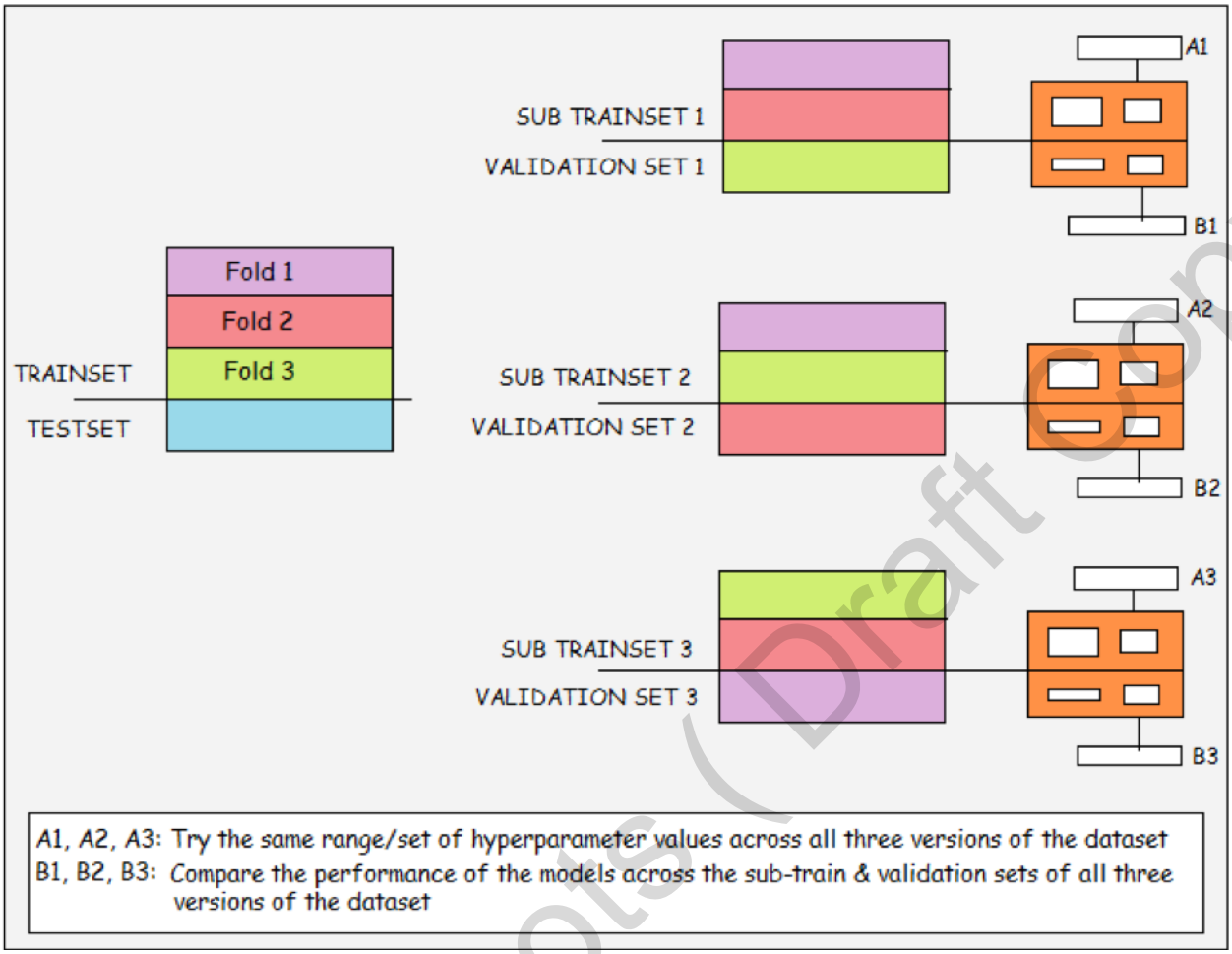
The sub-train set & validation set are then used to train and test the model over a range of hyperparameter values, and the performance of the model over all these values is recorded (step 2 in image shown below). From these ranges of performances, we chose the hyperparameter value that yielded the best generalization across the sub-train set & validation set. We then train & test the model using this chosen hyperparameter value on the original train-test split.(step 3 in image below)



K FOLD CROSS VALIDATION:

K fold cross validation is a more thorough form of cross validation, where the sample at hand is utilized to its full extent, in helping us determine the optimal hyperparameter value(s) for the model. Here, **step 2** of the basic cross validation technique just discussed is modified as shown in the image below.

Different versions of the sub-train & validation sets are created by segregating the train set into '**K**' folds (K = 3 in the image shown below). The model is then trained using a range of hyperparameters over all K folds of the data and the performance of the model is recorded across all iterations. The hyperparameter that gives us the best generalization among all the iterations is then used to train & test the model over the original train-test split.



BIAS & VARIANCE TRADE OFF:

The more a model is constrained/regularized, the lesser its learning capacity (ie: the more "biased" it becomes). In other words the model becomes less responsive to the variation of the data during prediction and will constantly perform badly.

The lesser the regularization, the more the model overfits the training data. Its performance then becomes localized to the training data and does not transfer to other samples. This results in a model that is inconsistent (high variation) in its performance. It may perform extremely well on some data, but may perform mundanely or very badly with respect to some other data.

By performing K fold cross validation and choosing the hyperparameter(s) that exhibits the best generality, we have better chances at achieving an optimal bias-variance tradeoff.

4. BASICS TRAINING & EVALUATION

(LOGISTIC REGRESSION)

BASICS TRAINING & EVALUATION (LOGISTIC REGRESSION)

In the previous book in the chapters on Logistic Regression we implemented the algorithm for the same, from scratch and applied it to a toy data set. We then visualized the separating hyperplane at different iterations of the optimization process. In this chapter we shall explore using the fundamentals of using the scikit learn library for training and evaluation purposes.

NOMENCLATURE / ABBREVIATIONS:

For the purposes of clarity we will be using a specific pattern to name the variables we will using while performing machine learning operations:

1. Capital **X**: refers to the feature set in **tabular format**. In other words, the data points are arranged in a matrix where the columns represent features of the data. Scikit Learn uses tabular data as input instead of the formal linear algebraic matrix form, where vectors are arranged column wise. This is simply for the ease of interpretability/readability of the user.
2. Small **x** : refers to a single data point.
3. Small **y**: refers to the vector containing the labels corresponding to each datapoint in **X**.
4. **tr** or **train**: refers to train set.
5. **ts** or **test**: refers to test set.
6. Suffix **pred**: refers to model predictions (probabilities or labels)
7. Suffix **proba**: refers to probabilities
8. Suffix **acc**: refers to accuracy.
9. Suffix **prec**: refers to precision.
10. Suffix **rec**: refers to recall
11. **df**: refers to a pandas DataFrame
12. **df_Xy**: refers to dataframe of feature set (X) and corresponding labels (y)
13. **df_X**: refers to dataframe containing only feature set - similarly for df_y
14. **idx**: refers to indexes.
15. **model_class**: refers to a predictor Class in Scikit Learn.
16. **model_instance**: refers to an instance of some model_class (ie: untrained model).
17. **model**: refers to a trained model.

MODEL INSTANTIATION:

For binary classification using the Logistic Regression, we first instantiate the Logistic Regression predictor Class as shown below:

```
from sklearn.linear_model import LogisticRegression

model_class = LogisticRegression
model_instance = model_class()

model_instance
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

The above code creates an instance of the LogisticRegression class with following default hyperparameters:

```
LogisticRegression(penalty = 'l2',
                           C = 1.0,
                class_weight = None,
                      solver = 'liblinear')
```

The Hyperparameter 'C' represents the inverse of regularization strength.

The 'solver' hyperparameter represents the kind of optimization method used to implement the Logistic Regression algorithm. Scikit Learn provides us with a set of optimization algorithms to choose from. Some of them (newton-cg, lbfgs, etc) use matrix methods combined with second order derivatives, while others (sag, saga) use some form of gradient descent optimization. Each has their own pros and cons with respect to the size of data that needs to be handled and the type of regularization desired.

The 'class_weight' hyperparameter is used when one encounters imbalanced data. Most machine learning models work best when no particular class (or set of classes) overly dominates the class distribution. This hyperparameter allows us to account for any imbalance in the input data, by providing class weighing strategies.

TRAINING:

We can then train the instantiated model by "fitting it to the labeled data (ie: X_train, y_train) using the 'fit' method as shown below:

```
model = model_instance.fit(X_train, y_train)
```

PREDICTING:

The model now becomes capable of predicting on other samples of the data (ie; X_test). Prediction is achieved by using the '**predict**' method as shown below:

```
y_pred = model.predict(X_test)
y_pred
```

```
array([0, 0, 0, 1, 0, 1, 0, 1, 1, 0], dtype=int64)
```

Scikit Learn provides the **'predict_proba'** method for most predictor classes, if one desires to interpret the model's predictions probabilistically. This is important in the case of Binary Classification as this allows us to manually decide the probability "**threshold**" at which we want the model to decide whether it belongs to a particular class/label. The predict method discussed earlier uses a default threshold of 50% for its decisions (ie: if the probability predicted for class_1 > 0.5, then the predict that the data point belongs to that class_1).

```
y_pred_proba = model.predict_proba(X_test).round(3)
y_pred_proba
```

```
array([[0.99 , 0.01 ],
       [1.   , 0.   ],
       [0.998, 0.002],
       [0.015, 0.985],
       [0.986, 0.014],
       [0.015, 0.985],
       [1.   , 0.   ],
       [0.021, 0.979],
       [0.   , 1.   ],
       [1.   , 0.   ]])
```

The predict_proba method returns a matrix, the columns of which represent the probabilities predicted by the model for each class/label. The dataframe of the outputs of the 'predict' and the 'predict_proba' methods shown below clarifies this further:

```
df_predictions = pd.DataFrame(y_pred_proba, columns = ['proba_0', 'proba_1'])
df_predictions = df_predictions.assign(y_pred = y_pred)

df_predictions.head(6)
```

	proba_0	proba_1	y_pred
0	0.990	0.010	0
1	1.000	0.000	0
2	0.998	0.002	0
3	0.015	0.985	1
4	0.986	0.014	0
5	0.015	0.985	1

LOG LOSS:

We use Scikit Learn's log_loss function to find the log loss produced by the model.

```
from sklearn.metrics import log_loss

y_train_pred_proba = model.predict_proba(X_train)
logloss = log_loss(y_train, y_train_pred_proba, labels=model.classes_)
```

TRAIN - TEST SPLIT:

We use the StratifiedShuffleSplit Class to split our datasets into train and test sets such that the same distribution of the label classes is maintained for each split. It also shuffles the data before splitting.

```
from sklearn.model_selection import StratifiedShuffleSplit

SSS_splitter = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=0)

train_idxs, test_idxs = list(SSS_splitter.split(df_X_train, df_y_train))[0]

X_train, y_train = df_X.values[train_idxs], df_y.values[train_idxs]
X_test, y_test = df_X.values[test_idxs], df_y.values[test_idxs]
```

K - FOLD SPLITTING FOR CROSS VALIDATION:

We use the StratifiedKFold Class to split our data into the desired "folds" of sub train & validation sets, for the sake of hyperparameter tuning and cross validation.

```
n_folds = 3
SKF_splitter = StratifiedKFold(n_splits=n_folds)

for sub_train_idx, val_idx in SKF_splitter.split(X_train, y_train):

    X_sub_train, X_val = X[sub_train_idx], X[val_idx]
    y_sub_train, y_val = y[sub_train_idx], y[val_idx]

#--- COLLECT ALL {SUB_TRAIN_SET, VAL_SET} PAIRS FOR CROSS VALIDATION PURPOSES
```

THE CONFUSION MATRIX (ACCURACY, PRECISION & RECALL):

The Confusion Matrix is a matrix that helps us visualize a model's prediction performance from multiple perspectives. It is a square matrix, whose columns represent predicted values and the rows represent actual values as shown below.

We use the confusion_matrix function to determine the confusion matrix:

```
from sklearn.metrics import confusion_matrix

con_matrix = confusion_matrix(y, y_pred)
con_matrix
```

```
array([[40,  2],
       [ 0, 72]], dtype=int64)
```

```
df_confusion_matrix = pd.DataFrame(con_matrix)
df_confusion_matrix.columns = ['pred_0', 'pred_1']
df_confusion_matrix.index   = ['true_0', 'true_1']

df_confusion_matrix
```

	pred_0	pred_1
true_0	40	2
true_1	0	72

The diagonal of the confusion matrix represents correct predictions. The following core performance metrics can be derived from the confusion matrix:

1. Accuracy
2. Precision of prediction with respect to each class &
3. Recall of prediction with respect to each class
4. F1 Score.

Shown below are the definitions of the above performance metrics.

```
actual_total = con_matrix.sum(axis = 1) #---2D VECTOR
pred_total = con_matrix.sum(axis = 0) #-----2D VECTOR
correct_preds = con_matrix.diagonal() #-----2D VECTOR

acc = correct_preds.sum()/actual_total.sum()

prec_0 = correct_pred[0]/pred_total[0]
rec_0 = correct_preds[0]/actual_total[0]

prec_1 = correct_pred[1]/pred_total[1]
rec_1 = correct_preds[1]/actual_total[1]
```

Accuracy basically expresses the percentage of total correct predictions. The Precision of prediction with respect to a class is the percentage of correct predictions among the total predictions made for that class.The recall of prediction with respect to a class is the percentage of that class recovered within the total predictions made for that class.

So with reference to the example confusion matrix shown earlier we have:

```
1 acc
```

```
0.9824561403508771
```

```
1 prec_0, rec_0
```

```
(1.0, 0.9523809523809523)
```

```
1 prec_1, rec_1
```

```
(0.972972972972973, 1.0)
```

The model under consideration could now be evaluated as follows:

1. The model predicts correctly 98.24% of the time.
2. With respect to class_0, the model is 100% precise, and it recovers 95.23% of all the data points belonging to that class.
3. With respect to class_1, the model is 97.297% precise and it recovers 100% of all the data points belonging to that class.

It should be noted that in case of imbalanced classes, the accuracy metric can be misleading. Say that our classification task involves predicting on a dataset with an imbalanced 90:10 class ratio. A model that always predicts the larger class will achieve 90% accuracy. Thus it is necessary to observe the Precision & Recall performances of the model to get a clearer picture of how a particular accuracy is being achieved

In many cases accuracy is of secondary importance. Consider the task of classifying whether a person is infected with a highly contagious disease or not - Here the primary aim of the classification task would be to achieve close to 100% recall on the 'infected' class (which is generally very small in size compared to the 'uninfected' class) and secondly to achieve as much precision as possible with respect to the bigger 'uninfected' class.

F1 SCORE:

The F1 score is the harmonic mean of the precision and recall. F1 scores require that both the precision & recall performance of the model with respect to a particular class be balanced and high to indicate a good performance. In other words, the higher the F1 score the more balanced and greater the values of precision & recall.

$$F_1 = \left(\frac{2}{\mathrm{recall}^{-1} + \mathrm{precision}^{-1}} \right) = 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}}$$

As with all other scores discussed earlier, the f1 score too has a range from 0 to 1. F1 score is useful if one expects balanced precision-recall performance from the predictive model, since this metric rewards such performances. But in cases of extremely imbalanced datasets, where we have to choose between recall & precision, this metric fails to provide deeper insight.

In most cases studying **the precision-recall curve** of a model is the best way to go. Irrespective of the specific classification requirements of the task at hand, this method provides one with all the information required to make informed judgements about a model's performance characteristics.

MODEL PREDICTION THRESHOLDING & THE PRECISION-RECALL CURVE:

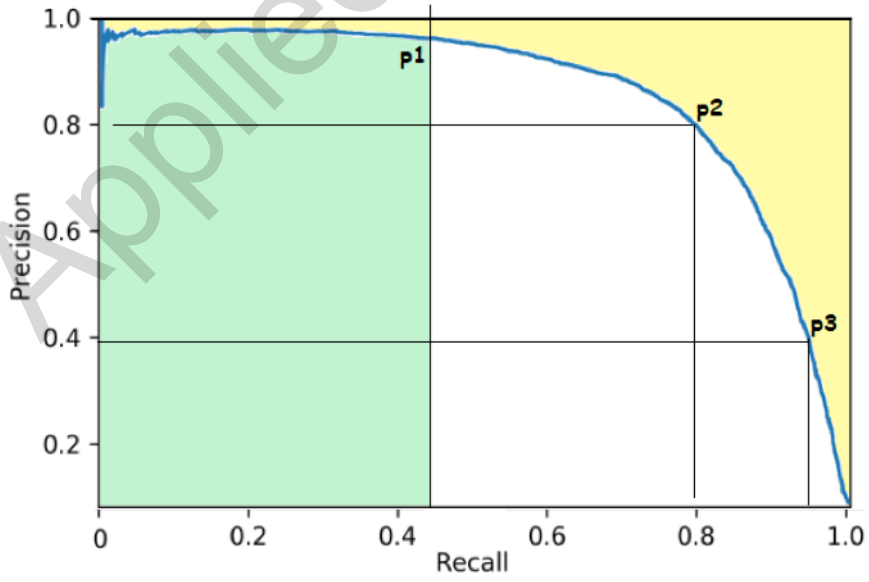
Consider the code shown below. The prediction mechanism of the model is "set" such that it predicts that a data point belongs to class_1 only if the probability predicted by the model for that class is greater than 60%.

```
probas_both_classes = model.predict_proba(X)
proba_class_1 = probas_both_cls[:, 1]

threshold_class_1 = 0.6

y_pred = np.array([1 if i > threshold_class_1 else 0 for i in proba_class_1])
```

Altering the prediction threshold of the model alters the precision & recall performances of the model. Generally, the higher the threshold set for a class, the more precise the model's predictions for that class will become, while at the same time resulting in a reduction in its recall performance and simultaneously vice-versa for the other class.

Thus the same model will have different precision and recall performances at different probability thresholds. A visual representation of this phenomenon is called the Precision-Recall Curve.

The plot shown below is an example of a Precision-Recall Curve depicting a model's performance over a range of thresholds for a particular label/class:



Using this plot one could make an informed decision as to the ideal threshold level one should use, given the requirements of the classification task.

If we desire to optimize the model for precision and recall is of secondary importance, we could use a model that falls on the right extreme of the green zone shown above (point **p1**). A model corresponding to this point on the curve would have the maximum recall possible while maintaining the precision levels greater than 95%.

If we desire a balanced precision-recall performance, we could choose a model that corresponds to point **p2** on the curve.

If we desire to optimize the model for recall and precision is of secondary importance, we could use a model that corresponds to point **p3** on the curve.

Precision - Recall curves can also be used to compare multiple models. Generally the greater the **Area Under the Curve** (ie: lesser the yellow region), the more robust the model is.

THE RECEIVER OPERATING CHARACTERISTIC (ROC) CURVE:

The ROC curve is another way to visualize a model's performance on a binary classification task. Here the class of interest is considered as the positive class and the other class is considered as the negative class. It consists of plotting the False Positive Rate (**FPR**) versus the True Positive Rate (**TPR**) of the model over a range of probability thresholds.

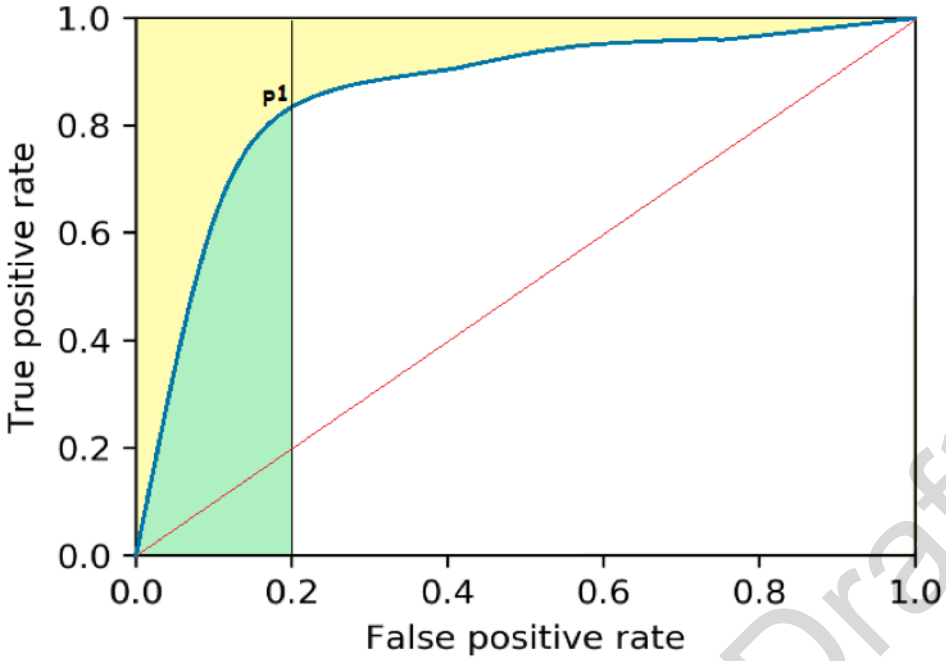
The **TPR** (or sensitivity) is another name for recall:

TPR = **P** (prediction = correct | prediction = positive) =
n_correct_positive_predictions / n_total_positive_points

Whereas **FPR** (or false alarm rate):

FPR = P (prediction = wrong | prediction = negative) =
n_wrong_positive_predictions / n_total_negative_points

The plot shown below is an example of a **ROC** curve depicting a model's performance over a range of thresholds for a particular label/class of interest:



Given that the ROC curve of the model in question has the profile shown above and say that the maximum tolerance for wrong positive predictions is 20%, then the best possible performance will be achieved at the probability threshold corresponding to point **p1**. In general, the greater the **AUC** (or lesser the yellow region), the more robust the model.

The red line in the ROC curve represents the performance that a random predictor would have.

For the examples in this book, we use a decomposed version of the precision recall curve, where we separately plot a precision curve and a recall curve, with respect to the various prediction thresholds chosen between 0 and 1, for both - the minority and the majority classes (ie: 4 plots in total). This will provide all the information required, for us to make an informed decision, about the right model to be chosen for the binary classification task at hand.

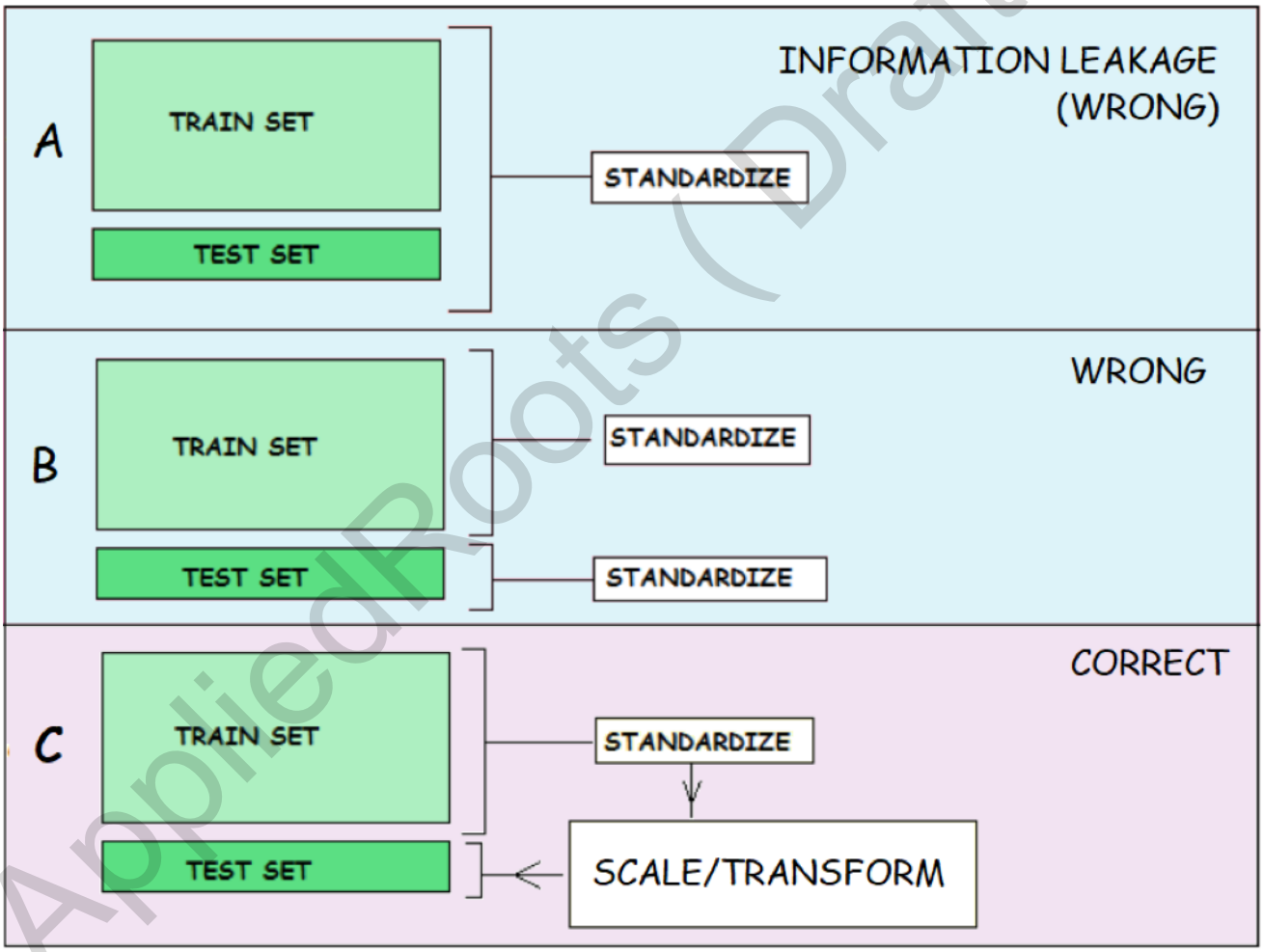
5. INFORMATION LEAKAGE

INFORMATION LEAKAGE

Information leakage or data leakage is the phenomenon where the model gets access to information from the test set during its training phase. This most often happens due to improper sequence in the application of data transformation/scaling/modification tasks on the dataset.

Data leakage often results in unrealistically high levels of performance on the test set, because the model is being run on data that it has already seen , in some capacity in the training set.

The image below describes information leakage using data standardization as an example:



EXAMPLE_A:

Shows the data being standardized before splitting. This causes the train set to be transformed using the information present in the test set (even though they are not yet split).

EXAMPLE_C:

Shows the right way. All transformation/scaling/modification of data are done after splitting the data. The test set is transformed based on information derived from the train set. Thus the model is not exposed to the test set before the testing phase. In other words, the test set is used only for model evaluation purposes, where it represents real world data that the model has not yet been exposed to.

Thus in the case of standardization, the standardizing scaler is "fit" to the train set which results in the scaler gaining information as to how the data need to be transformed. This information is then used to transform the train set and the test set as shown below.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler.fit(X_train)

X_train_standardized = scaler.transform(X_train)
X_test_standardized  = scaler.transform(X_test)
```

The same logic applies to all forms of pre-training data modifications like feature selection and dimensionality reduction. In the case of feature selection for example, the best features are selected by analysing the feature & label correlations in the train set. Once the best features are thus selected, the same features are then chosen from the test set also. The test set is not analysed or referred to for any purposes.

Note that during K fold cross validation, while training and testing for each fold of the train set, standardization is done only after splitting each fold into sub-train & sub-test sets. Each sub-test set is then standardized based on a scaler function "fitted" onto their corresponding sub-train sets.

EXAMPLE_B:

Shows the test set being standardized independent of the train set. There is no information leakage in this case but it is still not advisable. Since the test set is considerably smaller than the train set there could be slight differences in the distributions of the two and hence performance measures obtained by predicting on the independently standardized or transformed test sets are generally not reliable.

INFORMATION ERROR DUE TO DUPLICATE DATA POINTS :

This form of information leakage happens when your dataset contains several identical points. Occurrence of duplicate data is quite common in real world data, in such cases you may experience data leakage simply due to the fact that your train and test set may contain the same data point, even though they may correspond to different observations. This can be fixed

by deduplicating your dataset prior to splitting into train and test sets.

Information leakage due to duplicate data can also occur if we upsample imbalanced datasets prior to splitting. This can cause the same duplicated points to be present in the train and test set after splitting. Information leakage due to upsampling can be avoided by first splitting the data and then upsampling only the train set.

IMPLICIT LEAKAGE IN TEMPORAL DATA:

Data leakage can also occur if dependencies exist between your test and train set. This is commonly noticed in specific cases of data where time itself is an implicit feature and hence the sequence of data contains information that could factor into future predictions. Consider the following scenario - we have in our data a group of temporally ordered data points **x1, x2, x3** and **x4.** Now suppose during splitting, the data points **x1, x3** and **x4** end up in the train set and **x2** ends up in the test set. Here, we have most likely created data leakage because by training on points **x3** and **x4** and testing on point **x2**, we created an unrealistic situation in which we train our model on future knowledge, relative to the test set's point in time. In a real-world scenario, our model clearly would not have any knowledge of the future.

Fixing this problem involves ensuring that the test-train split is also split across time. So, everything in your training set should occur before everything in your test set. Thus the model is evaluated as if it were acting on incoming real-world data.

6. THE MACHINE LEARNING PIPELINE

THE MACHINE LEARNING PIPELINE

A pipeline in machine learning, basically denotes a carefully designed sequence of steps which help us arrive at a reliable predictor, for the machine learning task we wish to automate.

BINARY CLASSIFICATION HELPER FUNCTIONS:

For the sake of simplicity, helper functions have been created that automate most of the data preprocessing, EDA and machine learning/model evaluation tasks. The reader is advised to first pay attention to the process pipeline and explore the code only after they have grasped the logic behind steps involved in the pipelines described.

A) DATA PREPROCESSING & EDA FUNCTIONS:

1. **df_Xy, dictO_code2feats = fn_preprocess(df_Xy_raw, labels_col_name):**

   TASKS PERFORMED:

   On data **df_Xy_raw** being fed to it, this function performs the following:

   a. Drops all rows with missing labels.

   b. Replaces missing values in any feature column with the mean of that column

   c. Replaces feature (column) names with generic codes: f1, f2, f2.....fn, labels.

   ARGS

   a. **df_Xy_raw** - The dataframe that contains the all data points and their respective labels in tabular format.

   b. **Labels_col_name** - The name of the column within that dataframe which represent the labels (ie: labels_col_name)

   RETURNS:

   a. **df_Xy**: Rectified Dataframe containing features & labels

   b. **dictO_code2feats**: Dictionary which gives actual feature (columns) names that correspond to the generic

   code names assigned to each feature (ie: dictO_codes2features)

2. **fn_label_distr(labels_) :**

   This function outputs a barplot describing the distribution of the labels.

3. IDXS_TR, IDXS_TS = FN_TR_TS_SPLIT(DF_XY, TS_SIZE = 0.2):

This function returns the indexes of the data split into train and test splits using scikit learn's stratified shuffle split .

4. LISTO_GOOD_FEATS = FN_FEAT_SELECT_CLS(DF_TR_RAW, PLOT = TRUE, THRESH_FEAT_LABEL = 0, THRESH_FEAT_FEAT = 1):

ARGS:

a. df_tr_raw - the train set with full features in dataframe format
(ie: Prior to feature selection)

KW_ARGS:

a. **plot** : Boolean - Default set to "True", outputs the following 2 plots:

1. Horizontal Bar Plot showing the degree of correlation (ANOVA - f ratio) of all features with respect to labels.
1. Covariance matrix showing degree of correlation (Spearman) of all features with respect to each other.

b. **thresh_feat_label**: Min threshold for feature-to-label correlation, below which the feature is ignored.

c. **thresh_feat_feat**: Max threshold for feature-to-feature correlation, above which the feature is ignored.

RETURNS

a. **listO_good_feats** : Selected features arranged in increasing order of importance.

5. FN_DISTR_LABEL_FEATS(DF_TR_, N_TOP_FEATS = 4) :

This function outputs a series of subplots that shows the distribution of each feature with respect to the Classes (each class id uniquely colored), thus providing some insights into the degree of separability.

ARGS:

a. **df_tr_ : df_tr_raw.iloc[: , listO_good_feats]** (ie:Train Set with its features/columns arranged in decreasing order of importance)

KW_ARGS:

a. **N_top_feats**: Number of features for which we want to compare class distributions. Default set to 4.

6. LISTO_OUTLIER_IDXS_C0 = FN_LOF_OUTLIER_IDXS(DF_XY_, N_CLUSTERS, N_NEIGHBORS):

This function uses the Local Outlier Factor measure to detect outliers in the data fed to it.

TASKS PERFORMED: this function performs the following on df_Xy_:

a. Performs Kmeans clustering on **df_X**

b. Detects outliers on each of the clusters identified in step **a**.

ARGS:

a. **n_clusters:** Number of clusters to be detected by the Kmeans algorithm

b. **n_neighbors:** Number of neighbouring points to consider while detecting outliers within each cluster.

RETURNS: Indexes of outliers detected by the Local Outlier Factor algorithm.

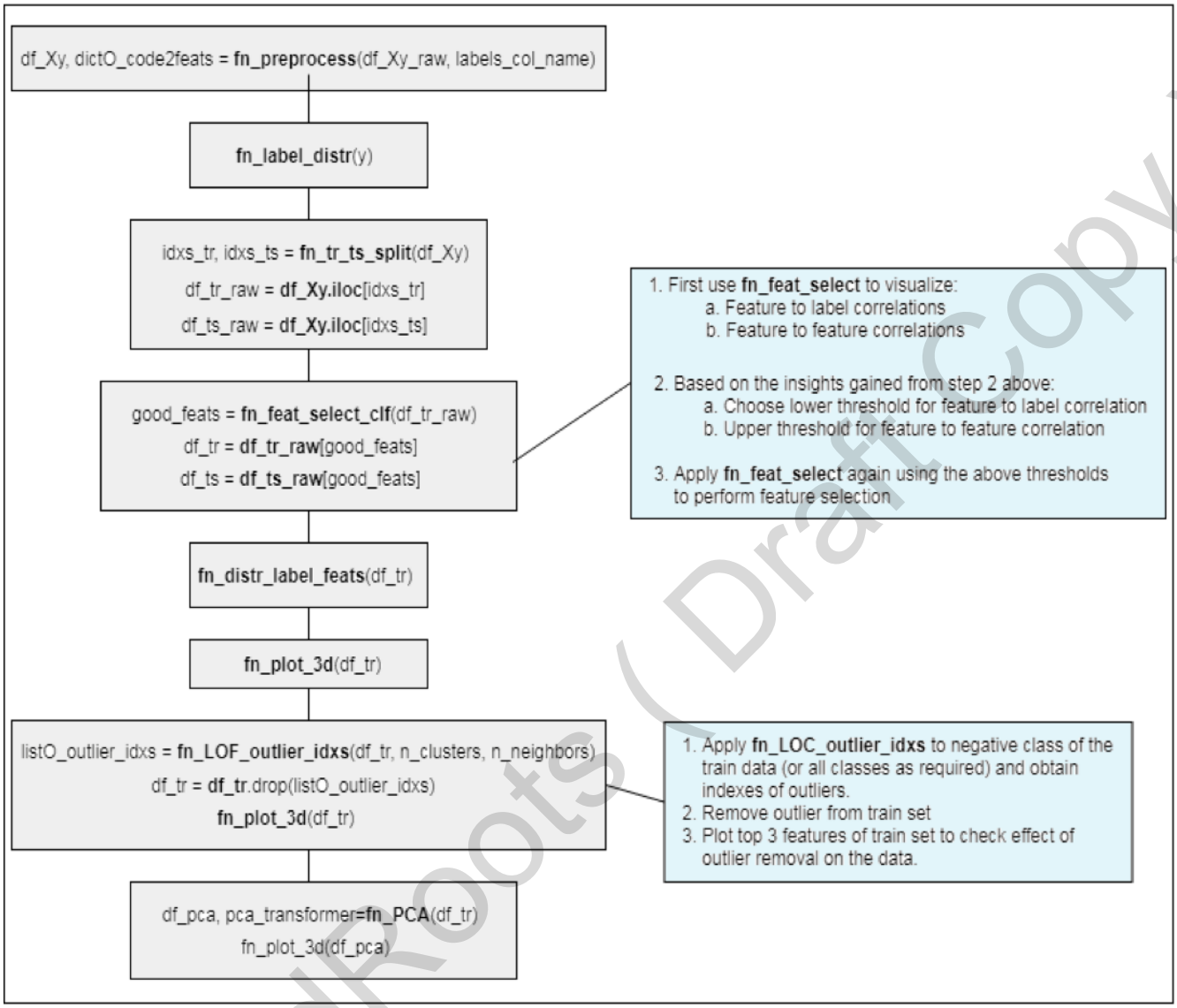
7. FN_PLOT_3D_BINARY(DF_TR_):

This function outputs a 3D scatter plot using the first 3 features of the dataset fed to it. Each class is uniquely colored, thus providing insight into the degree of separability.

8. DF_PCA, PCA_TRANSFORMER = FN_PCA(DF_TR_) :

This function is used to perform Principal Component transformation on the dataset fed to it. It returns the transformed feature set along with the pca transformer object, for transformation other data of the same kind.

The flow chart shown below visually describes the sequence of steps in which the previously introduced functions are used during the Data_Preprocessing_EDA stage:



B) MODEL TRAINING & EVALUATION FUNCTIONS:

9. df_tr_standardized, df_ts_standardized = fn_standardize_df(df_tr_, df_ts_):

This function returns standardized train & test sest corresponding to the original train & test sets fed to it as arguments. The test set is standardized using the standardizing transformer derived from the train set (ie: It is not standardized in isolation to the train set)

10. df_kfoldcv, dictO_model_instances, listO_invalid_models = fn_kfoldcv_clf_binary(df_tr_, model_class, parameter_grid, n_folds = 3, model_name_prefix = ''):

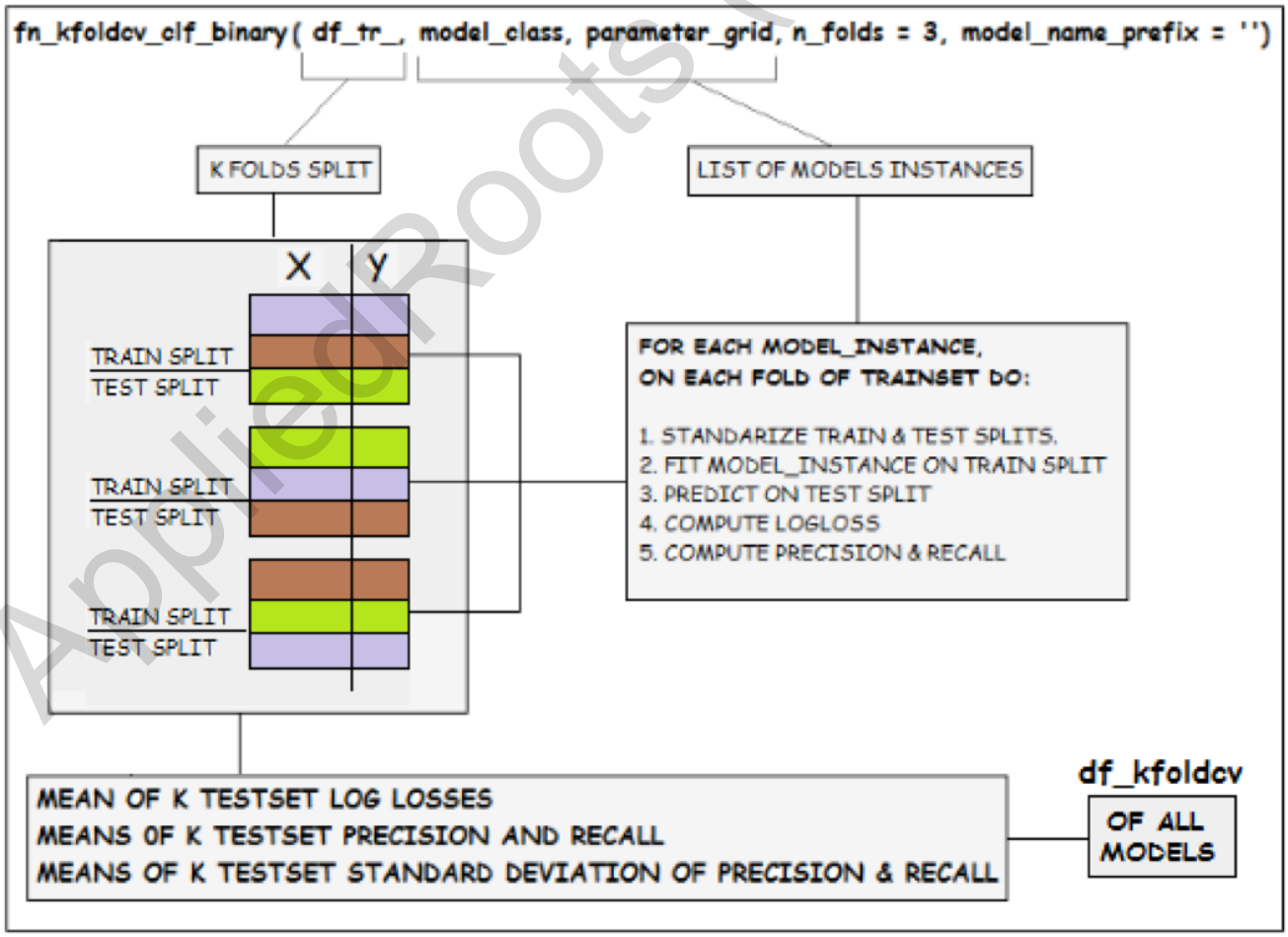
This function performs K fold cross validation. The Input argument "**parameter_grid**" is a dictionary defined in the format shown below. It specifies the range for each of the model's hyperparameters that we intend to tune for.

```
parameter_grid = dict(penalty = ['l2'],
                              C = [1e-2, 1, 1e2],
                         solver = ['lbfgs'],
                   class_weight = ['balanced'],
                       max_iter = [400])
```

Given the example parameter_grid shown above, the function instantiates models having the following hyperparameters:

```
{'model_0': {'penalty': 'l2',
             'C': 0.01,
             'solver': 'lbfgs',
             'class_weight': 'balanced',
             'max_iter': 400},
 'model_1': {'penalty': 'l2',
             'C': 1,
             'solver': 'lbfgs',
             'class_weight': 'balanced',
             'max_iter': 400},
 'model_2': {'penalty': 'l2',
             'C': 100.0,
             'solver': 'lbfgs',
             'class_weight': 'balanced',
             'max_iter': 400}}
```

The function then performs K fold cross validation using each of these model instances. The illustration shown below describes the tasks the function performs using models instantiated by the "**parameter_grid**" input parameter.



