1. Introduction

Our dataset consists of MRI images classified into four categories: glioma, meningioma, pituitary tumor, and no tumor.
Glioma, pituitary tumor, and meningioma are three distinct types of brain tumors, each originating from different cell types and locations within the brain. Gliomas arise from glial cells, which are supportive cells in the central nervous system, and can be highly aggressive, with glioblastoma being the most malignant form. Pituitary tumors develop in the pituitary gland, a small gland at the base of the brain that regulates various hormones; these tumors can cause hormonal imbalances and symptoms depending on the hormones affected, but are typically benign. Meningiomas originate from the meninges, the protective membranes covering the brain and spinal cord, and are usually slow-growing and benign, though they can cause significant symptoms and complications due to their size and location. Each of these tumors requires different diagnostic approaches and treatment strategies tailored to their unique characteristics and impacts on the patient's health.
Our project aims to enhance the accuracy and efficiency of brain tumor diagnosis by leveraging advanced machine learning and deep learning techniques. Initially, we concentrated on traditional machine learning techniques, including Principal Component Analysis (PCA), Non-negative Matrix Factorization (NMF), LASSO Logistic Regression, Decision Tree Classifier, Random Forest with Recursive Feature Elimination, and Support Vector Classifier (SVC) to classify the MRI images and identify the most critical features. Subsequently, we advanced to deep learning techniques, particularly Convolutional Neural Networks (CNNs) such as ResNet50 and VGG16, as well as the Transformer-based Vision Transformer (ViT). These methods are renowned for their ability to automatically learn complex patterns and features from images.
By utilizing these sophisticated models, we seek to automate the classification of MRI images into the aforementioned categories, thereby assisting doctors in diagnosing tumors. This assistance could not only reduces the time required for diagnosis but also minimizes human error, providing more consistent and reliable results. Our approach could streamline the diagnostic process, allowing healthcare professionals to focus more on patient care and less on manual image analysis.

2. Data Preparation

We started the analysis by creating the `df` object to represent the DataFrame contained in the pickle file named "complete_df.pkl".

```
from google.colab import drive
drive.mount('/content/drive')

df = pd.read_pickle("/content/drive/MyDrive/ProgettoDataMining/code/complete\_df.pkl")
```

To ensure that all MRI images were in 8-bit, we applied the `normalize_to_8bit` fuction. This function normalizes an image with 12-bit or 14-bit pixel values to an 8-bit image. This process involves several steps: first, the image is converted to a float32 data type for precision during normalization. The pixel values are then shifted so that the minimum value becomes 0 and scaled so that the maximum value becomes 1. Following this, the pixel values are further scaled to the range 0-255. Finally, the normalized image is converted back to an 8-bit format (uint8) and returned. This ensures that the pixel values of the input image are properly scaled and adjusted for 8-bit representation.

```
def normalize_to_8bit(image):
    image = image.astype(np.float32)
    image = image -  image.min()
    image = image / image.max()
    image = image * 255
    norm_image = image.astype(np.uint8)
    return norm_image
```

Additionally, we converted all images of the `df` in greyscale. To achieve this, we applied the `transform_grayscale` function. This function converts a color image to a grayscale image. If the input image has more than two dimensions, it is assumed to be a color image with red, green, and blue channels. The function extracts these channels and applies the standard formula for converting RGB to grayscale: $\text{gray} = 0.2989 \times \text{red} + 0.5870 \times \text{green} + 0.1140 \times \text{blue}$. This results in a single-channel grayscale image. The function then prints the original shape and range of values of the image, as well as the shape of the transformed grayscale image, and returns the grayscale image. If the input image is already in grayscale (i.e., has only two dimensions), the function simply prints the shape and range of values and returns the original image.

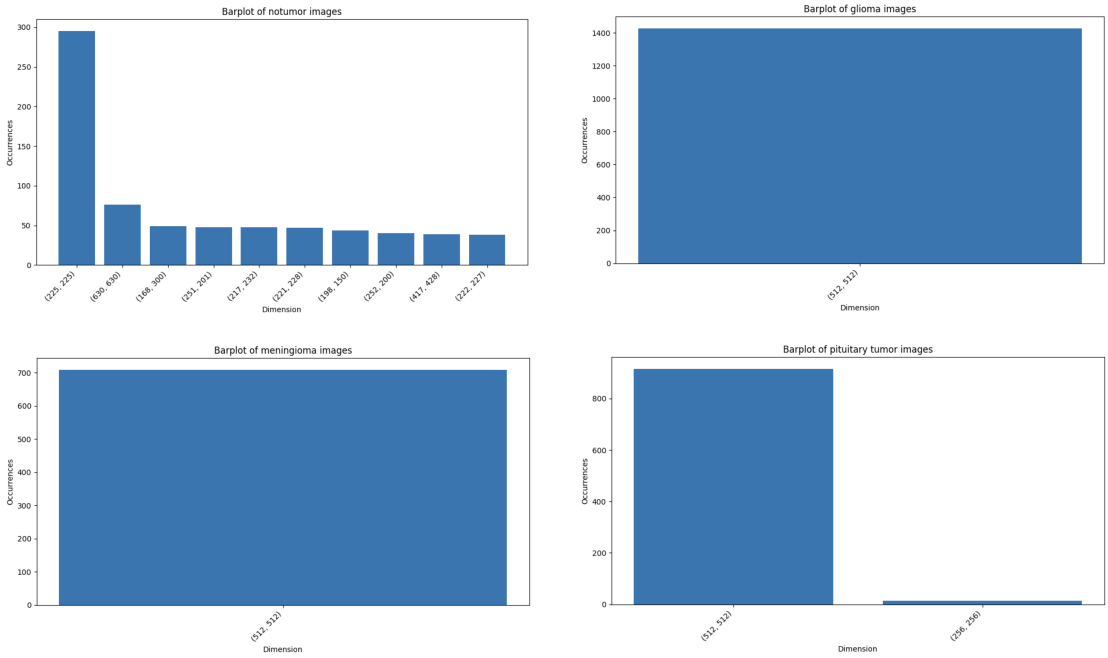
```
def transform_grayscale(image):
  if len(image.shape) > 2:
      red = image[:,:,0]
      green = image[:,:,1]
      blue = image[:,:,2]
      gray = 0.2989 * red + 0.5870 * green + 0.1140 * blue
      print("The shape was", image.shape, "and now is", gray.shape)
      print(" The range of values is ",np.min(image),np.max(image))
      return gray
  print("The shape is", image.shape)
  print(" The range of values is ",np.min(image),np.max(image))
  return image
```

To perform statistical analysis and visualization on the labeled image data stored in `df`, we defined a class named `Statistic_of_Dataframe`. This class encapsulates functionalities for counting image shapes for each class and creating bar plots to show the distribution of these shapes.

Specifically, the class includes a constructor that initializes an empty dictionary, `self.count`, to store counts of image shapes. The `count_types_image_for_every_class` method iterates over each unique label in the the DataFrame, counts the occurrences of different image shapes for each label, and stores these counts in `self.count`. Another method, `create_graphics`, generates bar plots to visualize the distribution of image shapes for each class stored in `self.count`. This method sorts the image shapes by their frequency in descending order, selects the top `num_item_to_show` shapes, and creates bar plots to visually represent the data.
After defining the `Statistic_of_Dataframe` class, we created an instance named `u`. Then, we called `count_types_image_for_every_class(df)` on `u` to populate the shape counts, followed by `create_graphics()` to produce and display the bar plots. This approach allowed us to effectively analyze and visualize the distribution of image shapes across different tumor types in the dataset.

```
class Statistic_of_Dataframe:

    def __init__(self):
        self.count = {}

    def count_types_image_for_every_class(self, df):
        for label in df["Label"].unique():
            x = df.loc[df['Label'] == label].reset_index(drop=True)
            self.count[label] = {}
            y = self.count[label]
            for i in range(len(x)):
                image = x.at[i, "Image"]
                shape = image.shape
                if shape in y:
                    y[shape] += 1
                else:
                    y[shape] = 1

    def create_graphics(self, num_item_to_show=10):
        if self.count =={}:
            print("Count is empty. Please implement the
    count_types_image_for_every_class method ")
        for label in self.count.keys():
            y = self.count[label]
            x_axis = []
            y_axis = []
            sorted_tuple_list = sorted(y.items(), key=lambda x: x[1], reverse=
    True)
            self.count[label] = dict(sorted_tuple_list)
            for item in sorted_tuple_list:
                x_axis.append(str(item[0]))
                y_axis.append(item[1])
            if len(x_axis) > num_item_to_show:
                x_axis = x_axis[:num_item_to_show]
                y_axis = y_axis[:num_item_to_show]
            plt.figure(figsize=(10, 6))
            plt.bar(x_axis, y_axis)
            plt.xlabel('Dimension')
            plt.ylabel('Occurrences')
            plt.title(f'Barplot of {label} images')
            plt.xticks(rotation=45, ha='right')
            plt.tight_layout()
            plt.show()
```

```
u = Statistic_of_Dataframe()
u.count_types_image_for_every_class(df)
u.create_graphics()
```



The bar plots reveal distinct patterns in the distribution of image dimensions across the different classes. The "notumor" class shows a majority of images with dimensions of (225, 225), along with some other less frequent dimensions. In contrast, the "glioma" and "meningioma" classes display uniformity, with all images sized at (512, 512). The "pituitary tumor" class predominantly consists of images with dimensions of (512, 512), though it also includes a smaller number of images sized at (256, 256).

