MAGNETIC RESONANCE IMAGING CLASSIFICATION

Using SVM and Logistic Regression

~Manasha A

Performing image classification on a dataset of brain tumour images using the scikit-learn library to train and evaluate two classification models. Namely, Logistic regression and support vector machine (SVM). Classification of MRI scans to verify whether they show any sign of brain tumour or not. We are also going to be demonstrating the process of training and evaluating classification models along with visualisation to display images and their predicted classes. The main components of tumour classification are preprocessing the acquired dataset, splitting them into testing and training datasets followed by training and evaluation.

Let us take a look at the classification models and datasets to better understand the process of tumour detection.

Table of Contents

1. Support Vector machine (SVM) 2
   1.1 Loss Function/ Hinge loss: 3
   1.2 Cost function: 3
   1.3 Training SVM 4
   1.3.1 Prediction 4
   1.4 Code Explanation of SVM algorithm implementation 4
   1.4.1 Output: 7
2. LOGISTIC REGRESSION: 8
   2.1 What is Logistic regression 8
   2.3 Training: 10
   2.4 Testing: 10
   2.5 Code Explanation: 10
   2.5.1 Output: 12
3. Classification of MRI scans 14
   3.1 Dataset for MRI tumour classification: 14
   3.2 Steps involved in implementing the algorithm: 15
   3.3 Code explanation for MRI classification: 15
4. Conclusion 20

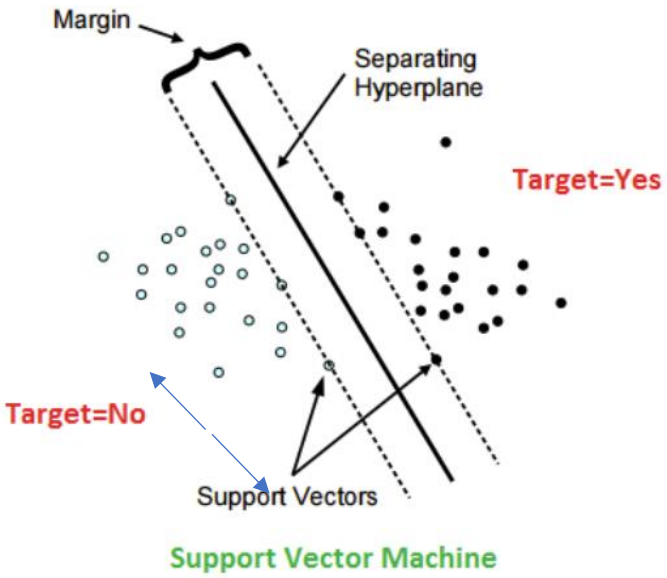
1. Support Vector machine (SVM)

Let us assume that it is even hard for us to distinctly spot the difference between a cat and a dog, then we can only imagine it will be harder for our computers to spot them. Therefore, we use a frontier which best segregates two classes by a hyper-plane and tell their difference.

Use a linear model and try to find a linear decision boundary(hyperplane) that best separates the data. The best hyperplane is the one that yields the largest separation/margin between both classes. So, we choose the hyperplane so that the distance from it to the nearest data point on each side is maximized.

The hyperplane/ line segregating the datasets chosen are said to be equidistant from the closest data values from the distinct datasets, the hyperplane should also have the largest margin between the 2 classes. The support vectors are the data points that are closest to the separating hyperplane; these points are on the boundary of the slab. SVM works by mapping data to a high dimensional feature space so that data points can be categorised, even when the data are not otherwise linearly separable (This is done by kernel function of SVM classifier).

When classifying datasets, unoptimized decision boundary could result in greater misclassifications of the data. Hence, only support vectors are important and all other training examples can be ignored.



The nearest points to the hyperplane are called the support vectors, these are the most difficult data points to classify and they have direct impact on the optimum location of the decision surface. In the above image, the data points which are highlighted are the support vectors of this training set. The length of the margin is given by $2/\omega$ where ω is the weight. If yi = 1, i.e., we take into account the blue dataset then ω.xi – b >= 1 otherwise if yi = -1, i.e., we take into account the green dataset then ω.xi – b <= 1

H1 and H2 are the planes such that H1.ω. xi – b = +1 and H2.ω. xi – b = -1 respectively. The red plane in the middle is the median H0 where ω.xi – b =0.

d+ = shortest distance to the closest positive point

d- = shortest distance to the closest negative point

margin = d+ + d-

Finding the optimal hyperplane is an optimisation problem and can be solved using optimisation techniques (by using Lagrange multiplier). We know that the distance between H0 and H1 is given by Ax+By+c = 0 => |Ax0 +By0 +c| / sqrt(A^2 +B^2), so,

|ω.xi + b| / ||ω|| = 1/||ω|| (since, ω.xi + b = 1)

⇨ The distance between H1 and H2 is 2/||ω|| aka margin

In order to maximise the margin, we thus need to minimise ||ω||. With the condition that there are no data points between H1 and H2:

yi = 1 if ω.xi – b >= 1

yi = -1 if ω.xi – b <= 1

can be combined into yi(xi, ω) >=1 ,in this combined single inequality form we aim to find a hyperplane defined by 'ω' and 'b' that separates the data points with the margin of at least 1. The optimisation now becomes minimising ||ω|| subject to the constraint yi(xi, ω) >=1 for all data points.

By solving this optimization problem, we can obtain the optimal weight vector 'ω' and bias term 'b', which define the decision boundary of the SVM and allow us to classify new data points.

1.1 Loss Function/ Hinge loss:

It is designed to quantify the error or 'loss' associated with the misclassification of data points. In the context of SVM, hinge loss measures the 'hinge' or margin violation, which is the distance between a data point and the decision boundary of the SVM. The hinge loss function encourages the SVM to correctly classify data points while maximizing the margin.

L = max (0, 1-yi(ω.xi – b)) is the function.