The '**model_name_prefix**' keyword argument lets the user give a prefix to the names that the function assigns to each of the models. For example, for the **parameter_grid** example just previously described, the model names the three resulting models as: model_1, model_2 & model_3. If the keyword argument **model_name_prefix = "logistic_"**, the models will then be named: logistic_model_1, logistic_model_2 & logistic_model_3.

Apart from the dataframe mentioned above (ie: df_kfoldcv), the function also returns:

- A dictionary - **dictO_model_instances**, that maps the model names to the respective hyperparameters they were trained on.
- A list - **listO_invalid_models**, which is a list of parameters for which models could not be created due to constraints of the algorithms used in scikit learn (see Scikit Learn's manual for Logistic Regression).

11. fn_performance_models_data(listO_models, df_tr_standard, listO_thresholds, legend)

This function plots the precision-recall curves for all the models listed in the "**listO_models**" parameter, when they are trained on **df_Xy_**. The precision_recall curves are plotted by computing the precision & recall for the models at the prediction thresholds specified by the "**listO_thresholds**" parameter.

The "**legend**" parameter is simply the names of the models listed in the "**listO_models**" parameter. These names are the same as the index of the dataframe "**df_kfoldcv**" returned by **fn_kfoldcv_clf_binary**.

Four plots are outputted by the function:

1. The recall curve of class_1 at the various prediction thresholds specified.
2. The precision curve of class_1 at the various prediction thresholds specified.
3. The recall curve of class_0 at the various prediction thresholds specified.
4. The precision curve of class_0 at the various prediction thresholds specified.

The analysis of these 4 plots gives all the information required to help us choose the best possible model suited for the classification task at hand.

CLASSIFICATION OF WISCONSIN BREAST CANCER DATASET:

DATA DESCRIPTION:

The dataset contains 559 entries consisting of 30 features, which were computed from digitized images of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image. Labels indicating whether the cancer is benign (class_0) or malignant (class_1) are provided for each entry.

```
from sklearn.datasets import load_breast_cancer

data = load_breast_cancer()

df = pd.DataFrame(data['data'])
df.columns = data['feature_names']
df_Xy_raw = df.assign(target = data['target'])


df_Xy_raw.info()
```

```
0   mean radius              569 non-null    float64
1   mean texture             569 non-null    float64
2   mean perimeter           569 non-null    float64
3   mean area                569 non-null    float64
4   mean smoothness          569 non-null    float64
5   mean compactness         569 non-null    float64
6   mean concavity           569 non-null    float64
7   mean concave points      569 non-null    float64
8   mean symmetry            569 non-null    float64
9   mean fractal dimension   569 non-null    float64
10  radius error             569 non-null    float64
11  texture error            569 non-null    float64
12  perimeter error          569 non-null    float64
13  area error               569 non-null    float64
14  smoothness error         569 non-null    float64
15  compactness error        569 non-null    float64
16  concavity error          569 non-null    float64
17  concave points error     569 non-null    float64
18  symmetry error           569 non-null    float64
19  fractal dimension error  569 non-null    float64
20  worst radius             569 non-null    float64
21  worst texture            569 non-null    float64
22  worst perimeter          569 non-null    float64
23  worst area               569 non-null    float64
24  worst smoothness         569 non-null    float64
25  worst compactness        569 non-null    float64
26  worst concavity          569 non-null    float64
27  worst concave points     569 non-null    float64
28  worst symmetry           569 non-null    float64
29  worst fractal dimension  569 non-null    float64
30  target                   569 non-null    int64
```

A) DATA PREPROCESSING & EDA STAGE:

STEP_1: Applying **fn_preprocess_data** we have:

```
labels_col_name = 'target'

df_Xy, dictO_code2feats = fn_preprocess(df_Xy_raw, labels_col_name)
print(df_Xy.head(10))
```

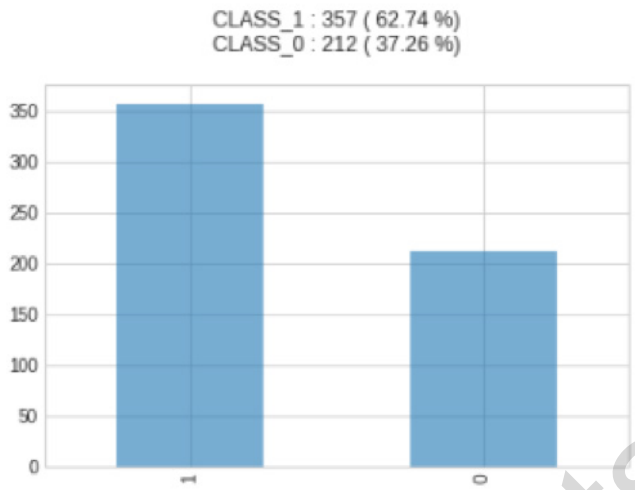
```
>>> 1/3: CHECKING FOR ROWS WITH MISSING LABELS
*** NO LABELS MISSING
>>> 2/3: CHECKING FOR MISSING FEATURE VALUES
*** NO MISSING VALUES
>>> 3/3: RENAMING COLUMNS WITH GENERIC NAMES
```

	f0	f1	f2	f3	...	f27	f28	f29	labels
0	17.99	10.38	122.80	1001.0	...	0.2654	0.4601	0.11890	0
1	20.57	17.77	132.90	1326.0	...	0.1860	0.2750	0.08902	0
2	19.69	21.25	130.00	1203.0	...	0.2430	0.3613	0.08758	0
3	11.42	20.38	77.58	386.1	...	0.2575	0.6638	0.17300	0
4	20.29	14.34	135.10	1297.0	...	0.1625	0.2364	0.07678	0
5	12.45	15.70	82.57	477.1	...	0.1741	0.3985	0.12440	0
6	18.25	19.98	119.60	1040.0	...	0.1932	0.3063	0.08368	0
7	13.71	20.83	90.20	577.9	...	0.1556	0.3196	0.11510	0
8	13.00	21.82	87.50	519.8	...	0.2060	0.4378	0.10720	0
9	12.46	24.04	83.97	475.9	...	0.2210	0.4366	0.20750	0

As can be seen from the feedback that the function provides, the dataset is free of missing values, so the function just renames the columns with generic feature names and returns dataframe df_Xy.

STEP_2: Applying **fn_show_label_distribution** we have:

```
y = df_Xy.labels.values

fn_label_distr(y)
```



We see that the data is moderately imbalanced and hence make a note to use this information during hyperparameter tuning in the model train & evaluation stage.

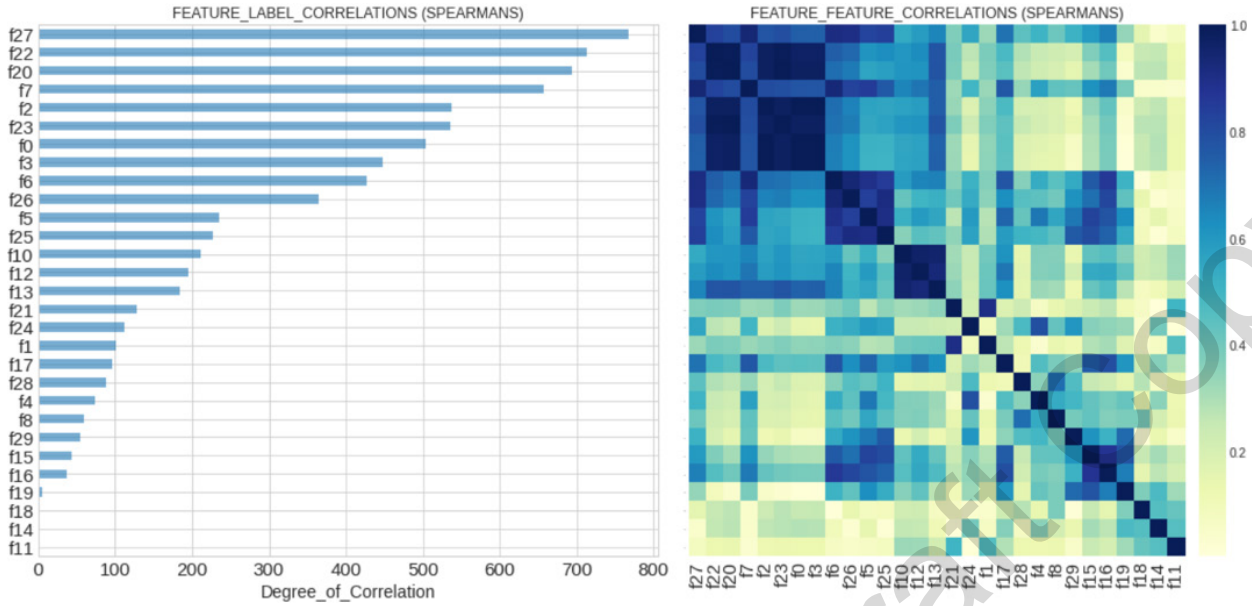
STEP_3: Applying **fn_tr_ts_split** we have:

```
idxs_tr, idxs_ts = fn_tr_ts_split(df_Xy, ts_size = 0.2)

df_tr_raw = df_Xy.iloc[idxs_tr]
df_ts_raw = df_Xy.iloc[idxs_ts]

df_tr_raw.shape, df_ts_raw.shape
```

```
((455, 31), (114, 31))
```

STEP_4: Applying **fn_feat_select_clf** we have:

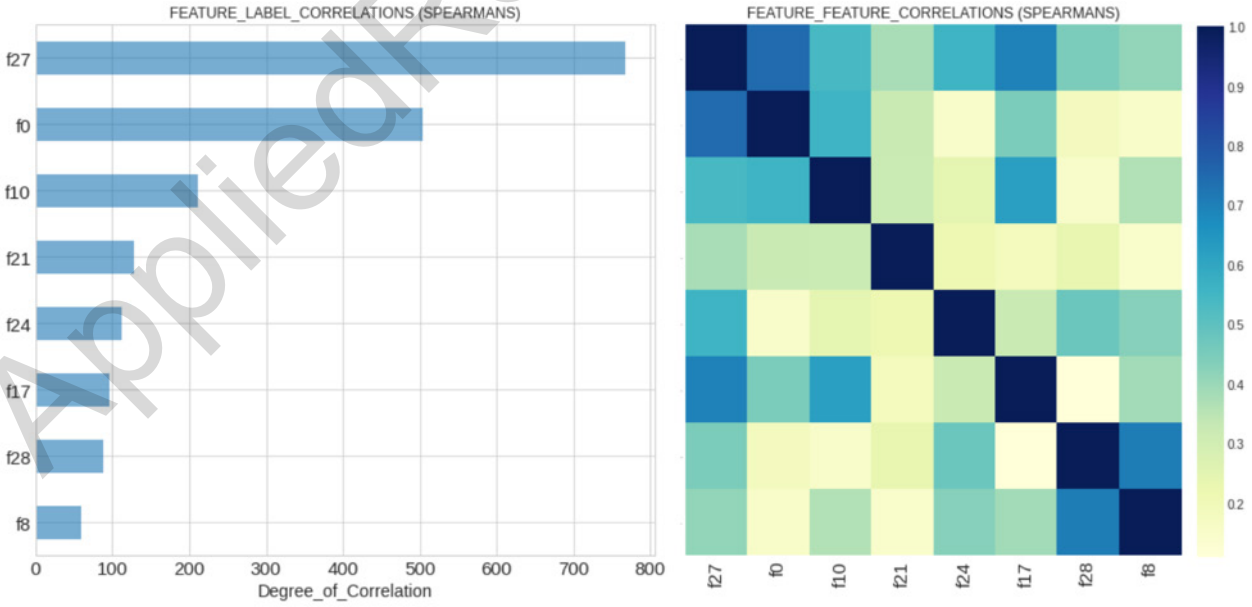
```
listO_good_feats = fn_feat_select_clf(df_tr_raw, plot = True)
```



Using the feedback we get from the above plots we choose a feature-to-label threshold of 10 and a feature-to-feature correlation threshold of 0.75 as shown below:

```
list0_good_feats = fn_feat_select_clf(df_tr_raw, thresh_feat_label = 10,
                                                 thresh_feat_feat = 0.75,
                                                        plot = True)

```

This results in the following output, the original feature set is reduced to a smaller set of features:



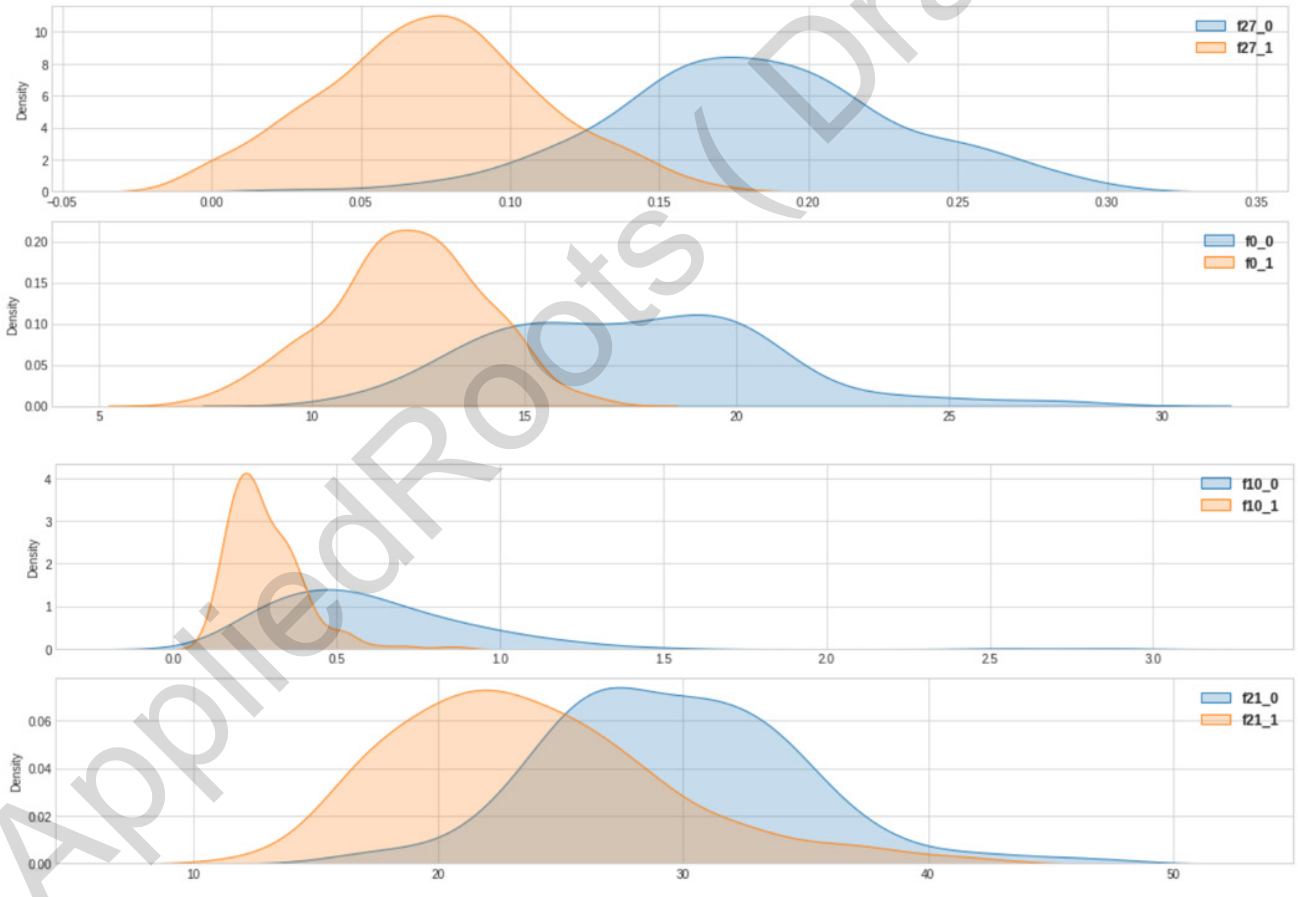
We then use only the features selected above, thus getting the final form of our train & test sets:

```
df_tr = df_tr_raw[list0_good_feats + ['labels']]
df_tr.index = range(len(df_tr))

df_ts = df_ts_raw[list0_good_feats + ['labels']]
df_ts.index = range(len(df_ts))

df_tr.shape, df_ts.shape
```

```
((455, 9), (114, 9))
```

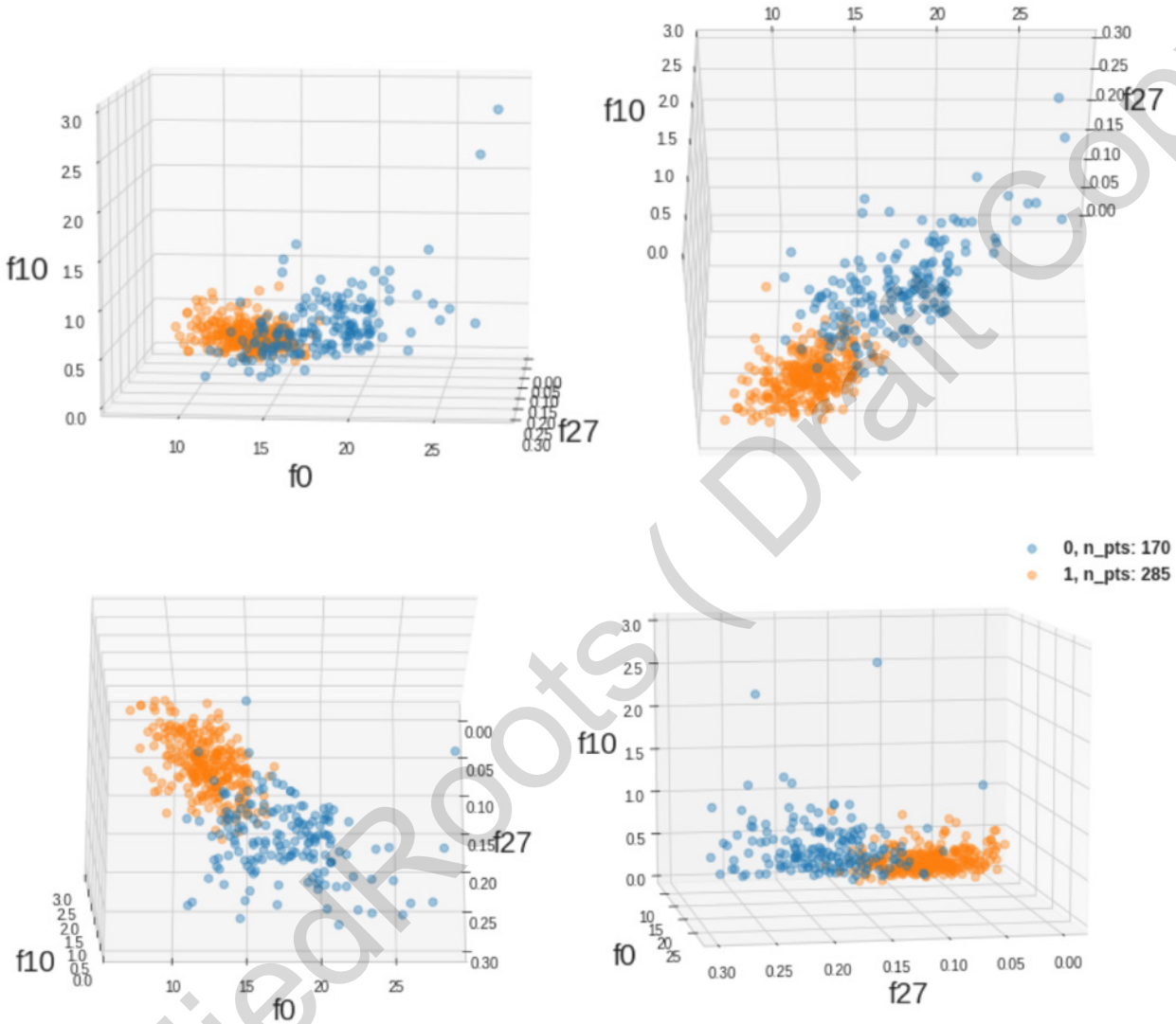
STEP_5: Applying **fn_distr_label_feats** we have:

```
fn_label_distr_feats(df_tr, n_top_feats = 4)
```



STEP_6: Applying **fn_plot_3d_binary** on train data we have:

```
1   fn_plot_3d_binary(df_tr)
```



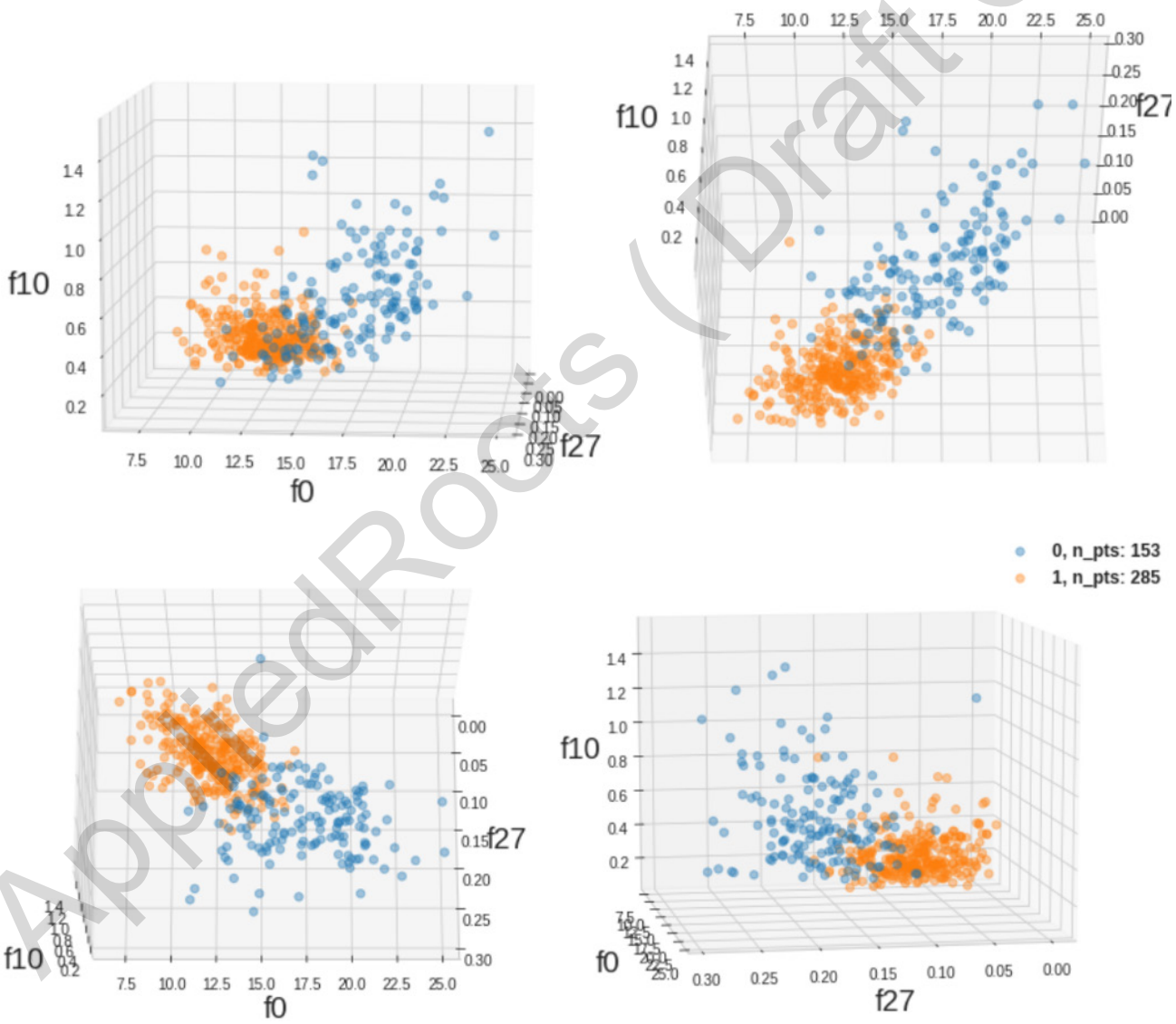
By analysing the distribution plot and the 3D plot outputted by the previous two functions respectively, it is possible to gain insights into the degree of separability between the two classes.

It can be seen from the above plots that there is a good degree of separation between the two classes

STEP_7: Applying **fn_LOF_outlier_idxs** on the train set and then applying **fn_plot_3d_binary** on the outlier free data we have:

```
df_Xy_ = df_tr[df_tr.labels == 0]
n_clusters = 3
n_neighbors = 10

list0_outlier_idxs_c0 = fn_LOF(df_Xy_, n_clusters, n_neighbors)

print(len(list0_outlier_idxs_c0))

fn_plot_3d_clf(df_tr.drop(list0_outlier_idxs_c0))
```

```
100% (3 of 3) |############################| Elapsed Time: 0:00:00 Time:  0:00:00
17
```

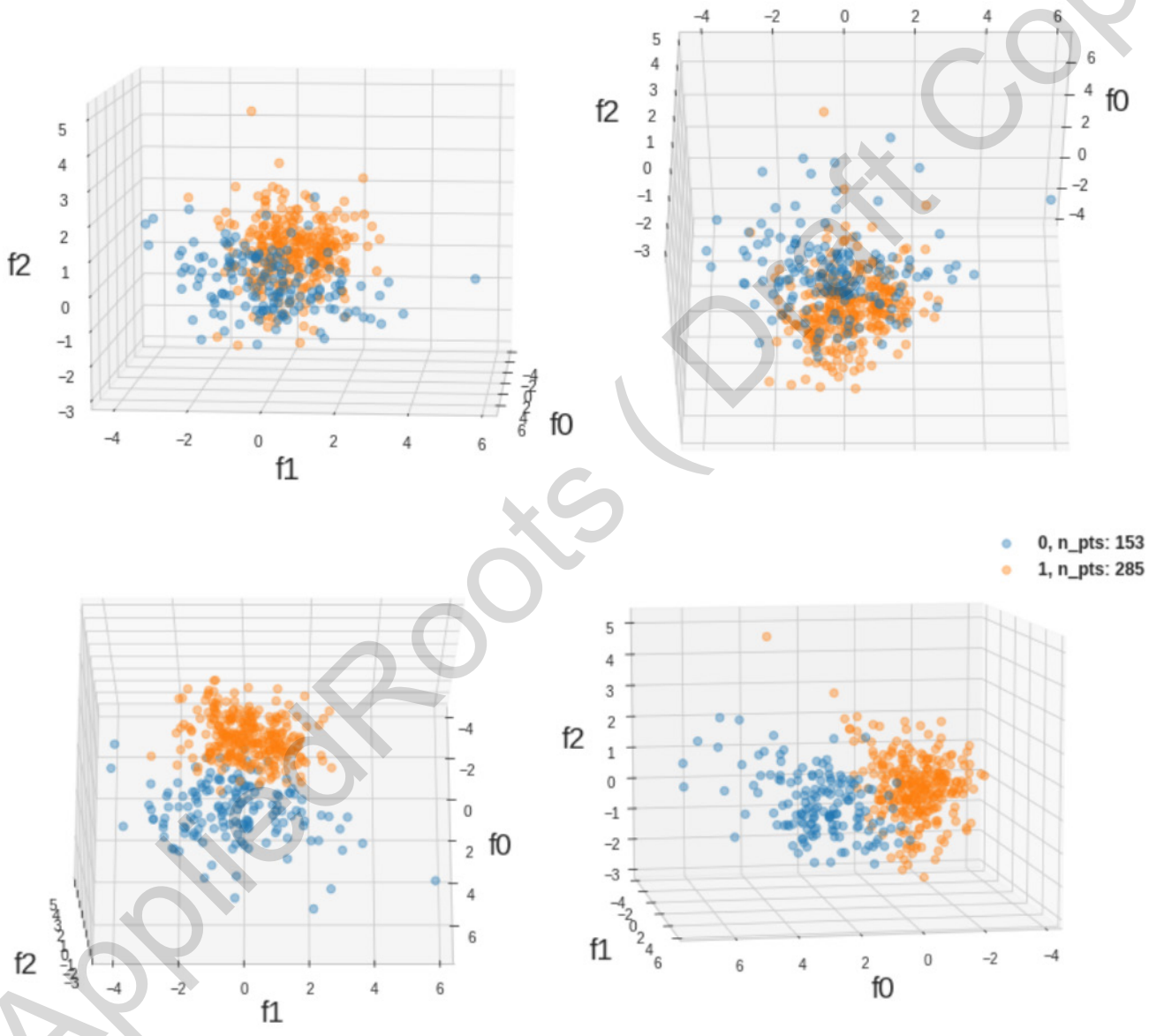


As can be seen the data contains lesser outliers for each of the classes. This helps the model find a better separation boundary. We drop the outlier form the train set as shown below.

```
df_tr = df_tr.drop(list0_outlier_idxs_c0)
df_tr.index = range(len(df_tr))
```

STEP_8: Applying **fn_PCA** on the train set and then applying **fn_plot_3d_binary** on the transformed data data we have:

```
df_pca, pca_transformer = fn_PCA(df_tr)

fn_plot_3d_clf(df_pca)
```



The transformed data shown above shows a good degree of separation between the classes, and so it too could be used to train a classifier model.

B) MODEL TRAINING & EVALUATION:

The model training and evaluation stage basically involves three main steps:

1. K Fold Cross Validation with Hyperparameter tuning.
2. Plotting and analysing precision-recall curves, for a selected set of models, which were obtained by filtering the output of the first step, for models that have best performances. This is done by training these selected models on the full train set (instead of subsets or "folds" of the train set as done in the previous step) and plotting their precision-recall values at different thresholds from zero to one. We then choose the model with the best curve as our final predictor.
3. Train the chosen model on the original test set and then predict on both, the train set and the test set, to check for generalization on unseen data.

STEP_9: Applying function **fn_kfoldcv_clf_binary** we have:

```
from sklearn.linear_model import LogisticRegression

df_tr_ = df_tr
model_class = LogisticRegression

parameter_grid = dict(penalty = ['l2', 'l1', 'elasticnet'],
                              C = [10**i for i in range(-15, 15)],
                         solver = ['lbfgs', 'newton-cg', 'saga'],
                   class_weight = ['balanced'],
                       max_iter = [10_000])

z = fn_kfoldcv_clf_binary(df_tr_, model_class, parameter_grid,  n_folds = 3)

df_kfoldcv, dict0_model_instances, list0_invalid_models = z
df_kfoldcv.describe()
```

```
100% (270 of 270) |######################| Elapsed Time: 0:00:45 Time:  0:00:45

SOME MODELS INVALID - CHECK list0_invalid_models
```

	mean_loss	mean_rec_0	mean_rec_1	mean_prec_0	mean_prec_1	std_loss	std_rec_0	std_rec_1	std_prec_1	std_prec_1
count	120.000000	120.000000	120.000000	120.000000	120.000000	1.200000e+02	120.000000	120.000000	120.000000	120.000000
mean	0.369666	88.614150	86.290550	92.706313	91.344662	1.344003e-02	9.689508	11.614875	5.742604	5.994221
std	0.282145	17.197054	15.776503	8.105613	7.881724	1.165230e-02	17.992834	16.677460	8.840292	8.742357
min	0.090784	50.000000	50.000000	67.466000	67.466000	9.661917e-15	0.382000	1.796000	0.041500	0.021000
25%	0.110477	95.335500	90.578000	93.297500	90.420000	9.537680e-05	0.908500	3.735875	0.908500	1.461500
50%	0.134279	96.986500	93.591000	96.986500	95.913000	1.197629e-02	0.908500	4.375500	0.908500	1.913000
75%	0.693144	96.986500	94.571500	96.986500	96.410500	2.160001e-02	2.703500	5.737500	5.578500	6.209000
max	0.707454	97.512500	96.532500	97.958500	97.433000	3.044603e-02	50.000000	50.000000	32.534000	32.534000

NOTE:

1. Notice that in the above function, for the **parameter_grid** hyperparameter, we choose:**class_weight = 'balanced'**. This is because while checking the label distribution using **fn_label_distr,** we observed that the distribution is skewed/unbalanced. In general it is a good practice to use this keyword argument.
1. The feedback (print out) provided by the function indicates that our **parameter_grid** yields some invalid parameter combinations, this is to be expected as some parameter combinations can be incompatible.

We then filter out the best performing models as shown below:

```
dff = df_kfoldcv_gridsearch
n = 5

df1 = dff.sort_values(by = 'ts_mean_rec_1', ascending = False)[:n]
df2 = dff.sort_values(by = 'ts_mean_prec_1', ascending = False)[:n]
df3 = dff.sort_values(by = 'ts_std_rec_1', ascending = True)[:n]
df4 = dff.sort_values(by = 'ts_mean_loss', ascending = True)[:n]
df5 = dff.sort_values(by = 'ts_std_loss', ascending = False)[:n]

df_filtered_cv = pd.concat([df1, df2, df3, df4, df5]).drop_duplicates()
df_filtered_cv = df_filtered_cv[df_filtered_cv.ts_mean_loss < 0.3]
df_filtered_cv = df_filtered_cv.sort_values(by = 'ts_mean_rec_1', ascending = False)[:3]
df_filtered_cv
```

	ts_mean_rec_1	ts_mean_prec_1	ts_mean_rec_0	ts_mean_prec_0	ts_mean_loss	ts_std_loss	ts_mean_acc	ts_std_rec_1	ts_std_rec_0
model_42	0.982456	0.965564	0.934641	0.966122	0.151102	0.008104	0.965753	0.004962	0.018486
model_44	0.982456	0.965564	0.934641	0.966122	0.151104	0.008105	0.965753	0.004962	0.018486
model_179	0.968421	0.976147	0.954248	0.944748	0.110468	0.021588	0.963470	0.022739	0.040290

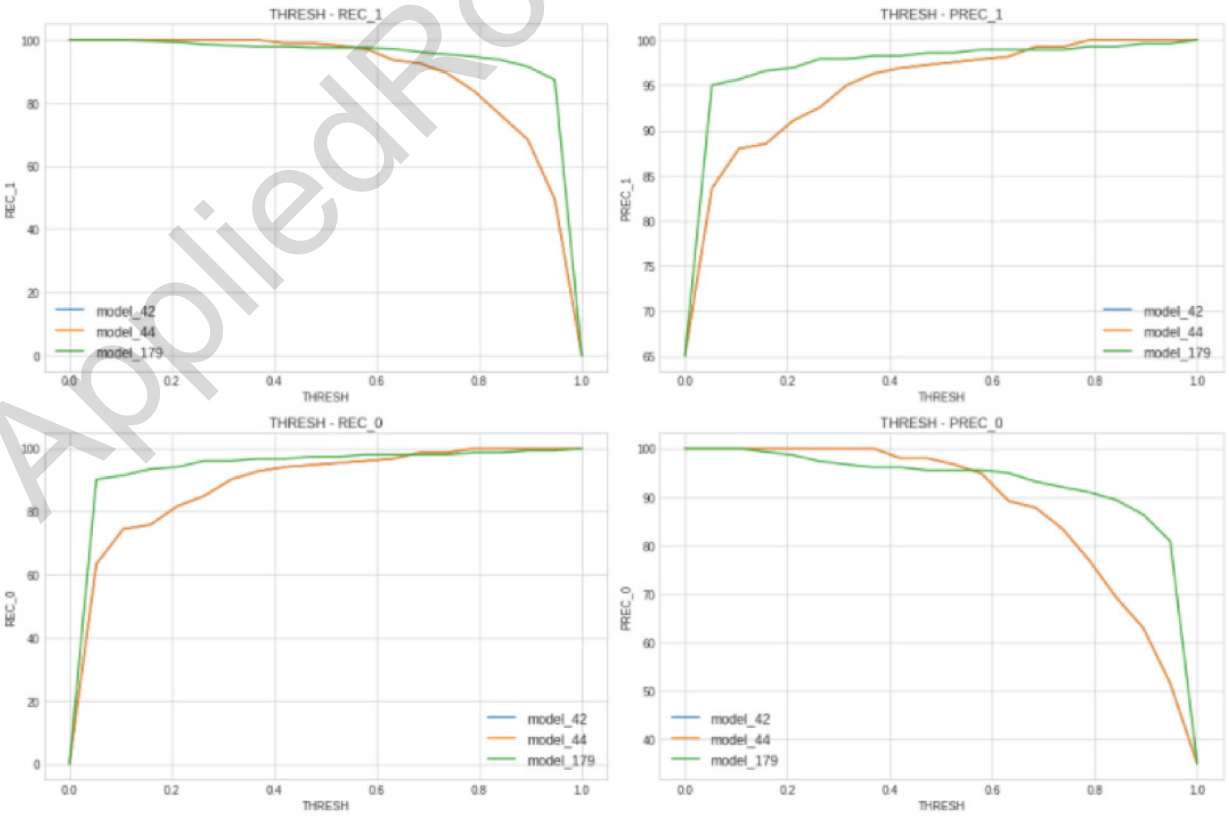
We can check the parameters of the above filtered models as shown below:

```
display(dict0_model_instances['model_42'],
        dict0_model_instances['model_44'],
        dict0_model_instances['model_179'])
```

```
LogisticRegression(C=0.1, class_weight='balanced', dual=False,
                   fit_intercept=True, intercept_scaling=1, l1_ratio=None,
                   max_iter=10000, multi_class='auto', n_jobs=None,
                   penalty='l2', random_state=None, solver='lbfgs', tol=0.0001,
                   verbose=0, warm_start=False)
LogisticRegression(C=0.1, class_weight='balanced', dual=False,
                   fit_intercept=True, intercept_scaling=1, l1_ratio=None,
                   max_iter=10000, multi_class='auto', n_jobs=None,
                   penalty='l2', random_state=None, solver='saga', tol=0.0001,
                   verbose=0, warm_start=False)
LogisticRegression(C=100000000000000, class_weight='balanced', dual=False,
                   fit_intercept=True, intercept_scaling=1, l1_ratio=None,
                   max_iter=10000, multi_class='auto', n_jobs=None,
                   penalty='l1', random_state=None, solver='saga', tol=0.0001,
                   verbose=0, warm_start=False)
```

STEP_10: Applying function **fn_performance_models_data** we have:

```
df_tr_standard, df_ts_standard, scaler = ai.fn_standardize_df(df_tr_set, df_ts_set)

list0_model_names = list(df_filtered_cv.index)
X_train, y_train = df_tr_standard.iloc[:, :-1].values, df_tr_standard.iloc[:, -1].values
list0_models = [dict0_model_instances[i].fit(X_train, y_train) for i in list0_model_names]

list0_thresholds = np.linspace(0, 1, 20)
legend = list0_model_names

fn_performance_models_data(list0_models, df_tr_standard, list0_thresholds, legend)
```



It can be seen from the plots above that model_179 gives the best performance. Let us assume that we need to achieve maximum **recall** for class_1 (malignant) but with minimal impact on the performance for class_0 (benign). After studying the plots above we observe the following:

1. The values of rec_1 and rec_0 seem to be above 90% around the 40% probability threshold.
2. At the 40% probability threshold, the values of prec_1 & prec_0 are still quite high.

Thus we conclude that **model_179**, set at a probability threshold of around 40%, is good for our classification purposes. We then train the **model_179** on the entire train set and test its performance across the train and test sets as shown below:

```
df_Xy_ = df_tr_standard
model_ = dict0_model_instances['model_179'].fit(X_train, y_train)
thresh = 0.4

fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.0602
ACCURACY: 97.489
--------------------
```

	prec	re
class_0	96.104	96.73
class_1	98.239	97.89

STEP_11: Finally we test the chosen model on the test set to check if the train set performance of the model generalizes to data it was not exposed to while training. For this, we use the function **fn_test_model** as shown below:

```
df_Xy_ = df_ts_standard

fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.0604
ACCURACY: 97.368
--------------------
```

	prec	rec
class_0	93.333	100.000
class_1	100.000	95.833

As can be inferred from the above results, the model generalizes well to the test set and manages to recover 95% of the 'malignant' class, while maintaining the precision & recall of the 'benign' class high.

The function also indicates that the model has an accuracy of around 96.5% and a Log Loss of around 0.06.

Note that the log loss performance of the model is based solely on the probability predictions of the model (model.predict_proba) and not on label/class prediction (model.predict). As defined previously, the log loss is simply the mean of the **negative log probability** values predicted for the correct class over the entire data set, whereas the other performance metrics (acc, prec, rec) are based on the counts or frequency of the correct & wrong predictions for each class at the probability **threshold** selected. The probability threshold selected for class prediction has no effect on the log loss.

It is possible that a model can predict the correct class even though its confidence for that prediction maybe less (say 55%) and if this behaviour of the model is consistent over a major part of the data set, then the model will have a good "class prediction" performance (accuracy, precision, recall), but still have a high log loss.

Generally a low log loss would correspond to a good predictive performance, but the relationship between them can be non linear. In other words if we collect the **accuracy-log loss performances** of say 10 Logistic Regression models which were trained on the same data but with their hyperparameters tuned differently and sort these pairs of values by accuracy, the log losses need not be perfectly sorted.

7. UNBALANCED BINARY CLASSIFICATION

UNBALANCED BINARY CLASSIFICATION

In the previous chapter we performed Binary classification on the Breast Cancer data set. The classes/labels of this data set were more or less balanced and neither class dominated the other. In this chapter we shall perform Binary Classification on a data set with unbalanced class ratio.

CLASSIFICATION OF CREDIT CARD FRAUD DATASET:

DATA DESCRIPTION:

The data represents credit card transactions that occurred over two days in September 2013 by European cardholders. The dataset is credited to the Machine Learning Group at the Free University of Brussels.