3. Features Extraction

Given the vast number of features — totaling 1000 — in our dataset, this section focuses on reducing data dimensionality and extracting the most significant components using Principal Component Analysis (PCA) and Non-Negative Matrix Factorization (NMF). These techniques help to streamline our data, making it more manageable and highlighting the key features that contribute most to our analysis.

3.1 Libraries

These imports collectively set up the environment for data manipulation, image processing, dimensionality reduction, model evaluation, and timing.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import random
import cv2
from sklearn.decomposition import PCA
from sklearn.metrics import mean_squared_error
from sklearn.decomposition import NMF
from sklearn.model_selection import KFold
import time
```

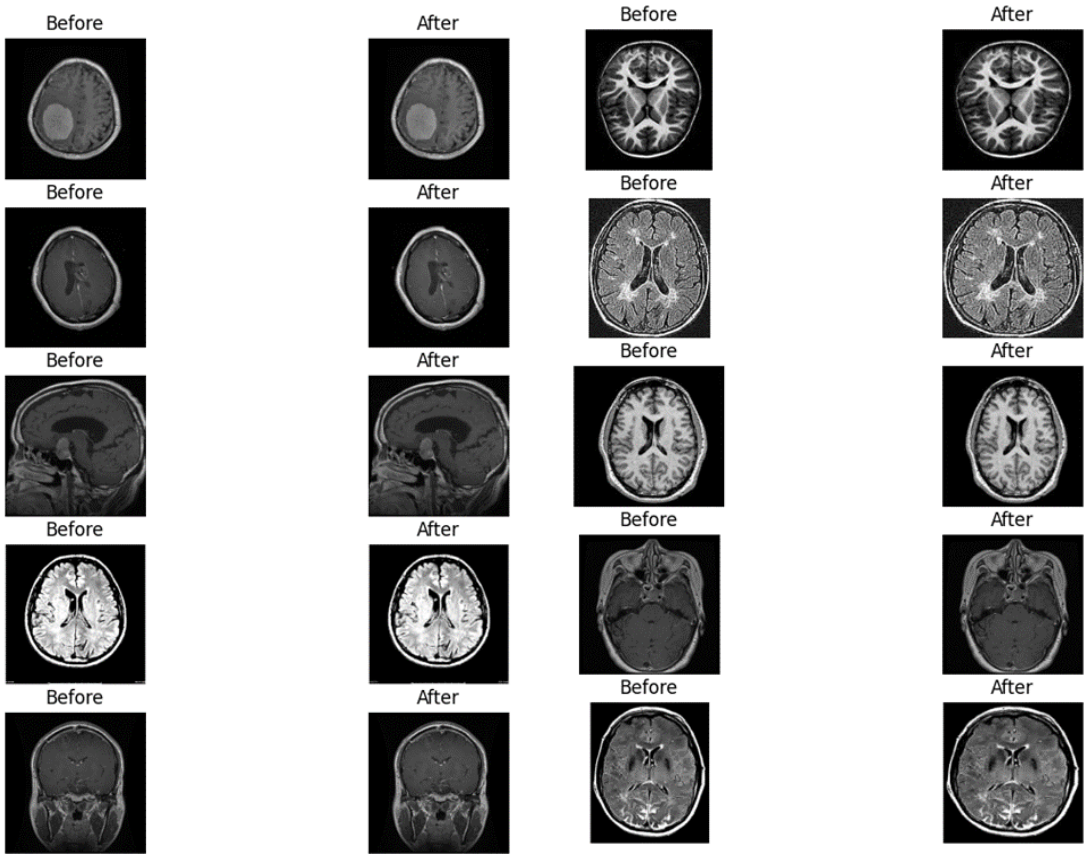
3.2 Resizing

We proceeded with data pre-processing by resizing the images to make them comparable and suitable for inclusion in statistical and deep learning models to extract valuable information.
To perform image resizing operations on the labeled image data stored in `df`, we defined a class named `Resize`. This class encapsulates functionalities for resizing images and providing visual feedback on the resizing operation through side-by-side comparisons. Specifically, the class includes a constructor that initializes the original DataFrame (`df`) and the desired size for resizing (`size`). It defines a `resize` method that iterates through each image in the DataFrame, resizes it to the specified dimensions, and updates the DataFrame with the resized images. Additionally, it defines the `show_resize` method that randomly selects a specified number of examples from the original DataFrame and displays a comparison of images before and after the resizing side by side. This setup allows for a clear visual comparison of the effects of image resizing.
After defining the `Resize` class, we created an instance named `resizer` with the desired image size of (150, 150) pixels. We applied the `resize` method to resize the images and used the `show_resize` method to display 10 examples of images before and after resizing, using a random seed set to 20 for reproducibility. Finally, we updated the original DataFrame with the resized DataFrame, ensuring that `df` now contains images resized to (150, 150) dimensions.

```
class Resize:

    def __init__(self, df, size):
        self.dataframe = df
        self.size = size
        self.new_df = None
```

```
    def resize(self):
        self.new_df = self.dataframe.copy()
        for i in range(len(self.dataframe)):
            image = self.new_df.at[i, "Image"]
            new_image = cv2.resize(image, self.size)
            self.new_df.at[i, "Image"] = new_image

    def show_resize(self, num_example, random_seed=50):
        if self.new_df is None:
            print("You have to apply the resize method before showing the resized images.")
        else:
            random.seed(random_seed)
            fig, axes = plt.subplots(num_example, 2, figsize=(8, 20))
            i = 0
            iterations = 0
            while i < num_example and iterations < 100:
                index = random.randint(0, len(self.dataframe) - 1)
                if pd.isna(self.new_df.at[index, "Image"]).all():
                    print("There is no image at index", index)
                    iterations += 1
                else:
                    image = self.dataframe.at[index, "Image"]
                    new_image = self.new_df.at[index, "Image"]
                    axes[i, 0].imshow(image, cmap="gray")
                    axes[i, 0].set_title("Before")
                    axes[i, 0].axis('off')
                    axes[i, 1].imshow(new_image, cmap="gray")
                    axes[i, 1].set_title("After")
                    axes[i, 1].axis('off')
                    i += 1
                    iterations += 1
            plt.show()
```

```
size = (150, 150)
num_example = 10

resizer = Resize(df, size)
resizer.resize()
resizer.show_resize(num_example, 20)
df = resizer.new_df
```



3.3 Balanced subsample from the DataFrame

To ensure a balanced representation of each category in our dataset, we developed a function named `select_balanced_subdf` that takes a DataFrame containing labeled data and creates a balanced subsample from it. In detail, the function calculates the number of unique labels and determines the number of observations to select per category to achieve the desired total number of observations. It iterates through each unique label, randomly sampling the specified number of observations per category and concatenating them into a new DataFrame. If the resulting DataFrame has fewer observations than desired, the function supplements the remaining observations from a reserve category. This approach guarantees that each category is adequately represented in the subsample.
We applied this function to our original DataFrame (`df`) to create a balanced subsample (`subsampled_df`) with 1000 total observations. To ensure reproducibility and a clean index, we shuffled the selected subset randomly with a fixed random seed and reset the index. This balanced subsample is now ready for further analysis, providing a fair representation of each category.