L = 0 if y.f(x) >=1, in this case the loss/error is zero when the data point lies in the correct side of the decision boundary and L = 1-y.f(x) otherwise, the further we are away from the decision boundary the higher the loss is.

$$L = \begin{cases} 0, & \text{y. f(x)} \geq 1 \\ 1 - \text{y. f(x)}, & otherwise \end{cases}$$

1.2 Cost function:

$$J = \lambda \, ||\omega||^2 + \frac{1}{n} \sum_{i=1}^{n} \max\left(0, 1 - \text{yi}(\omega . \text{xi} - \text{b})\right)$$

In cost function, we are minimising the loss and maximising length. If the data point is on the correct side of the boundary then Ji = $\lambda \, ||\omega||^2$, else Ji = $\lambda \, ||\omega||^2 + 1 - \text{yi}(\omega . \text{xi} - \text{b})$

To calculate the weights, we get the derivative/gradient of cost function:

If y.f(x) >=1 then,

$\frac{dJi}{d\omega k} = 2\lambda \omega k$ for $\frac{dJi}{db} = 0$

Else,

$\frac{dJi}{d\omega k} = 2\lambda \omega k - yi.\ xik$ for $\frac{dJi}{db} = yi$

Update rule:

If y.f(x) >=1 then,

$\omega = \omega - \alpha.d\omega = \omega - \alpha.2\lambda\omega$

$b = b - \alpha.db = b$

else

$\omega = \omega - \alpha.d\omega = \omega - \alpha.(2\lambda\omega - yi.xi)$

$b = b - \alpha.db = b - \alpha.yi$

1.3 Training SVM

- ⇨ Initialise weights
- ⇨ Make sure y belongs to {-1,1}
- ⇨ Application of update rule for n iterations

1.3.1 Prediction

- ⇨ Calculate y = sign (ω.x – b) {i.e., if greater than zero then class +1, otherwise if lesser than zero then class -1}

1.4 Code Explanation of SVM algorithm implementation

```
import numpy as np
import matplotlib.pyplot as plt
```

Importing the necessary libraries as np and plt respectively.

```
class SVM:
    def __init__(self, learning_rate=0.001, lambda_param=0.01, n_iters=1000):
        self.lr = learning_rate
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.w = None
        self.b = None
```

In class SVM, the __init__ method is used to initialise the object's attributes and it is used only within classes (it is similar to a constructor like in java or C++). The input parameters for this method are learning rate set to 0.001, lam

bda parameter set to 0.01 and the number of iterations are set to 1000 by default. We are declaring new variables for the inputs and initialising weight and bias as none.

```
def fit(self, X, y):
        n_samples, n_features = X.shape

        y_ = np.where(y <= 0, -1, 1)

        # Initialize weights
        self.w = np.zeros(n_features)
        self.b = 0

        for _ in range(self.n_iters):
            for idx, x_i in enumerate(X):
                condition = y_[idx] * (np.dot(x_i, self.w) - self.b) >= 1
                if condition:
                    self.w -= self.lr * (2 * self.lambda_param * self.w)
                else:
                    self.w -= self.lr * (2 * self.lambda_param * self.w -
np.dot(x_i, y_[idx]))
                    self.b -= self.lr * y_[idx]
```

Defining fit method that gets input as training data X and y. Assigning the size of the n-dimensional array to n_samples and n_features using X.shape.

y_ is assigned the class values as either -1 or +1, where y<=0 it is -1 otherwise it is +1. Then we initialise the weights w with array of zeroes with size as n_features and b with 0.

We then check the condition yi(ω.xi – b) >= 1, then if the condition is true then we use the update rule depending on the different gradients $\omega = \omega - \alpha.2\lambda\omega$ and bias is not updated.

Else, $\omega = \omega - \alpha.(2\lambda\omega - yi.xi)$ and $b = b - \alpha.yi$ are updated accordingly.

```
def predict(self, X):
        approx = np.dot(X, self.w) - self.b
        return np.sign(approx)
```

Calculating y = sign (ω.x – b) to make it either +1 or -1

```
# Testing
if __name__ == "__main__":
    from sklearn.model_selection import train_test_split
    from sklearn import datasets
```

Setting up the main function and importing desired modules.

```
X, y = datasets.make_blobs(
        n_samples=50, n_features=2, centers=2, cluster_std=1.05,
random_state=40
    )
    y = np.where(y == 0, -1, 1)

    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=123
    )
```

Creating an example dataset with 50 samples and 2 features and splitting the dataset into training and testing sets.

```
clf = SVM()
    clf.fit(X_train, y_train)
    predictions = clf.predict(X_test)
```

Setting up SVM (creating an instance of class SVM) and loading it with the training data. Get the predictions using the predict method defined above.

```
def accuracy(y_true, y_pred):
        accuracy = np.sum(y_true == y_pred) / len(y_true)
        return accuracy

    print("SVM classification accuracy:", accuracy(y_test, predictions))
```

Calculating the accuracy of the prediction and printing them.

```
def visualize_svm():
        def get_hyperplane_value(x, w, b, offset):
            return (-w[0] * x + b + offset) / w[1]
```

Inside the visualise_svm() function , we define a function that returns the value of the hyperplane at a given point. This function takes 4 input parameters the input value ‘x’, the SVM’s weight vector ‘w’, the SVM’s bias term ‘b’, and an offset for the hyperplane.

```
fig = plt.figure()
        ax = fig.add_subplot(1, 1, 1)
        plt.scatter(X[:, 0], X[:, 1], marker="o", c=y)
```

plt.scatter function is then used to plot the data points, where X[:, 0] and X[:, 1] represent the x and y coordinates of the data points, respectively, and y represents the corresponding labels.

```
x0_1 = np.amin(X[:, 0])
        x0_2 = np.amax(X[:, 0])

        x1_1 = get_hyperplane_value(x0_1, clf.w, clf.b, 0)
        x1_2 = get_hyperplane_value(x0_2, clf.w, clf.b, 0)

        x1_1_m = get_hyperplane_value(x0_1, clf.w, clf.b, -1)
        x1_2_m = get_hyperplane_value(x0_2, clf.w, clf.b, -1)

        x1_1_p = get_hyperplane_value(x0_1, clf.w, clf.b, 1)
        x1_2_p = get_hyperplane_value(x0_2, clf.w, clf.b, 1)
```