It contains only numeric input variables which are the result of a PCA transformation applied on the cardholder's original recorded features.The only features which have not been transformed with PCA are 'Time' and 'Amount'. features. 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction amount.

Each record is classified as either normal (class_0) or fraudulent (class_1). The transactions are heavily skewed towards class_0. Specifically, there are 492 fraudulent credit card transactions out of a total of 2,84,807 transactions, which is a total of about 0.172% of all transactions.

#	Column	Non-Null Count	Dtype
0	Time	284796 non-null	float64
1	V1	284796 non-null	float64
2	V2	284796 non-null	float64
3	V3	284796 non-null	float64
4	V4	284796 non-null	float64
5	V5	284796 non-null	float64
6	V6	284796 non-null	float64
7	V7	284796 non-null	float64
8	V8	284796 non-null	float64
9	V9	284796 non-null	float64
10	V10	284796 non-null	float64
11	V11	284796 non-null	float64
12	V12	284796 non-null	float64
13	V13	284796 non-null	float64
14	V14	284796 non-null	float64
15	V15	284796 non-null	float64
16	V16	284796 non-null	float64
17	V17	284796 non-null	float64
18	V18	284796 non-null	float64
19	V19	284796 non-null	float64
20	V20	284796 non-null	float64
21	V21	284796 non-null	float64
22	V22	284796 non-null	float64
23	V23	284796 non-null	float64
24	V24	284796 non-null	float64
25	V25	284796 non-null	float64
26	V26	284796 non-null	float64
27	V27	284796 non-null	float64
28	V28	284796 non-null	float64
29	Amount	284796 non-null	float64
30	Class	284807 non-null	int64

A) DATA PREPROCESSING & EDA STAGE:

STEP_1: Applying fn_preprocess_data we have:

```
[17]  1  labels_col_name = 'Class'
      2
      3  df_Xy, dict0_code2feats = fn_preprocess(df_Xy_raw, labels_col_name)
      4  print(df_Xy.head(10))
```

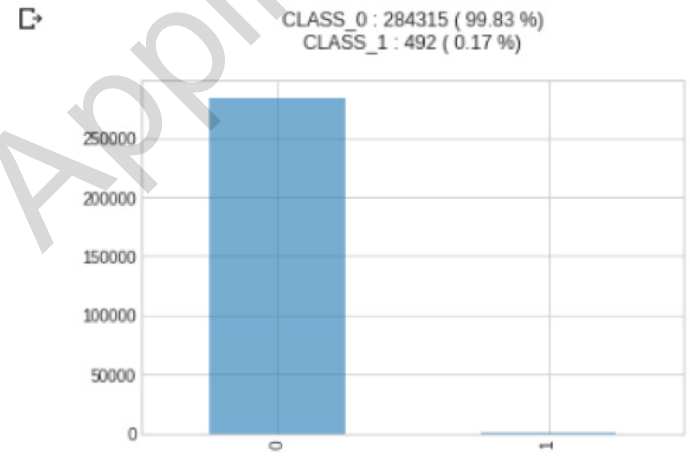
```
>>> 1/3: CHECKING FOR ROWS WITH MISSING LABELS
*** NO LABELS MISSING
>>> 2/3: CHECKING FOR MISSING FEATURE VALUES
*** 330 FEATURE VALUES MISSING! REPLACING WITH MEAN OF FEATURE
>>> 3/3: RENAMING COLUMNS WITH GENERIC NAMES
```

	f0	f1	f2	f3	...	f27	f28	f29	labels
0	0.0	-1.359807	-0.072781	2.536347	...	0.133558	-0.021053	149.62	0
1	0.0	1.191857	0.266151	0.166480	...	-0.008983	0.014724	2.69	0
2	1.0	-1.358354	-1.340163	1.773209	...	-0.055353	-0.059752	378.66	0
3	1.0	-0.966272	-0.185226	1.792993	...	0.062723	0.061458	123.50	0
4	2.0	-1.158233	0.877737	1.548718	...	0.219422	0.215153	69.99	0
5	2.0	-0.425966	0.960523	1.141109	...	0.253844	0.081080	3.67	0
6	4.0	1.229658	0.141004	0.045371	...	0.034507	0.005168	4.99	0
7	7.0	-0.644269	1.417964	1.074380	...	-1.206921	-1.085339	40.80	0
8	7.0	-0.894286	0.286157	-0.113192	...	0.011747	0.142404	93.20	0
9	9.0	-0.338262	1.119593	1.044367	...	0.246219	0.083076	3.68	0

As can be seen from the feedback that the function provides, the dataset has missing values, so the function replaces these missing values with the means of their respective columns. It also renames the columns with generic feature names and returns data frame df_Xy.

STEP_2: Applying fn_show_label_distribution we have:

```
[18]  1  y = df_Xy.labels.values
      2
      3  fn_label_distr(y)
```



It can be seen that the data is extremely imbalanced, with class_1 forming just 0.17% of the entire data.

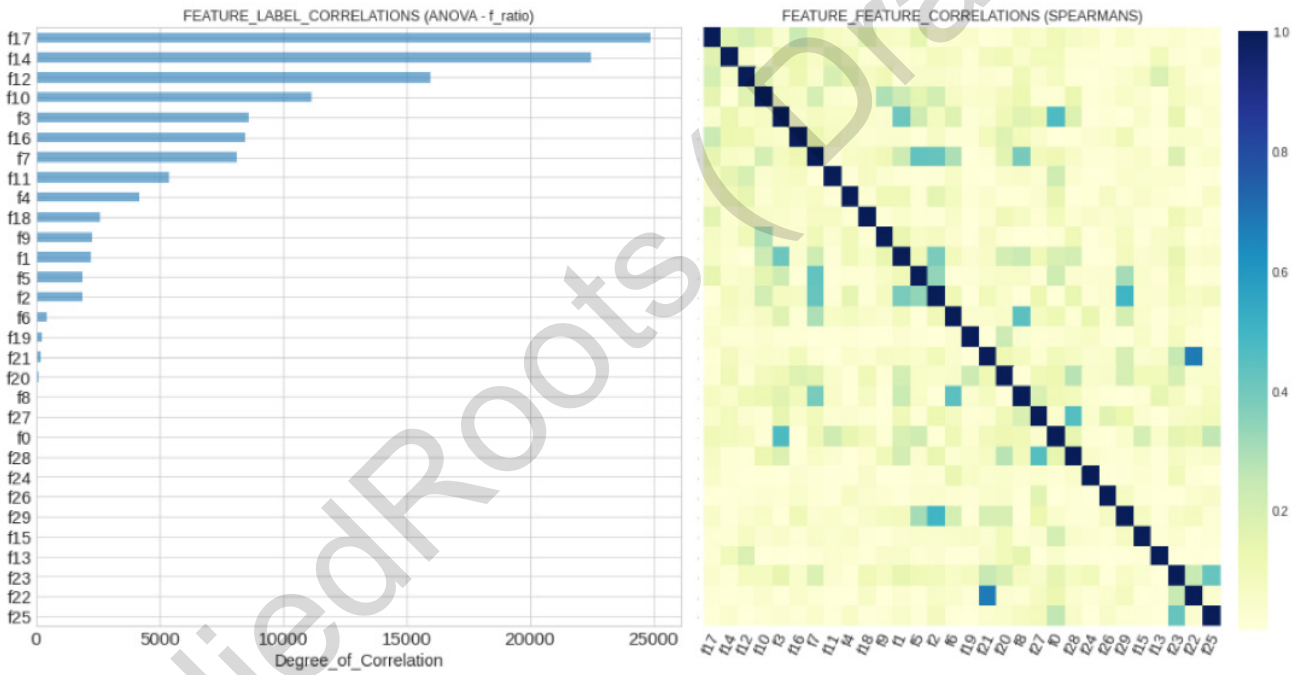
STEP_3: Applying fn_tr_ts_split we have:

```
[19]  1  idxs_tr, idxs_ts = fn_tr_ts_split(df_Xy, ts_size = 0.2)
      2
      3  df_tr_raw = df_Xy.iloc[idxs_tr]
      4  df_ts_raw = df_Xy.iloc[idxs_ts]
      5
      6  df_tr_raw.shape, df_ts_raw.shape
```

```
((227845, 31), (56962, 31))
```

STEP_4: Applying fn_feat_select_clf we have:

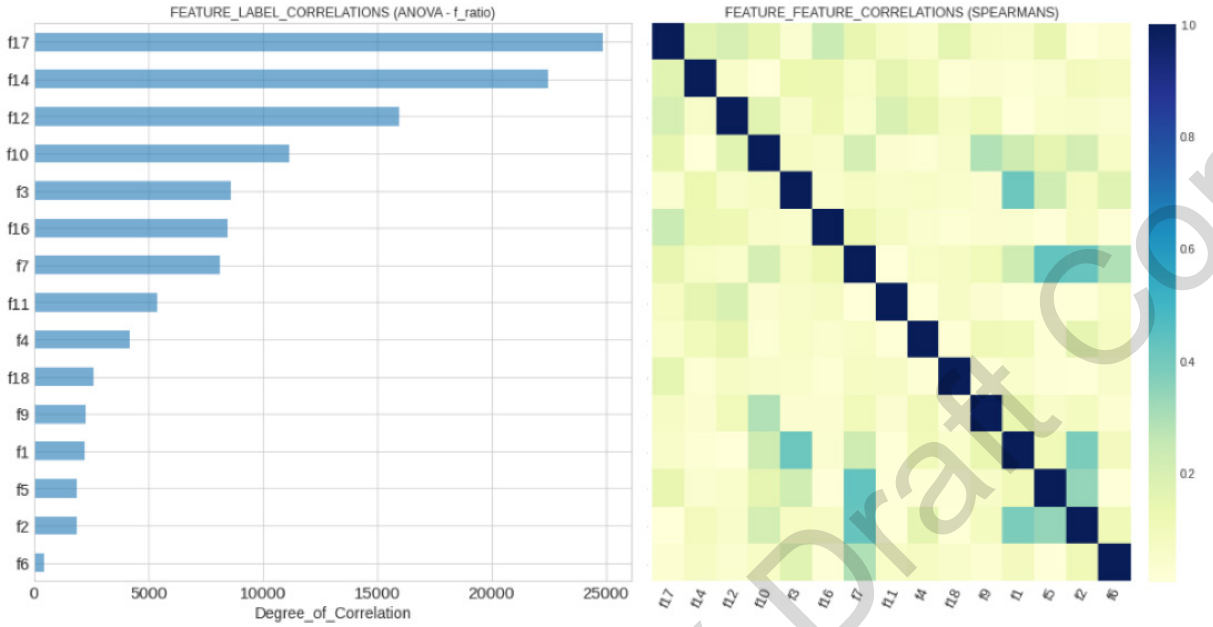
```
1  list0_good_feats = fn_feat_select_clf(df_tr_raw, plot = True)
```



It can be seen from the correlation matrix that there is no correlation between the features. This is expected since the features provided were already PCA transformed to begin with. Using the feedback we get from the above plots we choose a **feature-to-label** threshold of 300 and a **feature-to-feature** correlation threshold of 0.9 as shown below:

```
1  list0_good_feats = fn_feat_select_clf(df_tr_raw, plot = True,
2                                        thresh_feat_label = 300,
3                                        thresh_feat_feat = 0.9)
```

This results in the following output, the original feature set is reduced to a smaller set of features:



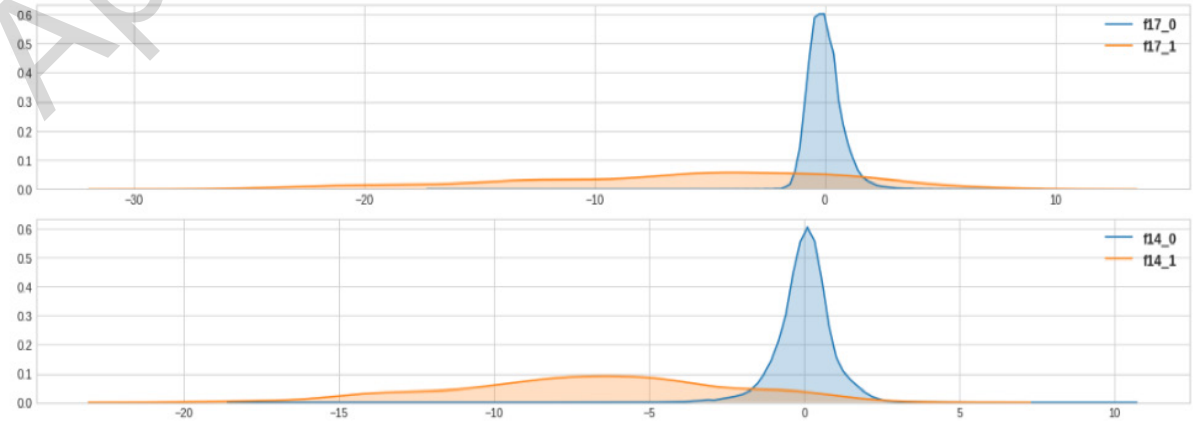
We then use only the features selected above, thus getting the final form of our train & test sets:

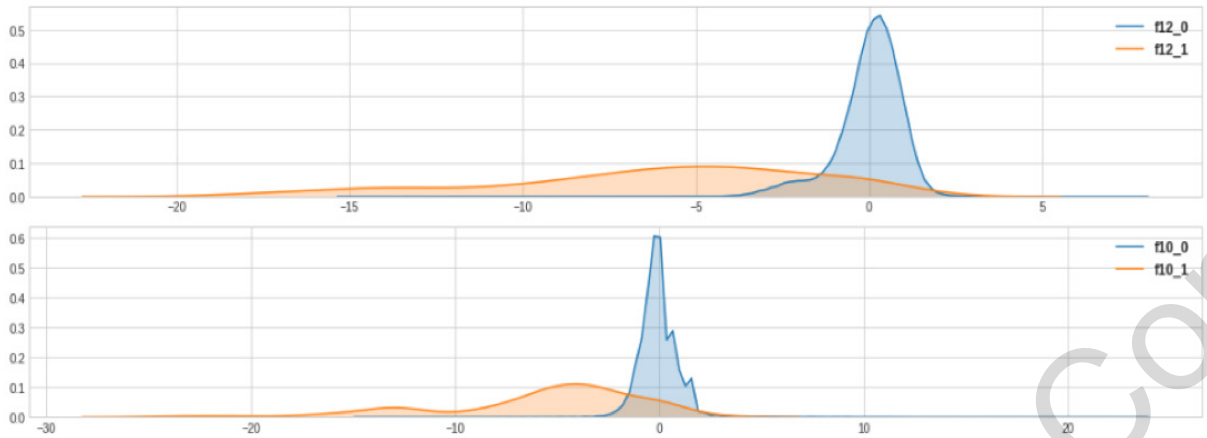
```
df_tr = df_tr_raw[list0_good_feats + ['labels']]
df_tr.index = range(len(df_tr))

df_ts = df_ts_raw[list0_good_feats + ['labels']]
df_ts.index = range(len(df_ts))

df_tr.shape, df_ts.shape
```

```
((227845, 16), (56962, 16))
```

STEP_5: Applying fn_compare_class_distr we have:

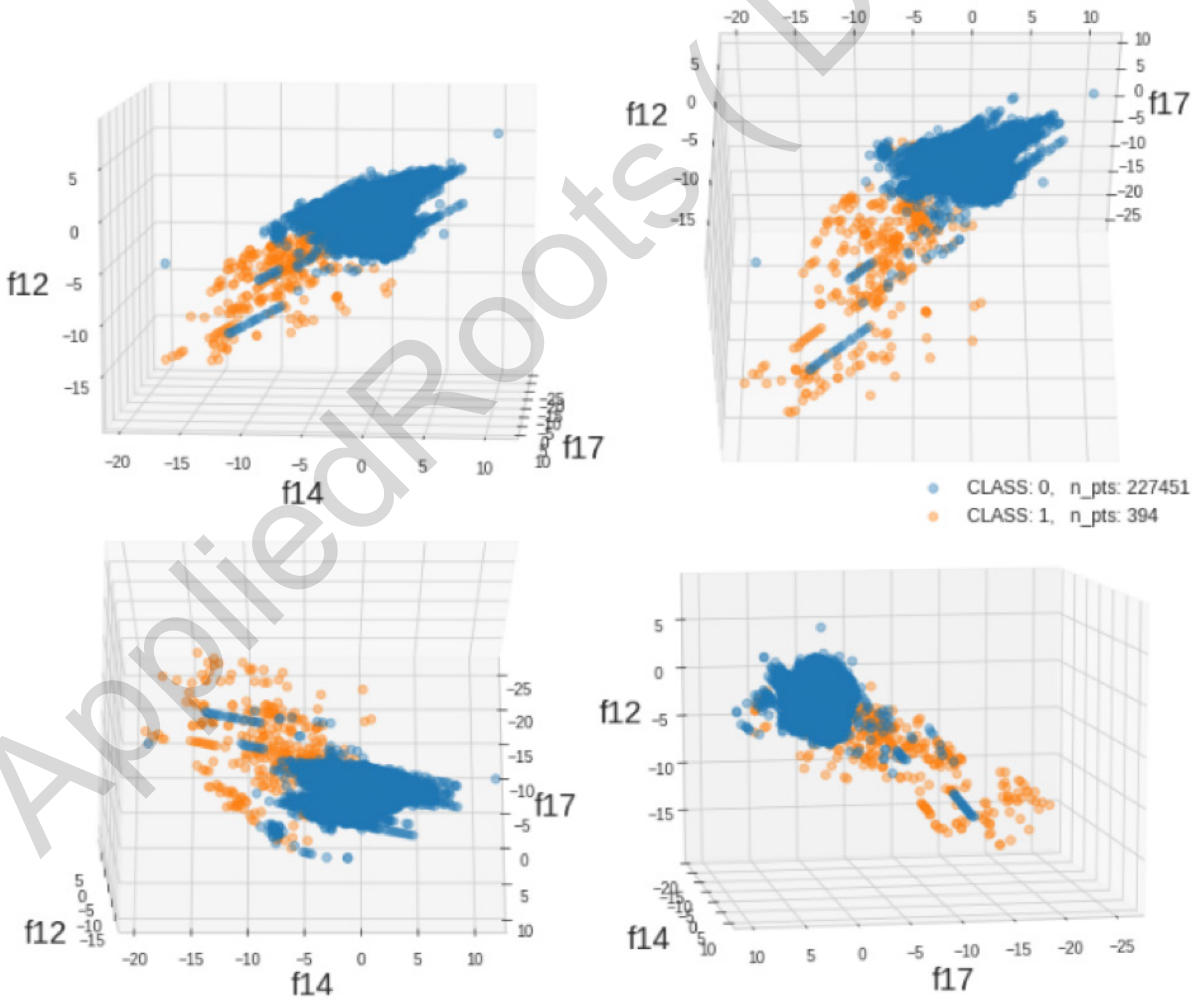
```
fn_label_distr_feats(df_tr, n_top_feats = 4)
```





STEP_6: Applying fn_plot_3d_binary on train data we have:

```
1   fn_plot_3d_binary(df_tr)
```

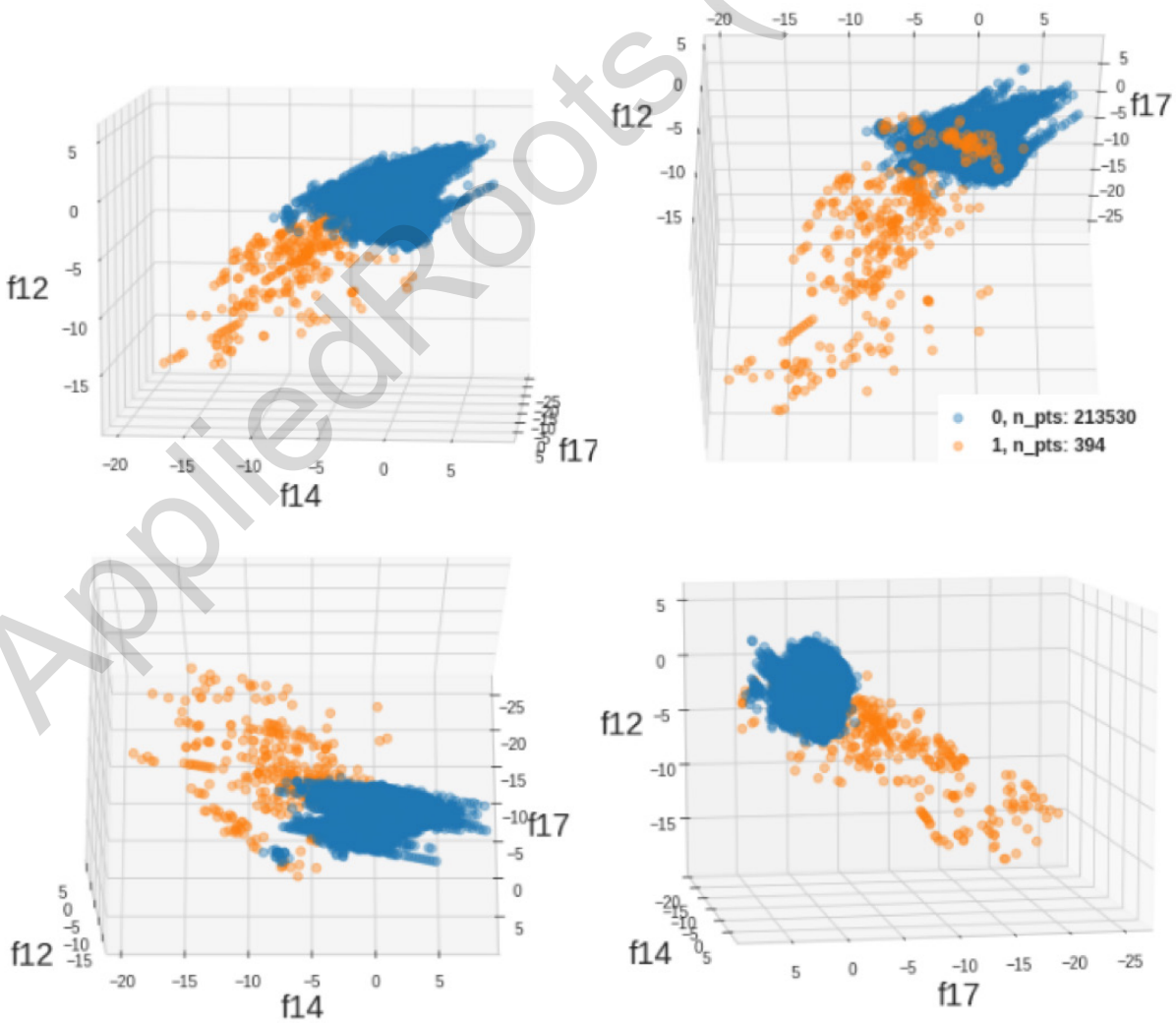


By analysing the distribution plots obtained from step_5 and the 3D plots outputted above, it can be seen that there is a fair degree of separation between the two classes. It can also be observed from the distribution plots, that the distribution of class_1 is quite spread out compared to that of class_0.

STEP_7 : Applying **fn_LOF_outlier_idxs** on the train set and then applying **fn_plot_3d_binary** on the outlier free data we have:

```
df_Xy_ = df_tr[df_tr.labels == 0]
n_clusters = 10
n_neighbors = 30

list0_outlier_idxs_c0 = fn_LOF_outlier_idxs(df_Xy_, n_clusters, n_neighbors)

print(len(list0_outlier_idxs_c0))

fn_plot_3d_binary(df_tr.drop(list0_outlier_idxs_c0))
```

```
100% (10 of 10) |########################| Elapsed Time: 0:01:45 Time:  0:01:45
13921
```

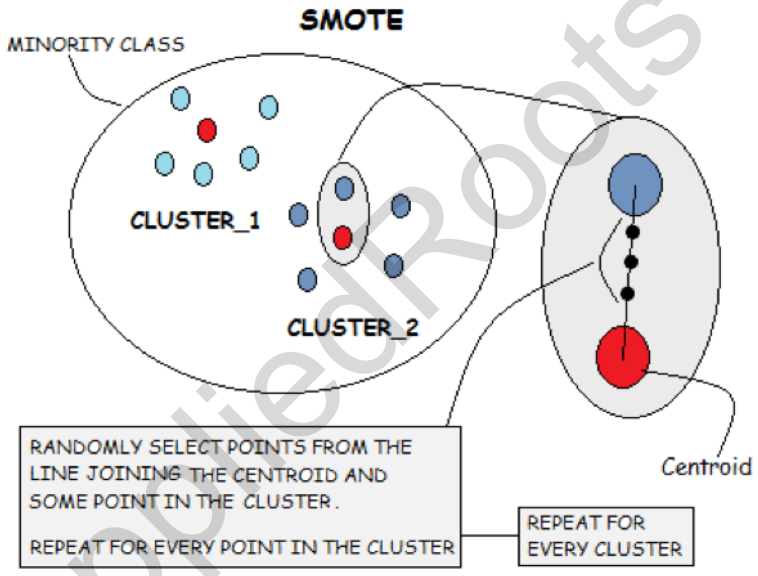


By analysing the distribution plots obtained from step_5 and the 3D plots outputted above, it can be seen that there is a fair degree of separation between the two classes. It can also be observed from the distribution plots, that the distribution of class_1 is quite spread out compared to that of class_0.

STEP_8: (UPSAMPLING CLASS_1):

At this point we introduce a new function called **fn_smote**, which performs the task of upsampling data fed to it, by using a method called **synthetic minority oversampling technique** or smote for short. Upsampling the minority class for highly imbalanced data sets (while at the same time downsampling the majority class) is a precautionary measure to ensure that our model learns an optimal decision surface, that is an outcome of being trained on a more or less balanced dataset.

The smote technique creates synthetic (ie: new) data points, whose locations in the feature space are based on the bounds imposed by, the distribution of the actual data points of the minority class. In other words, if the minority class were enclosed by some imaginary surface, formed by their outermost points, then these synthetic points should lie inside this surface. We implement this in fn_smote by performing the actions outlined in the image below:



Using **fn_smote** we have:

```
n_clusters = 10
n_smotes_per_pt = 150

df_smote =  fn_smote(df_tr_, n_clusters, n_smotes_per_pt, label = 1)
df_tr_c1_oversample = pd.concat([df_tr[df_tr.labels == 1], df_smote])

df_tr_c1_oversample.shape
```

```
(59494, 16)
```

As can be seen from the code above, we first choose to cluster the minority class in the train set into 10 clusters and we then generate 150 synthetic points for each actual data point present. These synthetic points are then combined with the actual minority class data points and we thus obtain 59,494 points represented as the data frame df_tr_c1_ovesample.

We then undersample the majority class as shown below (in this particular case we sample 50%):

```
df_tr_c0_undersample = df_tr[df_tr.labels == 0].sample(frac = 0.5)
df_tr_c0_undersample.shape
```

```
(113726, 16)
```

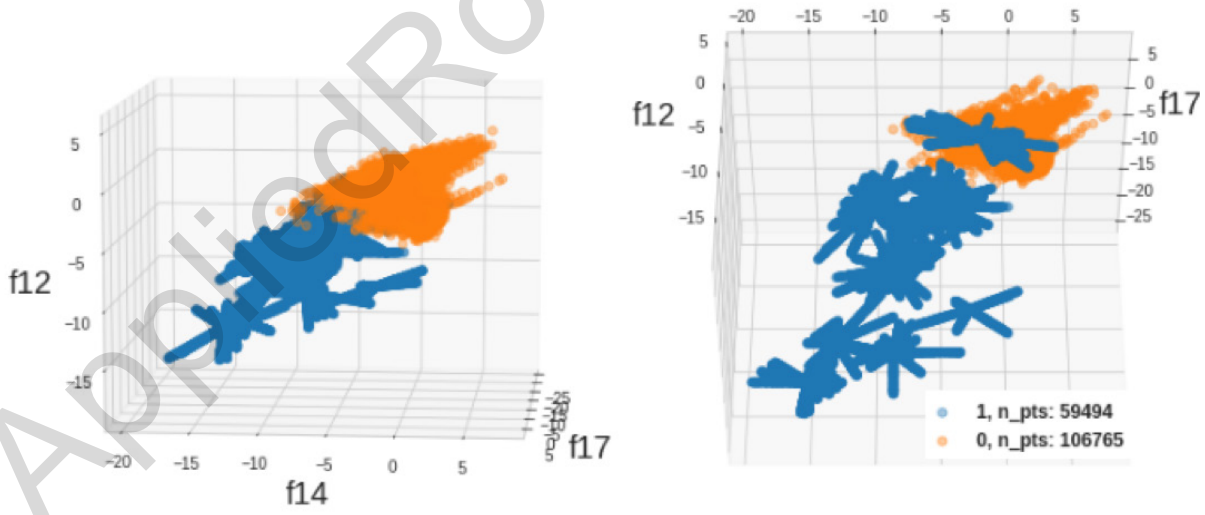
After the above two sampling processes we have a synthetic train set with their classes much more balanced (**ie: n_class_1_pts: 59494 & class_0: 113726**):

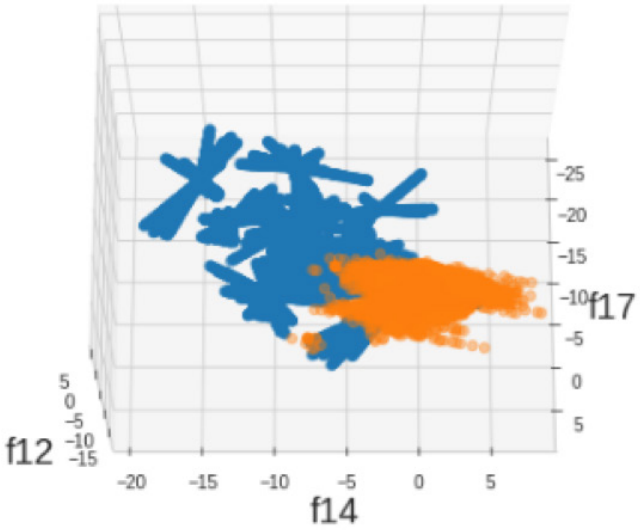
```
df_tr_synth = pd.concat([df_tr_c1_oversample, df_tr_c0_undersample])
df_tr_synth.index = range(len(df_tr_synth))
df_tr_synth.shape
```

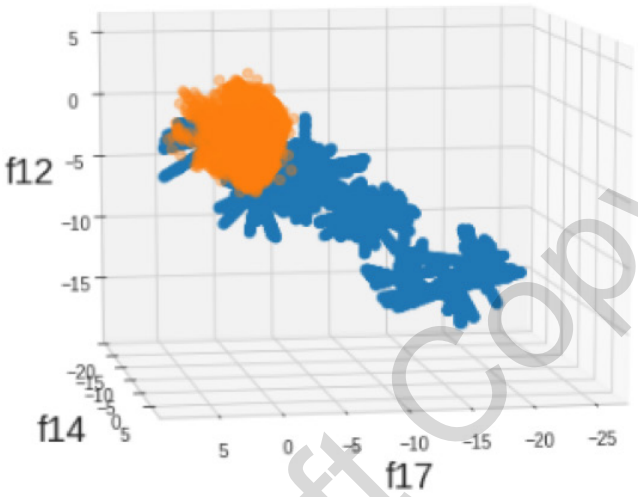
```
(173220, 16)
```

Applying **fn_plot_3d_binary** on the synthetic train data we have:

```
fn_plot_3d_binary(df_tr_synth)
```







Comparing the above set of plots with those for the original highly imbalanced data, we can see that the distribution of both the classes remain same, even though the data has been artificially balanced.

B) MODEL TRAINING & EVALUATION:

In the case of imbalanced datasets, the steps for model training and evaluation are as follows:

STEP_9: Perform K fold cross validation using the **synthetic train set** and then choose a subset of models that perform the best.

STEP_10: These selected set of models are then retrained on the entire **synthetic train set**. Precision-recall curves are then plotted by using these models to predict on the **train set**.

STEP_11: We choose the model with the best precision-recall curve as our final model and then test it on the **test set** to check for generalization to new data.

STEP_12: We then plot the precision-recall curves, for the predictions of the final model, on the following datasets:

a. The synthetic train set.

b. The actual train set.

c. A bunch of datasets that are obtained by under sampling the majority class to be comparable to the size of the minority class.

Step_12 helps us understand the effect of imbalance in data, by comparing the model's precision-recall performance for both balanced and imbalanced cases.

Implementing the steps described above we have:

STEP_10: Applying fn_kfoldcv on the synthetic train set we have:

```
from sklearn.linear_model import LogisticRegression

df_tr_ = df_tr_synth
model_class = LogisticRegression

parameter_grid = dict(penalty = ['l2'],
                      C = [10**i for i in range(-15, 15)],
                      solver = ['lbfgs'],
                      class_weight = ['balanced'],
                      max_iter = [100_000])

z = fn_kfoldcv_clf_binary(df_tr_, model_class, parameter_grid)

df_kfoldcv, dict0_model_instances, list0_invalid_models = z
df_kfoldcv.describe()
```

```
100% (30 of 30) |########################| Elapsed Time: 0:01:13 Time:  0:01:13
```

	mean_loss	mean_rec_0	mean_rec_1	mean_prec_0	mean_prec_1	std_loss	std_rec_0	std_rec_1	std_prec_1	std_prec_1
count	30.000000	30.000000	30.000000	30.000000	30.000000	3.000000e+01	30.00000	30.000000	30.000000	30.000000
mean	0.313581	85.792733	94.824517	92.730450	96.491467	2.794002e-02	13.41820	3.891583	5.648250	2.287067
std	0.270461	7.565811	2.961101	2.560835	0.810441	1.961233e-02	8.22742	3.985831	3.894842	1.740173
min	0.106334	76.593500	91.097500	89.655000	95.487000	1.841738e-11	6.20800	0.595000	2.232500	0.085500
25%	0.106345	76.868000	91.172000	89.751500	95.506000	5.408843e-04	6.21050	0.613000	2.235000	0.963500
50%	0.108771	91.823500	97.168000	94.707000	97.118750	3.997550e-02	6.82450	0.673250	2.518500	0.965250
75%	0.688002	92.429500	97.260000	95.012000	97.163500	3.997732e-02	23.13200	8.811000	10.248500	4.457000
max	0.693147	92.432000	97.262500	95.014000	97.165000	5.643067e-02	23.40650	8.902500	10.345000	4.513000

We then filter out the best performing models as shown below:

```
df = df_kfoldcv

df1 = df[df.std_rec_1 < 0.6][:5]
df2 = df[df.mean_rec_1 > 95][:5]
df3 = df[df.mean_rec_0 > 95][:5]

df_filtered_cv = pd.concat([df1, df2, df3]).drop_duplicates()
df_filtered_cv = df_filtered_cv.sort_values(by = 'mean_prec_1', ascending = False)[:3]
df_filtered_cv
```

	mean_loss	mean_rec_0	mean_rec_1	mean_prec_0	mean_prec_1	std_loss	std_rec_0	std_rec_1	std_prec_1	std_prec_1
model_16	0.106334	92.4250	97.261	95.0140	97.1590	0.039807	6.2210	0.604	2.2440	0.9730
model_13	0.113258	89.9835	97.125	93.9545	97.1415	0.040578	8.8725	0.846	3.5775	0.7855
model_15	0.107590	92.0925	97.211	94.8555	97.0960	0.039998	6.5695	0.595	2.4145	1.0130

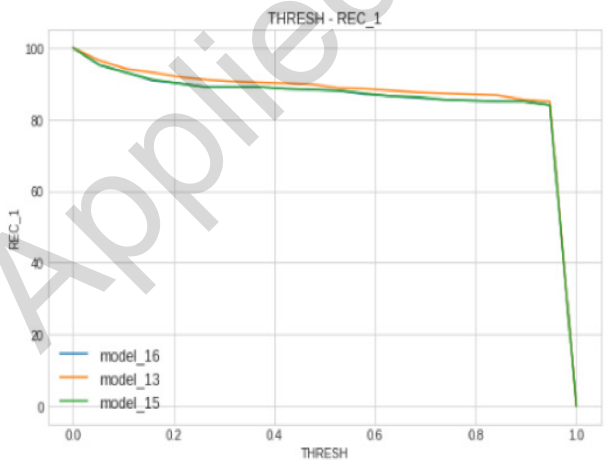
We can check the parameters of the above filtered models as shown below:

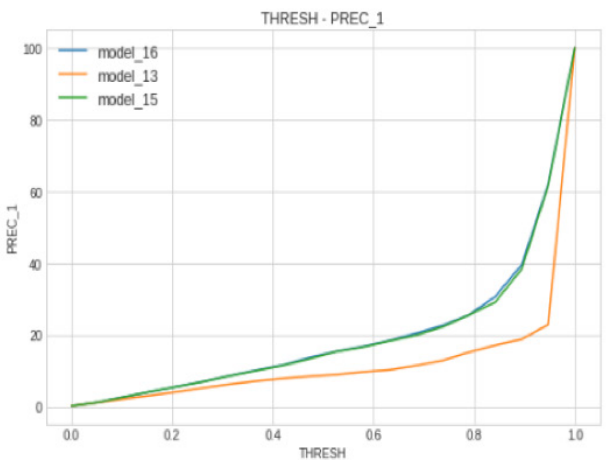
```
dict0_model_instances['model_16'], dict0_model_instances['model_13'], dict0_model_instances['model_18']
```

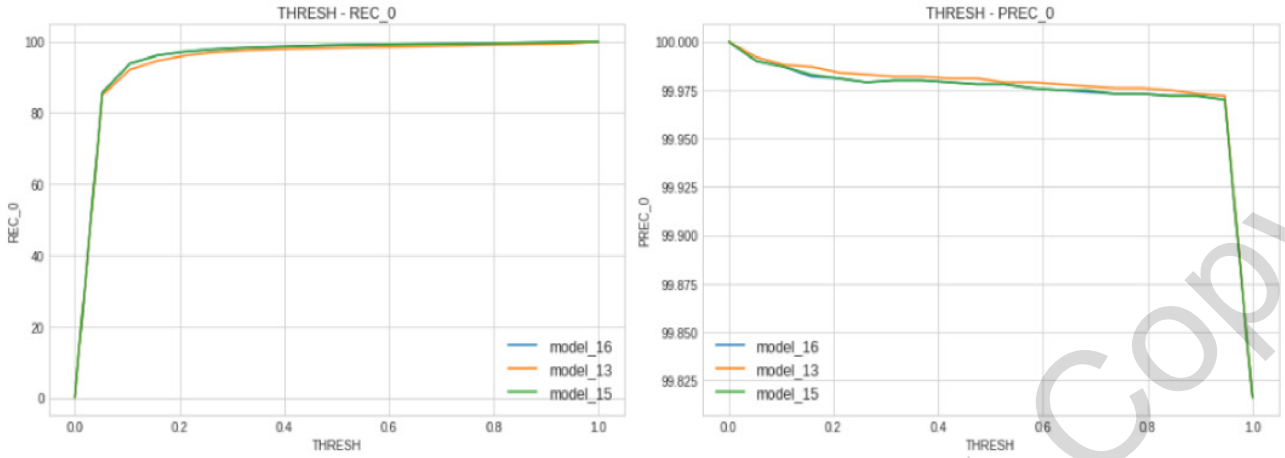
```
(LogisticRegression(C=10, class_weight='balanced', dual=False,
                    fit_intercept=True, intercept_scaling=1, l1_ratio=None,
                    max_iter=100000, multi_class='auto', n_jobs=None,
                    penalty='l2', random_state=None, solver='lbfgs', tol=0.0001,
                    verbose=0, warm_start=False),
 LogisticRegression(C=0.01, class_weight='balanced', dual=False,
                    fit_intercept=True, intercept_scaling=1, l1_ratio=None,
                    max_iter=100000, multi_class='auto', n_jobs=None,
                    penalty='l2', random_state=None, solver='lbfgs', tol=0.0001,
                    verbose=0, warm_start=False),
 LogisticRegression(C=1000, class_weight='balanced', dual=False,
                    fit_intercept=True, intercept_scaling=1, l1_ratio=None,
                    max_iter=100000, multi_class='auto', n_jobs=None,
                    penalty='l2', random_state=None, solver='lbfgs', tol=0.0001,
                    verbose=0, warm_start=False))
```

STEP_11: Applying function **fn_plot_perform_curves** we have:

```
df_tr_standard, df_ts_standard = fn_standardize_df(df_tr, df_ts)

list0_model_names = list(df_filtered_cv.index)
X_train, y_train = df_tr_synth.iloc[:, :-1].values, df_tr_synth.iloc[:, -1].values
list0_models = [dict0_model_instances[i].fit(X_train, y_train) for i in list0_model_names]

df_Xy_ = df_tr_standard
list0_thresholds = np.linspace(0, 1, 20)
legend = list0_model_names

%time fn_performance_models_data(list0_models, df_Xy_, list0_thresholds, legend)
```







From the above plots we observe that:

1. All the selected models generally have similar Precision-Recall curves with only slight differences
2. The top-right plot indicates that the precision of class_1 remains low till around the 85% threshold.
3. The below-right plot indicates that the precision of class_0 never falls below 98% and remains above 99.9% till around the 90% threshold.
4. The 20% threshold seems most suitable, since it affords a good balance between recall of class_1 and class_0 (ie: Both are greater than 90%).

We choose model_13 as our predictor and retest its performance by fitting the model to the full synthetic train set and checking its predictions (at 20% threshold) on the train set:

```
df_Xy_ = df_tr_standard
model_ = dict0_model_instances['model_13'].fit(X_train, y_train)
thresh = 0.2

fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.0708
ACCURACY: 95.856
--------------------
```

	prec	rec
class_0	99.985	95.863
class_1	3.947	92.132

STEP_12: MODEL TESTING:

Testing model_13 at the same threshold on the test set we have:

```
df_Xy_ = df_ts_standard

fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.0837
ACCURACY: 95.527
--------------------
```

	prec	rec
class_0	99.989	95.530
class_1	3.493	93.878

The performance of the chosen model is quite similar across the train and test sets and hence we conclude that the model generalizes well to data outside the train set.