```
def select_balanced_subdf(df, num_observations, random_state=1):

    num_label = len(df["Label"].unique())
    num_observations_for_one_category = num_observations // num_label
    new = pd.DataFrame()
    indexes = []

    for label in df["Label"].unique():
        subset = df.loc[df['Label'] == label]
        indexes.extend(subset.index)
        y = subset.sample(n=num_observations_for_one_category, random_state=
    random_state)
        new = pd.concat([new, y]).reset_index(drop=True)

    if len(new) < num_observations:
        reserve = df.loc[~df.index.isin(indexes)]
```

```
        y = reserve.sample(n=num_observations - len(new), random_state=
    random_state)
        new = pd.concat([new, y]).reset_index(drop=True)

    return new
```

```
num_observations = 1000
subsampled_df = select_balanced_subdf(df, num_observations)
subsampled_df= subsampled_df.sample(frac=1, random_state=42).reset_index(drop=
    True)
```

3.4 Normalized matrix

To prepare our image dataset for Principal Component Analysis (PCA), we needed to convert the images into a suitable format and normalize their pixel values. To achieve this, we developed a a function named `create_matrix_of_rows_images_normalized` designed to process images stored in a DataFrame and create a normalized matrix, where each row corresponds to a flattened and normalized version of an image.
Specifically, the function begins by determining the number of pixels in a flattened version of the first image to set the matrix dimensions. It then creates an empty matrix with rows corresponding to the number of images and columns corresponding to the number of pixels in each flattened image. Iterating through the DataFrame, the function flattens each image into a 1D array and assigns it to the respective row in the matrix. After processing all images, the function normalizes the pixel values by scaling them to the range [0, 1] by dividing by 255.
We applied this function to the balanced subsample DataFrame (`subsampled_df`), resulting in a normalized image matrix (`image_matrix`) with dimensions corresponding to the number of images and the number of pixels per image. This preparation is crucial for PCA, which requires normalized input data.

```
def create_matrix_of_rows_images_normalized(df):

    dimension = len(df.at[0, "Image"].flatten())
    image_matrix = np.empty((len(df), dimension))

    for i in range(len(df)):
        image = df.at[i, "Image"]
        print(i)
        image = image.flatten()
        image_matrix[i, :] = image

    image_matrix = image_matrix / 255
    return image_matrix
```

```
image_matrix = create_matrix_of_rows_images_normalized(subsampled_df)

image_matrix.shape
```

```
(1000, 22500)
```

3.5 Principal Component Analysis

To analyze the dimensionality of our image dataset, we applied Principal Component Analysis (PCA) using the normalized image matrix (`image_matrix`).
Principal Component Analysis (PCA) is a statistical technique used to reduce the dimensionality of a dataset while preserving as much variance as possible. It works by transforming the data into a new coordinate system, where the greatest variances by any projection of the data lie along the first coordinates, called principal components. These principal components are a set of linearly uncorrelated variables that are ordered by the amount of variance they capture from the original dataset. Essentially, PCA simplifies the complexity of high-dimensional data by reducing the number of dimensions without significant loss of information, making it easier

to analyze and visualize the underlying patterns and structures.
First, we initialized the PCA model with whitening. Whitening further transforms the data by centering it and scaling it so that each feature has unit variance. This whitening step ensures that the transformed data has a mean of 0 and a variance of 1 for each feature, effectively removing any correlations between features and standardizing their scales. Whitening is particularly useful as it can improve the interpretability and performance of PCA by ensuring that all dimensions contribute equally to the variance calculation. Next, we fitted the PCA model to the normalized image matrix. This process extracted the principal components, which capture the directions in the feature space that maximize the variance. These principal components are stored in the `principal_components` array. To represent the original data in the reduced-dimensional space, we transformed the image matrix using the PCA model, resulting in the `loading` matrix. In `loading`, each row corresponds to the coordinates of the original data points in the new coordinate system defined by the principal components. Finally, by applying `pca.inverse_transform(loading)`, we reconstructed the original data (`image_matrix`) from from the transformed data (`loading`), resulting in `X_reconstructed`. This reconstruction attempts to approximate the original data using a reduced number of principal components, effectively reversing the transformation performed during the dimensionality reduction phase by PCA.

```
pca = PCA(whiten=True)

pca.fit(image_matrix)

principal_components = pca.components_

loading = pca.transform(image_matrix)

X_reconstructed = pca.inverse_transform(loading)
```

When utilizing PCA, the `explained_variance_ratio_` attribute provides a list of values that indicate the percentage of variance explained by each of the computed principal components. These values are sorted in descending order, implying that the first principal component explains the maximum variance in the data, the second component explains the second highest variance, and so forth. This attribute is crucial for understanding how much information each principal component retains from the original data, offering insights into the relative significance of each component in describing the overall data variability. By invoking `pca.explained_variance_ratio_`, we computed and assigned these explained variance ratios to the `variance_ratio` variable.
We used `np.cumsum(variance_ratio)` to compute the cumulative sum of the explained variance ratios. This means each value in the resulting array is the sum of all previous values in `variance_ratio`. The cumulative sum provides a visualization of how much of the total variance in the data is explained using the first N principal components. This is useful for deciding how many principal components to include in the analysis based on how much of the total variance one wishes to retain.
Additionally, in PCA, the `explained_variance_` attribute provides an array where each element indicates the absolute amount of variance explained by its corresponding principal components. By invoking `pca.explained_variance_`, we computed and assigned these explained variances to the `variance` variable. This attribute directly quantifies the variance explained by each principal component in the original units of measurement, offering a complementary perspective to the normalized variance ratios.

```
variance_ratio = pca.explained_variance_ratio_

cumulative_variance_ratio = np.cumsum(variance_ratio)

variance = pca.explained_variance_
```

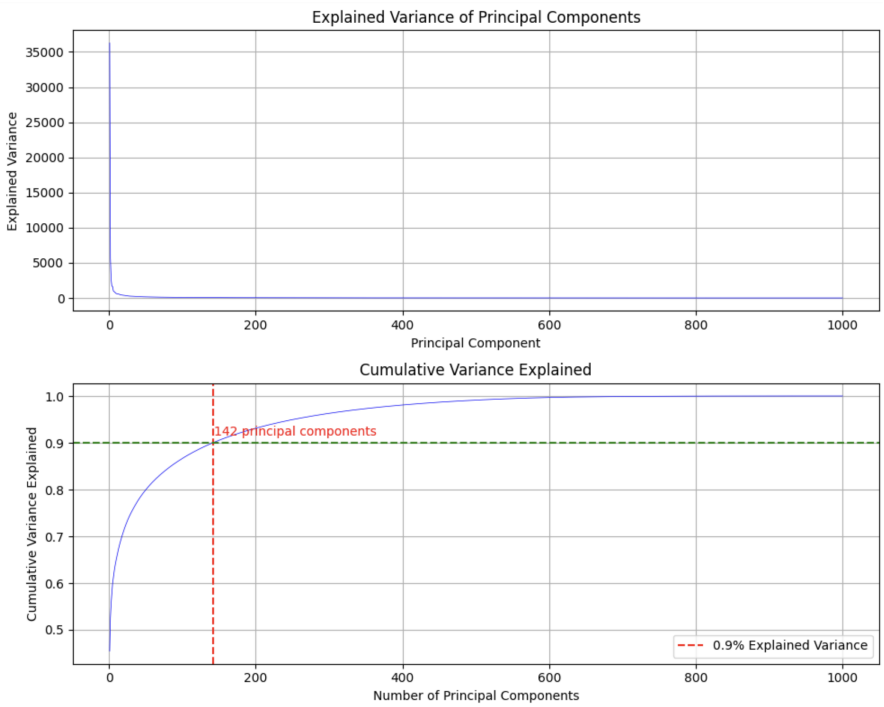
To determine the number of principal components required to retain a specified amount of variance, we set a threshold of 90%. As mentioned previously, `cumulative_variance_ratio` is an array where each element indicates the accumulated proportion of variance explained by including up to that many principal components. We used the `np.argmax()` function to find

the index of the first element in `cumulative_variance_ratio` where the cumulative explained variance equals or exceeds 90%. Given Python's use of 0-based indexing, we added 1 is added to this index to obtain the actual count of principal components. Consequently, `n_components_90` holds the number of principal components required to explain at least 90% of the total variance in the dataset.