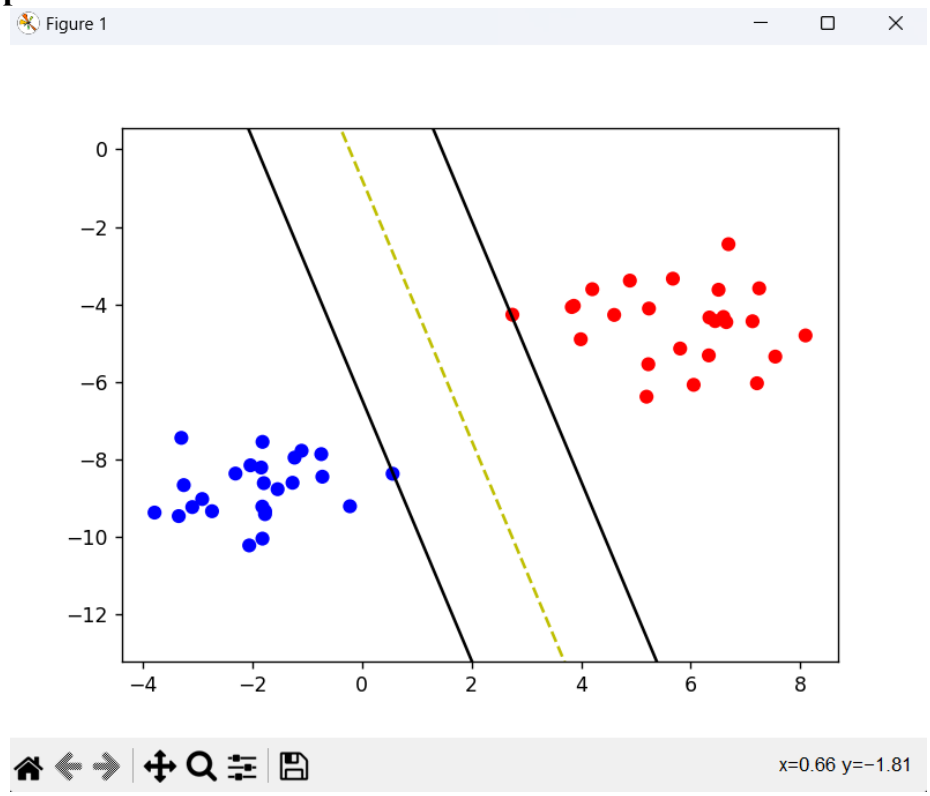
In this part, the minimum and maximum values of the x-coordinate (X[:, 0]) are calculated using np.amin and np.amax, respectively. These values are then used to determine the corresponding y-coordinate values of the hyperplane and margins. The get_hyperplane_value function is called multiple times with different offset values (0, -1, and 1) to calculate these y-coordinate values.

```
plt.plot([x0_1, x0_2], [x1_1, x1_2], "y--")
        plt.plot([x0_1, x0_2], [x1_1_m, x1_2_m], "k")
        plt.plot([x0_1, x0_2], [x1_1_p, x1_2_p], "k")
```

```
        plt.ylim([np.amin(X[:, 1]) - 3, np.amax(X[:, 1]) + 3])
        plt.show()

    visualize_svm()
```

The minimum and maximum values of the y-coordinate (X[:, 1]) are calculated using np.amin and np.amax, respectively. The y-axis limits of the plot are set to be slightly larger than the range of y-values to ensure all data points are visible. Finally, plt.show() is called to display the plot.

1.4.1 Output:




```
(.venv) PS C:\Users\manas\OneDrive\Desktop\mfc> python -u "c:\U
SVM classification accuracy: 1.0
(.venv) PS C:\Users\manas\OneDrive\Desktop\mfc>
```

2. LOGISTIC REGRESSION:

2.1 What is Logistic regression

Logistic regression is a classification algorithm. It is used to predict a binary outcome based on a set of independent variables. It uses a sigmoid curve to show the relationship between the target and independent variables. However, caution should be exercised, logistic regression works best with large data sets that have an almost equal occurrence of values in target variables. It is a type of regression analysis. Regression is a type of predictive modelling technique which is used to find the relationship between the dependant variable (usually known as 'Y') and either one or more independent variables(usually known as 'X'). For example: accessing how much of an impact the dependant variable might have on the independent variable, like if a soft drinks company is sponsoring a football match then regression is used for determining if the ads being displayed during the match have accounted for any increase in sales or not.

Linear regression is usually used for predictive analysis. It essentially determines the extent to which there is a linear relationship between a dependent variable and one or more independent variables. In terms of output, linear regression will give a trend line plotted amongst a set of data points. We use linear regression to predict the sales of a company based on the cost spent on online advertisements, or to see how the change in the GDP might affect the stock price of a company. Whereas, Logistic regression is essentially used to calculate (or predict) the probability of a binary (yes/no) event occurring itself. It is better to use logistic regression when the output/dependant variable tends to be dichotomous.

The independent variables/ data belong to these types:

Continuous: Continuous data refers to measurements that can take on any value within a certain range, such as temperature in degrees Celsius or weight in grams. In technical terms, continuous data can be classified as either interval data or ratio data. Interval data means that the intervals between each value are evenly divided, while ratio data has equally divided intervals and a meaningful "zero" point. For example, temperature in degrees Celsius falls under interval data because the difference between 10 and 11 degrees Celsius is the same as the difference between 30 and 31 degrees Celsius. However, there is no true zero in temperature, meaning that a temperature of zero degrees Celsius does not indicate the absence of temperature. Conversely, weight in grams is considered ratio data as it has equally divided intervals and a true zero. In other words, if something weighs zero grams, it means it has no weight at all.

Discrete, ordinal: Data which can be placed into some type of order in a scale. Ex: customer survey, good reads reviews etc.

Discrete, nominal: Data which fits into named groups which do not represent any kind of order or scale. Ex: eye colour: blue, brown, green.

2.2 Advantages of Logistic regression:

A machine learning model can be described as a mathematical depiction of a real-world process. The process of setting up a machine learning model requires training and testing the model. Training is the process of finding patterns in the input data, so that the model can map

a particular input (say, an image) to some kind of output, like a label. Logistic regression is easier to train and implement as compared to other methods.