STEP_13: COMPARING MODEL PERFORMANCE ON BALANCED & IMBALANCED DATA

For the sake of implementing the above task we create the following datasets by undersampling the majority class. These datasets represent data where the class imbalance is not so extreme as in the original data.

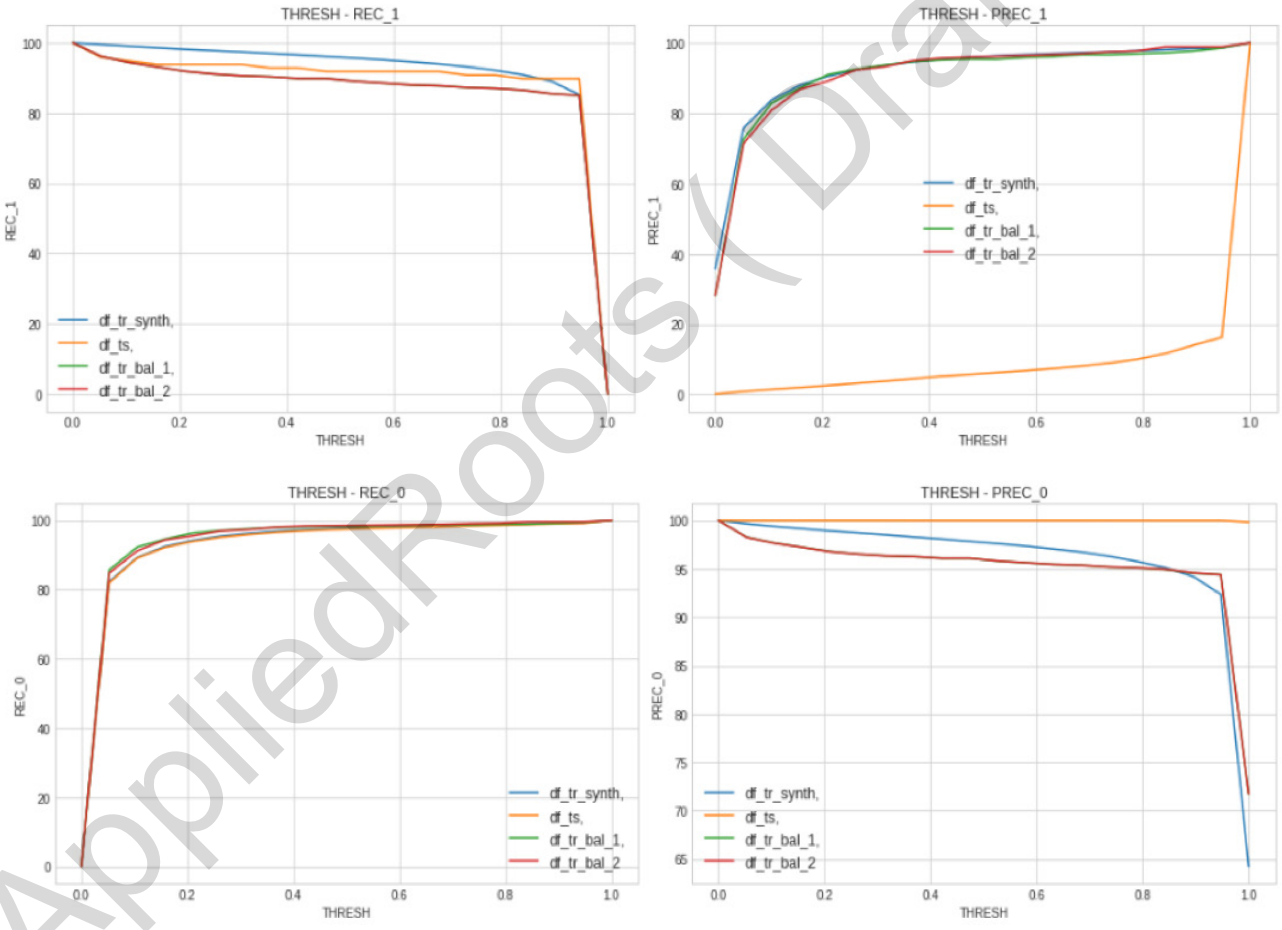
```
df_tr_c1 = df_tr_standard[df_tr_standard.labels == 1]
df_ts_c1 = df_ts[df_ts.labels == 1]

df_tr_c0_downsample_1 = df_tr_standard[df_tr_standard.labels == 0].sample(1000)
df_tr_c0_downsample_2 = df_tr_standard[df_tr_standard.labels == 0].sample(1000)

df_ts_c1_upsample_1 = df_ts_c1.sample(frac = 250, replace = True)
df_ts_c1_upsample_2 = df_ts_c1.sample(frac = 580, replace = True)

df_tr_bal_1 = pd.concat([df_tr_c0_downsample_1, df_tr_c1])
df_tr_bal_2 = pd.concat([df_tr_c0_downsample_2, df_tr_c1])

df_ts_bal_1 = pd.concat([df_ts_0, df_ts_c1_upsample_1])
df_ts_bal_2 = pd.concat([df_ts_0, df_ts_c1_upsample_2])
```

We then use another new function fn_performance_model_datas to compare the precision-recall curves of the model when predicted on multiple datasets. The following five datasets are predicted on and their performance curves plotted:

a. **df_tr_synth** (Synthetic train set)
b. **df_ts** (The original test data.)
c. **df_tr_bal_1 & df_tr_bal_2** (just created above)

NOTE: **fn_performance_model_datas** is used to plot the precision recall curves of one model made to predict on multiple datasets, whereas **fn_performance_models_data** is used to plot the precision recall curves of multiple models made to predict on a single dataset. The difference in the function names is only in which word in the name is plural (ie: data - models or datas - model).

```
1  list0_df_Xy_s = [df_tr_synth, df_ts, df_tr_bal_1, df_tr_bal_2]
2  legend = 'df_tr_synth, df_ts, df_tr_bal_1, df_tr_bal_2'.split()
3
4  list0_thresholds = np.linspace(0, 1, 20)
5
6  fn_performance_model_datas(model_, list0_df_Xy_s, legend, list0_thresholds)
```



From the plots above we observe that:

a. As seen from the two plots on the right, it is the precision metric that gets most affected by imbalance in data.

b. The precision curves for **class_1** on the **synthetic test set** and the other 2 **balanced data sets** are almost identical and show good performance (ie: reaches 80% around the 10% threshold).

3. The precision curves for **class_1** on the **test set** remains below 20% till around the 90% threshold)
4. The precision curves for **class_0** on the **test set** does not fall below 98%.
5. The recall performance of the model for the both classes, on **all the five datasets** used, are more or less similar. Also, they are consistent with recall performance observed for the **train set**.

While evaluating a model that has to perform on imbalanced datasets, studying its performance on the original imbalance train set and also balanced versions of the train set, gives us a good perspective on the effect of the imbalance on the performance metrics.

The reason that highly imbalanced data sets produce such effects on precision can explained as follows:

1. In the Binary classification task above, say the recall for the class_0 is 99%. This means that 1% of the majority class got wrongly classified as class_1. 1% of class_0 is around 2300 points. Now, even if the recall for class_1 is 100% (ie: around 300 points), the number of class_0 points classified as class_1 hugely outnumbers that of class_1. Thus the precision for class_1 remains very low, even though only 1% of the other class was wrongly classified.
2. Imbalance has the renverse effect when considering the majority class. In other words precision for class_0 remains high irrespective of the model's performance with respect to class_1. Assume all the class_1 points are wrongly classified as class_0 (300 points) and assume that the recall for class_0 is just 50%. This means there are around 100,000 points classified correctly. The precision for class_0 will still be above 99%.

This means that the decision boundary learnt by the model is actually quite good and it is the imbalance in the data that produces poor precision performance with respect to class_1.

8. LINEAR REGRESSION

LINEAR REGRESSION

PERFORMANCE METRICS:

The following three performance metrics will be used for evaluating a regression model's performance - the Mean Absolute Error, the Mean Absolute Percentage Error and the r2 score. The Root Mean Squared Error can also be used, but it is not as interpretable as three metrics just mentioned.

MEAN ABSOLUTE ERROR (MAE):

is defined as the mean of the absolute errors between actual label and predicted label values for the entire data set.

$$MAE(y, f(X)) = \frac{1}{n}\sum_{i}\left(|y_i - f(x_i)|\right)$$

MEAN ABSOLUTE PERCENTAGE ERROR (MAPE):

is defined as the mean of the ratios between the absolute errors and the absolute value of the labels for the entire data set. The values of MAPE will lie within the interval 0 to 1. Here the errors are expressed as percentages of the actual value being predicted and hence MAPE is much more interpretable than MAE.

$$MAPE(y, f(X)) = \frac{1}{n}\sum_{i}\left(\frac{|y_i - f(x_i)|}{|y_i|}\right)$$

R2 SCORE (r2):

is defined with respect to two values:

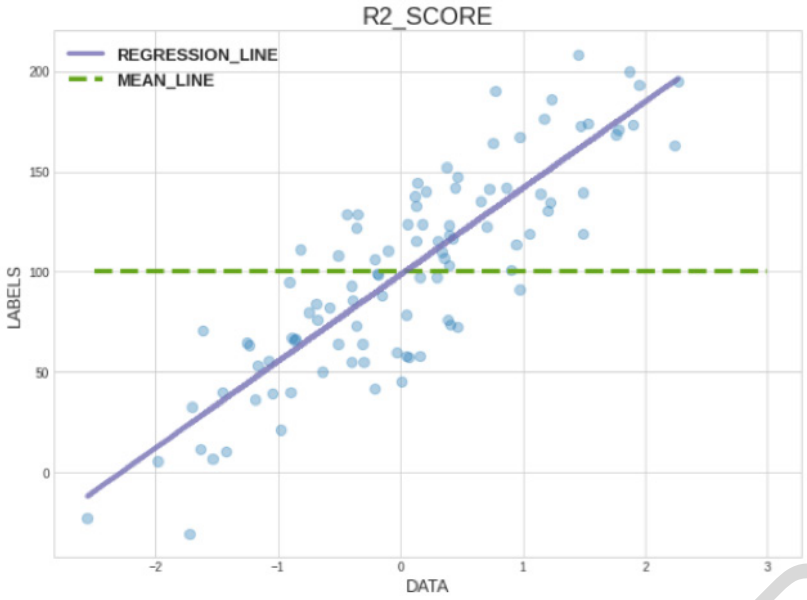
1. The variance (ie: squared loss) of the labels values from the mean value of the labels (ie: **var_mean**).
2. The variance of the label values from the regression surface predicted by the model (ie: **var_reg**).

It is defined as:

$$r2(y, f(x)) = \frac{var\_mean - var\_reg}{var\_mean}$$

The values of the r2 score lie usually within the interval of 0 to 1. The smaller the variance of the labels from the regression surface (ie: **var_reg**), higher the r2 score will be and when this variance is equal to zero, the r2 score will be equal to one. When **var_reg** is equal to **var_mean**,

then the r2 score will be equal to zero. If **var_reg** is more than **var_mean**, then the r2 score will be negative.



REGRESSION WITH OUTLIER RESILIENCE

The Linear Regression algorithm in its basic form is quite susceptible to the effect of any outliers in the data. Consider the image shown below, note that the data contains four outliers, whereas the rest of the data indicate a definite correlation with the labels. The image shows us the difference between the “Preferred Regression line” and the “Actual Regression line” learnt by a basic linear regression model.

To overcome this weakness, modified regressions methods like RANSAC regression and Huber regression, have been developed, which are more resilient to the presence of outliers in the data.

RANSAC REGRESSION:

RANSAC stands for Random Sample Consensus. In this method the following steps are implemented:

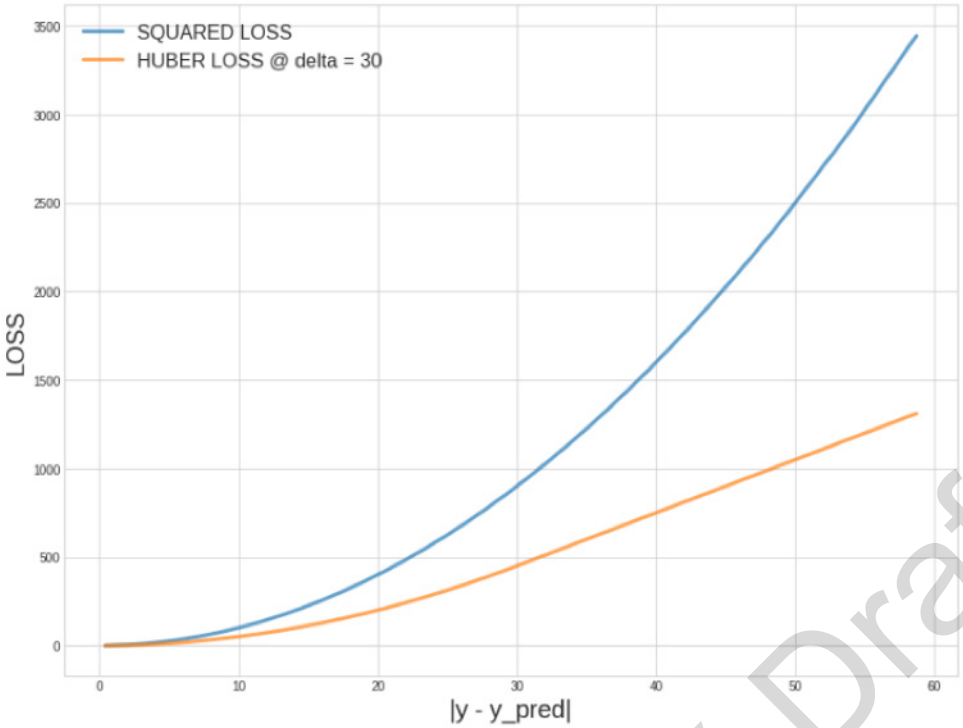
1. Randomly select **d+1** number of points from the dataset, where d is equal to the number of dimensions of the data.
2. Fit a Linear regression model to the selected data and then count the number of data points that lie within a pre-specified distance from the regression surface thus learnt. This pre-specified distance is called the **residual threshold**. Generally the **MAD** (median absolute deviation) of the labels is chosen as the residual threshold.
3. Repeat steps 2 & 3 for a suitable number of iterations depending on the size of the data set.
4. The iteration/model that produces the regression surface, which results in the most number of data points, lying within the specified residual threshold is then identified.
5. Only the specific data points that correspond to the model identified in step 4 are then used as the training data and the rest of the dataset is ignored. The Linear regression model that results from training on this subset of data is then used for prediction purposes.

HUBER REGRESSION:

Huber regression uses a modified loss function which advantageously combines the best features of squared error and absolute error. It uses a **loss threshold parameter** δ such that for all errors below this threshold half the squared error is used and for errors beyond it, a value proportional to the absolute error, as shown below:

$$L(y, y\_pred) = \frac{1}{n}\sum_{i} \begin{cases} \frac{1}{2}(y - y\_pred)_i^2 & \text{if } |y - y\_pred|_i <= \delta \\ \delta |y - y\_pred|_i - \frac{1}{2}\delta^2 & \text{Otherwise} \end{cases}$$

The plot below compares Squared Loss and Huber Loss. As can be seen, the squared error loss increases rapidly at a quadratic rate as the difference between the actual and predicted values increases. Thus a few large errors (outliers) can disproportionately affect the outcome of a regression.



Huber loss on the other hand, applies a variant of the absolute error (as shown in the equation above) for values of the residual (ie: y - y_pred) that are greater than the delta value (δ) specified, and half squared error for all other residual values thus resulting in a more gradual increase in loss as the values of the residual increases.

Note the difference in how RANSAC & Huber regression treat outlier data points. The RANSAC regressor ignores outliers while training on data, whereas the Huber regressor incorporates all data points during its training, but reduces the effect of outliers by suitably modifying the loss function.

REGRESSION EXAMPLE: BOSTON HOUSING PRICES

DATA DESCRIPTION:

The dataset contains 506 entries consisting of 13 features, which were collected by the U.S Census Service concerning housing in the area of Boston in 1978. The target variable (MEDV) in this dataset are the median values of owner-occupied homes in units of 1000 dollars. Given below is a description of the features and labels (target variable):

```
- CRIM     per capita crime rate by town
- ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
- INDUS    proportion of non-retail business acres per town
- CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
- NOX      nitric oxides concentration (parts per 10 million)
- RM       average number of rooms per dwelling
- AGE      proportion of owner-occupied units built prior to 1940
- DIS      weighted distances to five Boston employment centres
```

```
- RAD      index of accessibility to radial highways
- TAX      full-value property-tax rate per $10,000
- PTRATIO  pupil-teacher ratio by town
- LSTAT    % lower status of the population
- MEDV     Median value of owner-occupied homes in $1000's
```

```
1   from sklearn.datasets import load_boston
2
3   data = load_boston()
4
5   df = pd.DataFrame(data['data'])
6   df.columns = data['feature_names']
7   df_Xy_raw = df.assign(labels = data['target'])
8
9
10  df_Xy_raw.info()
```

```
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   CRIM     506 non-null    float64
 1   ZN       506 non-null    float64
 2   INDUS    506 non-null    float64
 3   CHAS     506 non-null    float64
 4   NOX      506 non-null    float64
 5   RM       506 non-null    float64
 6   AGE      506 non-null    float64
 7   DIS      506 non-null    float64
 8   RAD      506 non-null    float64
 9   TAX      506 non-null    float64
 10  PTRATIO  506 non-null    float64
 11  LSTAT    506 non-null    float64
 12  labels   506 non-null    float64
```

A) DATA PREPROCESSING & EDA:

As with the case of binary classification, helper functions have been created that automate most of the preprocessing-EDA and training-evaluation tasks for linear regression also. These functions perform similar tasks to the ones performed in the case of classification, except that they have been modified to suit the purposes of regression. The suffixes "**clf**" and "**reg**" are used to differentiate between classification & regression tasks respectively.

HELPER FUNCTIONS FOR DATA PREPROCESSING & EDA:

1. df_Xy, dictO_code2feats = fn_preprocess_data(df_Xy_raw, labels_col_name)
2. fn_label_distr_reg(y)
3. idxs_tr, idxs_ts = fn_train_test_split_reg(df_Xy, test_size = 0.2, std_thresh = 2)
4. good_feats = fn_feat_select_reg(df_tr_raw)
5. fn_scatter_feat_label(df_tr_, n_top_feats = 6)
6. fn_plot_3d_reg(df_tr_)

The fourth function **fn_feat_select_reg** computes the Spearman's coefficient between the features and the labels (since the labels is a continuous random variable) instead of anova f_ratio that was used during the classification tasks.

The fifth function **fn_scatter_feat_label** is used to create scatter plots of each important feature (derived using **fn_feat_select_reg**) and the labels.

The sixth function, **fn_plot_3d_reg** is used to create 3d plots between the top two feats and the labels. The plots outputted by the previous two functions give us insight into how easy it will be to fit a regression surface to the data.

STEP_1: Applying **fn_preprocess_data** we have:

```
labels_col_name = 'labels'

df_Xy, dict0_code2feats = fn_preprocess_data(df_Xy_raw, labels_col_name)
print(df_Xy.head(10))
```

```
>>> 1/3: CHECKING FOR ROWS WITH MISSING LABELS
*** NO LABELS MISSING
>>> 2/3: CHECKING FOR MISSING FEATURE VALUES
*** NO MISSING VALUES
>>> 3/3: RENAMING COLUMNS WITH GENERIC NAMES
```

	f0	f1	f2	f3	f4	...	f9	f10	f11	f12	labels
0	0.00632	18.0	2.31	0.0	0.538	...	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	...	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	...	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	...	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	...	222.0	18.7	396.90	5.33	36.2
5	0.02985	0.0	2.18	0.0	0.458	...	222.0	18.7	394.12	5.21	28.7
6	0.08829	12.5	7.87	0.0	0.524	...	311.0	15.2	395.60	12.43	22.9
7	0.14455	12.5	7.87	0.0	0.524	...	311.0	15.2	396.90	19.15	27.1
8	0.21124	12.5	7.87	0.0	0.524	...	311.0	15.2	386.63	29.93	16.5
9	0.17004	12.5	7.87	0.0	0.524	...	311.0	15.2	386.71	17.10	18.9

From the feedback provided by the function above we see that the data does not contain any missing labels or feature values.

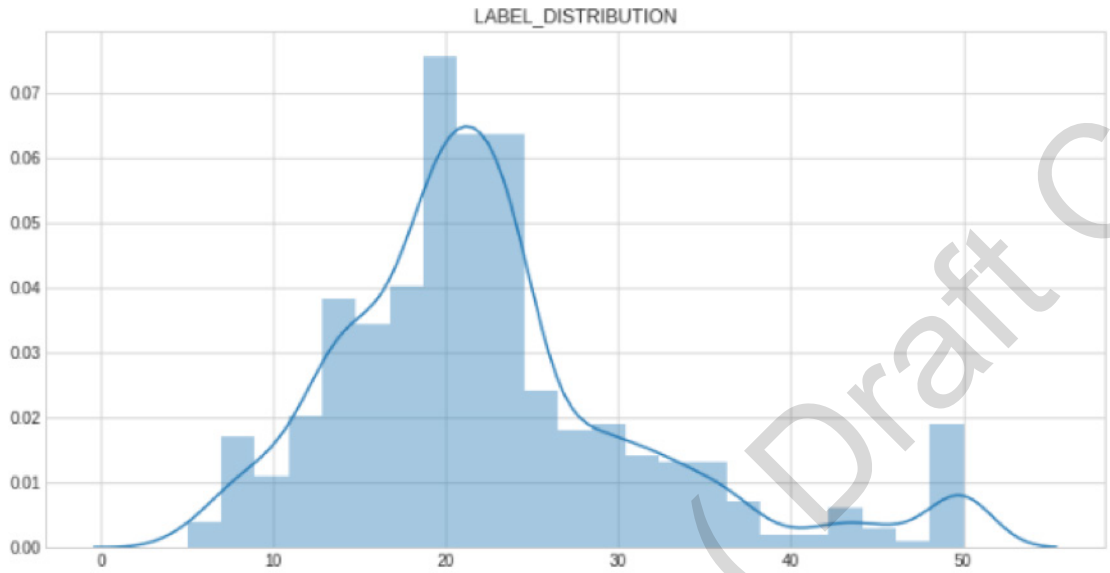
STEP_2: Applying **fn_train_test_split_reg** we have:

```
idxs_tr, idxs_ts = fn_train_test_split_reg(df_Xy, test_size = 0.2)

df_tr_raw = df_Xy.iloc[idxs_tr]
df_ts_raw = df_Xy.iloc[idxs_ts]

df_tr_raw.shape, df_ts_raw.shape
```

```
((404, 14), (102, 14))
```

STEP_3: Applying **fn_label_distr_reg** on train set labels we have:

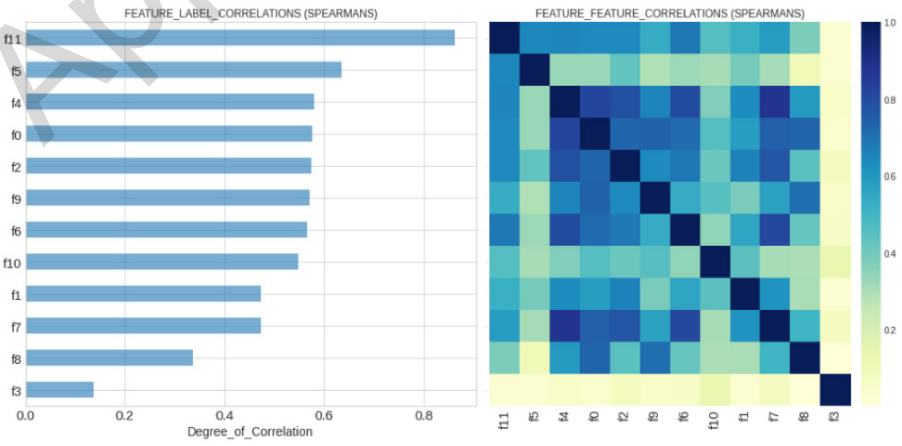
```
y_ = df_tr_raw.labels.values

fn_show_label_distribution(y_)
```



As can be observed from the distribution plot above, the values of the labels are mostly contained in the interval of [10, 30]. The distribution has a longer tail on the right side with Values extending upto 50. The bin containing labels whose values are equal to 50 show a sudden spike/irregularity with respect to the rest of the distribution. This is indicative of the values of apartments/houses being thresholded at a maximum value of 50 or they represent outliers.

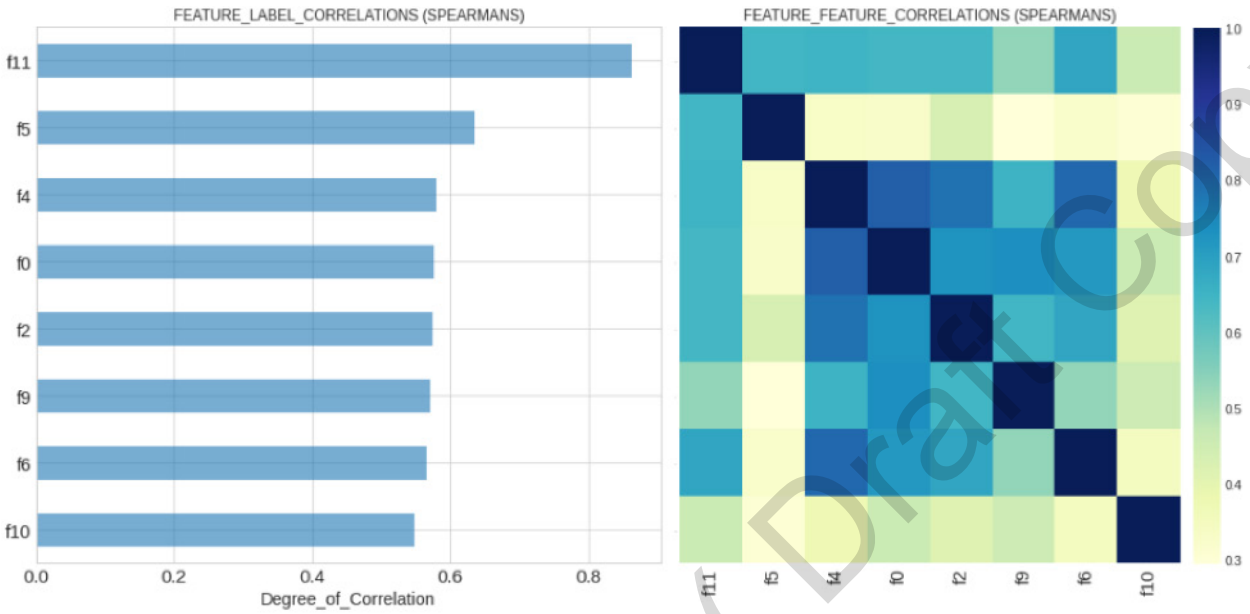
STEP_4: Applying **fn_feat_select_reg** we first sort the features in decreasing order of importance and study the bar plot and heatmap outputted by the function to decide upon suitable thresholds for feature selection.

```
good_feats = fn_feat_select_reg(df_tr_raw, plot = True)
```



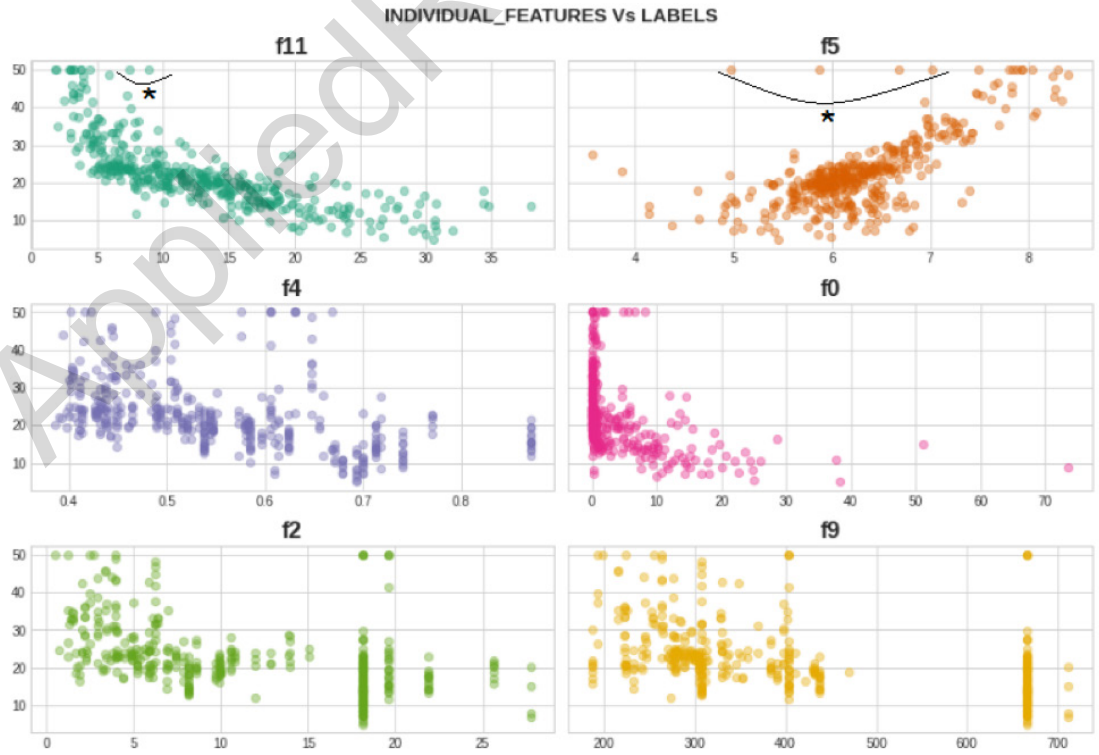
Using the thresholds shown below we arrive at the following set of useful features:

```
good_feats = fn_feat_select_reg(df_tr_raw, thresh_feat_label = 0.5, thresh_feat_feat = 0.7)
```



STEP_5: Applying **fn_scatter_feat_label** we have:

```
df_tr_raw = df_tr_raw[good_feats + ['labels']]
df_ts_raw = df_ts_raw[good_feats + ['labels']]

fn_scatter_feat_label(df_tr_raw, n_top_feats = 6)
```



Studying the scatter plots above, we see that the first 2 important features **f11** and **f5**, show a good amount of correlation with respect to the labels. Also, some data points for these features (indicated by the "**star**" symbols), whose labels are equal to 50 do not generally have the same correlation as the rest of the data, suggesting that they might be outliers. This is consistent with the observations made in **step_3** while analysing the label distribution of the train set.

The scatter plot of feature **f0** with its **labels** show an over concentration of data points at the lower spectrum of **f0** irrespective of the label values. Thus this feature does not provide an adequate correlation with respect to the labels (ie: price of apartment). On looking up what feature **f0** represents, we find that it represents crime rate, which explains the lack of correlation of this variable with respect to its labels. In other words, irrespective of the price of the apartments, most people prefer to live in areas with lower crime rates. In other words, most of the data collected are from places with lesser or no crime rate.

We drop the data points detected as outliers and also drop feature **f0** from the dataset as shown below:

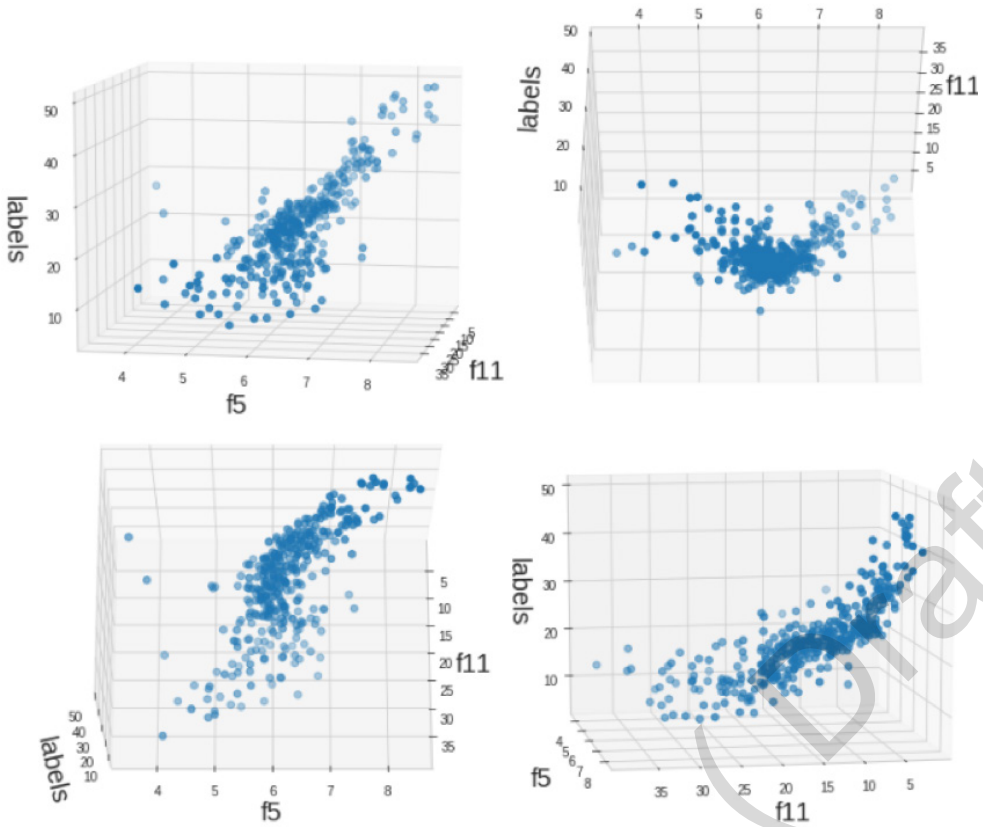
```
df_Xy = df_tr_raw

condition_1 = ((df_Xy.labels > 49) & (df_Xy.f11 > 5))
condition_2 = ((df_Xy.labels > 49) & (df_Xy.f5 < 7.5))

df1 = df_Xy[condition_1]
df2 = df_Xy[condition_2]
idxs = pd.concat([df1, df2]).drop_duplicates().index

df_tr = df_Xy.drop(idxs).drop('f0', axis = 1)
df_tr.shape, df_Xy.shape
```

```
((398, 8), (404, 9))
```

The same process is repeated with the test set. We thus arrive at our final train & test set which are ready for model training & testing.

STEP_6: Applying **fn_plot_3d_reg** we visualize the combined scatter plot of the top two import features with respect to their labels are different visual orientations:

```
fn_plot_3d_reg(df_tr)
```



The scatter plots above indicate that the data is quite suitable for the purposes of regression.

B) MODEL TRAINING & EVALUATION:

The procedure used for Model Training and Evaluation in the case of regression tasks are similar to those used in the case of classification; with a few key differences. We will be using the following two model classes from the Scikit Learn library within our specifically implemented helper functions :

1. **sklearn.linear_model.RANSACRegressor( )**: The ransac regressor model has one main hyperparameter that we need to tune for - the residual threshold. As discussed earlier, the Median Absolute Deviation (MAD) of the labels is generally used as the reference for the residual threshold value.

   While tuning for this parameter we feed the model various values relative to the label's MAD value. In our present example, we use the product of the MAD with twenty equispaced values between 0.2 and 2 as our range of values for the residual threshold.

1. **sklearn.linear_model.HuberRegressor( ):** The huber regressor model has two main hyperparameters to be tuned:
   - Epsilon: This represents the residual value (|y - y_pred|) below which squared loss is calculated and above which the absolute error.
   - Alpha: L2 regularization parameter.

HELPER FUNCTIONS FOR MODEL TRAINING & EVALUATION:

1. **df_kfoldcv, dictO_model_instances = fn_kfold_cv(df_tr, model_class, param_grid)**
   This is the same function that was used to perform K Fold Cross Validation for the classification task, but this time we perform it on the two regression models described above

2. **fn_performance_models_data_reg(df_tr_standardized, dictO_best_models)** This function visualizes the performance of the regression models provided within **dictO_best_models**, by plotting the CDF curves for the Absolute Error and Percentage Absolute Error and also provides information about the following performance metrics: MAE, MAPE and r2_score.

STEPS FOR MODEL TRAINING AND EVALUATION FOR REGRESSION TASK:

1. As in the case for classification, during the model training and evaluation stage for regression, we first train and evaluate the two regression models described above (RANSAC & Huber) for a range/combination of hyperparameters using **fn_kfold_cv**. The models are evaluated with respect to the three evaluation metrics described earlier: Mean Absolute Percentage Error, r2 score and Mean Absolute Error, in that order of importance.
2. We then filter out the best models among these, by choosing the models that display the most consistency (generalization) and best performance across all the K folds of the train-evaluation set.
3. We then use **fn_performance_models_data_reg** to compare the selected best models, by analysing the CDF plots of the MAE and MAPE for all these models as well as comparing their r2 scores. The best model for the task is then selected based on the information thus gleaned.
4. The best model is then used to predict on the test set to check if the chosen model's performance generalizes to new unseen data.

STEP_7: (Continued from Data Processing and EDA stage):

Shown below, the RANSAC regression model is trained and cross validated and then the best models are filtered out using suitable conditional statements:

```
from sklearn.linear_model import RANSACRegressor
from scipy.stats import median_absolute_deviation

model_class = RANSACRegressor
y_tr = df_tr.iloc[:, -1].values.ravel()
list0_thresh =  np.linspace(0.5, 2, 20).round(2) * median_absolute_deviation(y_tr)

parameter_grid = dict(residual_threshold = list0_thresh,
                      max_trials = [1000])
prefix = 'ransac_'

z = fn_kfoldcv(df_tr, model_class, parameter_grid, model_name_prefix = prefix)
df_kfoldcv_ransac, dict0_model_instances_ransac = z
df_kfoldcv_ransac.describe()
```

	mean_mae	std_mae	mean_mape	std_mape	mean_r2	std_r2
count	20.000000	20.000000	20.000000	20.000000	20.000000	20.000000
mean	3.601017	0.593003	0.575717	0.522944	0.594267	0.152362
std	0.251108	0.258738	0.924090	1.299651	0.079425	0.078207
min	3.122333	0.110330	0.199333	0.021453	0.374667	0.037774
25%	3.421500	0.451205	0.270333	0.101774	0.544833	0.102937
50%	3.621000	0.584829	0.338333	0.180280	0.600667	0.146055
75%	3.724917	0.713229	0.413667	0.296397	0.652000	0.190241
max	4.221333	1.221927	4.387667	5.941392	0.702667	0.355678

Filtering out the best RANSAC models:

```
df = df_kfoldcv_ransac

df1 = df[df.std_mae < 0.4][:3]
df2 = df1[df1.mean_mape < 0.4][:3]

df_filtered_ransac = df2.drop_duplicates()[:2]
df_filtered_ransac
```

	mean_mae	std_mae	mean_mape	std_mape	mean_r2	std_r2
ransac_model_19	3.326000	0.267447	0.281333	0.067539	0.649000	0.061682
ransac_model_8	3.341333	0.385781	0.376667	0.249202	0.690667	0.071098

STEP_7: is then repeated on the Huber regression model and the best models filtered out just as shown earlier:

```
from sklearn.linear_model import RANSACRegressor
from scipy.stats import median_absolute_deviation

model_class = RANSACRegressor
y_tr = df_tr.iloc[:, -1].values.ravel()
listO_thresh =  np.linspace(0.5, 2, 20).round(2) * median_absolute_deviation(y_tr)

parameter_grid = dict(residual_threshold = listO_thresh,
                      max_trials = [1000])
prefix = 'ransac_'

z = fn_kfoldcv(df_tr, model_class, parameter_grid, model_name_prefix = prefix)
df_kfoldcv_ransac, dictO_model_instances_ransac = z
df_kfoldcv_ransac.describe()
```

	mean_mae	std_mae	mean_mape	std_mape	mean_r2	std_r2
count	20.000000	20.000000	20.000000	20.000000	20.000000	20.000000
mean	3.601017	0.593003	0.575717	0.522944	0.594267	0.152362
std	0.251108	0.258738	0.924090	1.299651	0.079425	0.078207
min	3.122333	0.110330	0.199333	0.021453	0.374667	0.037774
25%	3.421500	0.451205	0.270333	0.101774	0.544833	0.102937
50%	3.621000	0.584829	0.338333	0.180280	0.600667	0.146055
75%	3.724917	0.713229	0.413667	0.296397	0.652000	0.190241
max	4.221333	1.221927	4.387667	5.941392	0.702667	0.355678

Filtering out the best Huber models:

```
df = df_kfoldcv_ransac

df1 = df[df.std_mae < 0.4][:3]
df2 = df1[df1.mean_mape < 0.4][:3]

df_filtered_ransac = df2.drop_duplicates()[:2]
df_filtered_ransac
```

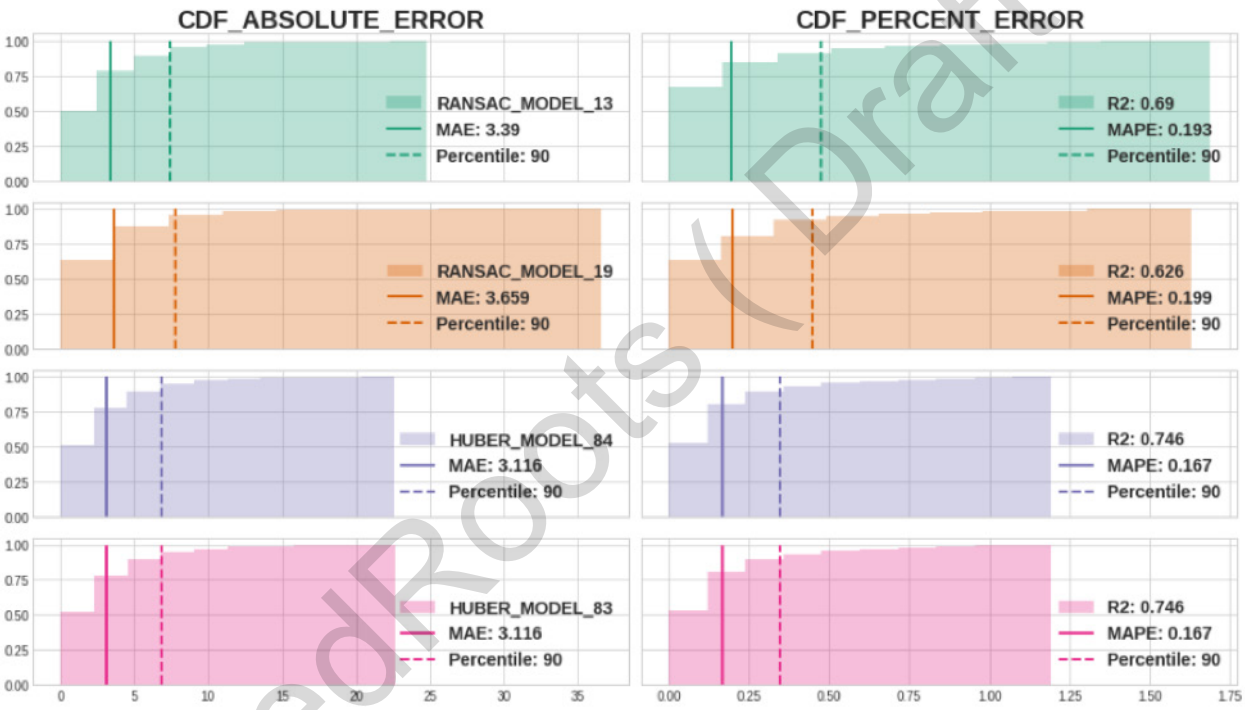
	mean_mae	std_mae	mean_mape	std_mape	mean_r2	std_r2
ransac_model_19	3.326000	0.267447	0.281333	0.067539	0.649000	0.061682
ransac_model_8	3.341333	0.385781	0.376667	0.249202	0.690667	0.071098

STEP_8: VISUALIZING THE PERFORMANCES OF THE BEST MODELS:

We first insert the best models (RANSAC & Huber) into a single dictionary and then fed to **fn_performance_models_data_reg** where they are retrained on the full train set and their prediction performances then evaluated on the same:

```
df_tr_standardized, df_ts_standardized = fn_standardize_df(df_tr, df_ts)

d1 = {k:dict0_model_instances_ransac[k] for k in df_filtered_ransac.index}
d2 = {k:dict0_model_instances_huber[k] for k in df_filtered_huber.index}
dict0_best_models = {**d1, **d2}
```

```
fn_perform_models_data_reg(df_tr_standardized, dict0_best_models,  percentiles = [90])
```



From the above visualization we see that the Huber models perform better than the RANSAC models. Both the Huber models shown above have similar performances and so we randomly choose HUBER_MODEL_84 as our final model. We then test the chosen model on the test set as shown below:

```
df = df_tr_standardized
X_tr, y_tr = df.iloc[:, :-1].values, df.iloc[:, -1].values

best_model = dict0_best_models['huber_model_177'].fit(X_tr, y_tr)

fn_test_model_reg(df_ts_standardized, best_model)
```

	MAE	MAPE	R2
Performance:	3.276	0.153	0.675

From the results above we see that the chosen model generalizes to the test set quite well.