```
threshold = 0.90

n_components_90 = np.argmax(cumulative_variance_ratio >= threshold) + 1
```

To visualize the variance explained by the principal components and the cumulative variance explained, we created a figure containing two subplots. The first subplot shows the explained variance of each principal component, providing insight into how much variance each component captures individually. The second subplot illustrates the cumulative variance explained by incrementally adding principal components, helping us determine the number of components needed to reach a specific threshold of explained variance. We highlighted the point where 90% of the variance is explained using vertical and horizontal dashed lines and text annotations. This visual representation aids in understanding the contribution of each principal component and the cumulative effect of including multiple components in the analysis.

```
plt.figure(figsize=(10, 8))

plt.subplot(2, 1, 1)
plt.plot(np.arange(1, len(variance) + 1), variance, linestyle='-', color='b',
    linewidth=0.5)
plt.title('Explained Variance of Principal Components')
plt.xlabel('Principal Component')
plt.ylabel('Explained Variance')
plt.grid(True)

print("\n"* 1)

plt.subplot(2, 1, 2)
plt.plot(np.arange(1, len(cumulative_variance_ratio) + 1),
    cumulative_variance_ratio, linestyle='-', color='b', linewidth=0.5)
plt.title('Cumulative Variance Explained')
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Variance Explained')
plt.axvline(x=n_components_90, color='r', linestyle='--', label='90% Explained
    Variance')
plt.axhline(y=0.90, color='g', linestyle='--')
plt.text(n_components_90 + 1, 0.91, f'{n_components_90} principal components', va
    ='bottom', ha='left', color='r')
plt.grid(True)
plt.legend()

plt.tight_layout()
plt.show()
```



To perform Principal Component Analysis with the objective of retaining 90% of the variance in the image dataset, we initialized the PCA model with whitening and set the number of components to `n_components_90`, which is the number needed to explain 90% of the variance. We then fitted the PCA model to the normalized image matrix (`image_matrix`). This process extracted the principal components (eigenvectors) from the fitted PCA model, which are stored in the `principal_components` array. These components define new axes in the feature space that capture the maximum variance within the dataset. Using the `transform()` method, we projected the original data onto these principal components, resulting in a lower-dimensional representation of the data stored in the `loading` matrix. To evaluate the effectiveness of the dimensionality reduction, we reconstructed the original image data from this lower-dimensional representation using `pca.inverse_transform(loading)`, resulting in `X_reconstructed`. To assess the quality of this reconstruction, we calculated the reconstruction error by comparing the original and reconstructed data using the `mean_squared_error` function. This error metric quantifies the average squared differences between the original pixel values and the reconstructed pixel values, providing a measure of how accurately the PCA model reproduces the original data using the selected principal components. The calculated reconstruction error was then printed to evaluate the extent of information loss due to the dimensionality reduction process. The dimensions of the `principal_components` matrix were retrieved, with the number of rows corresponding to the number of principal components needed to explain 90% of the variance, and the number of columns corresponding to the number of features in the original image matrix (total number of pixels in each flattened image). Thus, the shape of this matrix provides insight into the new feature space defined by the PCA.

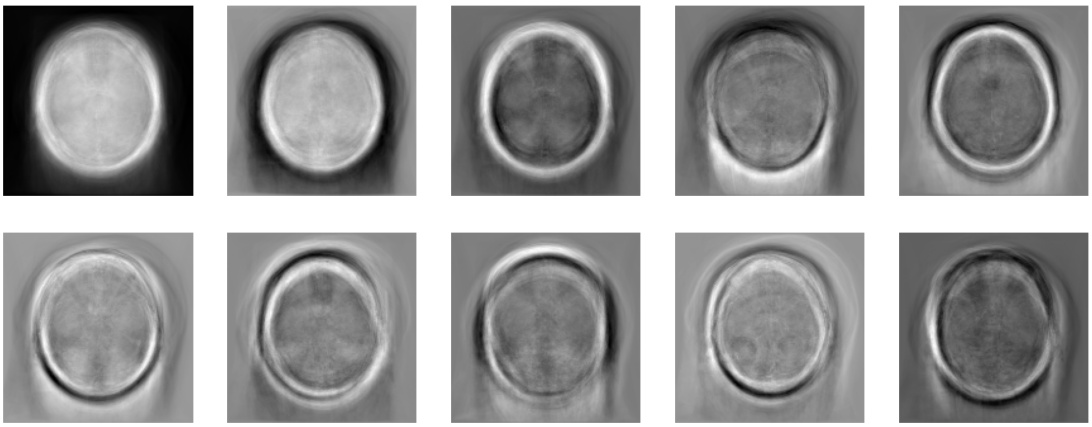
```
pca = PCA(whiten=True, n_components=n_components_90)

pca.fit(image_matrix)

principal_component= pca.components_

loading = pca.transform(image_matrix)

X_reconstructed = pca.inverse_transform(loading)

reconstrucitn_error = mean_squared_error(image_matrix,X_reconstructed)
print(reconstrucitn_error)

```

```
principal_component.shape
```

```
0.3567830058958513

(142, 22500)
```

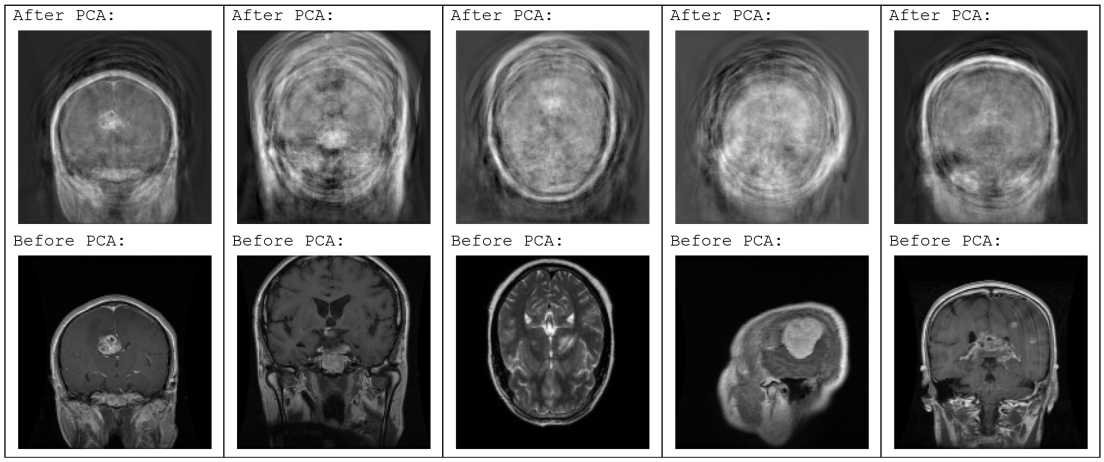
We visually examined the principal components obtained from the PCA by selecting the top 10 components and reshaping each one back into the original image dimensions. Each component was displayed using a grayscale colormap, with the axis turned off for clarity.

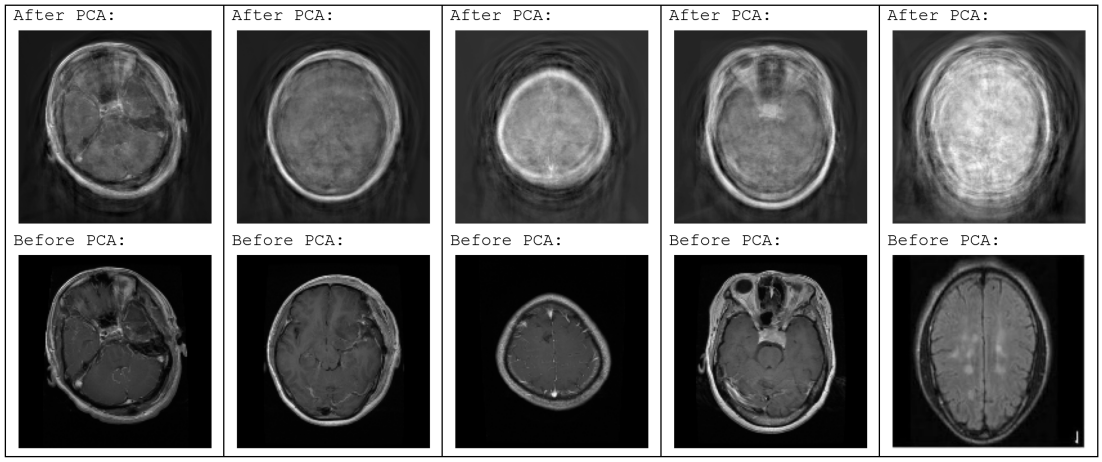
```
num_component = 10
for i in range(num_component):
    plt.imshow(principal_component[i, :].reshape(size), cmap="gray")
    plt.axis('off')
    plt.show()
```



We selected 10 random examples from the dataset to compare the original images with the reconstructed images after applying PCA. By initializing the random number generator with a seed value of 74, we ensured the reproducibility of our random selections. Each image was displayed using a grayscale colormap with the axis turned off for clarity, and the results were clearly separated for easy comparison. This visual comparison allowed us to assess the effectiveness of PCA in retaining the essential features of the original images despite the reduction in dimensionality.