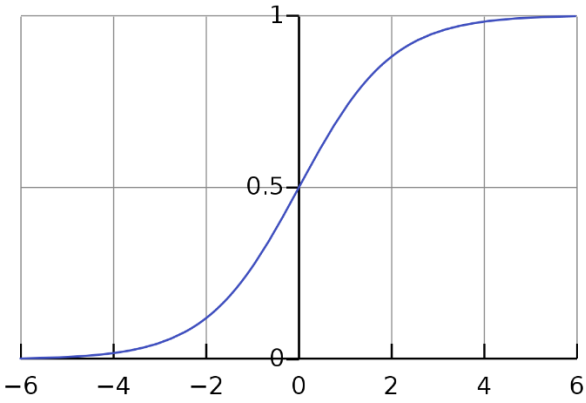
Logistic regression works well for cases where the dataset is linearly separable: A dataset is said to be linearly separable if it is possible to draw a straight line that can separate the two classes of data from each other. Logistic regression is used when your Y variable can take only two values, and if the data is linearly separable, it is more efficient to classify it into two separate classes.

There are certain assumptions that need to be made to implement logistic regression, they are:

- ⇨ The dependant variable is binary/ dichotomous
- ⇨ There should be no or only very little multicollinearity between the predictor/independent variables. (In statistics tests like 'Spearman's rank correlation coefficient' are used to calculate the correlation between the independent variables.
- ⇨ The independent variables should be linearly related to the log odds
- ⇨ It requires fairly large sample sizes.

Let us better understand this model by implementing in python:

As we already know, Best fitting line/ hyperplane for linear regression is given by y = wx + b, for logistic regression we are trying to create the probabilities of event occurrence with our datasets. So, we plot our equation in a sigmoid function and achieve a probability distribution between zero and one.



A sigmoid function is a real function that squashes an input real value between 0 and 1, no matter the size of the input value. It is also called the squashing function.

$$\text{sig}(Z) = \frac{1}{1+e^{-z}}$$

applying sigmoid function on y = wx +b, we get

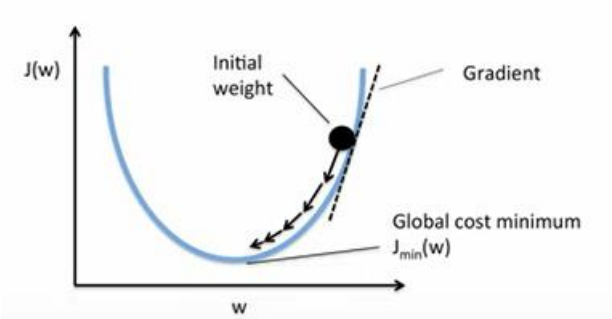
$$y = \frac{1}{1+e^{-wx+b}} = h(x)$$

$$J(w,b) = J(\theta) = \frac{1}{N}\sum_{i=1}^{n} [y^i \log\left(h(x) + \left(1 - y^i\right) \log\left(1 - h\left(x^i\right)\right)\right]]$$

Calculating the gradient descent,

$J'(\theta) = \left[\frac{dJ}{dw}\right] = \frac{1}{N}\sum_1^n 2xi(y - yi)$

$\left[\frac{dJ}{db}\right] = \frac{1}{N}\sum_1^n 2(y - yi)$



Gradient descent at any point gives us the direction to be taken to minimize the error.

w = w – α.dw

b = b - α.dw

where α is the learning rate, that tells how fast/slow gradient is in a particular direction. And dw and db are gradient descents.

Steps involved in Logistic regression:

2.3 Training:

- ⇨ Initialise weights as zero
- ⇨ Initialise bias as zero

Given a data point:

- ⇨ Predict result by using $y = \frac{1}{1+e^{-wx+b}}$
- ⇨ Calculate error
- ⇨ Use gradient descent to figure out new weight and bias values
- ⇨ Repeat n times

2.4 Testing:

Given a data point:

- ⇨ Put values from the data point into the equation $y = \frac{1}{1+e^{-wx+b}}$
- ⇨ Choose a label based on the probability

2.5 Code Explanation:

```
import numpy as np

def sigmoid(x):
    return 1/(1+np.exp(-x))
```

Importing necessary libraries, and defining sigmoid function.

```
class LogisticRegression():
```

```
    def __init__(self, lr=0.001, n_iters=1000):
        self.lr = lr
        self.n_iters = n_iters
        self.weights = None
        self.bias = None
```

Defining an __init__ function and initialising learning rate as 0.001 and number of iterations as 1000. Assigning values for lr, n_iters and initialising bias and weights as zero.

```
def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        for _ in range(self.n_iters):
            linear_pred = np.dot(X, self.weights) + self.bias
            predictions = sigmoid(linear_pred)

            dw = (1/n_samples) * np.dot(X.T, (predictions - y))
            db = (1/n_samples) * np.sum(predictions-y)

            self.weights = self.weights - self.lr*dw
            self.bias = self.bias - self.lr*db
```

getting the number of features and number of samples using X.shape, where X has the training data. Initialising an array of zeroes for n number of features to weights. In linear regression we calculate the predictions by weight times each x value plus bias for each iteration. Similar to that we are getting the linear predictions here and putting those values into a sigmoid function for each iteration. Further we calculate the gradients using $[\frac{dJ}{dw}] = \frac{1}{N}\sum 2xi(y - yi)$ and $[\frac{dJ}{db}] = \frac{1}{N}\sum 2(y - yi)$. As a last step in training the data using logistic regression we update the weights and biases.

```
def predict(self, X):
        linear_pred = np.dot(X, self.weights) + self.bias
        y_pred = sigmoid(linear_pred)
        class_pred = [0 if y<=0.5 else 1 for y in y_pred]
        return class_pred
```

after the training is done, we are defining a predict method. Y_pred is calculated by $y = \frac{1}{1+e^{-wx+b}}$. Y_pred now contains the probability and the values range in between 0 and 1. We classify predictions that lie between 0 and 0.5 into 0 label and 0.5 and 1 into 1 label.

To see how accurate the above implemented algorithm is, we are going to be classifying breast cancer data from scikit-learn datasets.

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn import datasets
import matplotlib.pyplot as plt
from LogisticRegression import LogisticRegression
```

First we import all the necessary libraries.