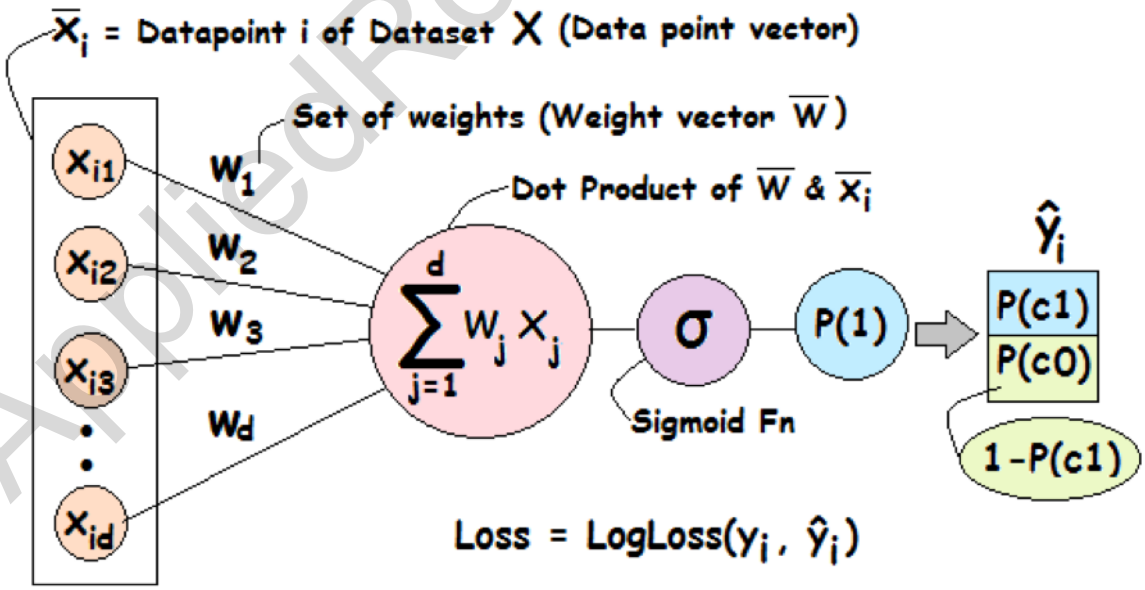
9. MULTICLASS CLASSIFICATION (LOGISTIC REGESSION)

MULTICLASS CLASSIFICATION (LOGISTIC REGESSION)

LOGISTIC REGRESSION BINARY CLASSIFICATION REVISITED:

Binary classification using Logistic regression involves finding optimal weight vector **W** that minimizes the log loss for the entire dataset **X**. The prediction mechanism can be represented as shown in the image below. Given a weight vector w and a data point **x_i**, the logistic regression model predicts by:

1. Computing the dot product between the **w** and **x_i** (we are ignoring the bias term here for the sake of simplicity).
2. The Dot product value is then passed through a sigmoid function σ to get a value that lies in the interval [0, 1]. This value represents the probability of the datapoint **x_i** belonging to **class 1** and since it is a binary classification task, one minus this value gives us the probability of **x_i** belonging to **class 0**.
3. The prediction will be in the form of a 2D vector $\hat{y}\_i$ composed of the two values described in the previous step.
4. The log loss for the prediction is then computed by taking the negative log of the probability predicted by the model for the correct class as indicated by its corresponding label y_i.
5. The log loss with respect to the entire data set is the mean of the log losses for all the data points. This is called the log loss performance of the model.



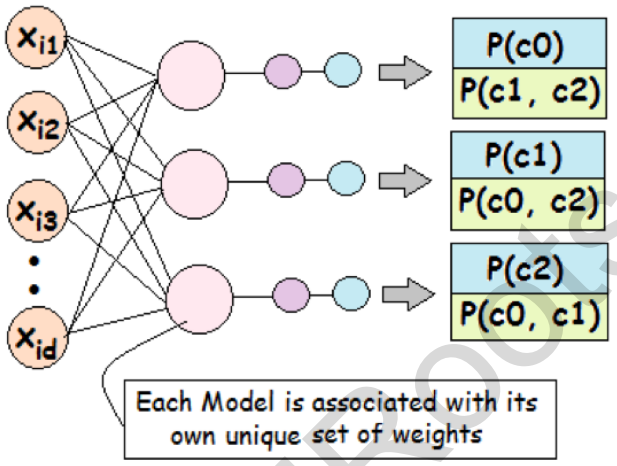
LOGISTIC REGRESSION - MULTICLASS CLASSIFICATION:

To adapt the Logistic Regression algorithm for multiclass classification tasks (more than 2 classes), one of the following two strategies are followed:

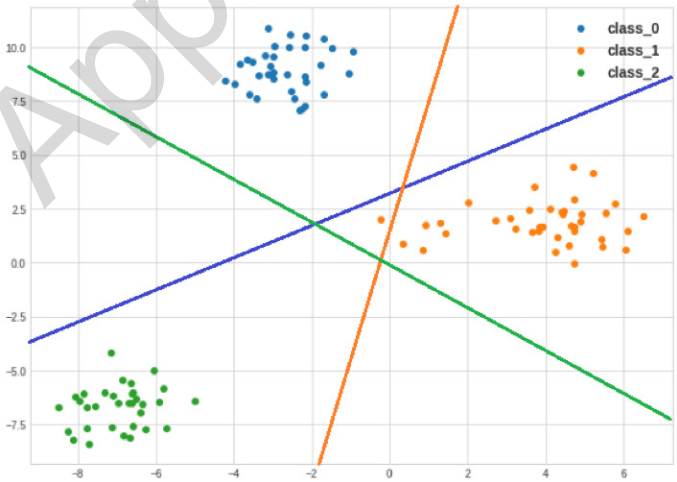
1. One Vs rest Strategy
2. Using Cross Entropy Loss

ONE Vs REST:

Here a unique Logistic Regression model is initiated for every class present. Each model is then assigned one unique class as main class and the combination of the remaining classes as the other class. In other words, there will be k binary classification models initiated for k classes, each performing binary classification with respect to one unique class and the combination of the remaining classes. The image below describes 3 class classification using this technique:



Each of the k models is optimized for its own unique weight vector by minimizing the log loss associated with it as much as possible. This will result in k separating boundaries. The image below shows the separating boundaries in the case of a 3 class classification problem for a 2D dataset.



In case of data points that fall in the space between the decision boundaries, the data point is assigned to the class of the separating boundary nearest to it.

Note that in one Vs rest model, the labels are modified such that there is one set of 2D label vectors for each model (ie: There will be K sets of label vectors for K classes). Each set uses value one to represent the particular class that the individual model is assigned to identify and value zero for the rest of the classes.

MULTICLASS CLASSIFICATION USING CROSS ENTROPY LOSS:

Cross entropy loss at its core, is a function that measures the "distance" between two probability distributions. To understand cross entropy we need to understand the following concepts:

1. Expected Value
2. Entropy
3. Likelihood
4. Likelihood ratio
5. KL Divergence

EXPECTED VALUE OF A RANDOM VARIABLE:

As discussed earlier in the chapter on probability and statistics, a random variable could be defined as the record of the output values of a "process" for a predefined number of observations. Some examples of random variable are:

1. Outcomes of a coin toss: Here the "process" is the coin tossing mechanism. The system has two unique "states" or outcomes - Heads & Tails. Say we record the outcome of 100 coin tosses, then this record is a random variable of size 100, that gives us information describing the system that produces these outcomes (ie: the coin tossing mechanism)
2. Say we have a box containing many balls, which have one of the following 4 values painted on them: 1, 2, 3 and 4. Then the outcome of randomly picking 100 balls from the box and recording the values painted on each ball is a random variable of size 100.

Consider the second example, say we repeat the process described using 3 different boxes, each having different ratios of the painted values on the balls. The output of such a process for box_1 could be :

Random_variable_box_1 = **[**1, 1, 4, 3, 2, 3, 2, 3, 4, 3..... (100 such recorded values)**]**

Say the final result of this process is as shown below:

	1	2	3	4
RV_box_1	25	25	25	25
RV_box_2	20	70	7	3
RV_box_3	5	3	2	90

The expected value of any Random Variable "X" is expressed as:

$$E(X) = \sum_i p(x_i)\, x_i$$

where:
$p(x_i)$ is the probability of the "ith" category/state of random variable X

Thus, the expected value for the random variable RV_box_2 will be :

(1 x 0.2) + (2 x 0.7) + (3 x 0.07) + (4 x 0.03) = 1.93

In other words, expected value is the mean or average value of the random variable.

ENTROPY:

Entropy is a basic quantity associated with any random variable, it can be interpreted as the average level of uncertainty inherent in the random variable's possible outcomes. It is mathematically defined as:

$$H(X) = -\sum_i p(x_i)\, \log p(x_i)$$

With reference to the example of the balls described earlier, we can intuitively say that the outcome of **RV_box_1** is more unpredictable than that of **RV_box_2** or **RV_box_3**, since it can take on any of the four possible output states (ie: values) with equal probability. Similarly, **RV_box_2** contains more uncertainty than **RV_box_3.** This can be verified by applying the above formula to each of these random variables.

H(RV_box_1) = - (0.25 x log(0.25) + 0.25 x log(0.25) + 0.25 x log(0.25) + 0.25 x log(0.25)) = 0.602

H(RV_box_2) = - (0.20 x log(0.20) + 0.70 x log(0.70) + 0.07 x log(0.07) + 0.03 x log(0.03)) = 0.37

H(RV_box_3) = - (0.05 x log(0.05) + 0.03 x log(0.03) + 0.02 x log(0.02) + 0.9 x log(0.9)) = 0.18

The entropy for **RV_box_3** is the least, indicating least uncertainty.

LIKELIHOOD:

Every Random Variable is associated with a probability distribution that describes the probabilities for each value belonging to the random variable. In the example described above we have three random variables that are associated with the same four values (ie: 1, 2, 3 & 4).

Now consider the scenario where we have a particular observation/outcome but we do not know the origin of this outcome. In other words, we do not know from which random variable, among the four described above, this particular observation belongs to. In such a scenario, we can estimate which random variable the outcome belongs to by using the concept of Likelihood. Higher the Likelihood associated with a random variable, greater the chances that the observation belongs to that particular random variable.

Say we have an outcome "2" (ie: the value got by picking one ball from one of the 3 boxes is 2), then:

1. The likelihood that the ball was picked from box_1 is 0.25 or 25%
2. The likelihood that the ball was picked from box_2 is 0.70 or 70%
3. The likelihood that the ball was picked from box_3 is 0.3 or 3%

In other words the likelihood of an outcome or value belonging to a particular distribution is the probability of observing that value in that distribution. Note the subtle difference between the concepts of probability and likelihood:

- Probability is a more fundamental concept than Likelihood. It concerns itself with a single probability distribution. It informs us about the chances of observing a particular outcome, given a single stochastic process (or random variable). The sum of all probabilities in a distribution always adds up to one.
- Likelihood is based on probability and is concerned with comparison between multiple probability distributions given a particular outcome. It tells us which probability distribution the outcome is most likely to belong to. Given a set of probability distributions, the likelihood values for a particular outcome need not add up to one.

LIKELIHOOD RATIO:

Likelihood ratio is a value used in relation to two probability distributions. Consider the following: Say we have 2 probability distributions **p** and **q**, with the same set of outcomes. In other words we have 2 stochastic processes which have the same set of values, but different probability distributions (like the balls in the 3 boxes). Given a particular outcome say **x1**, the likelihood ratio of **x1** with respect to **p** and **q** is defined as:

$$LR = \frac{P(x1|p)}{P(x1|q)}$$

- The Likelihood ratio expresses how much more likely it is that the observation **x1** has come from distribution p than distribution **q**.
- If the likelihood ratio is lesser than one then, the observation **x1** is most likely to have come from distribution **q**.
- If the likelihood ratio is equal to one then, the observation **x1** is equally likely to have come from distributions **p** or **q**.

The concept of Likelihood ratio can be further extended for a set of outcomes "i" as shown below:

$$LR = \prod_i \frac{P(x_i|p)}{P(x_i|q)} = \sum_i \log \frac{P(x_i|p)}{P(x_i|q)} \quad \text{where } i = 1 \text{ to } n$$

KL (KULLBACK - LIEBNER) DIVERGENCE:

KL Divergence between probability distributions p & q is mathematical expressed as:

$$D_{KL}(p||q) = \sum_i P(x_i|p) \log \frac{P(x_i|p)}{P(x_i|q)}$$

It is basically the **expected value of the likelihood ratio** – where the likelihood ratio expresses how much more likely the sampled data is from distribution p instead of distribution q. Note that KL Divergence is not a symmetric function and can thus yield a different value for D_KL(q || p). It can be further simplified as:

$$D_{KL}(p||q) = \sum_i P(x_i|p) \log P(x_i|p) - \sum_i P(x_i|p) \log P(x_i|q)$$

CROSS ENTROPY:

The cross entropy loss between distributions **p** and **q** is mathematically expressed as:

$$H(p, q) = H(p) + D_{KL}(p||q)$$

By substituting the expressions for entropy of **p** and KL divergence in the above equation we have:

$$H(p, q) = - \sum_i P(x_i|p) \log P(x_i|q)$$

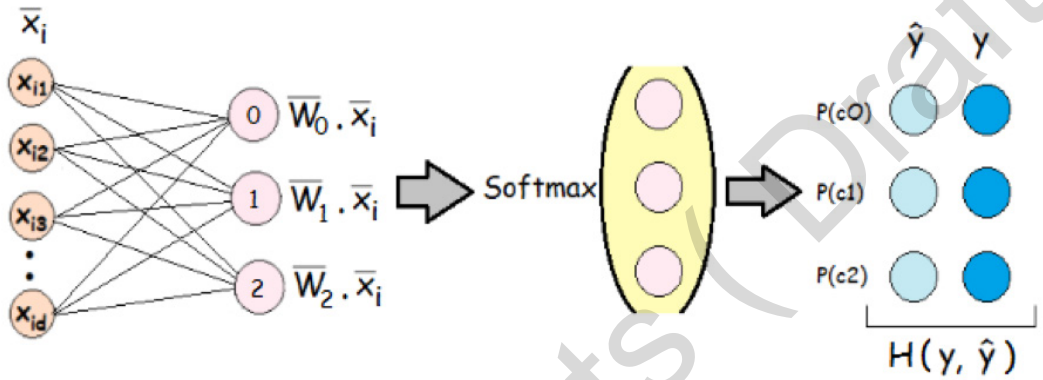
Here **p** is the reference distribution with which distribution **q** is being compared. The lesser the value of **H(p, q)**, the more similar distribution **q** is to distribution **p**. Note that **H(p, q)** is not equal to **H(q, p)** (ie: Cross entropy is not a symmetric function).

With reference to machine learning, the actual label vector y is distribution p and the predicted label vector ŷ is distribution q. Hence in terms of machine learning parlance we have:

$$H(y, \hat{y}) = -\sum_{i} P(y_i) \log P(\hat{y}_i) \quad \text{where i = number of classes}$$

MULTI CLASS LOGISTIC REGRESSION USING CROSS ENTROPY LOSS:

Multiclass logistic regression using cross entropy loss, is similar to the one Vs rest method, in that it too uses K models to classify K classes. The difference lies only in the fact that it does not pass the outputs of each of these classifiers through a Sigmoid function, but instead it considers the combined outputs of these K classifiers as a single vector and passes this vector through a softmax function. The image below describes 3 class classification using this technique:



The **softmax function** is a **function** that turns a vector of K real values into a vector of K real values that sum to 1. This is very useful because it converts the scores outputted by the K logistic regression models, to a normalized probability distribution.

The softmax function can be expressed in the form of a python list comprehension as:

$$\text{softmax}(z) = \left[\frac{e^{z_j}}{\sum_{i=1}^{k} e^{z_i}} \text{ for } j \text{ in } z \right] \quad \text{where z is a K dimensional vector}$$

Once the output scores of the K **individual** logistic regression models are passed through a softmax function, the resulting probability distribution represents the **composite** model's confidence (or uncertainty) about the class of the data point **x_i** fed to it. The cross entropy loss of this resulting probability distribution (predicted vector) with respect to the corresponding one hot encoded label vector, tells us how similar the predicted vector **($\hat{y}$)** is with respect to the label vector**(y)**.

In this form of classification, the weights of the individual logistic regression models are optimized to minimize the cross entropy loss between the predicted vector and the actual label vector. Note that, the resulting weights of this kind of model need not represent any separating boundary between the classes/labels.

The cross entropy Logistic Regression model is fundamental to understanding the structure of a neuron in neural network based machine learning models, which we will discuss in brief further along in this chapter.

MULTICLASS PERFORMANCE METRICS:

For accessing the performance of multi class models, we use the same performance metrics - precision and recall, but modified suitably such that they give us a sense of the overall performance of the model with respect to all the classes. The two main types of performance metrics are 'macro' performance and 'micro' performances.

MACRO PERFORMANCE METRICS:

The macro performance of the model refers to the mean performance of the model over all the classes. This could be applied to precision (macro precision score), recall (macro recall score) or any other performance metric like F1 score.

Macro performance measures have a tendency to be influenced by the performance of the model on the larger sized classes in the dataset. In other words, this metric will indicate good performance if the performance of the model on larger classes is good irrespective of its performance on the smaller sized classes and vice versa.

MICRO PERFORMANCE METRICS:

Micro performance metric uses the combined True Positive, True Negative, False Positive and False negative counts of all the classes to compute the overall Precision and Recall measures of the model. This performance metric is more resilient to the effect of imbalance in data.

MULTICLASS LOGISTIC REGRESSION EXAMPLE (PENGUINS SPECIES CLASSIFICATION):

DATA DESCRIPTION:

The dataset contains 344 entries about the Palmer Archipelago penguin found in Antarctica. It contains values of the following four features:

1. Culmen (beak) length.
2. Culmen depth.
3. Flipper length.
4. Body mass index.

Along with the corresponding subspecies of the penguin.That is: Adelie, Gentoo and Chinstrap.

```
data_path = 'drive/My Drive/1A_BOOK/CODE_5/DATA/'

df_original = pd.read_csv(data_path + 'PENGUINS.csv')
df_original.info()
```

```
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   culmen_length_mm   342 non-null    float64
 1   culmen_depth_mm    342 non-null    float64
 2   flipper_length_mm  342 non-null    float64
 3   body_mass_g        342 non-null    float64
 4   species            344 non-null    object
dtypes: float64(4), object(1)
memory usage: 13.6+ KB
```

DATA PREPROCESSING AND EDA:

STEP_1: Applying **fn_preprocess** we have:

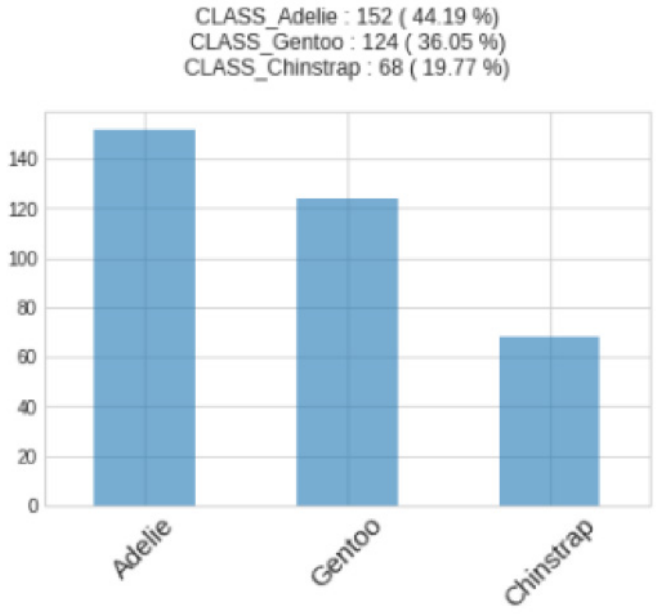
```
labels_col_name = 'species'

df_Xy, dict0_code2feats = fn_preprocess(df_original, labels_col_name)
print(df_Xy.sample(10))
```

```
>>> 1/3: CHECKING FOR ROWS WITH MISSING LABELS
*** NO LABELS MISSING
>>> 2/3: CHECKING FOR MISSING FEATURE VALUES
*** 8 FEATURE VALUES MISSING! REPLACING WITH MEAN OF FEATURE
>>> 3/3: RENAMING COLUMNS WITH GENERIC NAMES

       f0    f1     f2      f3     labels
284  45.8  14.2  219.0  4700.0     Gentoo
219  50.2  18.7  198.0  3775.0  Chinstrap
182  40.9  16.6  187.0  3200.0  Chinstrap
11   37.8  17.3  180.0  3700.0     Adelie
174  43.2  16.6  187.0  2900.0  Chinstrap
142  32.1  15.5  188.0  3050.0     Adelie
131  43.1  19.2  197.0  3500.0     Adelie
317  46.9  14.6  222.0  4875.0     Gentoo
154  51.3  19.2  193.0  3650.0  Chinstrap
187  47.5  16.8  199.0  3900.0  Chinstrap
```

As can be seen from the feedback that the function provides, the dataset has some missing value, which it replaces with the mean value of the feature. It then renames the columns with generic feature names and returns data frame df_Xy.

STEP_2: Applying **fn_show_label_distribution** we have:

```
y = df_Xy.labels.values

fn_label_distr(y)
```



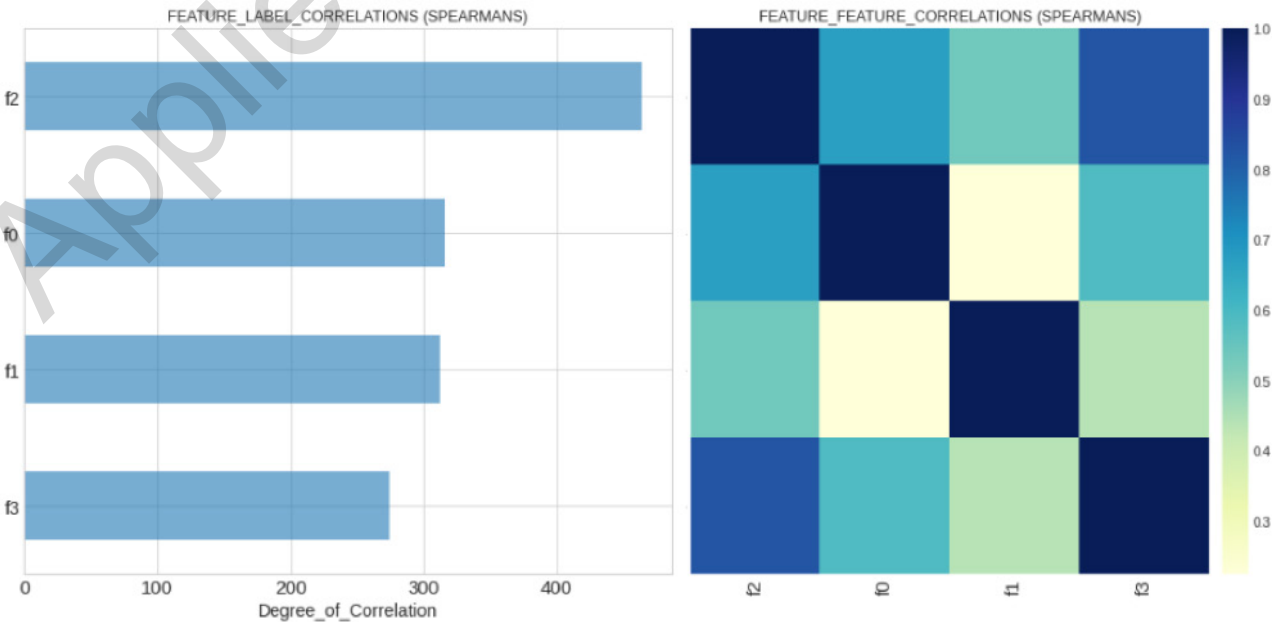
STEP_3: Applying **fn_tr_ts_split** we have:

```
idxs_tr, idxs_ts = fn_tr_ts_split(df_Xy, ts_size = 0.2)

df_tr_raw = df_Xy.iloc[idxs_tr]
df_ts_raw = df_Xy.iloc[idxs_ts]

df_tr_raw.shape, df_ts_raw.shape
```

```
((275, 5), (69, 5))
```

STEP_4: Applying **fn_feat_select_clf** we have:

```
list0_good_feats = fn_feat_select_clf(df_tr_raw, plot = True, figsize = (15, 7))
```



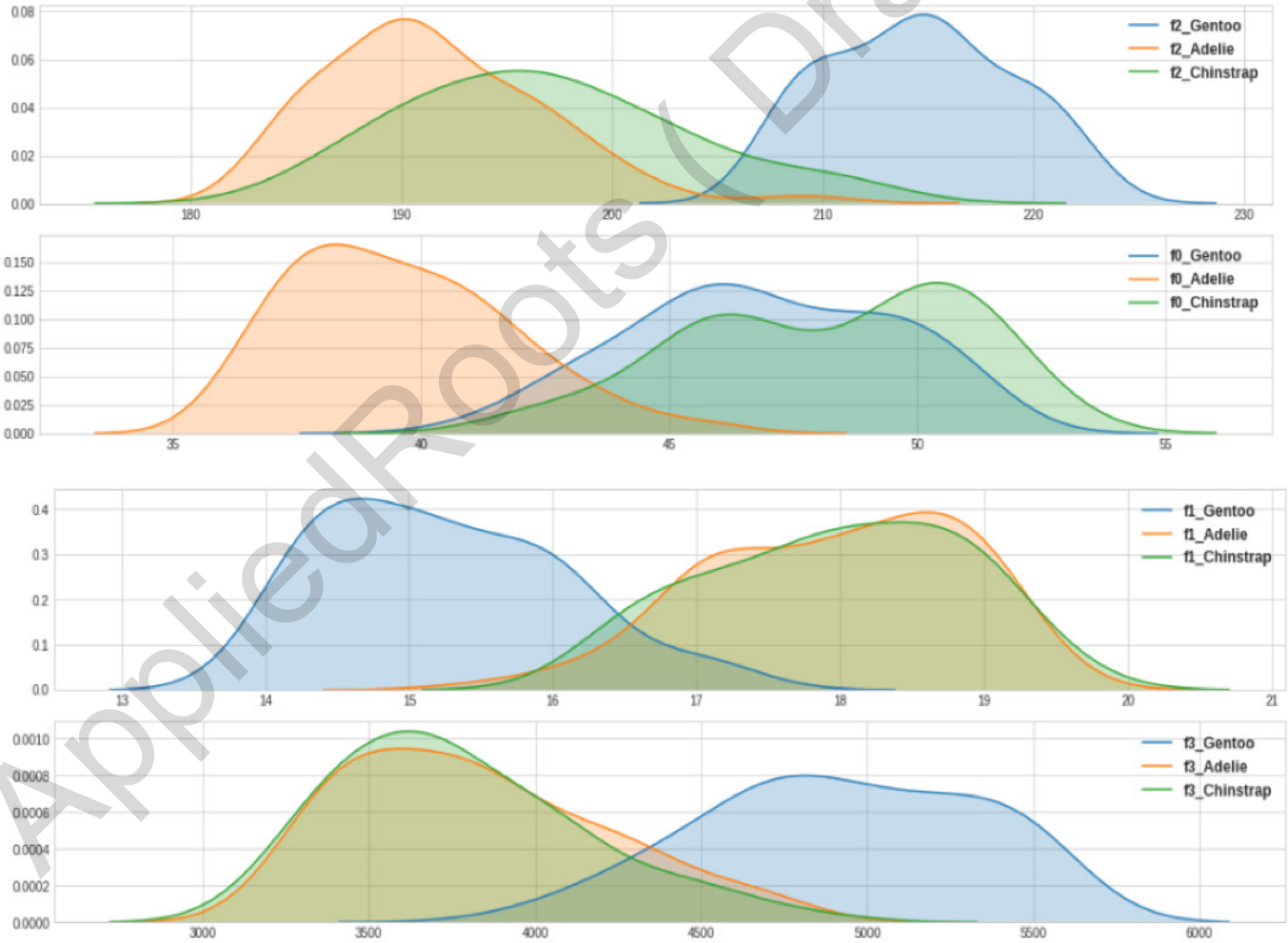
We thus arrive at the sequence of features arranged with respect to their importance. We rearrange the train and test sets in terms of the feature importance:

```
df_tr = df_tr_raw[list0_good_feats + ['labels']]
df_tr.index = range(len(df_tr))

df_ts = df_ts_raw[list0_good_feats + ['labels']]
df_ts.index = range(len(df_ts))

df_tr.shape, df_ts.shape
```

```
((275, 5), (69, 5))
```

STEP_6: Applying **fn_label_distr_feats** we have:

```
fn_label_distr_feats(df_tr)
```



We see that at least one of the three classes are separable with respect to the individual features.

STEP_7: Applying **fn_LOF_outlier_idxs** on all classes we have:

```
df_Xy_ = df_tr[df_tr.labels == 'Adelie']
n_clusters = 1
n_neighbors = 6

listO_outlier_idxs_Adelie = fn_LOF_outlier_idxs(df_Xy_, n_clusters, n_neighbors)
```

```
100% (1 of 1) |#########################| Elapsed Time: 0:00:00 Time:  0:00:00
```

```
df_Xy_ = df_tr[df_tr.labels == 'Gentoo']
n_clusters = 1
n_neighbors = 6

listO_outlier_idxs_Gentoo = fn_LOF_outlier_idxs(df_Xy_, n_clusters, n_neighbors)
```

```
100% (1 of 1) |#########################| Elapsed Time: 0:00:00 Time:  0:00:00
```

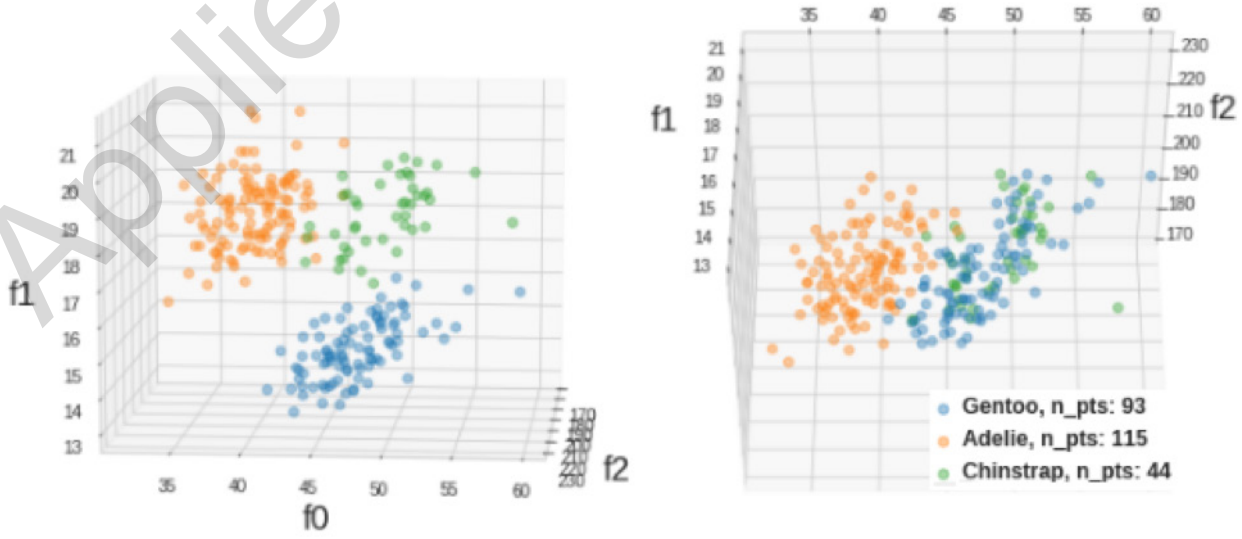
```
df_Xy_ = df_tr[df_tr.labels == 'Chinstrap']
n_clusters = 1
n_neighbors = 6

listO_outlier_idxs_Chinstrap = fn_LOF_outlier_idxs(df_Xy_, n_clusters, n_neighbors)
```

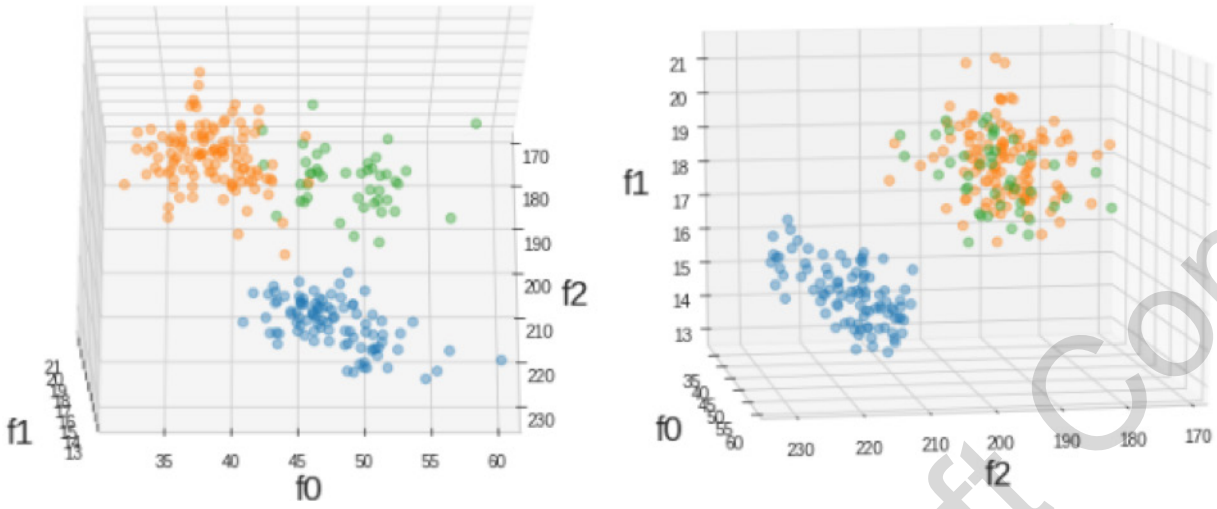
```
100% (1 of 1) |#########################| Elapsed Time: 0:00:00 Time:  0:00:00
```

STEP_8: Applying **fn_plot_3d_binary** on train data with outliers removed we have:

```
listO_outliers = listO_outlier_idxs_Adelie + listO_outlier_idxs_Gentoo + listO_outlier_idxs_Chinstrap

print(len(listO_outliers))
fn_plot_3d_binary(df_tr.drop(listO_outliers))
```

```
23
```





MODEL TRAINING AND EVALUATION:

In the case of multi class classification we use fn_koldcv_multi_clf which is similar to the K fold cross validation function used for binary classification, except that it is modified to compute the micro/macro precision/recall values described previously.

STEP_9: Applying **fn_koldcv_multi_clf** we have:

```
from sklearn.linear_model import LogisticRegression

df_tr_ = df_tr
model_class = LogisticRegression

parameter_grid = dict(penalty = ['l2', 'l1', 'elasticnet'],
                      C = [10**i for i in range(-15, 15)],
                multi_class = ['ovr', 'multinomial'],
                     solver = ['lbfgs', 'newton-cg', 'saga'],
               class_weight = ['balanced'],
                   max_iter = [10_000])

z = fn_kfoldcv_multi_clf(df_tr_, model_class, parameter_grid,  n_folds = 3, average = 'micro')

df_kfoldcv, dict0_model_instances, list0_invalid_models = z
df_kfoldcv.describe()
```

```
100% (540 of 540) |######################| Elapsed Time: 0:00:32 Time:  0:00:32

SOME MODELS INVALID - CHECK list0_invalid_models
```

	mean_ts_rec	std_ts_rec	mean_ts_prec	std_ts_prec	mean_ts_loss	std_ts_loss
count	240.000000	240.000000	240.000000	240.000000	240.000000	2.400000e+02
mean	84.594316	2.876890	84.594316	2.876890	0.589932	5.290714e-02
std	24.827679	4.812775	24.827679	4.812775	0.461082	9.297807e-02
min	25.071667	0.011261	25.071667	0.011261	0.042618	4.080227e-15
25%	95.273929	0.518119	95.273929	0.518119	0.105095	4.058963e-05
50%	96.735149	0.887568	96.735149	0.887568	0.358451	1.403004e-02
75%	97.818124	1.531558	97.818124	1.531558	1.098611	6.943504e-02
max	99.271381	31.137586	99.271381	31.137586	1.130523	3.559255e-01

We then filter out the best performing models as shown below:

```
df = df_kfoldcv

df1 = df[df.mean_ts_loss < 0.06]
df2 = df1[df1.mean_ts_rec > 97]

df_filtered_ransac = df2.drop_duplicates().sort_values(by = 'std_ts_rec', ascending = False)[:3]
df_filtered_ransac
```

	mean_ts_rec	std_ts_rec	mean_ts_prec	std_ts_prec	mean_ts_loss	std_ts_loss
model_99	98.542762	1.030469	98.542762	1.030469	0.042626	0.005255
model_100	98.542762	1.030469	98.542762	1.030469	0.042626	0.005254
model_101	98.542762	1.030469	98.542762	1.030469	0.042618	0.005264

STEP_10: We choose '**model_99**' as the best model and then train the same on the entire train set and check it performance as shown below:

```
df_tr_standardized, df_ts_standardized = fn_standardize_df(df_tr, df_ts)
```

```
df_Xy_ = df_tr_standardized
X_tr, y_tr = df_Xy_.iloc[:, :-1].values, df_Xy_.iloc[:, -1].values

model_ = dict0_model_instances['model_99'].fit(X_tr, y_tr)

fn_test_model_multi_clf(df_Xy_, model_)
```

```
====================
ACCURACY: 99.636
LOGLOSS: 0.02
====================
```

	precision	recall
class_Adelie	100.000	99.180
class_Chinstrap	98.182	100.000
class_Gentoo	100.000	100.000
MICRO	99.636	99.636
MACRO	99.394	99.727

STEP_11: We then test the chosen model on the test set to check for performance generalization. As shown below:

```
df_Xy_ = df_ts_standardized
X_ts, y_ts = df_Xy_.iloc[:, :-1].values, df_Xy_.iloc[:, -1].values

fn_test_model_multi_clf(df_Xy_, model_)
```

```
====================
ACCURACY: 98.551
LOGLOSS: 0.057
====================
```

	precision	recall
class_Adelie	96.774	100.000
class_Chinstrap	100.000	100.000
class_Gentoo	100.000	96.000
MICRO	98.551	98.551
MACRO	98.925	98.667

We see that the chosen model generalizes quite well to the test set.

LOG LOSS AND CROSS ENTROPY LOSS – A BRIEF DISCUSSION:

The Log loss is primarily a loss function derived specifically for the binary classification task. It leverages the fact that there are only two classes, and by specifically representing them using the integers **0 & 1** (and not 1 & 2 or any other pair of numbers), we can arrive at the formulation shown below. It computes the negative log of the probability predicted for the correct class. The average of this value computed for the entire data set, gives us the log loss performance of the model we are evaluating.

$$\text{Logloss for } \overline{x}_i = \underbrace{-y_i \log P(y_i = 1 \mid \overline{x}_i)}_{A} \underbrace{- (1 - y_i) \log P(y_i = 0 \mid \overline{x}_i)}_{B}$$

Only A gets "activated" when $y_i = 1$
Only B gets "activated" when $y_i = 0$

For log loss to be applicable, the primary probability distribution **y** (with respect to which the predicted probability distribution **ŷ** is being compared), has to be a vector that contains only binary values (ie: 0 or 1).