```
num_examples = 10
np.random.seed(78)
for i in range(num_examples):
    index = np.random.choice(X_reconstructed.shape[0])
    print("After PCA:")
    plt.imshow(X_reconstructed[index, :].reshape(size), cmap="gray")
    plt.axis('off')
    plt.show()
    print("Before PCA:")
    plt.imshow(image_matrix[index, :].reshape(size), cmap="gray")
    plt.axis('off')
    plt.show()
    print("\n" * 2)
```





After performing PCA, we observed that the images appeared more blurred and less detailed compared to the originals, likely due to the dimensionality reduction process that can result in the loss of some fine details. Additionally, the contrast between different regions was less pronounced, as the grayscale values seemed more averaged out, making it harder to distinguish specific features of the brain tissue.

3.6 Non-negative Matrix Factorization

Given the mediocre results from PCA, we attempted to use Non-Negative Matrix Factorization (NMF) as an alternative method for dimensionality reduction.

Non-negative Matrix Factorization (NMF) is a dimensionality reduction technique that decomposes a non-negative matrix V of size $m \times n$ into two non-negative matrices W (of size $m \times k$) and H (of size $k \times n$) such that $V \approx WH$. In NMF, the two matrices W and H have distinct but complementary roles in decomposing the original matrix V. The matrix W, often referred to as the basis matrix, contains columns that represents latent bases of the original matrix V. In practical applications, these columns can be viewed as fundamental components from which the original data can be reconstructed using linear combinations. The matrix H, known as the coefficient matrix, contains columns that indicate how to combine the bases (the columns of W) to reconstruct the columns of the original matrix V. The elements of H represent the contribution of each basis to the reconstruction of the data. The relationship between W and H can be expressed as $V \approx WH$, meaning the original matrix V is approximated as a linear combination of the columns of W weighted by the coefficients in H. The non-negativity constraint on W and H results in non-negative components that often have clearer and more intuitive interpretations compared to techniques like PCA, which can produce components with both positive and negative values. This is particularly beneficial in contexts such as image analysis where pixels are non-negative).

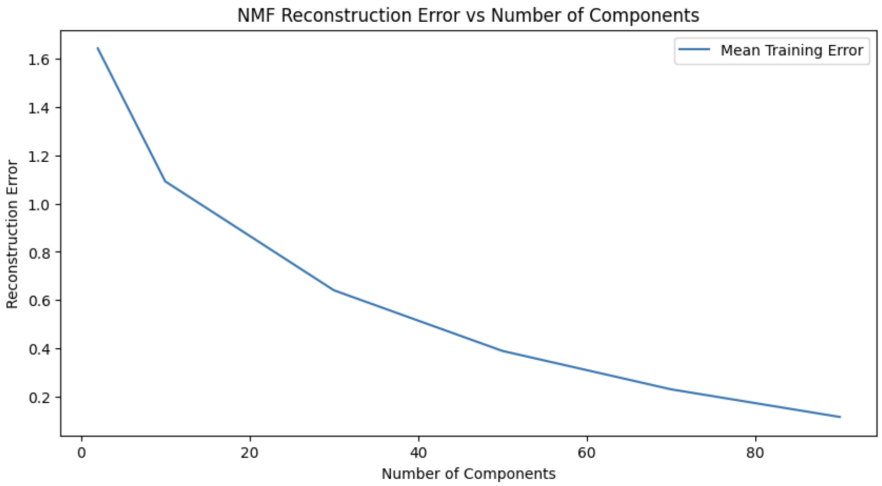
To identify the optimal number of components for NMF while maintaining reconstruction quality, we conducted an iterative evaluation on a small, balanced subset of images extracted from a larger dataset. We began by selecting 200 balanced observations from `subsampled_df` using the `select_balanced_subdf` function, ensuring representativeness of the overall data distribution. The selected images were then normalized into a matrix format using `create_matrix_of_rows_images_normalized`, which flattens each image into an array and scales the pixel values to the range [0, 1], resulting in a standardized input matrix for NMF (`small_sample`). We tested a range of ranks, starting with a rank of 2 and then from 10 to 99 in steps of 20, to determine the number of components. For each rank, we initialized an NMF model, fitted it to the normalized image matrix, and obtained the factorized matrices W and H. The data was then transformed into the NMF feature space (`encoded_images_train`) and subsequently reconstructed back into the original space (`decoded_images_train`). We computed the mean squared error (MSE) between the original and reconstructed images to evaluate the model's accuracy. This process was iterated for each rank, with the MSE, model instances, and W matrices stored for analysis. The elapsed time for each iteration was recorded to assess computational efficiency.

```
num_observations = 200
small_sample = select_balanced_subdf(subsampled_df, num_observations)
small_sample = create_matrix_of_rows_images_normalized(small_sample)

max_rank = 100
vector_1 = np.array([2])
vector_10 = np.arange(10, max_rank, 20)
ranks = np.concatenate((vector_1, vector_10))

error_vs_rank = []
models = []
W_tot = []
start_time_iteration = time.time()

for r in ranks:
    nmf = NMF(n_components=r, init='nndsvd', max_iter=5000)
    W = nmf.fit_transform(small_sample)
    H = nmf.components_
    elapsed_time_iteration = time.time() - start_time_iteration

    encoded_images_train = nmf.transform(small_sample)
    decoded_images_train = nmf.inverse_transform(encoded_images_train)
    train_error = mean_squared_error(small_sample, decoded_images_train)
    print(train_error)
    error_vs_rank.append(train_error)
    models.append(nmf)
    W_tot.append(W)

    print(f"Elapsed time for rank {r}: {elapsed_time_iteration:.2f} seconds\n")
    print("\n" * 3)
```

```
1.6443620686216813
Elapsed time for rank 2: 31.74 seconds

1.0923070897675637
Elapsed time for rank 10: 59.92 seconds

0.6403220899252511
Elapsed time for rank 30: 162.80 seconds

0.38851745786250974
Elapsed time for rank 50: 274.51 seconds

0.22917096557433478
Elapsed time for rank 70: 502.61 seconds

0.11454621622860202
Elapsed time for rank 90: 1028.49 seconds
```

To visually assess the impact of the number of components on the reconstruction error in NMF, we created a plot of the mean training error against the number of components.