```
bc = datasets.load_breast_cancer()
X, y = bc.data, bc.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=1234)
```

Load our dataset in bc. Assigning the data and target to X and y respectively. 'X' The input features or independent variables. 'y' is The target variable or dependent variable. 'test_size' specifies the proportion of the data that should be allocated to the test set. In this case, it is set to 0.2, indicating that 20% of the data will be used for testing, while the remaining 80% will be used for training. 'random_state' sets the random seed for reproducibility. By setting it to a specific value (in this case, 1234), the same random split will be generated each time the code is executed. The train_test_split function from scikit-learn returns a subset of input features for training, testing and a subset of target variables for training and testing respectively.

The Breast Cancer Wisconsin (Diagnostic) dataset

Classes	2
Samples per class	212(M),357(B)
Samples total	569
Dimensionality	30
Features	real, positive

```
clf = LogisticRegression(lr=0.01)
clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
```

Creating an instance 'clf' of class logistic regression and setting the learning rate as 0.01. fitting the train and test data using the fit method. Predicting the test data using predict method and assigning it to y_pred.

```
def accuracy(y_pred, y_test):
    return np.sum(y_pred==y_test)/len(y_test)

acc = accuracy(y_pred, y_test)
print(acc)
```

Defining a function that calculates the accuracy. It is the number of times y_pred = y_test for the values in y_test.

2.5.1 Output:

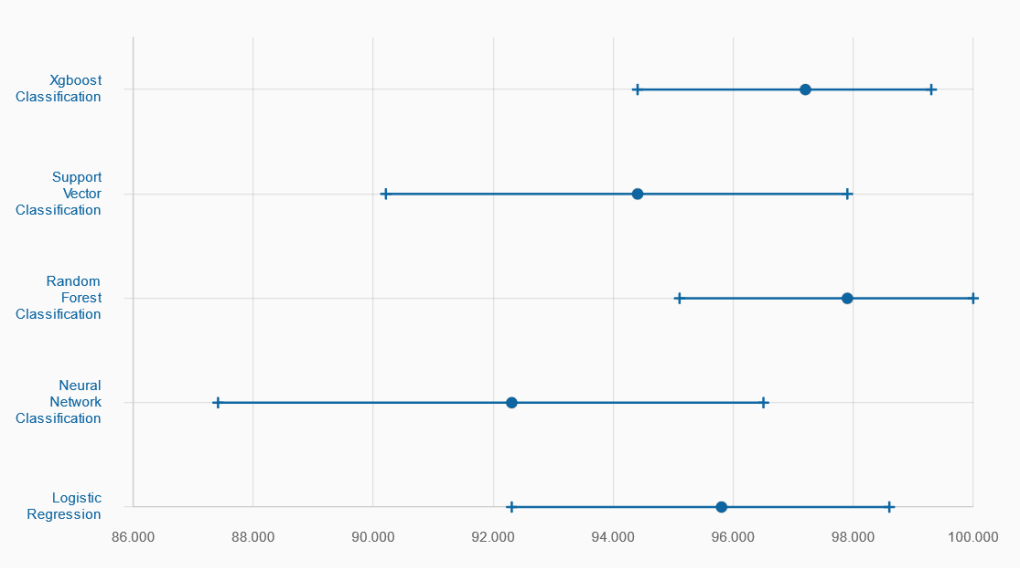
For learning rate = 0.01

```
ve\Desktop\mfc\train.py"
c:\Users\manas\OneDrive\Desktop
  return 1/(1+np.exp(-x))
0.9210526315789473
```

For learning rate = 0.001