Cross Entropy loss on the other hand, could be considered as a more general loss function which could be applied across any two probability distributions. It does not require that primary probability distribution to contain only binary values. If cross entropy is applied in cases where the primary probability distribution contains only binary values, then the resulting cross entropy loss is the same as the log loss. So it could be said that log loss is a particular case (or subclass) of cross entropy loss.

THE ML PIPELINE SUMMARY:

In general our machine learning pipeline consists of the following steps:

1. Preprocessing (ie:making data ready for analysis).
2. Visualizing label distribution.
3. Train-test splitting.
4. Data Balancing (Upsampling, down sampling, SMOTE).
5. Feature selection.
6. Outlier removal.
7. Visualizing the distribution/correlation of features wrt labels.
8. 3D scatter visualization of top three features.
9. 3D scatter visualization of dimensionally reduced data.
10. Hyperparameter tuning using K Fold cross validation.
11. Filtering out a subset of models based on carefully chosen conditions.
12. Choosing the best model for the task by visualizing performance of the selected models using precision-recall curves (for classification) or CDF plots (for regression).
13. Retraining the chosen model on the entire train set and rechecking model performance
14. Testing the trained model on the test set.

To implement the steps outlined above we used some basic statistical techniques, linear algebra and the following supervised/unsupervised techniques and algorithms:

1. K means algorithm for clustering.
2. Local Outlier Factor for outlier detection.
3. PCA technique for dimensionality reduction.
4. Logistic Regression algorithm for Classification tasks.
5. Linear Regression algorithm for Regression tasks.

It should be understood that, there are techniques and algorithms other than the ones used till now, which serve the same purposes. Once one grasps the reasons for each of the steps within the pipeline described above, most of machine learning is about gaining an understanding about the various algorithms/techniques available and making the most appropriate choices with respect to the tasks at hand. Some examples of other algorithms and techniques are given below:

1. CLUSTERING: Hierarchical Clustering, DB Scan, Spectral Clustering, etc.
2. OUTLIER DETECTION: Ransac, Isolation Forest, Autoencoders, etc.
3. DIMENSIONALITY REDUCTION: Kernel PCA, Tsne, Umap, Autoencoders, etc.

4. CLASSIFICATION: K Nearest Neighbours, Naive Bayes, Random Forest, Boosted Decision Trees, Neural Networks, etc
5. REGRESSION: K Nearest Neighbours, Random Forest, Boosted Decision Trees, Neural Networks, etc

DATA SHIFT:

Once a machine learning model is trained and tested, it is deployed into the real world to perform the task it was trained for. The ideal situation would be that the model's real world performance is similar to that observed during its testing. But this may not always be so. It is quite possible that the model's performance may be worse or maybe the model's performance depreciates with the passage of time. In most cases, this phenomenon can be explained via the concept of "data shift".

Data shift refers to the "shift" (or difference) in the real world data with respect to the training data. This difference in data causes the model to perform differently than what was observed during the training/testing stages. Data shift can be classified into three categories:

- **COVARIATE SHIFT:** This refers to the shift in the distribution of the features (or independent variables) of the train dataset and the real world data.
- **PRIOR PROBABILITY SHIFT:** This refers to the shift in the distribution of the labels (or target variables) of the train dataset and the real world data.
- **CONCEPT SHIFT:** This refers to the shift in the features - labels relationship in the train dataset and the real world data.

One of the methods used to create resilience to data shifts, is to make sure that the models are periodically retrained and tested with data that is infused with "newness". In other words, the existing data is mixed with a suitable proportion of new data, while at the same time, an equal proportion of data from the earliest portion of the timeline (oldest data) is dropped. This involves the additional tasks of collected new data and labelling it.

10. K NEAREST NEIGHBOURS

K NEAREST NEIGHBOURS

The K Nearest Neighbours (KNN) algorithm is one of the most basic machine learning algorithms. KNN is basically a distance based prediction technique, where given a query point, it looks for K nearest points in the train set and predicts the majority class (in case of classification) or mean value (in case of regression) of those neighbouring points as the class/value corresponding to the query point.

KNN models require the presence of the entire to make its predictions and its efficiency is very much dependent on the search/sorting techniques used for finding the K nearest neighbours.

The basic tuning parameter for KNN models is the number of neighbours parameter.
This refers to the number of closest data points the algorithm will consider as its neighbours will making its prediction. In its simplest version, the KNN algorithm considers only one neighbour (ie: the training data point closest to the query data) and simply predicts the class/values of that neighbour as the query point's class/value. This corresponds to extreme **overfitting**. As the number of neighbours is increased the model becomes more and more generalized and after a certain threshold, the model starts becoming **under fitted.**

SEARCH ALGORITHMS:

At its simplest form, the KNN algorithm uses **brute force** to find the K nearest neighbours of the query point it is given. In other words it computes the distances of all data points with respect to the query point, sorts these values and then chooses the top K data points as its neighbours. This strategy is suitable only for small data sets. For larger data sets, optimized space partitioning strategies are utilized. Two of the most commonly used strategies are:

1. KD TREE:

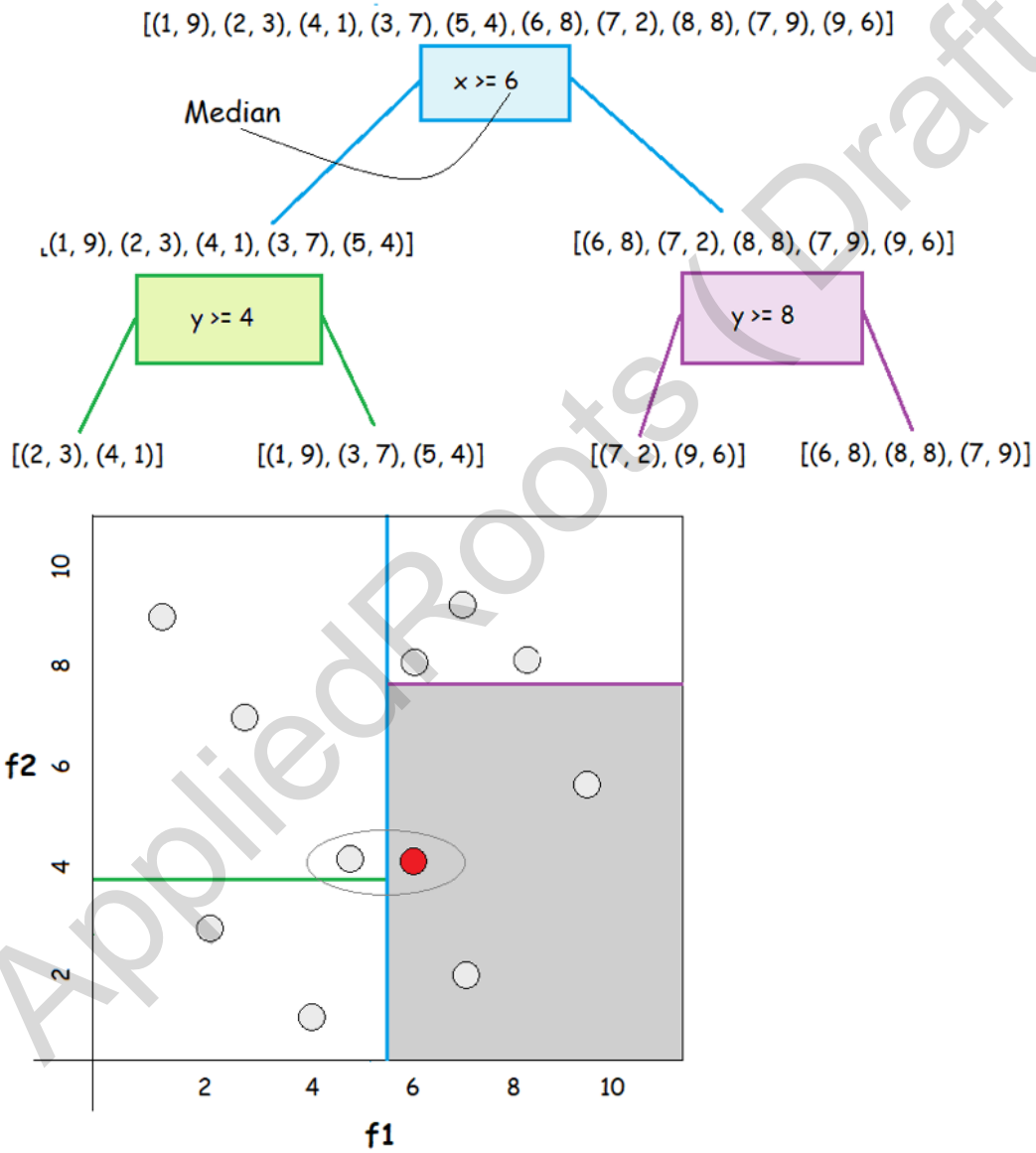
The KD Tree algorithm builds a data structure that organizes the data set as a **Binary Tree** and then when we want to find the nearest neighbour to a query point, we navigate down the tree to the branch that contains the nearest neighbours. Building a KD Tree essentially involves the following steps:

1. Randomly pick a feature (axis) from the dataset.
2. For that feature find the median value of all the data points.
3. Using this median value split the data into 2 approximately equal subgroups (two branches of the data tree)
4. Randomly choose any other feature except the features(s) chosen in the previous step(s).
5. For each data subgroup/branch find the median value and split it as described before (This results in 4 branches/subgroups at a tree depth of 2).

6. Recursively apply steps 4 & 5 until each branch has at least the prescribed number of data points.

The image shown below describes the process outlined above. The train dataset contains ten data points. For the first step, we use the median of feature **f1** to split the data space into 2 subgroups (Blue vertical line). The same step is recursively applied to these two resulting data sub groups using a different feature **(f2)** for median calculation.

So for the data points in the left subset, we get a new partition along the green horizontal line (median along feature f2 of data points in that subset) and in the same way for the data subset on the right side, we get a new partition along the purple line.



Now given an example query point (6, 4) - the red dot in the image above, its nearest neighbours will be the points in the shaded region as shown in the image above.

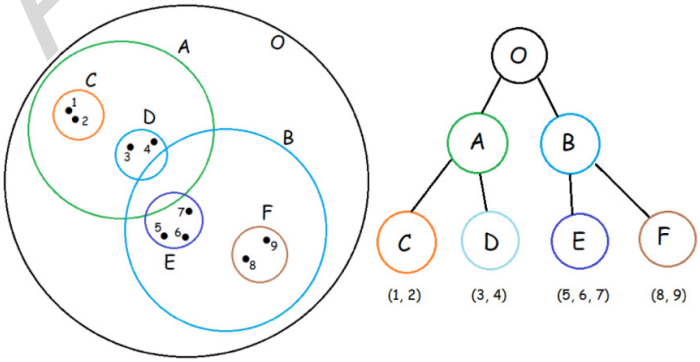
As can be seen from the image above, this form of data structuring decomposes the feature space using axis parallel planes (or lines in case of 2D data). Thus the nearest neighbours found using this method may not be completely accurate. In the above image the point encircled with the red point (query) is the closest neighbour, even though it belongs to a different "neighbourhood". Even so, the KD Tree method usually provides a quick and efficient neighbourhood search for KNN models.

2. BALL TREE:

The Ball tree algorithm is quite similar to KD Tree, except that it uses spherical distances to decompose the feature space instead of using axis parallel planes. It essentially involves the following steps:

1. Randomly select a point from the dataset and find the furthest point to this point.
2. Find the furthest point from the furthest point selected in step 1.
3. Draw a line joining the two points selected in steps 1 & 2 and project all data points on to this line.
4. Find the median of all projected data points on the line obtained in step 3. This divides the line into two segments and decomposes the dataset into two subgroups depending on which side of the median they fall.
5. For each line segment :
    a. Find the midpoint and distance of this midpoint to the farthest point in the data sub group corresponding to this line segment.
    b. Create a hypersphere at the midpoint, having its radius the same as the distance just computed.
6. Since the Ball Tree algorithm decomposes the feature space using the hyperspheres instead of axis parallel planes, it avoids the pitfall (sometimes wrongly grouped nearest points) of the KD Tree algorithm.

Since the Ball Tree algorithm decomposes the feature space using the hyperspheres instead of axis parallel planes, it avoids the pitfall (sometimes wrongly grouped nearest points) of the KD Tree algorithm.



KNN USING SCIKIT LEARN:

The Scikit Learn library provides the KNeighborsClassifier class to perform classification using the KNN algorithm. Described below are some of its most relevant parameters.

n_neighbors: This parameter refers to the number of neighbours to consider (default set to 5)

weights: This parameter refers to the "weight function to be used during prediction. The following options are provided (default set to 'uniform'):

1. 'uniform' : All points in each neighborhood are weighted equally.
2. 'distance': Weight points by the inverse of their distance. Closer neighbors of a query point will have a greater influence than neighbors which are further away.

algorithm: This parameter refers to the algorithm used to compute the nearest neighbors. The following options are provided (default set to 'auto'):

1. 'ball_tree' : will use BallTree
2. 'kd_tree': will use KDTree
3. 'brute': will use a brute-force search
4. 'auto': will attempt to decide the most appropriate algorithm based on the values passed to fit method.

leaf_size: This parameter refers to the termination condition (min neighbours per node) while creating a KD Tree or Ball Tree (default set to 30).

metric: This parameter refers to the distance metric to use for the tree. Some of the more relevant metrics that can be used are: 'euclidean', and 'manhattan' .

USING KNN ON BREAST CANCER DATA SET:

Since we have discussed the binary classification pipeline in quite some detail in the previous chapters, we will pick up from the "model training and evaluation" stage in the pipeline. We will be using the same set of machine learning Functions described in the previous chapters on Logistic Regression, but this time all of these functions are saved in a file "**aaic_mlfuncs.py**" .

We import the train and test sets as shown below:

```
import aaic_mlfuncs as ai
```

```
1  df_tr_set = pd.read_csv(data_path + 'df_tr_BC.csv')
2  df_ts_set = pd.read_csv(data_path + 'df_ts_BC.csv')
3
4  df_tr_set.shape, df_ts_set.shape
```

```
((438, 9), (114, 9))
```

```
X_train = df_tr_set.iloc[:, :-1].values
X_test  = df_ts_set.iloc[:, :-1].values

y_train = df_tr_set.iloc[:, -1].values
y_test  = df_ts_set.iloc[:, -1].values

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((438, 8), (114, 8), (438,), (114,))
```

We perform grid search with K fold cross validation as shown below:

```
from sklearn.neighbors import KNeighborsClassifier

model_class = KNeighborsClassifier

parameter_grid = dict(n_neighbors = range(5, 20, 5),
                      weights = ['uniform', 'distance'],
                      algorithm = ['ball_tree', 'kd_tree'],
                      leaf_size = range(10, 50, 10),
                      metric = ['euclidean', 'manhattan'])

z = ai.fn_kfoldcv_clf_binary(X_train, y_train, model_class,  parameter_grid)

df_kfoldcv_gridsearch, dict0_model_instances = z
df_kfoldcv_gridsearch.describe()
```

```
100% (96 of 96) |########################| Elapsed Time: 0:00:04 Time:  0:00:04
```

	ts_mean_rec_1	ts_mean_prec_1	ts_mean_rec_0	ts_mean_prec_0	ts_mean_loss	ts_std_loss	ts_mean_acc	ts_std_rec_1	ts_std_rec_0
count	96.000000	96.000000	96.000000	96.000000	96.000000	96.000000	96.000000	96.000000	96.000000
mean	0.988889	0.947575	0.896514	0.977824	0.235930	0.132641	0.956621	0.004962	0.049708
std	0.004280	0.007656	0.016929	0.008375	0.072292	0.062333	0.005217	0.003527	0.014434
min	0.982456	0.932570	0.862745	0.965758	0.125505	0.024457	0.947489	0.000000	0.033327
25%	0.985965	0.943547	0.888889	0.971869	0.190406	0.109462	0.954338	0.003722	0.040290
50%	0.989474	0.950118	0.901961	0.978375	0.226893	0.125092	0.956621	0.004962	0.048029
75%	0.992982	0.953047	0.908497	0.985926	0.320100	0.204104	0.961187	0.006203	0.048911
max	0.996491	0.956048	0.915033	0.993056	0.324404	0.208179	0.965753	0.009924	0.080049

We then filter the best models as shown below:

```
dff = df_kfoldcv_gridsearch
n = 5

df1 = dff.sort_values(by = 'ts_mean_rec_1', ascending = False)[:n]
df2 = dff.sort_values(by = 'ts_mean_prec_1', ascending = False)[:n]
df3 = dff.sort_values(by = 'ts_std_rec_1', ascending = True)[:n]
df4 = dff.sort_values(by = 'ts_mean_loss', ascending = True)[:n]
df5 = dff.sort_values(by = 'ts_std_loss', ascending = False)[:n]

df_filtered_cv = pd.concat([df1, df2, df3, df4, df5]).drop_duplicates()
df_filtered_cv = df_filtered_cv[df_filtered_cv.ts_mean_loss < 0.3]
df_filtered_cv = df_filtered_cv.sort_values(by = 'ts_mean_rec_1', ascending = False)[:5]
df_filtered_cv
```

	ts_mean_rec_1	ts_mean_prec_1	ts_mean_rec_0	ts_mean_prec_0	ts_mean_loss	ts_std_loss	ts_mean_acc	ts_std_rec_1	ts_std_rec_0
model_48	0.996491	0.953315	0.908497	0.993056	0.259035	0.131191	0.965753	0.004962	0.033327
model_95	0.992982	0.932570	0.862745	0.985926	0.125505	0.024457	0.947489	0.004962	0.080049
model_47	0.989474	0.953118	0.908497	0.978834	0.192016	0.109256	0.961187	0.000000	0.040290
model_66	0.989474	0.943547	0.888889	0.978375	0.194752	0.118993	0.954338	0.000000	0.040290
model_32	0.985965	0.956048	0.915033	0.971902	0.262598	0.131799	0.961187	0.009924	0.040290

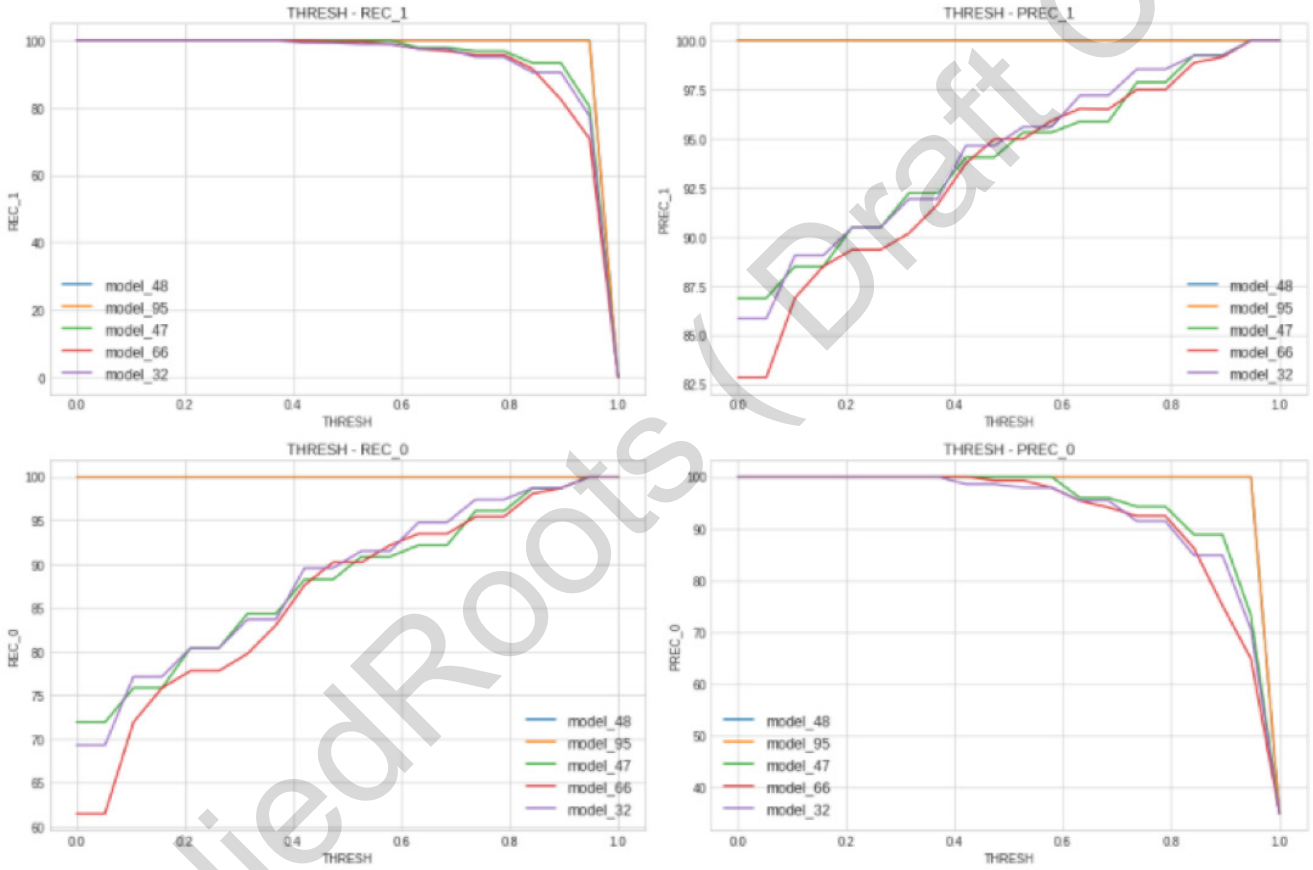
We can check the parameters of these chosen models as shown below:

```
display(dict0_model_instances['model_48'],
        dict0_model_instances['model_95'],
        dict0_model_instances['model_47'],
        dict0_model_instances['model_66'],
        dict0_model_instances['model_32'])
```

```
KNeighborsClassifier(algorithm='ball_tree', leaf_size=10, metric='euclidean',
                     metric_params=None, n_jobs=None, n_neighbors=10, p=2,
                     weights='distance')
KNeighborsClassifier(algorithm='kd_tree', leaf_size=40, metric='manhattan',
                     metric_params=None, n_jobs=None, n_neighbors=15, p=2,
                     weights='distance')
KNeighborsClassifier(algorithm='kd_tree', leaf_size=40, metric='manhattan',
                     metric_params=None, n_jobs=None, n_neighbors=10, p=2,
                     weights='uniform')
KNeighborsClassifier(algorithm='ball_tree', leaf_size=20, metric='euclidean',
                     metric_params=None, n_jobs=None, n_neighbors=15, p=2,
                     weights='uniform')
KNeighborsClassifier(algorithm='ball_tree', leaf_size=10, metric='euclidean',
                     metric_params=None, n_jobs=None, n_neighbors=10, p=2,
                     weights='uniform')
```

We then plot the precision - recall curves for the above selected models as shown below:

```
df_tr_standard, df_ts_standard, scaler = ai.fn_standardize_df(df_tr_set, df_ts_set)

listO_model_names = list(df_filtered_cv.index)
X_train, y_train = df_tr_standard.iloc[:, :-1].values, df_tr_standard.iloc[:, -1].values
listO_models = [dictO_model_instances[i].fit(X_train, y_train) for i in listO_model_names]

listO_thresholds = np.linspace(0, 1, 20)
legend = listO_model_names

ai.fn_performance_models_data(listO_models, df_tr_standard, listO_thresholds, legend)
```



Studying the precision recall curves of the various models above and assuming that we want to optimize for **precision**, we choose **model_32**, thresholded at 0.7 as our model for prediction purposes.

We then train the **model_32** on the entire train set and test its performance across the train and test sets as shown below:

```
df_Xy_ = df_tr_standard
model_ = dictO_model_instances['model_32'].fit(X_train, y_train)
thresh = 0.7

ai.fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.0957
ACCURACY: 95.89
--------------------
```

	prec	rec
class_0	91.411	97.386
class_1	98.545	95.088

```
1  df_Xy_ = df_ts_standard
2
3  ai.fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.0896
ACCURACY: 95.614
--------------------
```

	prec	rec
class_0	91.111	97.619
class_1	98.551	94.444

As can be seen from the train set and test set performances above, the model generalizes well and gives a good performance.

ADVANTAGES AND LIMITATIONS OF KNN:

KNN models are easy to interpret due to the simplicity of the algorithm and it often gives reasonable performance without needing too much tuning. In fact KNN can be used as a good "base-line" model with respect to which other more complex models can be evaluated.

KNN models are not often used as it basically requires the entire training data for the sake of prediction. This is a deterrent when dealing with large high dimensional datasets. KNN does not perform well with data that have more than a few hundred features.

11. NAIVE BAYES

NAIVE BAYES

Naive Bayes models are probabilistic **generative** models. All the previous models discussed so far are **discriminative** models . Generative modes are based on:

1. The joint probabilities $\mathbf{P(x_i, c_j)}$ (where $\mathbf{x_i}$, is some data point & $\mathbf{c_j}$ is some class **j** corresponding to $\mathbf{x_i}$).
2. Making predictions by using Bayes rule to calculate $\mathbf{P(c_j | x_i)}$.

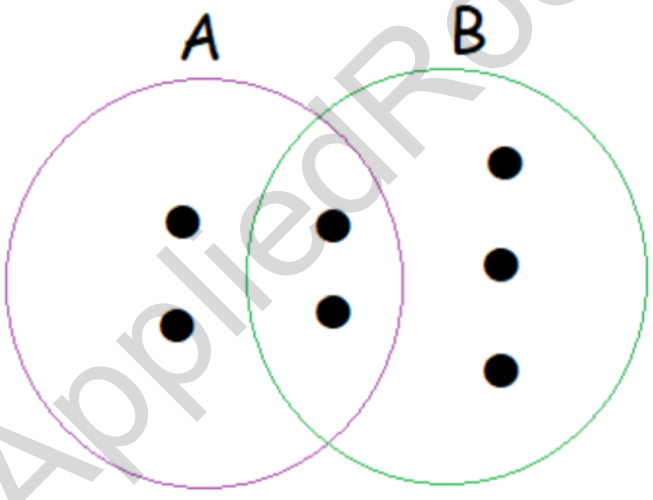
Discriminative classifiers essentially learn a mapping from inputs X to the class labels y. Generative models learn the distribution of the training data and thus can:

1. Predicting the probabilities for each class given a data point (ie: a set of features).
2. It also becomes capable of simulating/generating the data itself from scratch.

The Naive Bayes model uses the chain rule of probability to model the distribution of each class with respect to the features of the dataset. To do this it inspects each feature individually and collects their respective statistics.

THE CHAIN RULE:

The chain rule of probability basically expresses the joint probability of a group of dependent variables as the recursive product of conditional probabilities as described in the image below.



P(A) = 4/7
P(B) = 5/7
P(A, B) = P(A ∩ B) = 2/7

P(A|B) = 2/5 = P(A, B)/P(B)

Which implies: P(A, B) = P(A|B) P(B).
Similarly: P(A, B, C) = P(A|B, C) P(B, C)
Now, P(B, C) can be further decomposed as: P(B|C) P(B)

Hence: P(A, B, C) = P(A|B, C) P(B|C) P(B)

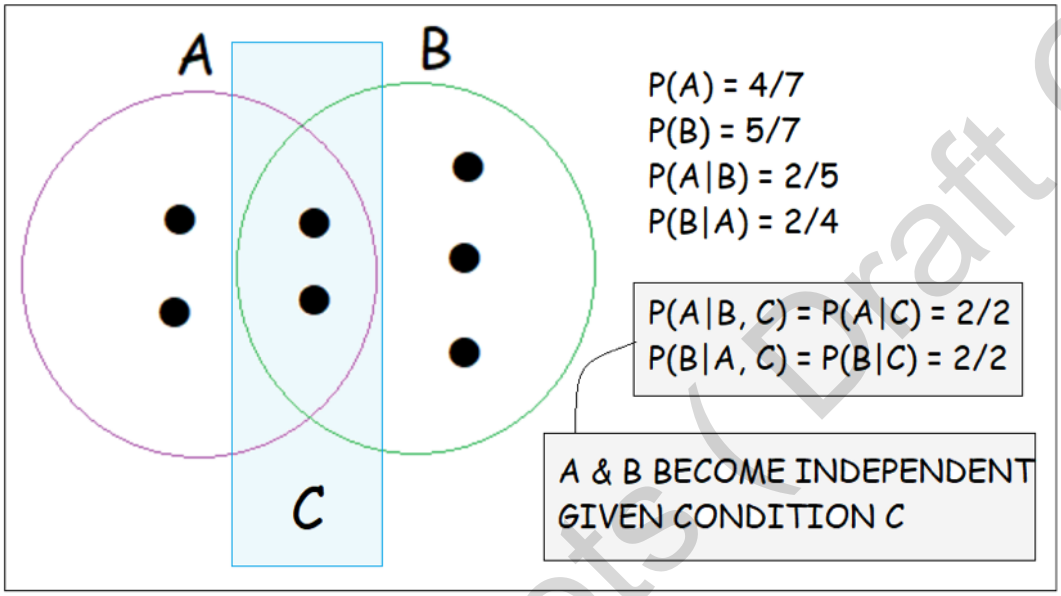
In general: P(x1, x2, x3...xn) = P(x1|x2, x3...xn) P(x2| x3, x4... xn) .....P(x_n-1|xn) P(xn)

CONDITIONAL INDEPENDENCE:

Two Random Variables A and B are considered conditionally independent, given a condition C if:

P(A|B, C) = P(A|C)

Consider the image shown below, A and B are conditionally independent given that we only consider data that belongs within region C



The Naive Bayes algorithm is so called because it makes the naive or simplistic assumption that all the features of the data set are independent of each other given that they belong to a certain class/label.

THE NAIVE BAYES ALGORITHM:

Consider a d dimensional dataset constraining n points and consists of k classes:

$D = \{x_i, y_i\}$ where:

1. $i = 1$ to n
2. $x \in \mathbb{R}^d$ (ie: x is d dimensional) and
3. $Y = \{C_1, C_2, C_3, \ldots C_j \ldots C_k\}$

Then the probability of some class $\mathbf{C_j}$ given a data point $\mathbf{x_i}$ is proportional to the joint probability of $\mathbf{C_j}$ & $\mathbf{x_i}$:

$$P(C_j | x_i) = P(C_j, x_i) / P(x_i)$$

OR: $P(C_j | x_i) \propto P(C_j, x_i)$

Since the data is d dimensional the above proportionality becomes modified to:

$$P(C_j \mid x_i) \propto P(C_j, x_i^1, x_i^2, x_i^3, \ldots\ldots x_i^d)$$

OR: $$P(C_j \mid x_i) \propto P(x_i^1, x_i^2, x_i^3, \ldots\ldots x_i^d, C_j)$$

Note that the **superscripts** in the above image refer to dimensions and not to exponentiation.

Now using the **chain rule** of probability, the above proportionality can be expressed as:

$$P(x_i^1, x_i^2, x_i^3, \ldots\ldots x_i^d, C_j)$$

$$= P(x_i^1 \mid x_i^2, x_i^3, \ldots\ldots x_i^d, C_j)\, P(x_i^2 \mid x_i^3, x_i^4 \ldots\ldots x_i^d, C_j) \ldots\ldots P(x_i^d \mid C_j)\, P(C_j)$$

If we now make the Naive assumption that all features are conditionally independent given $\mathbf{C}_j$, the above equation becomes:

$$P(x_i^1, x_i^2, x_i^3, \ldots\ldots x_i^d, C_j) = P(x_i^1 \mid C_j)\, P(x_i^2 \mid C_j)\, P(x_i^3 \mid C_j) \ldots\ldots P(x_i^d \mid C_j)\, P(C_j)$$

$$P(C_j \mid x_i) \propto P(C_j) \prod_{m=1}^{d} P(x_i^m \mid C_j) \quad \text{(A)}$$

OBJECTIVE FUNCTION:

The objective function of the Naive Bayes classification can thus be defined as shown below:

$$j^* = \underset{j}{\operatorname{argmax}} \sum_{j=1}^{k} P(C_j) \prod_{m=1}^{d} P(x_i^m \mid C_j)$$

For any given data point **xi** find that **j** (ie: Class) that maximises the above term. The above equation is called the **maximum a posteriori Rule** or **MAP** rule.

LAPLACE OR ADDITIVE SMOOTHING:

Laplace smoothing is a method used to "smooth" probabilities, especially when dealing with data sets whose features represent count values. For example consider the case of text classification, where the task is to classify given documents into say, positive sentiment or negative sentiment. To do this we use a collection of labelled text documents (labelled as positive or negative) for training. Each document in the train set is vectorized using the following strategy:

1. The set of all words present in the training documents is considered as the features of the documents.
2. Each document is then vectorized based on the count values of each word present in it.

Therefore if the set of all **d** words in the train set = **{W1, W2, W3.... Wd}**, then a particular document could be represented as a **d** dimensional vector: **doc_1 = [1, 2, 1 .... 0]** where each number represents the number of times each of those words appears in that particular document. Then the probability that some document **doc_i** belongs to **class** j is given by:

$P(C_j \mid doc\_i) = P(C_j)\ P(W1 \mid Cj)\ P(W2 \mid C_j)\ ....\ P(Wd \mid C_j)$

In the two class text classification task above, it could so happen that some words present in one particular class may not be present in the other. Let us say that word **W3** is not present in the negative class. Therefore the following becomes true:

P(W3 | neg) = 0

Therefore for all documents belonging to the negative class, the following becomes true, just because one word that exists in the total vocabulary does not exist in the documents of the negative class

P(neg | neg_doc) = P(neg) P(W1 | neg) P(W2 | neg) P(W3 | neg) .... P(Wd | neg)

P(neg | any_neg_doc) = 0

To avoid the above problem we use Laplace smoothing. In this technique the probability of a word for a given class, **P(W | class)** is modified as shown below.

$$P(Wi \mid Cj) = \frac{Count(Wi) \text{ in } Cj + \alpha}{Count(\text{all words}) \text{ in } Cj + \gamma\alpha}$$

Where:

1. γ is an attribute of Wi, signifying the number of states Wi can have. In this case γ **= 2** since Wi can have two states : Present for that particular class or not.
2. α is a tuning parameter, which can be any positive integer.

Using Laplace smoothing results in the value of **P(Wi | Cj)** to always be above zero and within 0.5. As α is increased **P(Wi | Cj)** tends towards 0.5. Smaller the value of α, greater will be the chances of overfitting to the train data, generally it is set to the value of one.

NAIVE BAYES USING SCIKIT LEARN:

The Scikit Learn library provides the **naive_bayes** module to perform classification using the Naive Bayes algorithm. There are basically two main types of Naive Bayes classifiers in this module:

1. **MultinomialNB:** This classifier is used when dealing with datasets whose features are basically count data (discrete random variables), like in the case of the text classification example just discussed previously.
2. **GaussianNB:** This classifier is used when dealing with datasets whose features are continuous Random Variables (like the features of the Breast Cancer Dataset). GaussianNB have their own version of probability smoothing called variance smoothing which uses the same underlying logic of Laplace smoothing, but suitably modified for continuous random variables.

USING NAIVE BAYES ON BREAST CANCER DATA SET:

Since we have discussed the binary classification pipeline in quite some detail in the previous chapters, we will pick up from the "model training and evaluation" stage in the pipeline. We will be using the same set of machine learning Functions saved in a file "**aaic_mlfuncs.py**" .

We import the train and test sets as shown below:

```
import aaic_mlfuncs as ai
```

```
1  df_tr_set = pd.read_csv(data_path + 'df_tr_BC.csv')
2  df_ts_set = pd.read_csv(data_path + 'df_ts_BC.csv')
3
4  df_tr_set.shape, df_ts_set.shape
```

```
((438, 9), (114, 9))
```

```
1  X_train = df_tr_set.iloc[:, :-1].values
2  X_test  = df_ts_set.iloc[:, :-1].values
3
4  y_train = df_tr_set.iloc[:, -1].values
5  y_test  = df_ts_set.iloc[:, -1].values
6
7  X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((438, 8), (114, 8), (438,), (114,))
```

We perform grid search with K fold cross validation using the **GaussianNB** classifier because we are dealing with **continuous** features. The code shown below demonstrates this.

```
from sklearn.naive_bayes import GaussianNB

model_class = GaussianNB
parameter_grid = dict(var_smoothing = [10**-i for i in range(2, 15)])

z = ai.fn_kfoldcv_clf_binary(X_train, y_train, model_class,  parameter_grid)

df_kfoldcv_gridsearch, dict0_model_instances = z
df_kfoldcv_gridsearch.describe()
```

```
100% (13 of 13) |########################| Elapsed Time: 0:00:00 Time:  0:00:00
```

	ts_mean_rec_1	ts_mean_prec_1	ts_mean_rec_0	ts_mean_prec_0	ts_mean_loss	ts_std_loss	ts_mean_acc	ts_std_rec_1	ts_std_rec_0
count	13.000000	13.000000	13.000000	13.000000	13.000000	13.000000	13.000000	13.000000	1.300000e+01
mean	0.968961	0.941944	0.892356	0.942020	0.460529	0.148292	0.941606	0.021994	9.449456e-03
std	0.001318	0.000082	0.000000	0.002171	0.000117	0.002178	0.000852	0.001821	1.805557e-18
min	0.968421	0.941910	0.892356	0.941130	0.460143	0.141094	0.941258	0.017891	9.449456e-03
25%	0.968421	0.941910	0.892356	0.941130	0.460566	0.148968	0.941258	0.022739	9.449456e-03
50%	0.968421	0.941910	0.892356	0.941130	0.460566	0.148977	0.941258	0.022739	9.449456e-03
75%	0.968421	0.941910	0.892356	0.941130	0.460566	0.148977	0.941258	0.022739	9.449456e-03
max	0.971930	0.942130	0.892356	0.946912	0.460566	0.148977	0.943525	0.022739	9.449456e-03

We then filter the best models as shown below:

```
dff = df_kfoldcv_gridsearch
n = 5

df1 = dff.sort_values(by = 'ts_mean_rec_1', ascending = False)[:n]
df2 = dff.sort_values(by = 'ts_mean_prec_1', ascending = False)[:n]
df3 = dff.sort_values(by = 'ts_std_rec_1', ascending = True)[:n]
df4 = dff.sort_values(by = 'ts_mean_loss', ascending = True)[:n]
df5 = dff.sort_values(by = 'ts_std_loss', ascending = False)[:n]

df_filtered_cv = pd.concat([df1, df2, df3, df4, df5]).drop_duplicates()
df_filtered_cv = df_filtered_cv[df_filtered_cv.ts_mean_loss < 0.5]
df_filtered_cv = df_filtered_cv.sort_values(by = 'ts_mean_rec_1', ascending = False)[:5]
df_filtered_cv
```

	ts_mean_rec_1	ts_mean_prec_1	ts_mean_rec_0	ts_mean_prec_0	ts_mean_loss	ts_std_loss	ts_mean_acc	ts_std_rec_1	ts_std_rec_0
model_0	0.971930	0.94213	0.892356	0.946912	0.460143	0.141094	0.943525	0.017891	0.009449
model_1	0.971930	0.94213	0.892356	0.946912	0.460512	0.148059	0.943525	0.017891	0.009449
model_2	0.968421	0.94191	0.892356	0.941130	0.460561	0.148884	0.941258	0.022739	0.009449
model_3	0.968421	0.94191	0.892356	0.941130	0.460566	0.148968	0.941258	0.022739	0.009449
model_4	0.968421	0.94191	0.892356	0.941130	0.460566	0.148977	0.941258	0.022739	0.009449

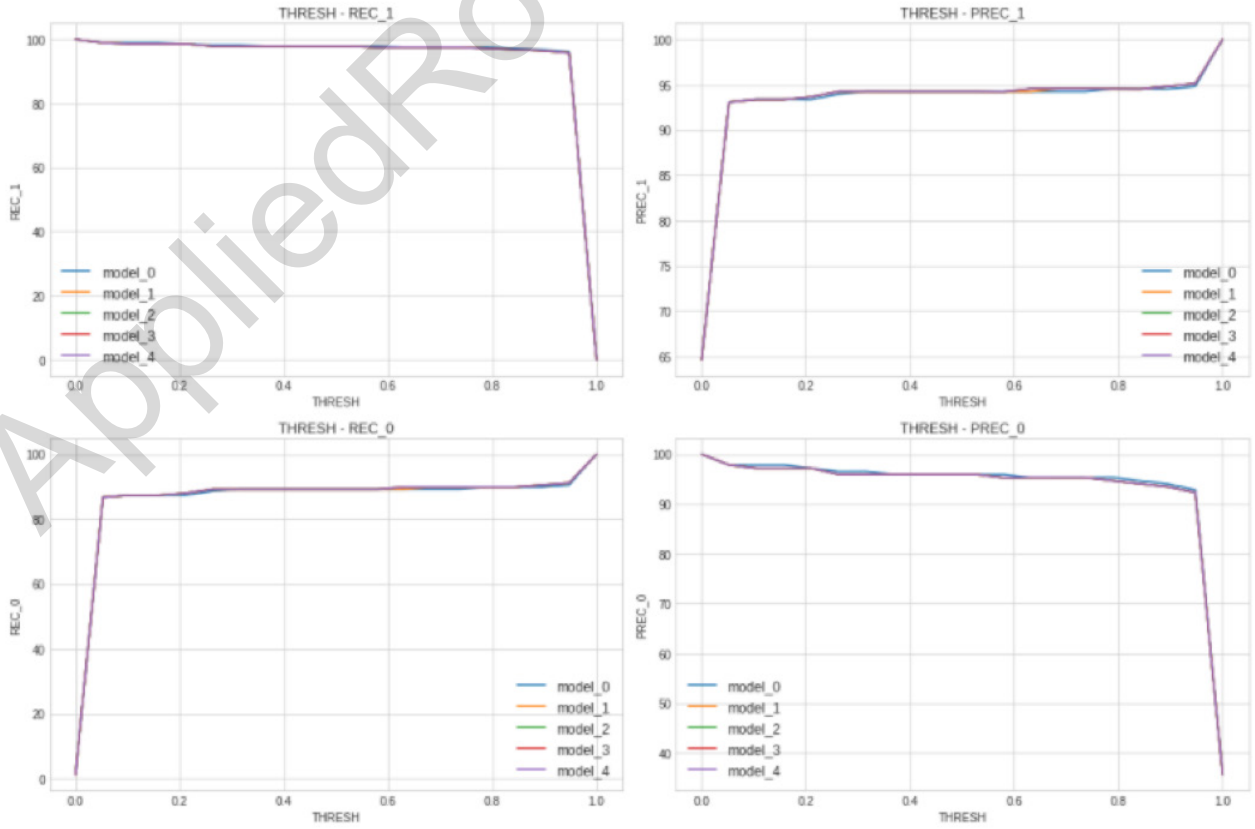
We can check the parameters of these chosen models as shown below:

```
display(dict0_model_instances['model_0'],
        dict0_model_instances['model_1'],
        dict0_model_instances['model_2'],
        dict0_model_instances['model_3'],
        dict0_model_instances['model_4'])
```

```
GaussianNB(priors=None, var_smoothing=0.01)
GaussianNB(priors=None, var_smoothing=0.001)
GaussianNB(priors=None, var_smoothing=0.0001)
GaussianNB(priors=None, var_smoothing=1e-05)
GaussianNB(priors=None, var_smoothing=1e-06)
```

We then plot the precision - recall curves for the above selected models as shown below:

```
df_tr_standard, df_ts_standard, scaler = ai.fn_standardize_df(df_tr_set, df_ts_set)

list0_model_names = list(df_filtered_cv.index)
X_train, y_train = df_tr_standard.iloc[:, :-1].values, df_tr_standard.iloc[:, -1].values
list0_models = [dict0_model_instances[i].fit(X_train, y_train) for i in list0_model_names]

list0_thresholds = np.linspace(0, 1, 20)
legend = list0_model_names

ai.fn_performance_models_data(list0_models, df_tr_standard, list0_thresholds, legend)
```



Since all models have very similar curves we choose **model_0** since it produces the least loss. Studying the precision recall curves of **model_0** and assuming that we want to optimize for **precision**, we choose **model_32**, thresholded at 0.8 as our model for prediction purposes. We then train the **model_0** on the entire train set and test its performance across the train and test sets as shown below:

```
df_Xy_ = df_tr_standard
model_ = dictO_model_instances['model_0'].fit(X_train, y_train)
thresh = 0.8

ai.fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.4131
ACCURACY: 94.808
--------------------
```

	prec	rec
class_0	95.302	89.873
class_1	94.558	97.544

```
df_Xy_ = df_ts_standard

ai.fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.2182
ACCURACY: 96.491
--------------------
```

	prec	rec
class_0	97.500	92.857
class_1	95.946	98.611

As can be seen from the train set and test set performances above, the model generalizes well and gives a good performance.