```
plt.figure(figsize=(10, 5))
plt.plot(ranks, error_vs_rank, label='Mean Training Error')
plt.xlabel('Number of Components')
plt.ylabel('Reconstruction Error')
plt.legend()
plt.title('NMF Reconstruction Error vs Number of Components')
plt.show()
```

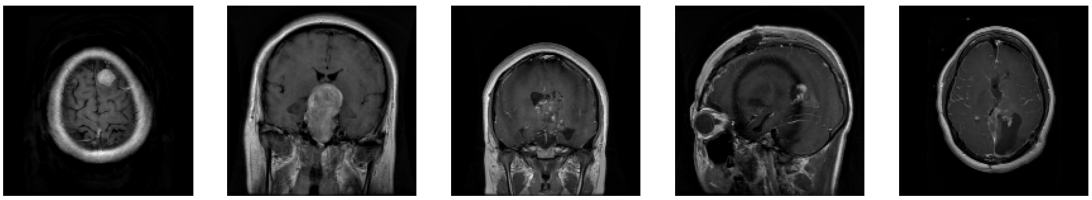


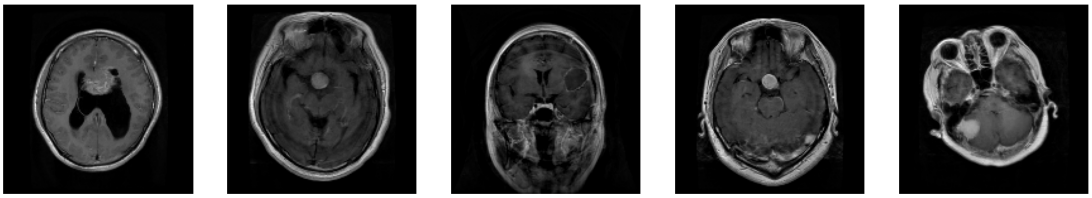
Observing the plot of reconstruction error versus the number of components for the NMF models, we see a significant decrease in error as the number of components increases. Starting with an error of around 1.6 for very few components, the reconstruction error drops sharply and continues to decline, reaching approximately 0.2 with 90 components. This trend indicates that increasing the number of components improves the reconstruction quality by reducing the error and capturing more details from the original data.
Based on this observation, we selected the 6th NMF model from our list of models, which corresponds exactly to the model with 90 components. From this model (`model`), we extracted the components matrix `H`, which represents the patterns or features identified in the data. Each row of `H` is a basis vector that describes a specific pattern, and the columns of `H` indicate how much each pattern contributes to reconstructing the original data points.

```
model = models[5]

H = model.components_
```

We visually examined the principal components obtained from the NMF by selecting the first 10 components and reshaping each one back into the original image dimensions. Each component was displayed using a grayscale colormap, with axis labels and ticks turned off for clarity.

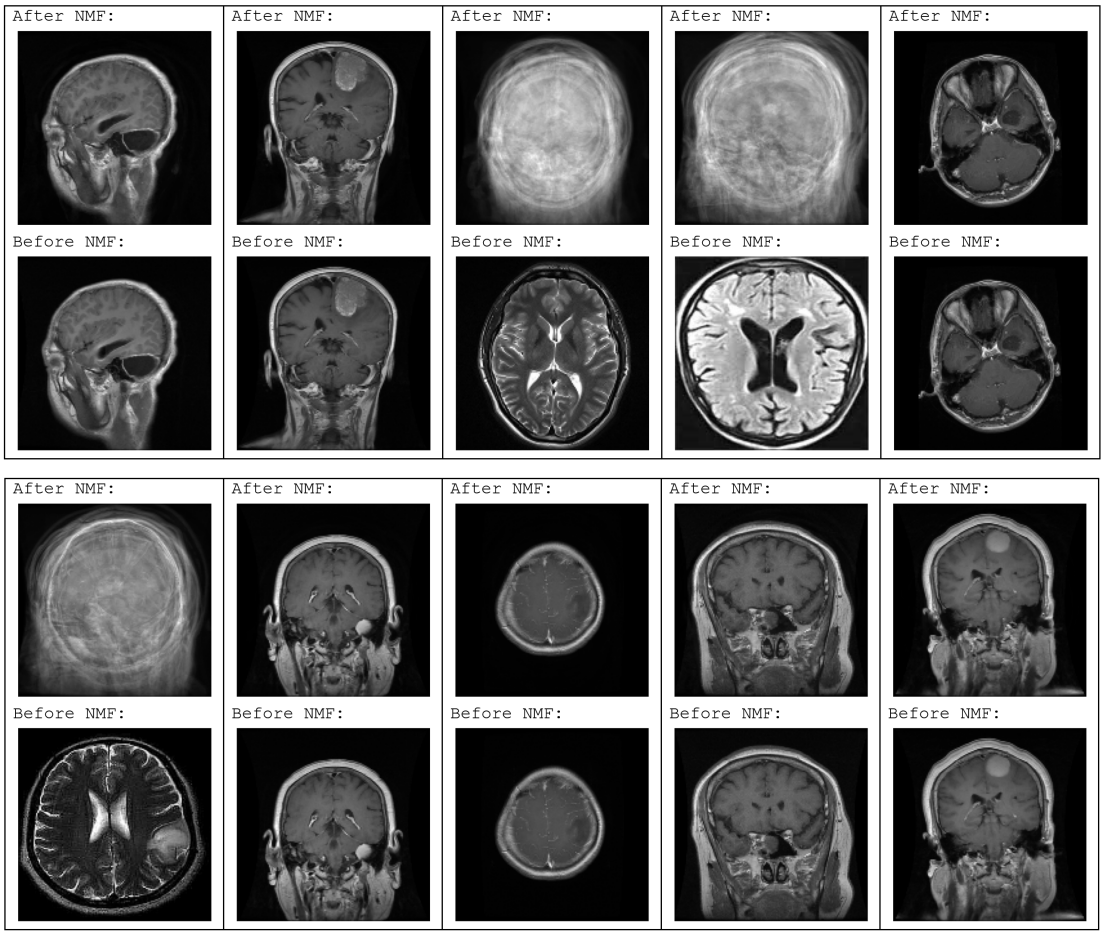
```
num_component = 10
for i in range(num_component):
    plt.imshow(H[i, :].reshape(size), cmap="gray")
    plt.axis('off')
    plt.show()
```





We randomly selected 10 images from the dataset for a detailed comparison between the original images and their reconstructions after applying NMF. By initializing the random number generator with a seed value of 40, we ensured reproducibility in our random selections. Each image was displayed using a grayscale colormap with the axis turned off for clarity, and the results were clearly separated for easy comparison. This visual comparison allowed us to observe the impact of the NMF transformation on the image quality and the preservation of key features.

```
num_examples = 10
np.random.seed(40)
for i in range(num_examples):
    index = np.random.choice(decoded_images_train.shape[0])
    print("Rows:",index)
    print("After nmf:")
    plt.imshow(decoded_images_train[index, :].reshape(size), cmap="gray")
    plt.axis('off')
    plt.show()
    print("Before nmf:")
    plt.imshow(small_sample[index, :].reshape(size), cmap="gray")
    plt.axis('off')
    plt.show()
    print("\n" * 10)
```



The results indicate that the data reduction did not significantly affect the main elements of the images, and the process to identify the optimal rank worked effectively. After reconstructing the images using NMF, we observed several differences and similarities compared to the original images. Most of the images post-NMF show a reasonable retention of the main anatomical structures of the brain, with the overall shape and major features preserved. Similar to the effects seen with PCA, there is noticeable blurriness and softening of details. Fine features are less distinct due to the dimensionality reduction and approximation inherent in NMF. Additionally, the contrast between different regions is somewhat reduced, with grayscale values appearing more homogenized, making it harder to distinguish subtle differences in tissue types.

4. Outlier Detection

In this chapter, we first applied an Autoencoder, followed by Isolation Forest and Non-Negative Matrix Factorization, to identify potential outliers.

4.1 Outlier Detection with Autoencoder

4.1.1 Libraries

These imports collectively set up the environment for performing data processing, model training, evaluation, and visualization necessary for developing and assessing an autoencoder model for outlier detection.

```
import cv2
import random
import gc
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from keras.applications.vgg16 import VGG16
from keras.applications.vgg16 import preprocess_input
from sklearn.metrics import mean_squared_error
from sklearn.utils.multiclass  import unique_labels
from sklearn.metrics import classification_report, confusion_matrix,
    ConfusionMatrixDisplay
from sklearn.model_selection import train_test_split
from tensorflow.keras.layers import  Layer
from tensorflow.keras.layers import Input, Flatten, Dense, Reshape
from tensorflow.keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import plot_model
from tensorflow.keras.models import load_model, save_model
```

4.1.2 Data Preparation

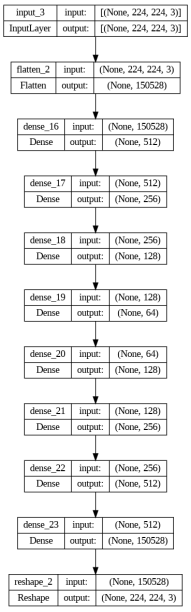
First, we created the `df` object to represent the DataFrame contained in the pickle file named "complete_df.pkl". To facilitate the analysis, we divided the data into separate DataFrames based on the tumor type. This was achieved by filtering the original DataFrame for each specific label. Additionally, deleting the original DataFrame after the segmentation helped optimize memory usage, ensuring efficient processing of the data subsets.