```
(.venv) PS C:\Users\manas\
c:\Users\manas\OneDrive\De
  return 1/(1+np.exp(-x))
0.8947368421052632
```

As we can see, the accuracy is higher for lower learning rate (α).



The above figure shows the accuracy of different types of ML training models.

The features based on which the above classification are done are Ten real-valued features computed for each cell nucleus:

a) radius (mean of distances from center to points on the perimeter)

b) texture (standard deviation of gray-scale values)

c) perimeter

d) area

e) smoothness (local variation in radius lengths)

f) compactness (perimeter^2 / area - 1.0)

g) concavity (severity of concave portions of the contour)

h) concave points (number of concave portions of the contour)

i) symmetry

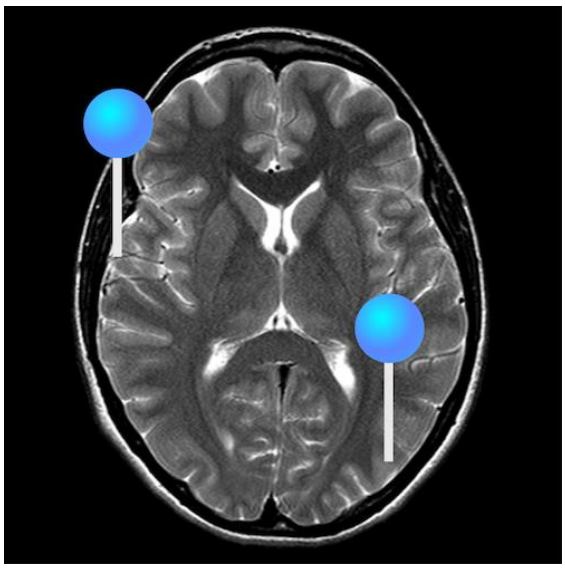
j) fractal dimension ("coastline approximation" - 1)

Now that we have a general idea about SVM and Logistic regression, let us look at it application in tumour classification.

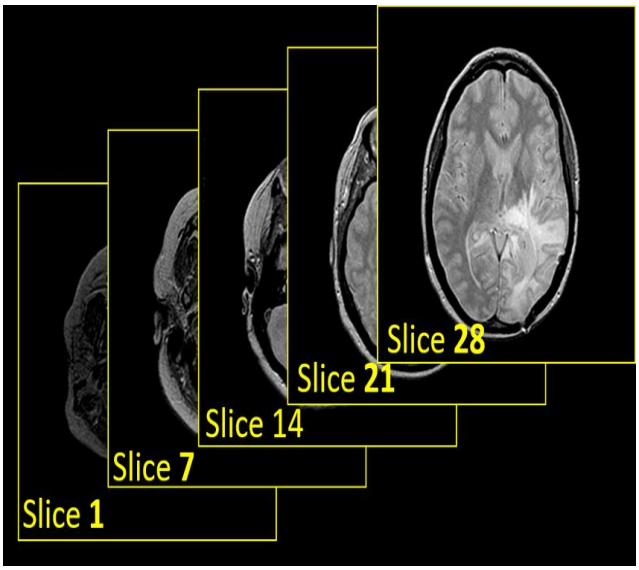
3. Classification of MRI scans

3.1 Dataset for MRI tumour classification:

Axial view of our brain:



One scan



Multiple scans stacked up together

In the above figure where there is one scan, The blue marking shows a single pixel, i.e., the smallest resolution unit such that we can tell the difference between it and its neighbours. In where multiple scans are stacked up together, we have collected multiple such images and here the smallest unit pixels are stacked up together to create a voxel. A voxel to a pixel is basically like a cube to a square.

Spatial resolution in imaging techniques:

In simple terms, it is the size of a single pixel times the distance between 2 pixels in different slices of image. (Size of pixel in 2D x distance between 2 pixels from different slices in 3D).

The intrinsic motion of the brain, tolerance capacity of the person and time of the scan directly contribute to the resolution (detail) of the MRI. This is why elaborate scans can be collected from corpses as their tolerance is not in picture and the blood flow is nil. Functional imaging produces 4D images usually as they are multiple voxels stacked up together (3D) and plotted along time (1D). The time measured as the time taken to iteratively go from one slice of the brain and after a while get back to the same slice but in different position possibly. Functional imaging, though very similar to anatomical imaging is still arguably better because it takes into account both temporal and spatial resolution. But we are going to be looking into classification of MRI for now though fMRI is arguably better.

The tumour dataset that we are using contains 2 classes. Namely, No tumour and pituitary tumour. Brain tumours are recognized as aggressive diseases affecting both children and adults, constituting a significant proportion (85 to 90 percent) of primary Central Nervous System (CNS) tumours. Each year, approximately 11,700 individuals receive a brain tumour diagnosis. The 5-year survival rate for cancerous brain or CNS tumours is around 34 percent for men and 36 percent for women. These tumours can be classified as benign, malignant, pituitary, and

more. To enhance patient life expectancy, it is crucial to employ accurate diagnostics, proper treatment planning, and timely interventions. Magnetic Resonance Imaging (MRI) has proven to be the most effective technique for brain tumour detection. However, the manual examination of the vast amount of generated image data by radiologists is prone to errors due to the complexity and unique properties of brain tumours.

Automated classification techniques utilizing Machine Learning (ML) and Artificial Intelligence (AI) have consistently demonstrated higher accuracy compared to manual classification. Classification models based on ML and deep learning is essential in the detection and classification of brain tumours. Such a system would revolutionize the field by providing more reliable and efficient analysis, enabling improved patient care and outcomes.

We are going to be using 2 out of the 4 classes mentioned earlier. Namely, pituitary tumour and no tumour and classify them.

3.2 Steps involved in implementing the algorithm:

- ⇨ Import all necessary libraries
- ⇨ Load and prepare data
- ⇨ Data visualisation
- ⇨ Pre-process the data to fit our ML models
- ⇨ Model training
- ⇨ Evaluation
- ⇨ Prediction
- ⇨ Testing

3.3 Code explanation for MRI classification:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import cv2
import os
```

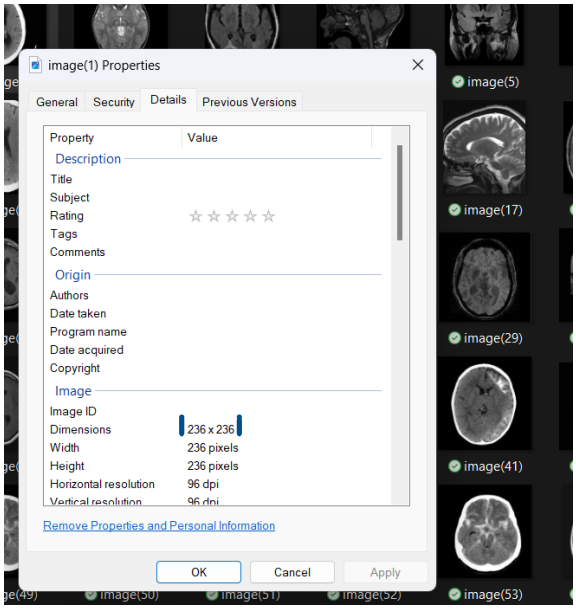
Importing all the necessary libraries such as numpy,pandas,matplotlib,scikit-learn etc.

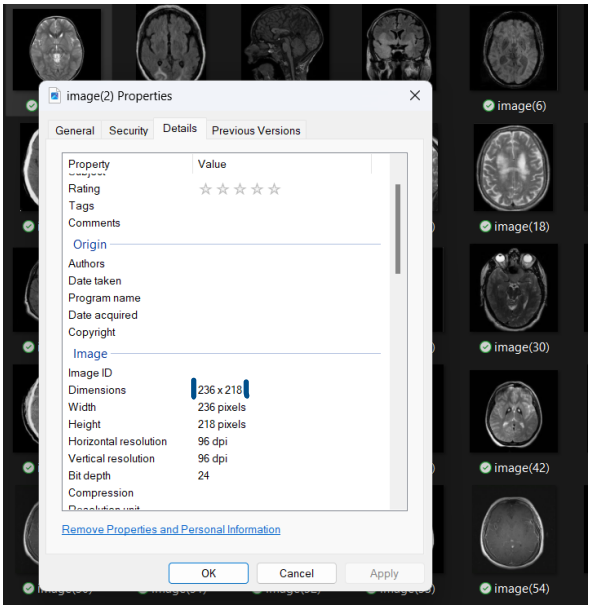
```
path = os.listdir(r'C:\Users\manas\OneDrive\Desktop\mfc\Training')
classes = {'no_tumor':0, 'pituitary_tumor':1}
```

Adding path of the files that consists of the training data that has already been downloaded and assigning it to the path variable. Labelling 0 and 1 for no tumour and pituitary tumour respectively.

```
X = []
Y = []
for cls in classes:
    pth = 'Training/'+cls
    for j in os.listdir(pth):   #folders in dir
        img = cv2.imread(pth+'/'+j, 0)
        img = cv2.resize(img, (200,200))
        X.append(img)
        Y.append(classes[cls])
```

Creating 2 lists X and Y, we later append the features/ training data/ dependant variable in X and the target/ independent variable in Y. Assigning the specific path of training dataset to pth and iterating the loop for every (no_tumour or pituitary tumour) cls. We are setting a single scan from the no tumour class to Y and from pituitary tumour class to X. But first we preprocess the image by reading the image in 2D and converting the image to 8-bit depth grayscale. The imread function defaults the second argument as zero. Therefore, we get a colour value from 0 to 255 for each pixel of our image. After converting the image to grayscale, we are resizing it to be 200x200 to make it capable of being passed into a ML model.