ADVANTAGES AND LIMITATIONS OF NAIVE BAYES:

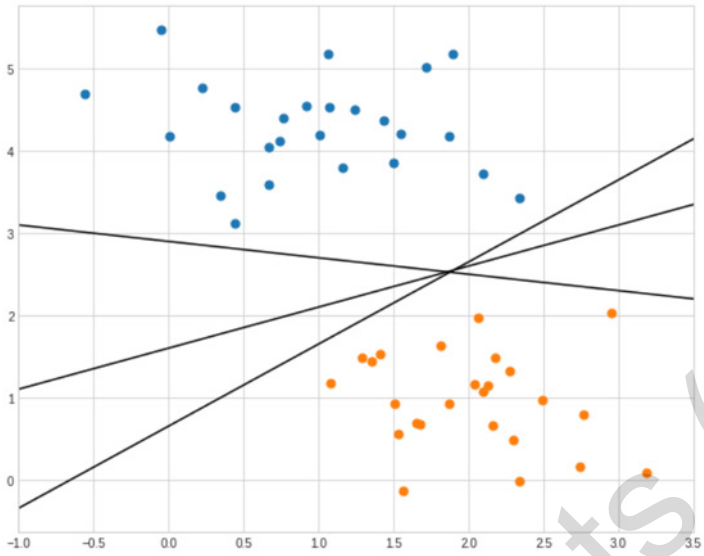
The main advantage of the Naive Bayes model is its interpretability since it is a probabilistic model. Naive bayes classifiers are fast to train, this is because all they are basically doing is inspecting each feature in the dataset individually and computing class label statistics. They are not too sensitive to parameter settings and generally give a good performance. They are well suited for high dimensional large datasets and perform equally well on continuous and categorical data.

Naive bayes models have generalization performance slightly less than parametric linear models. Laplace Smoothing turns out to be an over-head and a necessary requirement when probability of a feature turns out to be zero in a class. Naive bayes models are not suited for imbalanced data and so, training data must always be balanced before using it within the Naive bayes model.

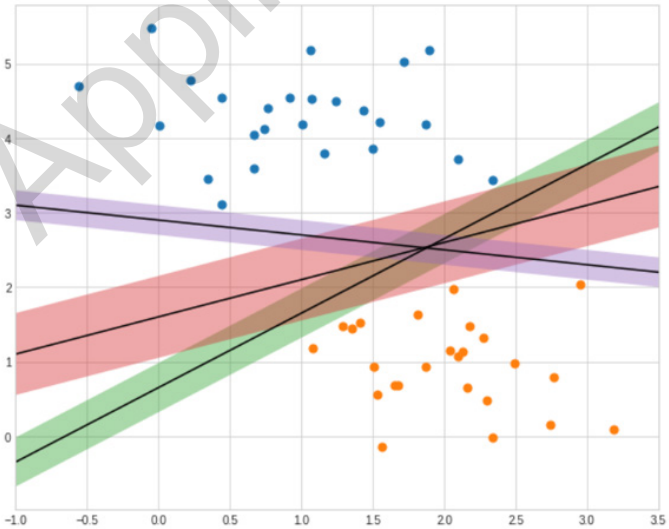
12. SUPPORT VECTOR MACHINES

SUPPORT VECTOR MACHINES

In the chapter on Logistic Regression we learnt about the technique of classification using linear hyperplanes. Consider the two dimensional dataset shown in the image below. As can be seen, there are multiple hyperplanes that can successfully classify the data points. Logistic regression chooses that hyperplane whose weight vector (**w**) produces the least logistic loss.

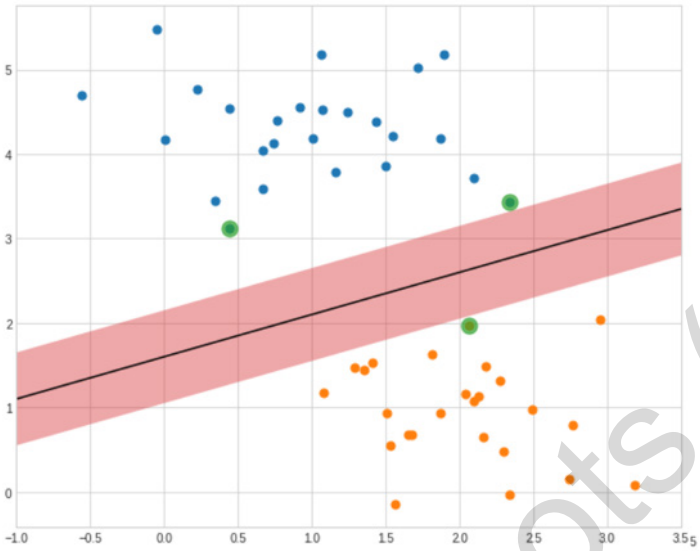


Another strategy for choosing the best hyperplane, is choosing that hyperplane that has the **maximum margin**. Consider the image shown below of the same dataset as earlier. Each of the hyperplanes now has an equi-spaced margin around them, the width of which is constrained by the positions of the closest data points to the hyperplane from the opposite classes. Now, the hyperplane with the largest margin can be considered as the best hyperplane. Support vector machine (SVM) is a classification algorithm that tries to find that weight vector (w) that corresponds to the hyperplane with the largest margin.



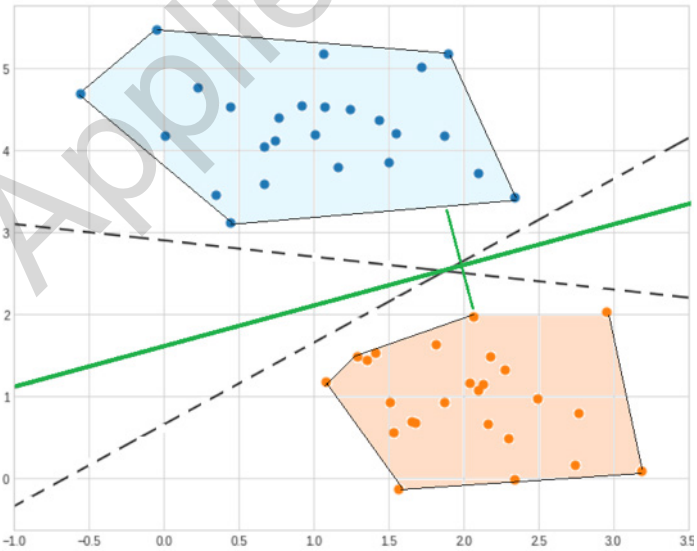
SUPPORT VECTORS:

For a given hyperplane support vectors are those data points that define the hyperplane margin. In other words, support vectors are those points from opposite classes that are nearest to the hyperplane being considered. In the image shown below, the points colored green are the closest points from opposite classes and they represent potential support vectors. The ones that produce the hyperplane with the largest margin are then chosen as the final support vectors. Once the support vectors are fixed we need not consider the rest of the data points for any classification purposes.



GEOMETRIC INTERPRETATION:

SVMs can be geometrically interpreted using the concept of **convex hulls**. Given a set of data points, a convex hull is the smallest polygon such that all points are inside the polygon or on its edges. The image below shows the convex hulls of the two data classes being examined.

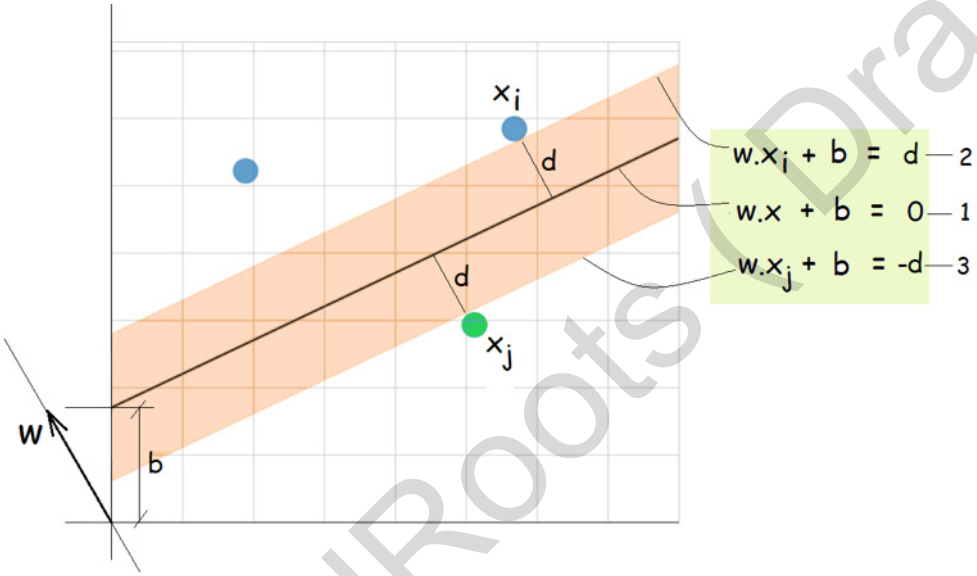


Given the convex hulls of the two classes, the hyperplane with the maximum margin will be that which bisects the shortest line joining the convex hulls.

THE SVM ALGORITHM (PRIMAL FORM):

For the sake of simplicity let us assume that the SVM algorithm first identifies potential support vector data points by first computing the dot products of all possible pairs of data points from opposite classes and then dropping all pairs whose dot product values fall below a specified threshold. In other words we select the K most similar pair of points where each point in the pair belongs to opposite classes.

Consider one such pair of potential support vector points $\mathbf{x}_i$ and $\mathbf{x}_j$ as shown in the image below.



Consider a weight vector **w** corresponding to some hyperplane between these two points, then the separating hyperplane will be represented by equation 1 shown above and the parallel hyperplanes at its margins will be represented by equations 2 and 3. Here d is the amount of offset the planes at the margins have with the main hyperplane. Then the expression for the width of the margin for a given pair of support vector points and some random hyperplane represented by weight vector **w** can be derived in the following way:

1. Subtract equations 1 & 2. We get the equation shown below:

$$w \cdot (x_i - x_j) = 2d$$

2. Scale the entire feature space such that **d = 1**. The right hand side of the above equations thus becomes equal to two as shown below:

$$w \cdot (x_i - x_j) = 2$$

3. Divide each side by the magnitude of the weight vector w. We get the following equation:

$$\frac{w}{||w||} \cdot (x_i - x_j) = \frac{2}{||w||}$$

4. The dot product at the right hand side of the above equation represents the width of the margin, which equates to **2/||w||.** Maximizing this value means maximizing the margin.

In the SVM formulation, the two classes are represented by numbers 1 and -1 where 1 is called the positive class, all these data points lie in the same direction as weight vector **w** and -1 is the negative class, all these data points lie in the opposite direction as **w**.

The SVM optimization function is basically about maximizing the margin such that all positive points in the potential pairs of support vector points are either above or on w.x + b = 1 and all negative points in the potential pairs of support vector points are either below or on w.x + b = -1. This can be mathematically as an optimization function as shown below:

$$w^* = \text{argmax}\ (2/||w||)$$

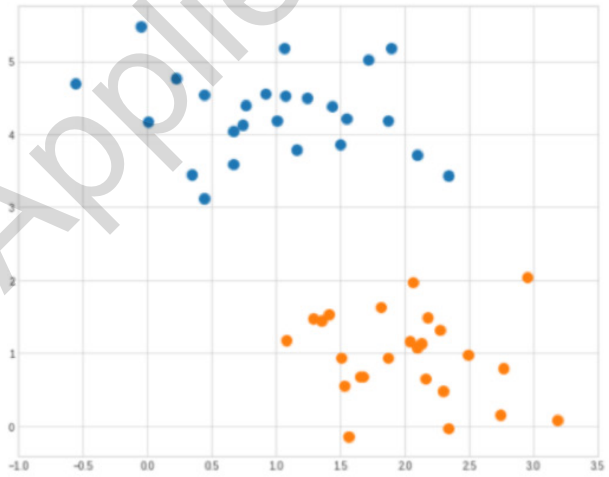
or

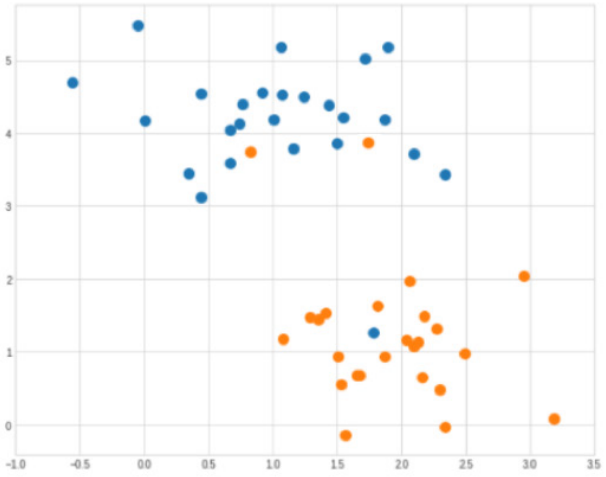
$$w^* = \text{argmin}\ (||w||/2)$$

such that: $y(w.x + b) >= 1$

where y represents labels (1, -1)

This kind of classifier is called a **hard margin** SVM classifier. It only works if the data it is supposed to classify is totally separable as shown in image on the left below. In cases of data of the type shown on the right side, where the classes are not completely separable, this formulation does not work.





SLACK VARIABLE:

To overcome the fact that SVMS do not work for data that is semi separable, we make it more accommodating to a few outliers within its margin. This is done by introducing the concept of a slack variable. The hard margin SVM classifier is modified using a slack variable ξ as shown below.

$$w^* = \text{argmin}\left(||w||/2 + \sum_i \xi_i \right) \quad \text{— 1}$$

where:

$\xi_i = 0$ if $y_i (w \cdot x_i + b) >= 1$

$\xi_i = 1 - y_i (w \cdot x_i + b)$ otherwise

Consider the two conditions of the equation above:

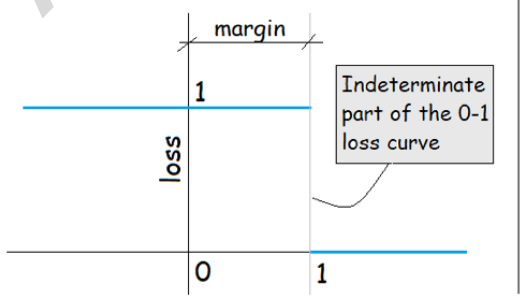
1. When **yi (w.xi + b) >= 1** this means that the point lies beyond or on the separating margins. In this case,ξ will be zero and the model behaves like a hard margin classifier.
2. When **yi (w.xi + b) < 1** this means that the point lies within the separating margins. In this case ξ will be a positive value directly proportional to the degree point **xi** is away from its respective margin. If the point lies on the main separating plane, the loss will be one and it will keep increasing as the point keeps moving in the wrong direction This is expressed in the image on the right side shown below. The loss of a soft margin is called the **hinge loss**. It is so called because its curve is a straight line that hinges on loss = 1 as can be seen in the image below.

The image of the left side represents the loss curve of a hard margin classifier. For the hard Margin classifier:

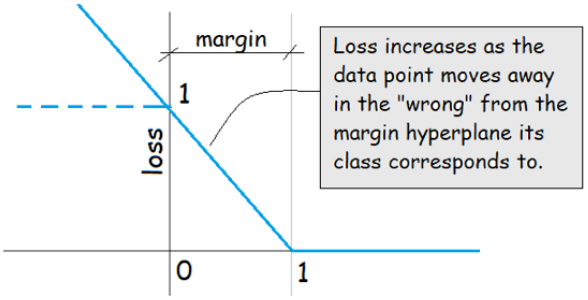
1. If **yi (w.xi + b) >= 1** (ie: for all points on or beyond the margin), the model produces a **loss = 0**.
2. For all other points it produces a **loss = 1.**

Hence its called the **zero-one** loss.

HARD MARGIN CLASSIFIER LOSS (0-1 LOSS)



SOFT MARGIN CLASSIFIER LOSS (HINGE LOSS)



Consider the terms in the equation one shown above. Say we use a variable **c** to modulate the value of the summation of all slack variable values ξ for all support vector points a shown below.

$$||w||/2 + c\sum_i \xi_i$$

The value of **c** allows us to trade-off between maximizing the margin (ie: Being more tolerant to misclassifications) and keeping the margin small (ie: Avoiding misclassification). In other words the value of **c** is inversely proportional to the degree of regularization.

$$w^* = \text{argmin}\left(c\sum_i \xi_i + ||w||^2\right) \quad \text{— 2}$$

where:

$\xi_i = 0$ if $y_i (w \cdot x_i + b) >= 1$

$\xi_i = 1 - y_i (w \cdot x_i + b)$ otherwise

c is a tuneable hyperparameter

In the above formulation we replaced **||w|| / 2** with **||w||2** since it monotonously increases with **||w||**. On inspecting Equation 2 shown above, we see that the first term represents the **loss** and the second term represents some form of **regularization**.

THE HINGE LOSS PERSPECTIVE:

The summation term of the slack variables ξ (first term) in the equation 2 shown above, is basically called the hinge loss and it can be interpreted in the following way:

Hinge loss $= \max(0, 1 - z_i)$

where: $z_i = 1 - y_i (w \cdot x_i + b)$

Therefore:

when $z_i >= 1$, Point xi is beyond/on the separating margins
Hinge loss will be = 0

when $z_i < 1$, Point xi is within the separating margins
Hinge loss will be = $1 - z_i$

Thus equation 2 shown earlier can be reframed as:

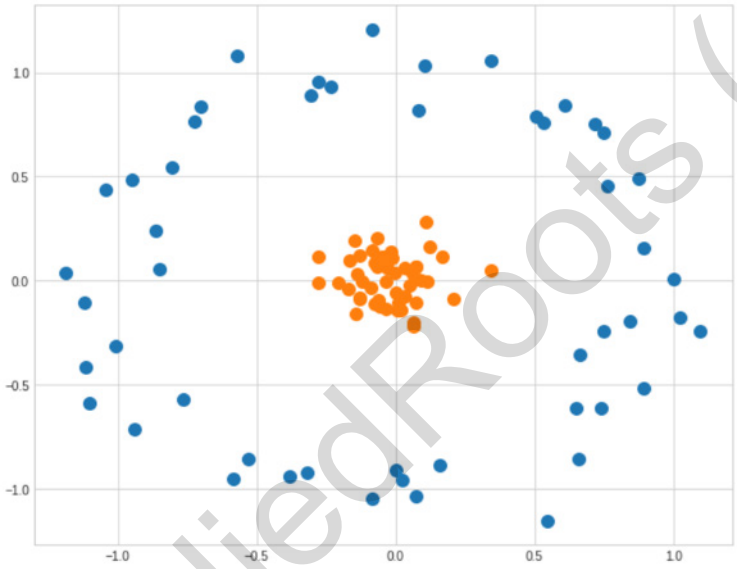
$$w^* = \text{argmin}\left(\max\left(0, \overbrace{1 - y_i (w \cdot x_i + b)}^{z_i}\right), \alpha(\text{reg})\right)$$

Where alpha is a variable that controls the magnitude of the regularization term (**reg**). The regularization term is either the **L1** norm or square of the **L2** norm of the **weight vector** of the separating plane. The higher the value of alpha , the stronger the regularization.

THE KERNEL TRICK (MAKING NON LINEAR DATA LINEARLY SEPARABLE):

Consider the data set shown in the plot below:

```
from sklearn.datasets import make_circles
X, y = make_circles(100, factor=.1, noise=.1)


plt.figure(figsize=(10, 8))
X1 = X[np.where(y == 0)]
plt.scatter(X1[:, 0], X1[:, 1],  s=100)
X2 = X[np.where(y == 1)]
plt.scatter(X2[:, 0], X2[:, 1],  s=100)
plt.show()
```



This kind of data is not separable using a linear classifier. But we could make it separable by feature engineering a new feature as shown below.

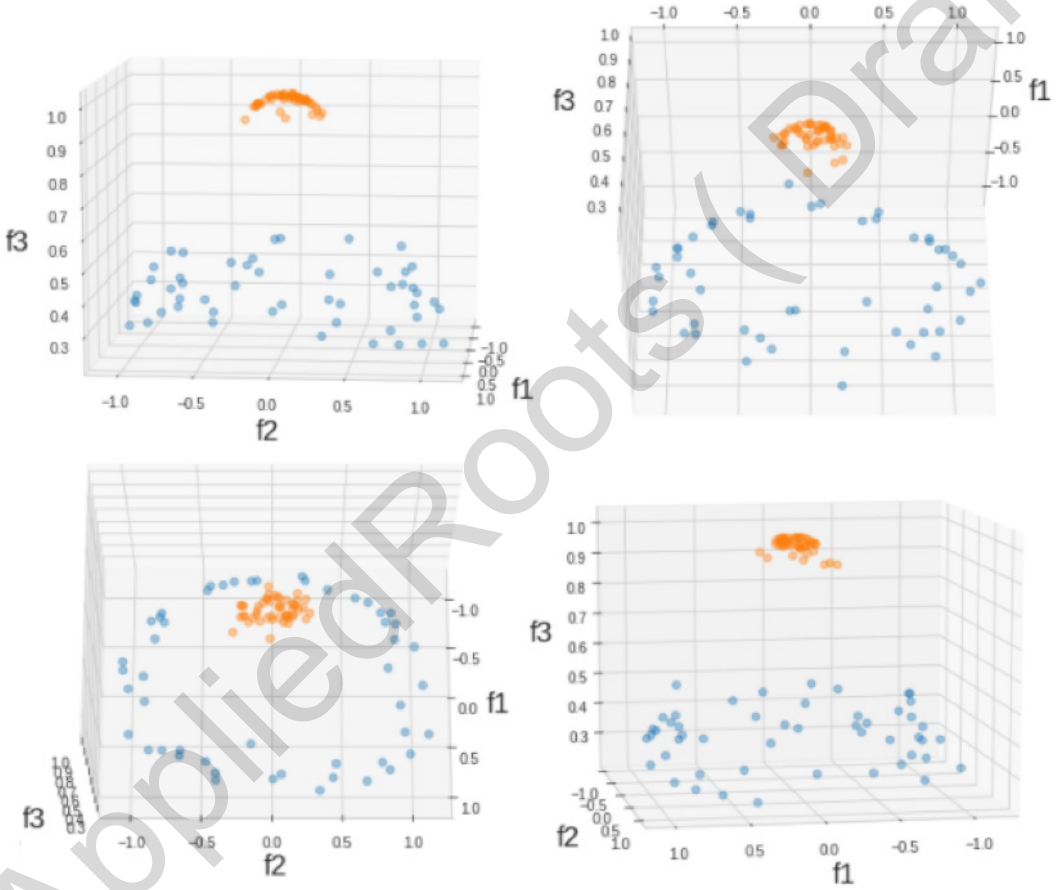
```
import pandas as pd

r = np.exp(-(X ** 2).sum(1))
XX = np.hstack((X, r.reshape(-1, 1)))
df = pd.DataFrame(XX, columns = 'f1 f2 f3'.split()).assign(labels = y)
df.head()
```

Create New feature using Radial Basis Function

	f1	f2	f3	labels
0	-0.724009	0.762198	0.331166	0
1	-0.584109	-0.954364	0.285934	0
2	-0.276611	0.953357	0.373289	0
3	0.003512	-0.141779	0.980087	1
4	0.079290	0.820475	0.506888	0

Plotting the newly engineered data we have:

```
import aaic_mlfuncs as ai
ai.fn_plot_3d_clf(df)
```



From the 3D plots shown above we see that the engineered data is linearly separable by a 2D plane. Using the existing lower dimensional non linearly separable data, we created a new feature using the radial basis function (RBF), thus increasing its dimensionality. This new higher dimensional data is now linearly separable. Thus we see that it is possible to separate non linear data by increasing its dimensionality using some suitable transformation functions like the RBF .

KERNEL FUNCTIONS:

A Kernel is a function that takes in two vectors and outputs the dot product those two vectors would produce, if they existed in a particular higher dimensional space associated with that kernel. In other words, it can be imagined that the Kernel function sequentially does the following:

1. tt first represent the 2 input vectors in a specific higher dimensional space
2. It then computes the dot product of the two vectors in this higher dimensional transformed space and returns it.

Though in actuality this is not the case, a kernel function (K) is a direct mapping between two vectors and their higher dimensional dot product (in some specific higher dimensional space associated with that particular kernel function).

Kernel function

$$K(x_i, x_j) = \varphi(x_i) \cdot \varphi(x_j)$$

where: φ is some transformation function implicitly associated with K (like the RBF)

THE DUAL FORM OF THE SVM CLASSIFIER:

The Dual form of the SVM is the outcome of further modifying the Dual form using advanced mathematical concepts like quadratic solvers and Langragian multipliers, to arrive at a form that expresses the optimization as a function of the dot products of the data points. The Dual form of SVM is shown below.

$$\lambda_i^* = \text{argmax}\left(\sum_{i=1}^{n}\lambda_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} y_i \lambda_i (x_i)\cdot(x_j)\lambda_j y_j\right)$$

$$\text{such that: } \lambda_i >= 0 \text{ for all } i \;\&\; \sum_{i=1}^{n}\lambda_i y_i = 0 \quad \text{where: } y_i = 1 \text{ or } -1$$

The values λ i in the equation shown above are called **Lagrangian Multipliers**. They can be viewed as weights corresponding to each data point, which are optimized for, such that the above expression is maximized. Given that we have found the Lagrangian Multipliers that maximize the above expression, we can then find the weight vector corresponding to the separating hyperplane by using the computation shown below.

$$w = \sum_{i=1}^{n} \lambda_i y_i x_i$$

The mathematical process/steps taken to arrive at the Dual form will not be discussed as it is beyond the scope of this book. Nonetheless, we can still get insights about what the SVM Dual Form equation does, by inspecting its elements. For the Dual Form expression to be maximized, the second term in the expression also has to be maximum. For that to happen the following should be true:

1. The resulting value of the product inside the double summation must be negative, thus making the entire second term positive. In other words **xi & xj** should belong to opposite classes, hence making the product of **yi & yj** negative.
2. The dot product between **xi & xj** must be large. In other words, they should be as similar as possible

Thus it can be said that, for the Dual form to be maximized, the data points **xi & xj** being considered should only be potential support vector points. Thus λi be such that it is zero or almost zero for all points, except for support vector points.

The **kernel trick** basically involves substituting the dot product of **xi & xj** with a suitable kernel function **K (xi, xj)**. Doing this implicitly means that:

1. We have increased the dimensionality of the dataset (by using the particular non linear transformation associated with the kernel function chosen).
2. We have then applied the Dual Form SVM optimization on this transformed higher dimensional dataset.

Thus if the original data is non linear, we can still separate it by using its higher dimensional linear separable counterpart by using the Kernel Trick.

SVM USING SCIKIT LEARN:

The Scikit Learn library provides three SVM implementations which we could use.

1. The **svm.LinearSVC** class
2. The **svm.SVC** class and
3. The **linear_model.SGDClassifier** class.

The second implementation uses the Dual Form optimization and has the capacity to perform the Kernel trick, whereas the other two are linear classifiers. The first classifier implements the Primal form using slack variables, whereas the third classifier implements the hinge loss form, using stochastic gradient descent for optimization.

Shown below are the classes that can perform SVM classification, with some of the more relevant parameters along with their default values displayed.

```
svm.LinearSVC(penalty='l2',
              C=1.0,
              class_weight=None,
              verbose=0,
              random_state=0)
```

```
svm.SVC(C=1.0,
        kernel='rbf',
        probability=True,
        class_weight=None,
        random_state=0)
```

```
linear_model.SGDClassifier(loss='hinge',
                           penalty='l2',
                           alpha=0.0001,
                           random_state=0,
                           class_weight=None)
```

USING SVM ON BREAST CANCER DATA SET:

Since we have discussed the binary classification pipeline in quite some detail in the previous chapters, we will pick up from the "model training and evaluation" stage in the pipeline. We will be using the same set of machine learning Functions saved in a file "**aaic_mlfuncs.py**" .

We import the train and test sets as shown below:

```
import aaic_mlfuncs as ai
```

```
df_tr_set = pd.read_csv(data_path + 'df_tr_BC.csv')
df_ts_set = pd.read_csv(data_path + 'df_ts_BC.csv')

df_tr_set.shape, df_ts_set.shape
```

```
((438, 9), (114, 9))
```

```
X_train = df_tr_set.iloc[:, :-1].values
X_test  = df_ts_set.iloc[:, :-1].values

y_train = df_tr_set.iloc[:, -1].values
y_test  = df_ts_set.iloc[:, -1].values

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((438, 8), (114, 8), (438,), (114,))
```

We Choose to perform grid search with K fold cross validation using the SVC classifier as shown below.

```
from sklearn.svm import SVC

model_class = SVC
parameter_grid = dict(C = np.linspace(0.1, 5, 100),
                      kernel = ['linear', 'rbf', 'poly', 'sigmoid'],
                 probability = [True],
                class_weight = ['balanced'],
                random_state = [0])

z = ai.fn_kfoldcv_clf_binary(X_train, y_train, model_class, parameter_grid)

df_kfoldcv_gridsearch, dict0_model_instances = z
df_kfoldcv_gridsearch.describe()
```

```
100% (400 of 400) |######################| Elapsed Time: 0:00:18 Time:  0:00:18
```

	ts_mean_rec_1	ts_mean_prec_1	ts_mean_rec_0	ts_mean_prec_0	ts_mean_loss	ts_std_loss	ts_mean_acc	ts_std_rec_1	ts_std_rec_0
count	400.000000	400.000000	400.000000	400.000000	400.000000	400.000000	400.000000	400.000000	400.000000
mean	0.984298	0.959945	0.924608	0.971233	0.115350	0.022400	0.963006	0.009956	0.019701
std	0.007592	0.020752	0.041421	0.013220	0.036403	0.011889	0.011003	0.004428	0.010411
min	0.961404	0.902054	0.804064	0.930440	0.067876	0.002107	0.927805	0.004962	0.000342
25%	0.978947	0.936706	0.877842	0.962022	0.078949	0.009085	0.950343	0.004962	0.009070
50%	0.982456	0.968601	0.942912	0.968241	0.101504	0.020065	0.968392	0.008595	0.018141
75%	0.992982	0.975692	0.955733	0.985648	0.140077	0.033684	0.972896	0.013129	0.023635
max	0.996491	0.982418	0.968312	0.993197	0.189509	0.053175	0.979668	0.019849	0.057879

We then filter the best models as shown below:

```
dff = df_kfoldcv_gridsearch
n = 5

df1 = dff.sort_values(by = 'ts_mean_rec_1', ascending = False)[:n]
df2 = dff.sort_values(by = 'ts_mean_prec_1', ascending = False)[:n]
df3 = dff.sort_values(by = 'ts_std_rec_1', ascending = True)[:n]
df4 = dff.sort_values(by = 'ts_mean_loss', ascending = True)[:n]
df5 = dff.sort_values(by = 'ts_std_loss', ascending = False)[:n]

df_filtered_cv = pd.concat([df1, df2, df3, df4, df5]).drop_duplicates()
df_filtered_cv = df_filtered_cv[df_filtered_cv.ts_mean_loss < 0.5]
df_filtered_cv = df_filtered_cv.sort_values(by = 'ts_mean_rec_1', ascending = False)[:5]
df_filtered_cv
```

	ts_mean_rec_1	ts_mean_prec_1	ts_mean_rec_0	ts_mean_prec_0	ts_mean_loss	ts_std_loss	ts_mean_acc	ts_std_rec_1	ts_std_rec_0
model_138	0.996491	0.928278	0.860789	0.993056	0.139251	0.007166	0.948091	0.004962	0.023336
model_202	0.996491	0.931443	0.867078	0.993197	0.137423	0.006537	0.950343	0.004962	0.030834
model_106	0.996491	0.931352	0.867199	0.993056	0.140596	0.005801	0.950359	0.004962	0.026185
model_206	0.996491	0.931443	0.867078	0.993197	0.137304	0.006592	0.950343	0.004962	0.030834
model_230	0.996491	0.928429	0.860668	0.993197	0.136423	0.007715	0.948076	0.004962	0.032475

We can inspect the parameters of these chosen models as shown below:

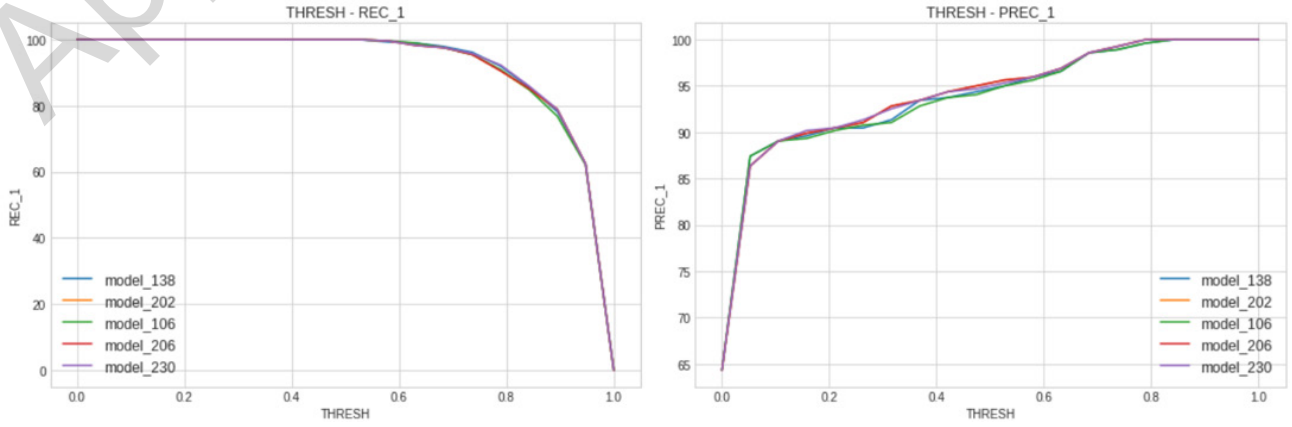
```
models = [dict0_model_instances[i] for i in df_filtered_cv.index]

display(models)
```

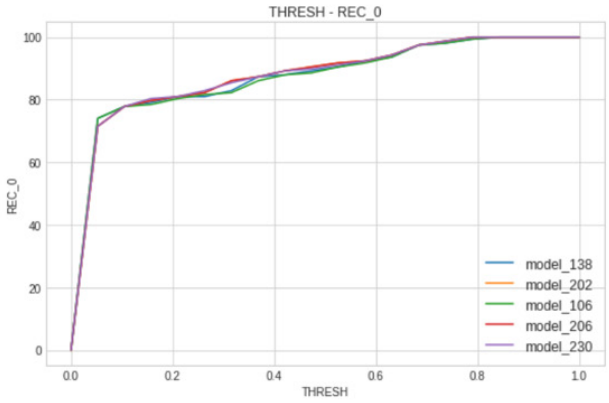
```
[SVC(C=1.782828282828283, break_ties=False, cache_size=200,
     class_weight='balanced', coef0=0.0, decision_function_shape='ovr', degree=3,
     gamma='scale', kernel='poly', max_iter=-1, probability=True, random_state=0,
     shrinking=True, tol=0.001, verbose=False),
 SVC(C=2.574747474747475, break_ties=False, cache_size=200,
     class_weight='balanced', coef0=0.0, decision_function_shape='ovr', degree=3,
     gamma='scale', kernel='poly', max_iter=-1, probability=True, random_state=0,
     shrinking=True, tol=0.001, verbose=False),
 SVC(C=1.386868686868687, break_ties=False, cache_size=200,
     class_weight='balanced', coef0=0.0, decision_function_shape='ovr', degree=3,
     gamma='scale', kernel='poly', max_iter=-1, probability=True, random_state=0,
     shrinking=True, tol=0.001, verbose=False),
 SVC(C=2.6242424242424245, break_ties=False, cache_size=200,
     class_weight='balanced', coef0=0.0, decision_function_shape='ovr', degree=3,
     gamma='scale', kernel='poly', max_iter=-1, probability=True, random_state=0,
     shrinking=True, tol=0.001, verbose=False),
 SVC(C=2.921212121212122, break_ties=False, cache_size=200,
     class_weight='balanced', coef0=0.0, decision_function_shape='ovr', degree=3,
     gamma='scale', kernel='poly', max_iter=-1, probability=True, random_state=0,
     shrinking=True, tol=0.001, verbose=False)]
```

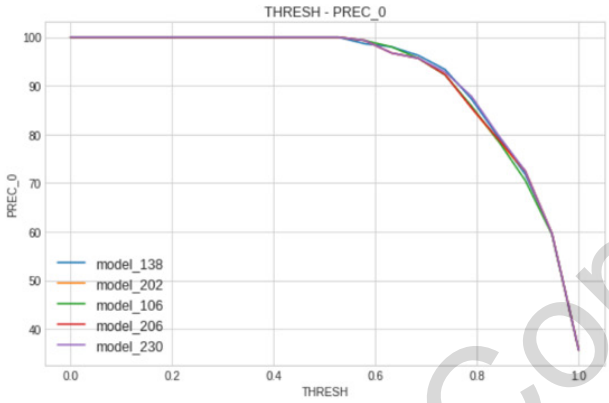
As can be seen from the parameters of the above models, the **kernel** that works the best for this dataset is the polynomial **kernel**.