```
from google.colab import drive
drive.mount('/content/drive')
file_path = '/content/drive/MyDrive/ProgettoDataMining/complete_df.pkl'
df = pd.read_pickle(file_path)

df_glioma = df[df['Label'] == 'glioma']
df_notumor = df[df['Label'] == 'notumor']
df_pituitary_tumor = df[df['Label'] == 'pituitary tumor']
df_meningioma = df[df['Label'] == 'meningioma']
del df
```

4.1.3 Creation of the Autoencoder model

We designed an autoencoder model for processing the MRI images with a shape of $224 \times 224 \times 3$. The model begins with an input layer that accepts images of this specified shape. The input is then flattened into a 1D array, which is subsequently passed through three dense layers with 512, 256, and 128 units, respectively, each using ReLU activation functions. This sequence compresses the image data into a more compact representation. The compressed data is then encoded into a latent vector of size 64 using another dense layer with ReLU activation. The decoder portion of the autoencoder reconstructs the image by passing the latent vector through three dense layers with 128, 256, and 512 units, respectively, also using ReLU activation functions. The final dense layer reshapes the data back to the original image dimensions, utilizing a sigmoid activation function to ensure pixel values are within the appropriate range. This reconstructed image is then reshaped to $224 \times 224 \times 3$. We compiled the complete autoencoder model and provided a summary of its architecture. Additionally, we generated a visual representation of the model architecture using the `plot_model` function, saving it as `model_plot.png` to visualize the structure and layer configurations.

```
input_img = Input(shape=(224, 224, 3))

x = Flatten()(input_img)

x = Dense(512, activation='relu')(x)
x = Dense(256, activation='relu')(x)
x = Dense(128, activation='relu')(x)

latent = Dense(64, activation='relu')(x)

x = Dense(128, activation='relu')(latent)
x = Dense(256, activation='relu')(x)
x = Dense(512, activation='relu')(x)

decoded = Dense(224*224*3, activation='sigmoid')(x)
decoded = Reshape((224, 224, 3))(decoded)

autoencoder = Model(input_img, decoded)

autoencoder.summary()

plot_model(autoencoder, to_file='model_plot.png', show_shapes=True,
    show_layer_names=True)
```



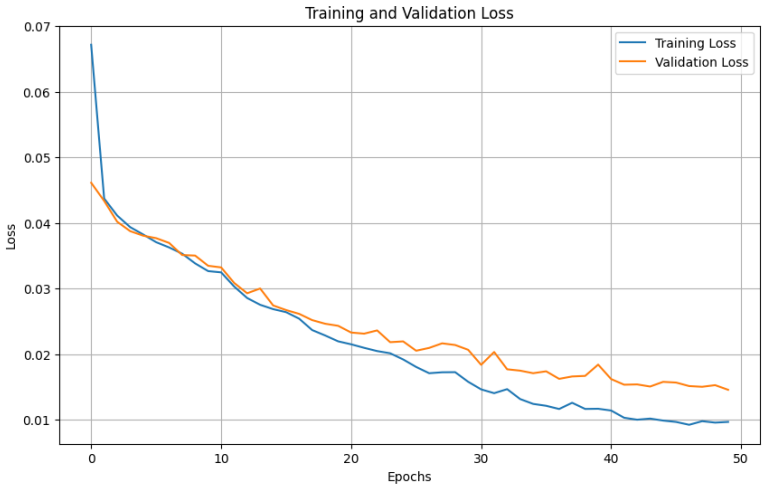
4.1.4 Training, tuning and external validation for notumor

After creating the autoencoder model, we proceeded with outlier detection for the "notumor" category.
The first step involved normalizing and reshaping the images. We iterated through the "Image" column of the `df_notumor` DataFrame, normalizing each image by dividing by 255 to scale the pixel values to the range [0, 1]. Each image was then reshaped to the required input dimensions for the model, specifically 224×224×3. These reshaped images were collected into a list, which was subsequently converted into a NumPy array with the appropriate shape. The data type of this array was set to float16 to optimize memory usage and computational efficiency.
To facilitate model training and evaluation, we split the dataset into training and testing sets using an 80-20 split. This was achieved using the `train_test_split` function from `scikit-learn`, with 20% of the data reserved for testing and a fixed random state to ensure reproducibility.

```
X = []
for i,x in enumerate(df_notumor["Image"].values):
    x = x/255
    new = x.reshape(1, 224, 224, 3)
    X.append(new)
X = np.vstack(X)
X= X.astype(np.float16)

X_train_notumor, X_test_notumor = train_test_split(X, test_size=0.2, random_state
    =42)

del X
```

Next, we compiled and trained the autoencoder model on the training data. The compilation step involved setting the optimizer to Adam with a learning rate of 0.0005 and using mean squared error (MSE) as the loss function. This configuration is aimed at efficiently minimizing the reconstruction error during training. The training process was conducted on the `X_train_notumor` dataset. We set the training parameters to 50 epochs, a batch size of 32, and enabled shuffling to ensure the training data was well mixed. Additionally, we used a validation split of 20% from the training data to monitor the model's performance on unseen data during training. The autoencoder was trained to learn the normal patterns in the "no tumor" images. The training history, including loss and validation loss metrics, was recorded for further analysis to evaluate the model's performance over the training epochs. The training process was monitored by plotting the training and validation loss.

```
autoencoder.compile(optimizer=Adam(learning_rate=0.0005), loss='mse')

history = autoencoder.fit(X_train_notumor,X_train_notumor,
                          epochs=50,
                          batch_size=32,
                          shuffle=True,
                          validation_split=0.2)

autoencoder = tf.keras.models.load_model('/content/drive/MyDrive/
    ProgettoDataMining/modello_salvato.keras')

plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```



The plot revealed a decrease in both losses over time, indicating effective learning and good model performance.

To simulate the detection of outliers for the "notumor" category, we generated a set of images from the tumor categories (glioma, meningioma, and pituitary tumor) to act as outliers. We conducted this simulation by randomly selecting 540 images from these tumor datasets. For each iteration, we set a random seed for reproducibility and randomly chose one of the tumor dataframes. We then sampled a single image from the chosen dataframe and normalized it by dividing by 255. This normalized sample was added to the simulation list. We split the simulated outliers into two subsets: the first 240 images were designated for validation (`X_val_outlier_notumor`), and the remaining 300 images were reserved for testing (`X_test_outlier_notumor`). Additionally, a portion of the "notumor" training set was selected for validation (`X_val_notumor`), specifically the images from indices 960 to 1200. The `X_test_notumor` set, defined earlier, remained unchanged. This setup allowed us to evaluate the autoencoder's ability to distinguish between normal "notumor" images and outliers from the other tumor categories during the validation and testing phases.

```
n = 540
simulation = []

for i in range(n):
  random.seed(i)
  df = random.choice([df_meningioma, df_glioma, df_pituitary_tumor])
  sample = df.sample(n=1, random_state=i)
  sample = (sample.iloc[0,0])/255
  simulation.append(sample)

X_val_outlier_notumor = simulation[:240]
X_test_outlier_notumor = simulation[240:]

X_val_notumor = X_train[960:1200]
```

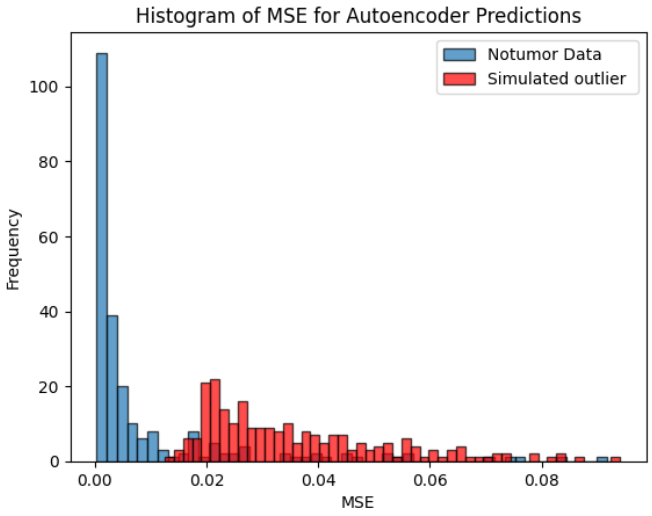
To detect outliers using our trained autoencoder, we established a method based on the mean squared error (MSE) between the original and reconstructed images. If the MSE exceeds a certain threshold, the image is considered an outlier.