As we can the dimensions of 2 training images are very different. We make it same for every image in the dataset.

```
np.unique(Y)
```

This command displays 0 and 1 as there are 2 target classes.

```
X = np.array(X)
Y = np.array(Y)
```

In this segment of the code X and Y are displayed in the form of an array.

```
pd.Series(Y).value_counts()
```

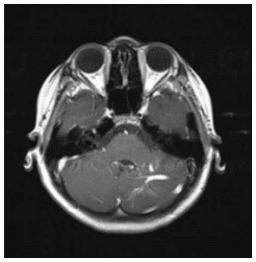
In this segment of the code, value_counts() function is used to get a Series containing counts of unique values. Hence, the values that will be returned in the form of a table are the number of samples with no tumour and number of samples with pituitary tumour.

```
X.shape, X_updated.shape
```

X.shape returns the number of samples and the dimension of the dataset. In our case, it will return 1222 samples in 200x200 dimension.

```
plt.imshow(X[0], cmap='gray')
```

this plot is to visualise our data just for better understanding purposes, we are displaying the first image in no_tumour class of training data.



Since sklearn toolkit works on only 2-D dataset and the dataset we have is 3-D (1222, 200, 200), the dataset needs to be converted to 2-D. This is done by flattening the image and making it (1222 ,40000) in dimension.

```
X_updated = X.reshape(len(X), -1)
X_updated.shape
```

X_updated will consist of a column array of 2 dimensions. Shape function displays the shape of the array.

```
xtrain, xtest, ytrain, ytest = train_test_split(X_updated, Y,
random_state=10,test_size=.20)
```

X: Input features or independent variables of the dataset. y: The target variable or dependent variable of the dataset.

The train_test_split function divides the data into training and test sets randomly.

test_size: This parameter specifies the proportion of the data that should be allocated to the test set. In this case, it is set to 0.2, indicating that 20% of the data will be used for testing, while the remaining 80% will be used for training.

random_state: we set it to a specific value (in this case, 10), the same random split will be generated each time the code is executed.

The train_test_split function then returns four variables:

- ⇨ X_train: The subset of the input features for training.
- ⇨ X_test: The subset of the input features for testing.
- ⇨ y_train: The subset of the target variable for training.
- ⇨ y_test: The subset of the target variable for testing.

```
xtrain.shape, xtest.shape
```

This is to show the number of samples we use for both testing and training. Here, it is (977,40000) for training and (245, 40000) for testing.

```
print(xtrain.max(), xtrain.min())
print(xtest.max(), xtest.min())
xtrain = xtrain/255
xtest = xtest/255
print(xtrain.max(), xtrain.min())
print(xtest.max(), xtest.min())
```

To make the X_train values lie between 0 and 1 we calculate the maximum values for training and testing data and divide it by 255. We divide the data by 255 here because all RGB values are from 0 to 255. After the scaling the maximum value is 1 and the minimum value is 0.

```
255 0
255 0
1.0 0.0
1.0 0.0
```

After the feature scaling, we can either choose to do feature selection or not. It is advised not to be implemented for our particular dataset because it leads to loss of data. Hence, we stick to the current data. Dimensionality reduction techniques like PCA (principal component analysis) might compress important spatial information present in our image dataset, since we use MRI data where each voxel might represent a feature. Advantages of feature scaling and feature selection are discussed later.

```
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
```

We are going to be comparing the accuracy of two models, Logistic regression and SVC (support vector classifier).

```
import warnings
warnings.filterwarnings('ignore')
```

Ignoring all warnings by using the method filterwarnings.

```
lg = LogisticRegression(C=0.1)
lg.fit(xtrain, ytrain)
```

Training the model using logistic regression, setting the learning rate to 0.1. Fitting the data.

```
sv = SVC()
sv.fit(xtrain, ytrain)
```

Training the model using SVC. Fitting the data. We use rbf kernel that is set by default.

```
print("Training Score:", lg.score(xtrain, ytrain))
print("Testing Score:", lg.score(xtest, ytest))
print("Training Score:", sv.score(xtrain, ytrain))
print("Testing Score:", sv.score(xtest, ytest))
```

Displaying the testing and training score of logistic regression and SVC.

```
Training Score: 1.0
Testing Score: 0.9591836734693877
Training Score: 0.9938587512794268
Testing Score: 0.963265306122449
```

We can infer from these scores that SVM is comparatively better than Logistic regression as there is more accuracy.

```
pred = sv.predict(xtest)
misclassified=np.where(ytest!=pred)
```

This generates an array of indices where the predictions are not equal to target variable. In other words, we can say that the samples have been misclassified. i.e., the prediction is not accurate.

```
misclassified
print("Total Misclassified Samples: ",len(misclassified[0]))
print(pred[36],ytest[36])
```

To better visualise how the misclassification actually happens, we can take a look at the value at prediction array index 36 and the value is 0 which is not the same as value at index 36 of the target testing data which is 1.