We then plot the precision - recall curves for the above selected models as shown below:

```
df_tr_standard, df_ts_standard, scaler = ai.fn_standardize_df(df_tr_set, df_ts_set)

list0_model_names = list(df_filtered_cv.index)
X_train, y_train = df_tr_standard.iloc[:, :-1].values, df_tr_standard.iloc[:, -1].values
list0_models = [dict0_model_instances[i].fit(X_train, y_train) for i in list0_model_names]

list0_thresholds = np.linspace(0, 1, 20)
legend = list0_model_names

ai.fn_performance_models_data(list0_models, df_tr_standard, list0_thresholds, legend)
```

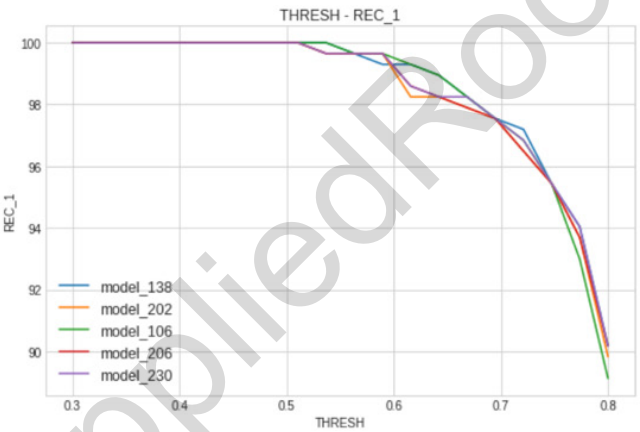


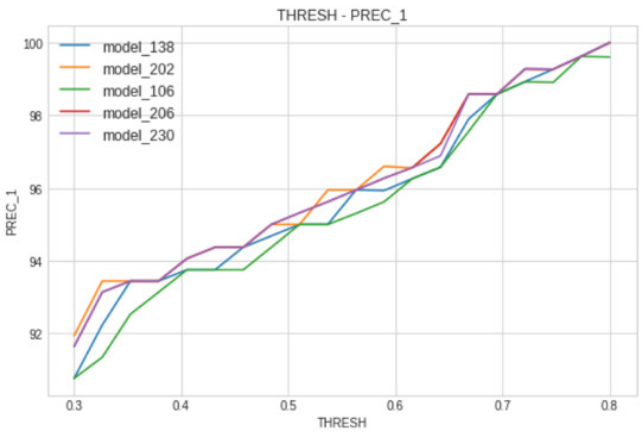


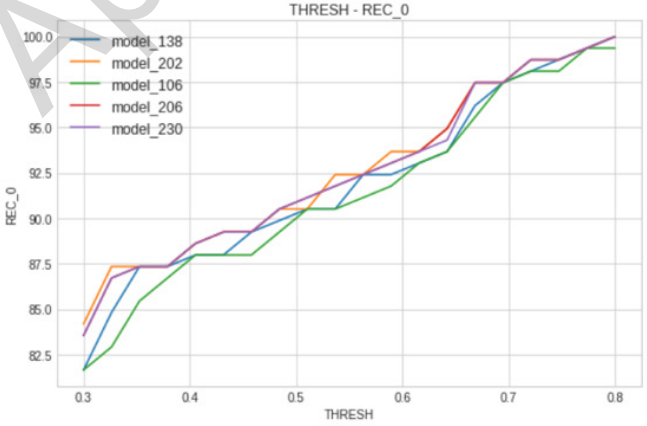


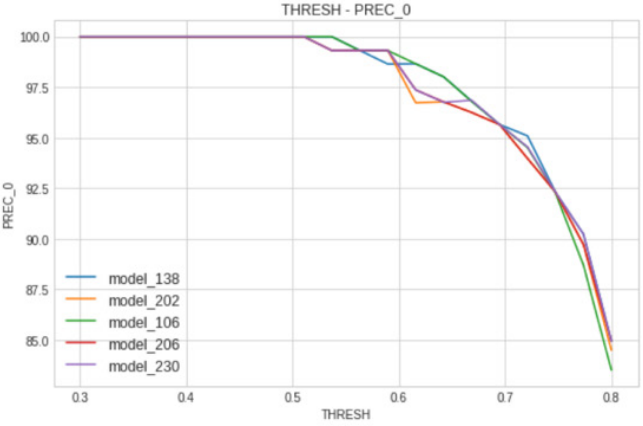
Assuming that we want to optimize for precision, we see that the best threshold lies in the range 0.5 to 0.8. We zoom into that range and look at the precision recall curves in more detail as shown below.

```
df_tr_standard, df_ts_standard, scaler = ai.fn_standardize_df(df_tr_set, df_ts_set)

list0_model_names = list(df_filtered_cv.index)
X_train, y_train = df_tr_standard.iloc[:, :-1].values, df_tr_standard.iloc[:, -1].values
list0_models = [dict0_model_instances[i].fit(X_train, y_train) for i in list0_model_names]

list0_thresholds = np.linspace(0.3, 0.8, 20)
legend = list0_model_names

ai.fn_performance_models_data(list0_models, df_tr_standard, list0_thresholds, legend)
```









From the above plots we see that model_230 thresholded at around 0.75, will most likely give best precision performance, while at the same time keeping all other metrics as high as possible. We then train the model_230 over the entire train set and check its performance across the train and test sets as shown below:

```
df_Xy_ = df_tr_standard
model_ = dict0_model_instances['model_230'].fit(X_train, y_train)
thresh = 0.75

ai.fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.0997
ACCURACY: 96.614
--------------------
```

	prec	rec
class_0	92.308	98.734
class_1	99.270	95.439

```
df_Xy_ = df_ts_standard

ai.fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.0858
ACCURACY: 96.491
--------------------
```

	prec	rec
class_0	91.304	100.000
class_1	100.000	94.444

From the performances indicated in the outputs above we see that the model gives a good performance and it also generalizes well to the test set.

ADVANTAGES AND LIMITATIONS OF SVMs:

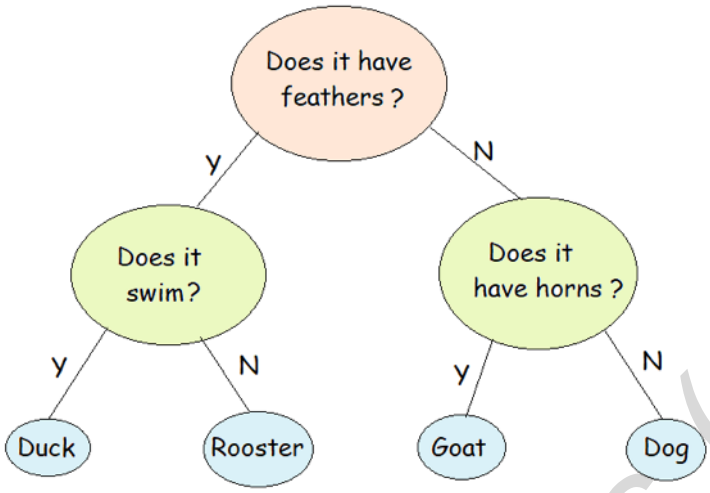
Support vector machines have the capacity to work on non linearly separable data due to the usage of the kernel trick. They work well on datasets that are high dimensional and even in situations where the number of data points is smaller than its dimensionality.

The computational requirement of the dual form kernelized implementation is quadratic in nature, since it has to compute the dot product of all the data points with each other. Thus kernelized SVMs do not scale well and become infeasible for datasets containing more than 30 - 40 thousand samples. In case of large datasets it is more practical to use the stochastic gradient descent based linear SVMs.

13. DECISION TREES

DECISION TREES

Decision Trees are machine learning models that learn a hierarchy of if-else questions that eventually lead to a decision. Consider the hierarchy of questions shown in the image below. This hierarchy can be used to classify any group containing four types of animals: Dog, goat, rooster & duck.



Each question leads to a yes/no (binary) split. Each of the two groups formed is then **recursively** subjected to such critical if-else questions, till we eventually reach splits that completely classify the data. Such an hierarchical structure is called a **Decision Tree**. The questions/conditions used to split the data are called **splitting conditions**. The top most node representing all the data is called the **root node**. The sub groups formed after each split are called **branch nodes** of the decision tree and the last (or terminal) branches of the decision tree are called **leaf nodes**. The total number of "levels" in the tree's hierarchy is called its **depth**.

In the example above, the data consists of three categorical features: Feathers, Swim and Horns (3D). Each data point (x_i) corresponds to a value/class (y_i) in the target variable/ (labels column), in this case: Duck, Rooster, Goat and Dog.

Decision trees work by basically dividing the features space using axes parallel separation boundaries as previously mentioned while discussing KD Trees in the chapter on KNNs.

CLASSIFICATION & REGRESSION TREES (CART):

"**Learning**" in the case of Decision Trees essentially involves discovering the right splitting conditions for each branch at every level (depth) of the tree's hierarchy. The strategies used for classification and regression to find the best splitting conditions are slightly different but are similar in essence.

SPLITTING STRATEGY FOR CLASSIFICATION :

Decision Trees use the concept of Entropy for choosing the best splitting conditions.eEntropy as discussed in the chapter on Multiclass Logistic regression, is basically a measure of the amount of randomness (lack of information/predictability) present within a random variable. Higher the randomness, more the entropy. Given a random variable y containing k classes/states, its entropy is defined as shown below.

$$H(y) = -\sum_{\text{for every class in y}} P(\text{class}) \times \log_2 P(\text{class}) \text{ ---Entropy}$$

P: Probability
H: Entropy

Consider the example shown below, where we compute the entropy (H) of three random variables y1, y2 & y3, each purposefully sampled from a population that contains two classes, such that they have the following proportions of 90:10, 50:50, 100: 0.

y1: [90% class_1, 10% class_2]
y2: [50% class_1, 50% class_2]
y3: [100% class_1, 0% class_2]

$$H(y1) = -(0.9 \times \log(0.9) + 0.1 \times \log(0.1)) = 0.46$$
$$H(y2) = -(0.5 \times \log(0.5) + 0.5 \times \log(0.5)) = 1$$
$$H(y3) = -(1 \times \log(1) + 0 \times \log(0)) = 0$$

As can be seen from the entropies of the three random variables, the more predictable the classes of the random variables are, the lesser is the entropy. The classes of the data points in random variable **y3** are completely predictable and hence it has zero entropy. In contrast, the classes of the data points in random variable **y2** are completely unpredictable and hence it has max entropy possible (ie: 1).

CLASSIFICATION FOR SINGLE FEATURE DATASET (1D):

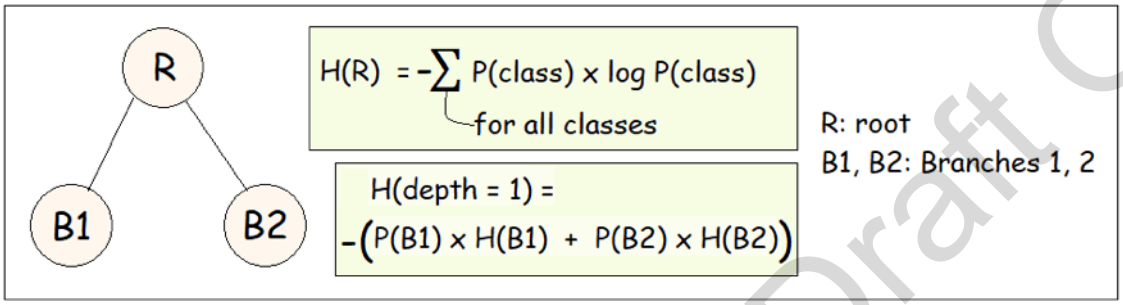
Consider a dataset {X, y} where X has only one feature **f1** that is numerical in nature. Say we split the data using one of the values **v_i** in the **f1** (ie: all data points that have their **f1 value >= v_i** are collected in one branch and the rest in the other branch).

If the collective entropy of the target variable y in the two branches is lesser than the entropy of y at the root (ie: the target variable y in its unsplit condition), then it means that the resulting split has produced more order (or reduced the randomness of y). This is because the classes in each node individually are now more predictable (greater probability of one unique class in each branch).

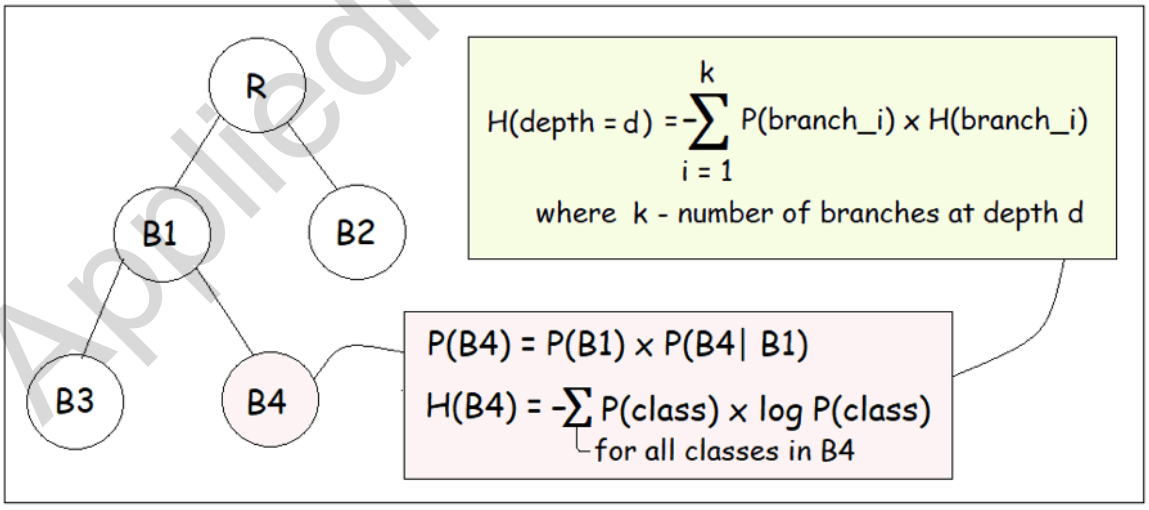
We find the beast split possible by following the steps described below:

1. We use each value v_i in the f1 as a splitting condition for the dataset (ie: is v >= v_i for v in f1). Thus we obtain two branches for every value v_i in the f1 used as the splitting condition.
2. The splitting condition (among all the splitting conditions) that results in branches that produce the least entropy is chosen as the splitting condition to use.

The image below describes the computation involved to calculate the collective entropy of branches at the first split (ie: Depth = 1)



3. Now if this process to find the best split described above, is recursively applied to each resulting branch (such that the two child nodes have lower entropy compared to its parent node), the tree's depth would keep growing, where each successive depth represents a grouping of the data such that the collective entropy of the target variable y is lesser than what it was at the depth/level preceding it.
4. The tree would stop growing in depth when all the resulting end nodes/ branches would contain only homogeneous data (ie: each leaf will contain only one of the two classes). The basic computation involved in such a process is described in the image below.



CLASSIFICATION FOR MULTI FEATURE DATASET:

The process for finding the splitting condition for multidimensional data remains the same except for one minor modification. First we find the best splitting condition with respect to each feature. Then we choose that feature that produces the best splitting condition with respect to all other features. This is performed recursively as before for all resulting branches until we reach a depth where all branches are pure (homogeneous).

SPLITTING STRATEGY FOR REGRESSION:

In case of regression, we use the mean squared error metric for choosing the right splitting condition.The values of the target variable **y** for regression will be continuous numerical values. Consider the case of a dataset {X, y} consisting of only one feature **f1**.

1. Let's say that we have split the original data into two branches based on some value **v_i** in **f1** (ie: all data points that have their **f1 value >= v_i** are collected in one branch and the rest in the other branch).
2. Then the prediction **y_i** corresponding to any data point **x_i** will be the mean value of the target variables in the branch that **x_i** belongs to.
3. The best splitting condition will be that which produces the least mean squared error for the entire data.
4. In case of multidimensional data, just as in the case for classification, we choose that splitting condition among all features that produces the best mean squared error value.
5. The splitting process described above is recursively applied to each resulting branch until we end up with leaves (end nodes) containing just one value.
6. Each data point when passed through this tree will eventually reach a leaf with a single value, which will be the same as its corresponding target value.

TREE PRUNING (REGULARIZATION):

In the above discussions on classification and regression using decision trees, we saw that if the decision tree is allowed to progress in depth without any constraints, it will eventually reach a stage where:

1. In the case of classification - The end nodes/leaves will be pure (ie: they will contain data points of only one particular class).
2. In the case of regression - The end nodes/leaves will contain only one value,

This means that the Decision tree has overfit/memorized the training data. To avoid overfitting tree pruning strategies are used. Some of the relevant ones being:

1. Limiting the maximum depth possible
2. Limiting the maximum number of leaves possible
3. Requiring a minimum number of data points in a node for it to be subject to further splitting.

By using the conditions outlined above we can stop the tree from growing all the way till the end and instead stop it when one of the above conditions is broken, thus preventing overfitting. This results in the following situations:

1. In case of classification we will end up with leaves/end nodes that may not be pure. Then each leave is assigned the class of the majority of the data points **x_i** in that node.
2. In case of regression we will end up with leaves/end nodes containing multiple data points (ie: multiple y values). Each leaf is then assigned the average **y** value of the data points **x_i** belonging to it.

DECISION TREE USING SCIKIT LEARN:

Shown below is the scikit learn class that performs classification using a decision tree. Some of the more relevant parameters along with their default values are also displayed.

```
DecisionTreeClassifier(criterion = 'gini,
                       max_depth = None,
                       min_samples_split = 2,
                       min_samples_leaf = 1)
```

CRITERION: Two options are provided by scikit learn as criteria for best splitting condition. Entropy and Gini impurity. Scikit learn uses the latter as its default choice. Gini Impurity is defined as shown below.

$$\text{gini impurity} = 1 - \sum_{\text{for all classes in the node}} P(\text{class})^2$$

Gini impurity is a monotonously increasing function with respect to entropy and hence it can be used as an alternative to entropy. This is preferred because gini impurity is computationally less intensive due the avoidance of log computations required for entropy.

MAX_DEPTH: This parameter refers to the maximum depth allowed for the tree. Scikit learn provides a default value of "None", which means the tree is allowed to grow in depth until all its end nodes or leaves are pure.

MIN_SAMPLES_SPLIT: This parameter refers to the minimum number of samples required in a branch for it to be considered viable for further splitting. Scikit learn provides a default value of two, which is the minimum possible.

MIN_SAMPLES_LEAF: This parameter refers to the minimum number of samples that each leaf should contain. Scikit learn provides a default value of one, which is the minimum possible.

USING DECISION TREE ON BREAST CANCER DATA SET:

Since we have discussed the binary classification pipeline in quite some detail in the previous chapters, we will pick up from the "model training and evaluation" stage in the pipeline. We will be using the same set of machine learning Functions saved in a file "**aaic_mlfuncs.py**" .

We import the train and test sets as shown below:

```
import aaic_mlfuncs as ai
```

```
df_tr_set = pd.read_csv(data_path + 'df_tr_BC.csv')
df_ts_set = pd.read_csv(data_path + 'df_ts_BC.csv')

df_tr_set.shape, df_ts_set.shape
```

```
((438, 9), (114, 9))
```

```
X_train = df_tr_set.iloc[:, :-1].values
X_test  = df_ts_set.iloc[:, :-1].values

y_train = df_tr_set.iloc[:, -1].values
y_test  = df_ts_set.iloc[:, -1].values

X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

```
((438, 8), (114, 8), (438,), (114,))
```

We Choose to perform grid search with K fold cross validation using the **DecisonTreeClassifier** class as shown below (using three depths : 3, 5 and None).

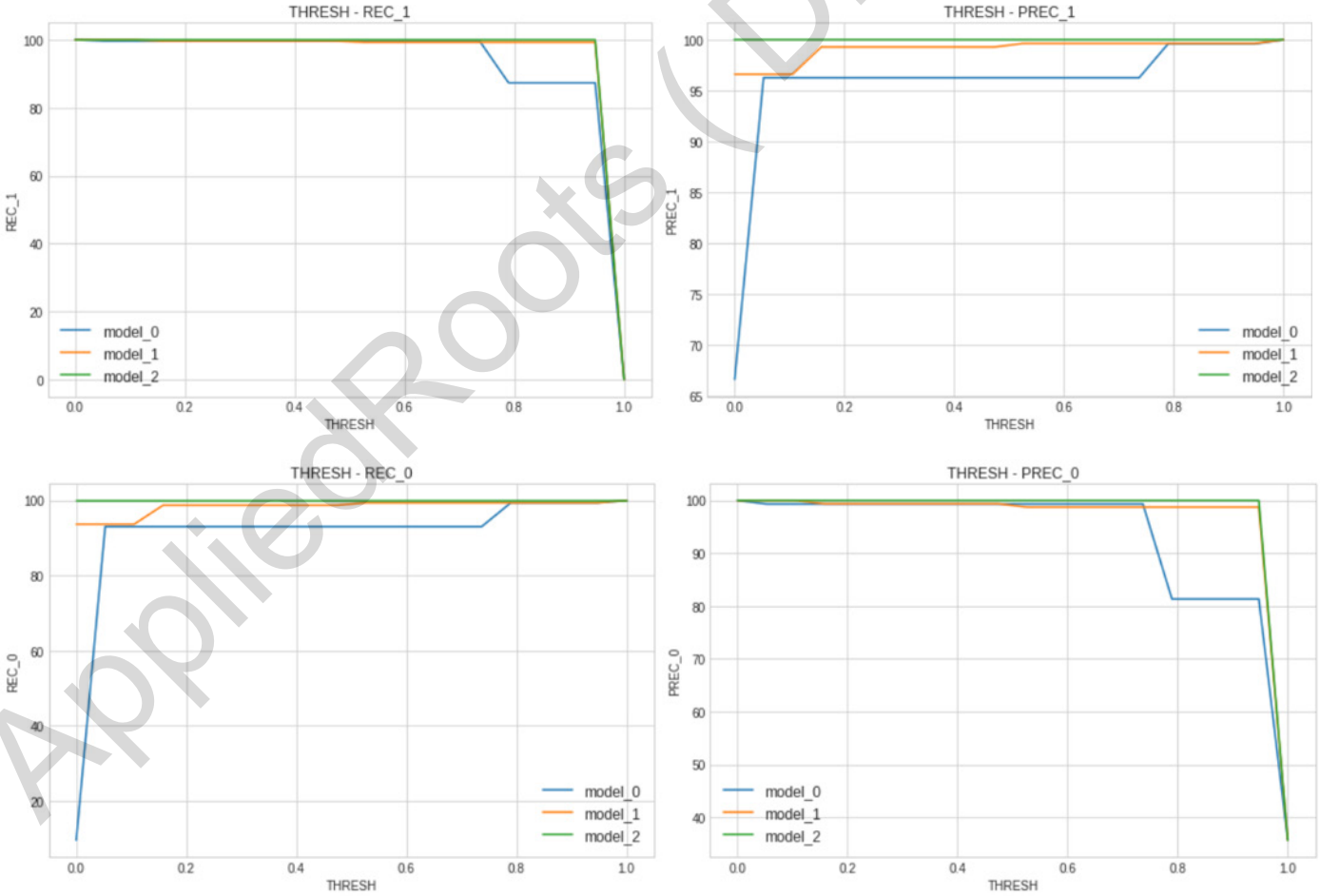
```
from sklearn.tree import DecisionTreeClassifier

model_class = DecisionTreeClassifier

parameter_grid = dict(max_depth = [3, 5, None])

z = ai.fn_kfoldcv_clf_binary(X_train, y_train, model_class, parameter_grid)

df_kfoldcv_gridsearch, dict0_model_instances = z
df_kfoldcv_gridsearch
```

```
100% (3 of 3) |########################| Elapsed Time: 0:00:00 Time:  0:00:00
```

	ts_mean_rec_1	ts_mean_prec_1	ts_mean_rec_0	ts_mean_prec_0	ts_mean_loss	ts_std_loss	ts_mean_acc	ts_std_rec_1	ts_std_rec_0
model_0	0.975439	0.932963	0.873367	0.951389	1.211920	0.672788	0.939036	0.017891	0.038925
model_1	0.943860	0.950570	0.911466	0.902371	1.894300	0.865404	0.932233	0.034735	0.031886
model_2	0.957895	0.948504	0.905298	0.924320	2.105623	1.062104	0.939036	0.030989	0.053112

We then plot the precision - recall curves for each of the three models as shown below:

```
df_tr_standard, df_ts_standard, scaler = ai.fn_standardize_df(df_tr_set, df_ts_set)

listO_model_names = list(df_kfoldcv_gridsearch.index)
X_train, y_train = df_tr_standard.iloc[:, :-1].values, df_tr_standard.iloc[:, -1].values
listO_models = [dictO_model_instances[i].fit(X_train, y_train) for i in listO_model_names]

listO_thresholds = np.linspace(0, 1, 20)
legend = listO_model_names

ai.fn_performance_models_data(listO_models, df_tr_standard, listO_thresholds, legend)
```



From the adobe plots it seems that **model_2** (ie: depth = None) performs the best and so we choose it for our prediction purposes. We then train the **model_2** over the entire train set and check its performance across the train and test sets as shown below:

```
df_Xy_ = df_tr_standard
model_ = dict0_model_instances['model_2'].fit(X_train, y_train)
thresh = 0.5

ai.fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 0.0
ACCURACY: 100.0
--------------------
```

	prec	rec
class_0	100.0	100.0
class_1	100.0	100.0

```
df_Xy_ = df_ts_standard

ai.fn_test_model_binary_clf(df_Xy_, model_, threshold_class_1 = thresh)
```

```
--------------------
LOGLOSS : 1.5149
ACCURACY: 95.614
--------------------
```

	prec	rec
class_0	95.122	92.857
class_1	95.890	97.222

As can be seen from the performances above, the model overfits the train set and does not generalize well to the testset, proving that when decision trees are allowed to grow as deep as possible, they memorize/overfit the train set. A model with lesser depth would give us better generalization, but we might lose out on performance.

ADVANTAGES AND LIMITATIONS OF DECISION TREES:

Decision Trees are simple to understand and interpret. They are completely unaffected by the scale of the individual features and so do not require feature scaling.

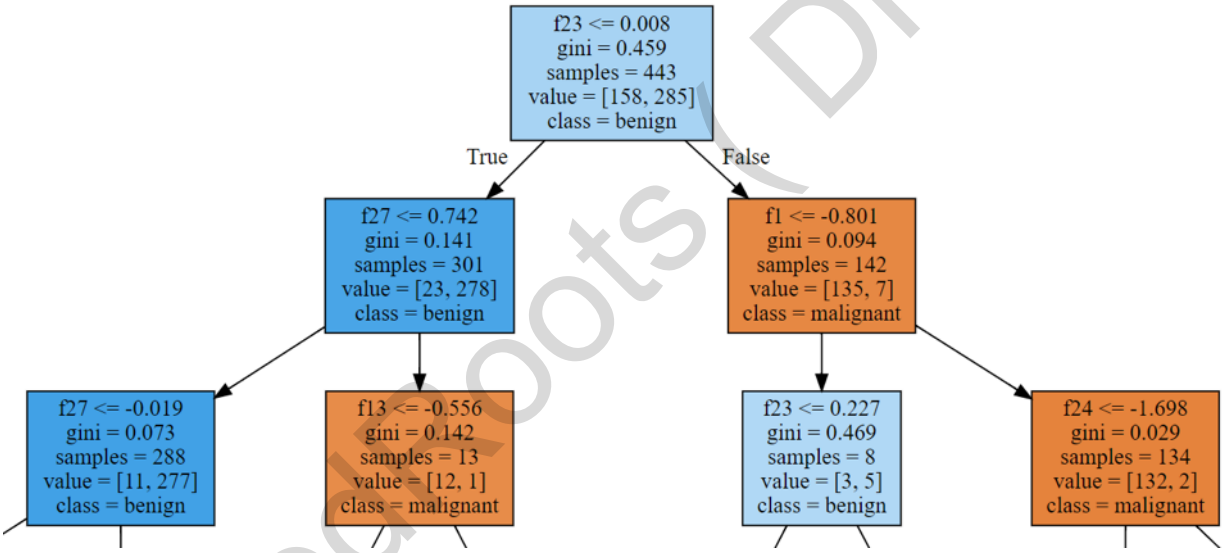
On the down side, Decision Trees cannot extrapolate to data outside of the feature ranges it was trained on. Even with pruning techniques discussed, they tend to overfit, resulting in poor generalization performance when compared to other models.

VISUALIZING DECISION TREES:

Shown below is the code used to visualize decision Trees.

```
import graphviz
from sklearn.tree import export_graphviz

export_graphviz(model_, out_file = data_path + 'BC_tree.dot',
                class_names = ['malignant', 'benign'],
                feature_names = df_tr_set.columns[:-1],
                filled = True)
```

```
with open(data_path + 'BC_tree.dot') as f:
    graph = f.read()

graphviz.Source(graph)
```



The above image is a visualization of the first few splits in the decision tree used for breast cancer predictions. Information about the splitting condition, the randomness measure, the class distribution and majority class is displayed for each node.

14. ENCODING CATEGORICAL VARIABLES

ENCODING CATEGORICAL VARIABLES

Non numerical random variables representing some categorical feature (of the subject on which the data is based) are called categorical variables. For instance in a dataset containing the heights, weights and gender information, "**gender**" is a categorical feature representing two categories - male and female. Suppose the same dataset had a feature called "**eye colors**", that would be another example of a categorical random variable.

Categorical random variables are **non-ordinal**. In other words, the values within such a random variable do not have any sequence or order with respect to each other. They are non numerical as opposed to numerical features like height, weight or age, where sequence and magnitude exist.

Categorical variables too contain information and so it could be used while training machine learning models. Since all information has to be represented in some numerical form for it to be "computable", various encoding strategies have been developed for encoding categorical variables. Some of the most relevant ones are discussed below.

ONE HOT ENCODING:

In one hot encoding, we represent a categorical variable as a group of binary variables, where each binary variable represents either the presence (1) or absence (0) of one particular category. In the table shown below, we have represented a categorical variable "Eye color" (first column) in one hot encoded form (columns 2,3 & 4). Thus in such a representation, n dimensional binary vectors are used to represent each of the n categories within the categorical variable. So, black is represented by the binary vector [1, 0, 0], brown by [0, 1, 0] and blue by [0, 0, 1].

Eye color	Black	Brown	Blue
Black	1	0	0
Black	1	0	0
Brown	0	1	0
Blue	0	0	1

DUMMY ENCODING:

The one hot encoding technique contains redundancy because we need only n-1 dimensions to binary encode n categories. Thus dropping any one of the columns in the above table, we can still represent all the categories perfectly. This is called dummy encoding, it is demonstrated in the table shown below. The category whose column is dropped is called the "**reference category**" and is represented by a zeros vector.

Eye color	Black	Brown
Black	1	0
Black	1	0
Brown	0	1
Blue	0	0

EFFECT ENCODING:

Effect encoding is similar to dummy encoding except that the reference category is represented by a vector of negative ones instead of zeroes. This is demonstrated in the table shown below.

Eye color	Black	Brown
Black	1	0
Black	1	0
Brown	0	1
Blue	-1	-1

ENCODING LARGE CATEGORICAL VARIABLES:

The encoding techniques discussed previously created a new column for every category that existed in the categorical variable. These methods are feasible in situations where the number of categories are not too large. Sometimes we have to deal with categorical random variables that have a large number of categories. Consider the example of location zip codes used as a feature in some dataset dealing with cities. There can be hundreds of zip codes given a country. Though zip codes can be numerical, they are actually categorical values since they do not contain any hierarchy or sequence. Representing such a random variable using the previous encoding techniques would increase the dimensionality of the dataset considerably. The following techniques could be used to handle such situations.

BIN COUNTING:

This is a frequency based encoding technique, where the categories are replaced with the count or percentage values of that category being considered. Thus in the case of the eye color categorical random variable, if the percentage of black, brown and blue is 60%, 35% and 5%, they are encoded using these values (ie: Black = 0.6, Brown = 0.35 and Blue = 0.5)

ENCODING USING TARGET MEAN:

This encoding technique is applicable only to regression situations (ie: labels are continuous values). Here each category is replaced by the mean of the label values corresponding to all data points with the same category.

$$\text{category} = \frac{1}{n} \sum_{\text{For labels belonging to category}} \text{label\_values}$$

ENCODING USING LABEL BASED RANKING:

This encoding technique like the previous one is applicable only to regression situations Rank base categorical encoding consists of the following steps:

1. Compute the mean value of the target for each category in the categorical random variable.
2. Order the categories in ascending order based on their corresponding mean values.
3. Rank the categories using numerical random variables starting from 0 to k minus one, where k is the number of categories.

Thus each category is replaced by a numerical value that indicates its “rank”.

PROBABILISTIC ENCODING:

In this technique, we encode the categories within a Categorical variable using numerical values based on conditional probability. Given a particular category we encode it with the conditional probability of the positive class given that particular category (zip code).

$$\text{category} = P(\text{positive class} | \text{category})$$

All categories within the categorical variable will be represented by some number between zero and one, based on the conditional probabilities computed as shown in the image above.

Using the techniques discussed above each categorical feature is represented by a numerical value and hence the number of features in the dataset remains unchanged.

15. TIME AND SPACE COMPLEXITY

TIME AND SPACE COMPLEXITY

Time and space complexity are metrics used to measure the performance of algorithms / Machine learning models with respect to computation time and memory consumption. The faster a model consumes time or space with the increase in the size of the data being processed, the higher its complexity.

Complexity Is not an exact quantitative measurement, instead it is meant to provide a **qualitative** understanding of:

1. How quickly a model can perform its task in relation to the scale of the inputs (ie: in relation to size of the data being processed). As the scale of the inputs increases, the model's time complexity tells us at what rate it will slow down (ie: At what rate the computation time required increases). This can help us reason about the feasibility of using a particular model when faced with large or constantly increasing data sizes.
2. How much storage capacity will be needed to store/save the model after it has been trained. This generally refers to the memory required to store the model's learned parameters.

BIG "O" NOTATION:

The big O notation is a convention used to rate models/algorithms based on how their run time or space requirements grow as the input size grows. This is also known as algorithm complexity. The letter "O" is used to refer to the "order of magnitude" of the train time, run time or space (memory) requirement of the model/algorithm.

Consider the case of the **Logistic Regression** algorithm. Here the basic operation is that of finding the distance of the data points from the separating hyperplane (ie: the dot product between the weight vector representing a particular hyperplane and a data point). If the size of the dataset is **n** and dimensionality is **d**, then this would mean that, the time required to train the algorithm (using gradient based optimization) will be **in the order of** (ie: proportional to) **n x d x E** where **E** is the number of epochs/iterations used to run gradient descent on the dataset. Since generally d & E are very small in comparison to **n**, we represent the **train time** complexity of Logistic Regression in big "O" notation as **O(n).**

Similarly the time required to predict on n new data points (run time complexity) using the algorithm will also be in **O(n)**.

The space complexity of a logistic regression model would be **O(d)**, since we only need to store the weight vector corresponding to the optimized hyperplane for further predictions.

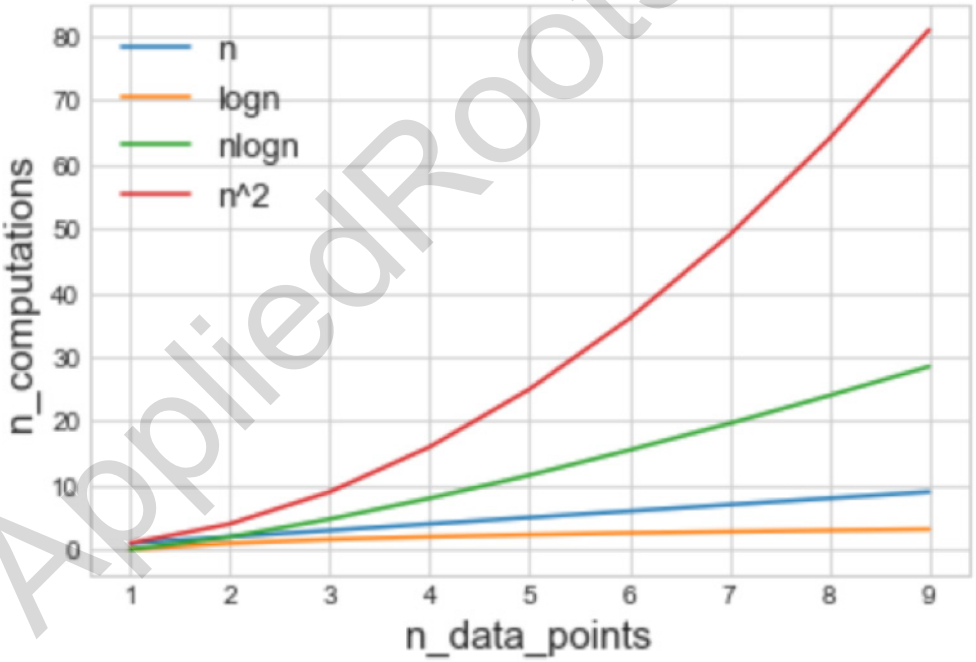
COMMON COMPUTATIONAL COMPLEXITIES:

Some of the most commonly encountered time/computational complexities are:

1. Linear complexity **O(n)**.
2. Quadratic complexity **O(n^2)**.
3. Log complexity **O(log n)**.
4. N log n complexity **O(nlogn)**.

Shown below is a plot that shows how the computational requirement of models having each of these complexities increases with increase in data size.

```
n = np.arange(1, 10)

plt.plot(n, n, label = 'n')
plt.plot(n, np.log2(n), label = 'logn')
plt.plot(n, n*np.log2(n), label = 'nlogn')
plt.plot(n, n**2, label = 'n^2')
plt.legend(fontsize = 13)
plt.xlabel('n_data_points', fontsize = 15)
plt.ylabel('n_computations', fontsize = 15)
plt.show()
```



As can be seen from the above plot models/algorithm with quadratic complexity (**ie: O(n^2))** become more and more infeasible as the size of the dataset increases. For 10,000 data points the order of computations required will be 10,00,00,000. This kind of complexity is associated with the dual form SVM implementation.

O(logn) complexity is associated with search algorithms like binary search or with the **run time** requirements of tree based algorithms like decision trees. It indicates a very low computation complexity

O(nlogn) complexity is associated with the **train time** requirements of tree based algorithms like decision trees.

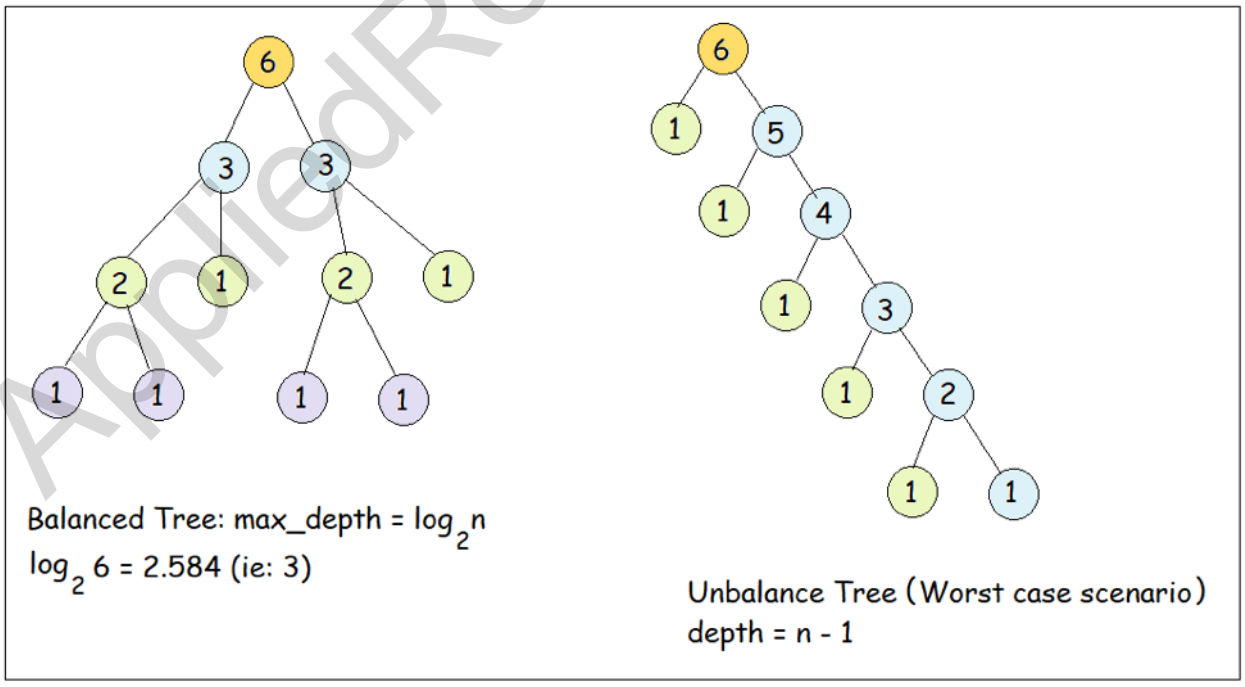
Linear complexity (ie: **O(n)**) is associated with the train time requirements of linear models like Logistic or Linear regression. Here the amount of time required grows proportionally to the size of the input data.

TIME & SPACE COMPLEXITIES OF SUPERVISED ML MODELS:

DECISION TREE BASED MODELS:

The basic computation in the case of Decision Trees is the Entropy or Gini Impurity calculation for each node (In case of Logistic Regression it is the dot product between the weight vector and a data point).

Shown below is the best and worst case scenarios for a decision tree depth based on six data points.



If the distribution of the features and the labels in the dataset is such that it results in a balanced tree, the maximum depth of the tree will be **log n**. If on the other hand the tree is highly imbalanced, then the worst case scenario would result in a tree of depth **n-1** (approximately n).

Since at each depth level, the decision tree has to compute entropy n times (ie: using each datapoint as its splitting condition), the train time complexity of a decision tree would be between **O(nlogn)** and **O(n^2)**. Generally it is closer to **O(nlogn)**.

The predict or run time complexity for decision trees will be between **O(logn)** and **O(n)**, since every new data point will pass through a path of decision nodes till it reaches the leaves. The max length of these paths will be the same as the depth of the tree.

The space complexity of decision tree based models is simply the number of nodes or splitting conditions it to be stored (excluding the end nodes or leaves).The maximum number of nodes possible in a binary tree is **2^(d+1) - 1**, where d is the depth of the tree. Hence the space complexity of a decision tree can be said to be : **O(2^(d+1))**

K NEAREST NEIGHBOURS:

The KNN model does not actually perform any training. Given a query point, all it does is - it refers to the training set and finds K nearest points to the query point within the train set and returns the majority class among the K nearest points as the class of the query point.

To improve the predict or run time performance of the model, generally a partitioning algorithm like the KD tree is used. The basic task while creating a KD tree is sorting the data points. A sorting algorithm like quicksort has a computational complexity of **O(nlogn)**. This sorting operation is performed once and is maintained while building the KDtree. At each node we split the data based on the median value of one of the datastet's features. This computation is negligible compared to the sorting operation and hence can be ignored. Thus the train time and runtime complexity of KNN models can be said to be **O(nlogn)** and since the maximum depth of a KD tree is log n, the run time complexity of a KNN classifier is **O(logn)**.

The space complexity of the KNN model is the same as that of decision trees.

SUPPORT VECTOR MACHINES:

THe Dual Formulation of SVMs have a train time complexity of **O(n^2)**. This is because it has to compute the dot product of all possible pairs of data points in the train set. The runtime and space time complexity of SVMs is the same as that of Logistic Regression. This is because once the hyperplane with the maximum margin is found, we use its weight vector (and margin size) to make predictions just as in the case of logistic regression.

NAIVE BAYES:

The train time complexity of the Multinomial Naive Bayes model is **O(n)**. This is because it computes for every feature, the conditional probability with respect to all the classes. In essence this would involve **d x n** computations, but since we express complexity as "in the order of", we say its train time complexity is **O(n).** The run time complexity of the NB model is also **O(n)**.

The number of probability values required to be stored for NB models is **d x k,** where d is the number of dimensions/features and k in the number of classes. So the space complexity of NB models can be said to be **O(d)**.