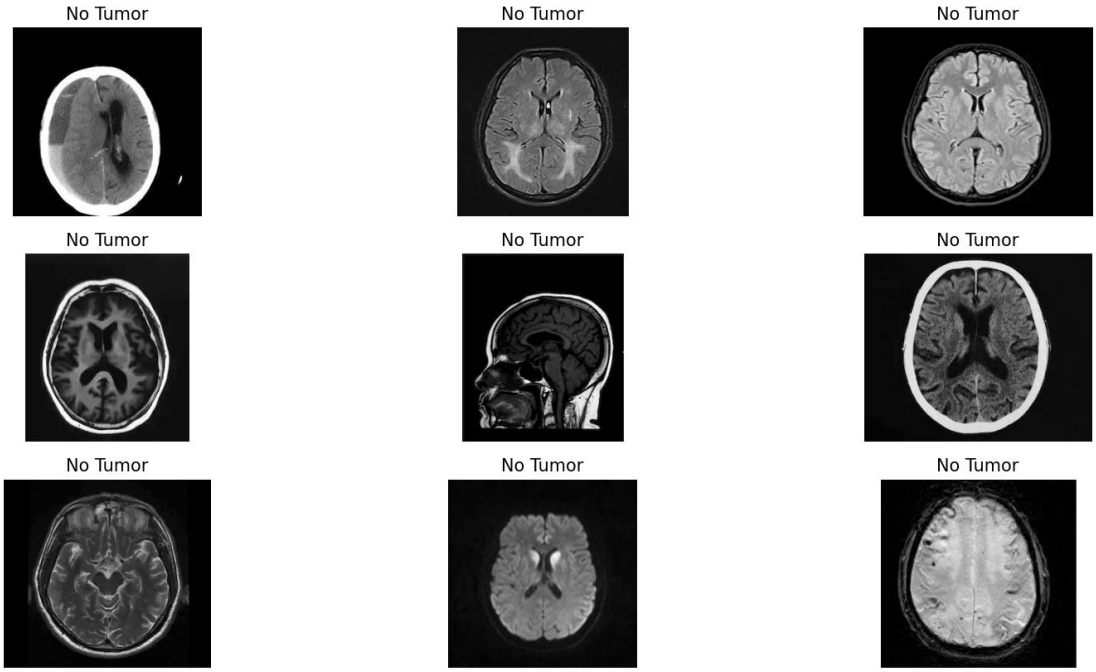
```
Total Misclassified Samples:  9
0 1
```

```
dec = {0:'No Tumor', 1:'Positive Tumor'}
```

Label 0 for no tumour and 1 for positive tumour.

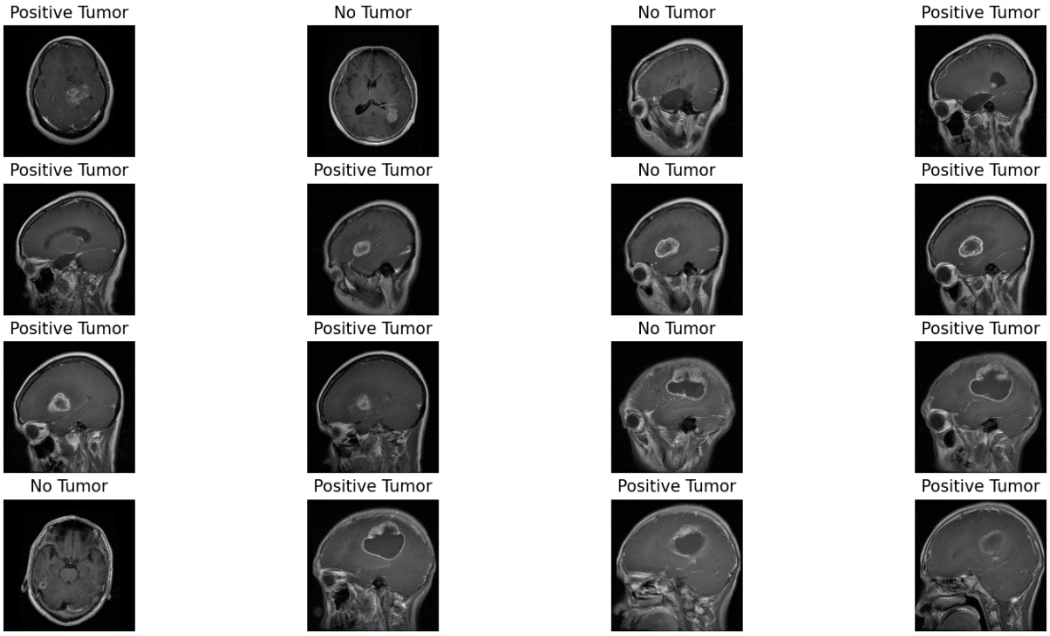
```
plt.figure(figsize=(12,8))
p = os.listdir(r'Testing/')
c=1
for i in os.listdir(r'Testing/no_tumor/')[:9]:
    plt.subplot(3,3,c)

    img = cv2.imread(r'Testing/no_tumor/'+i,0)
    img1 = cv2.resize(img, (200,200))
    img1 = img1.reshape(1,-1)/255
    p = sv.predict(img1)
    plt.title(dec[p[0]])
    plt.imshow(img, cmap='gray')
    plt.axis('off')
    c+=1
```

Now that the training part of the data is completed, we apply the testing/ unseen data to check the performance of the model. Reading the first 9 images from no_tumour class of testing data and using subplot to plot them in a single plot. Repeating the same technique for preprocessing and prediction. Predicting the outcome as either 0 or 1 and giving the plot title accordingly as either no tumour or pituitary tumour. The output derived from this is 100% accurate.



```
plt.figure(figsize=(12,8))
p = os.listdir(r'Testing/')
c=1
for i in os.listdir(r'Testing\pituitary_tumor')[:16]:
    plt.subplot(4,4,c)
    de = r'C:\Users\manas\OneDrive\Desktop\mfc\Testing\pituitary_tumor\\'
    img = cv2.imread(de[:-1]+i,0)
    img1 = cv2.resize(img, (200,200))
    img1 = img1.reshape(1,-1)/255
    p = sv.predict(img1)
    plt.title(dec[p[0]])
    plt.imshow(img, cmap='gray')
    plt.axis('off')
    c+=1
plt.show()
```

Repeating the same for pituitary class of testing data but changing the labels as positive and negative tumour.



Conclusion

SVM is a powerful machine learning algorithm that aims to find an optimal hyperplane to separate different classes in a dataset. It works by maximizing the margin between the support vectors, which are the closest data points to the decision boundary. SVM has been successfully applied to brain tumor image classification. On the other hand, logistic regression is a statistical model that predicts the probability of an instance belonging to a particular class. It uses a logistic function to model the relationship between the input features and the probability of a binary outcome. Logistic regression and SVM has been employed in brain tumor image classification tasks, demonstrating good accuracy in identifying tumor regions.