We visualized the range of MSE values to determine an appropriate threshold. First, we evaluated the MSE for each simulated outlier in the validation set. For each simulated outlier image in `X_val_outlier_notumor`, we reshaped the image to match the input shape of the autoencoder, predicted the reconstruction, and calculated the MSE between the predicted and actual images. These MSE values were stored in the `mse_simulation_validation_notumor` list. Next, we performed a similar evaluation for the "notumor" images in the validation set. Each image in `X_val_notumor` was reshaped, reconstructed using the autoencoder, and the MSE was calculated between the reconstructed and original images. These MSE values were stored in the `mse_tot_validation_notumor` list.

By comparing the MSE distributions of the "notumor" images and the simulated outliers, we could visualize the range of MSE values and identify an appropriate threshold to distinguish

between normal and anomalous images. To visualize the range of MSE values and identify an appropriate threshold for outlier detection, we plotted histograms of the MSE values for the validation sets. The first histogram represents the MSE distribution for the "notumor" validation set (`mse_tot_validation_notumor`), and the second histogram represents the MSE distribution for the simulated outliers (`mse_simulation_validation_notumor`). The histogram for the "no tumor" data is plotted with 50 bins, showing the frequency of different MSE values, and is labeled "No Tumor Data". The histogram for the simulated outliers, also with 50 bins, is plotted in red and labeled "Simulated Outliers". The x-axis represents the MSE values, while the y-axis indicates the frequency of these values. The title "Histogram of MSE for Autoencoder Predictions" summarizes the content of the plot.

```
mse_simulation_validation_notumor = []

for i  in range(len(X_val_outlier_notumor)):
  print(i)
  observation = X_val_outlier_notumor[i]
  processed_sample = observation.reshape(1,224,224,3)
  pred = autoencoder.predict(processed_sample)
  mse = mean_squared_error(pred.flatten(), observation.flatten())
  mse_simulation_validation_notumor.append(mse)

mse_tot_validation_notumor = []

for i in range(len(X_val_notumor)):
   print(i)
   real = X_val_notumor[i]
   preprocessed_real= real.reshape(1,224,224,3)
   pred = autoencoder.predict(preprocessed_real)
   mse = mean_squared_error(pred.flatten(), real.flatten())
   mse_tot_validation_notumor.append(mse)

plt.hist(mse_tot_validation_notumor, bins=50, edgecolor='k', alpha=0.7, label='
    Notumor Data')
plt.hist(mse_simulation_validation_notumor, bins=50, edgecolor='k', color='red',
    alpha=0.7, label='Simulated outlier ')
plt.xlabel('MSE')
plt.ylabel('Frequency')
plt.title('Histogram of MSE for Autoencoder Predictions')
plt.legend()
plt.show()
```



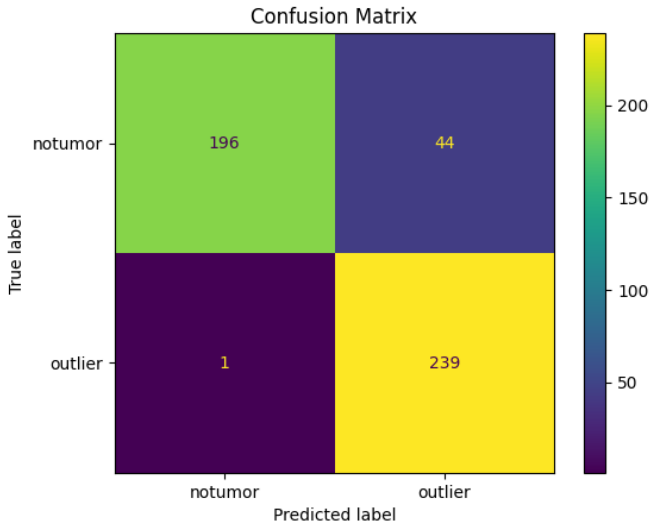
The histogram of MSE values for "no tumor" images (blue) shows that most values are clustered near zero, indicating that the autoencoder can accurately reconstruct these images. In contrast, the histogram of MSE values for simulated outliers (red) demonstrates a broader distribution with higher MSE values, highlighting the autoencoder's difficulty in reconstructing images from tumor categories on which it was not trained. This visualization is crucial for determining a

range of possible thresholds, which, as we can see from the graph, fall between 0.01 and 0.02. Based on the range suggested by the histogram, we conducted an analysis using a series of threshold values to validate their effectiveness for outlier detection. We created ground truth labels for the validation set, where images from the "notumor" category were labeled as "notumor" and simulated outlier images were labeled as "outlier". We then defined a range of thresholds from 0.010 to 0.025, evenly spaced in 10 steps. For each threshold value, we predicted whether each image in the validation set was "notumor" or "outlier" based on its MSE. If the MSE of an image was below the threshold, it was classified as "notumor"; otherwise, it was classified as "outlier". These predictions were then compared to the ground truth labels to evaluate the performance of each threshold. The evaluation metrics included the confusion matrix and the classification report, which provided detailed insights into the precision, recall, and F1-score for both classes.

```
list1 = ["notumor" for _ in range(len(mse_tot_validation_notumor))]
list2 = ["outlier" for _ in range(len(mse_simulation_validation_notumor))]

y_validation = list1 + list2

thresholds = np.linspace(0.010, 0.025, 10)

print()

for threshold in thresholds:
    print("For this thereshold value",threshold, "the confusion matrix is ")
    print("\n"*3)
    preds_1 = []
    preds_2 = []
    for error in mse_tot_validation_notumor:
        if error < threshold:
            pred = "notumor"
        else:
            pred = "outlier"
        preds_1.append(pred)
    for error in mse_simulation_validation_notumor:
        if error < threshold:
            pred = "notumor"
        else:
            pred = "outlier"
        preds_2.append(pred)
    y_pred_validation = preds_1 + preds_2
    evaluate_metrics(y_pred_validation, y_validation)
    print("\n"*10)
```

By iterating over the range of thresholds, we identified the optimal threshold value that best distinguished between normal and anomalous images, ensuring robust outlier detection for the "notumor" category. The chosen threshold value was 0.013. Below are the evaluation metrics for this threshold.

```
              precision    recall  f1-score   support

     notumor       0.99      0.82      0.90       240
     outlier       0.84      1.00      0.91       240

    accuracy                           0.91       480
   macro avg       0.92      0.91      0.91       480
weighted avg       0.92      0.91      0.91       480
```

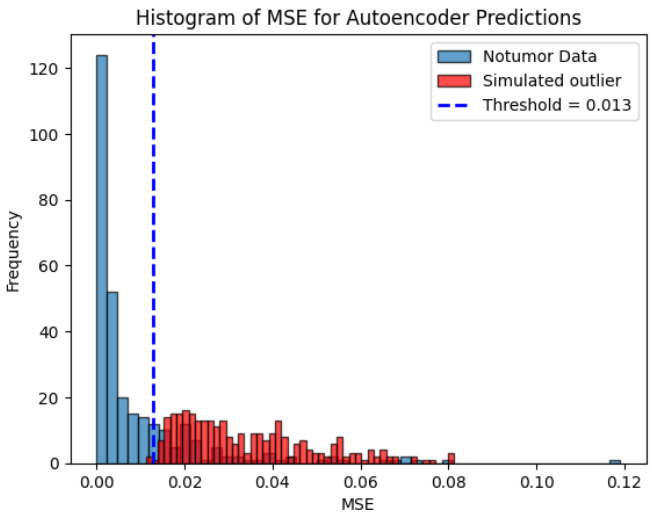


The confusion matrix for the threshold value of 0.013 reveals that out of 240 "no tumor" images, 196 were correctly classified, and 44 were misclassified as outliers. Conversely, out of 240 simulated outlier images, 239 were correctly identified as outliers, and only 1 was misclassified as "notumor".

To validate our autoencoder model and its ability to detect outliers, we tested it using a separate test set. This test set included images from both the "notumor" category and simulated outliers from other tumor categories. The goal was to assess the model's performance on data it had not seen during training or validation. First, we calculated the Mean Squared Error (MSE) for each image in the test set. For the "notumor" test set (`X_test_notumor`), each image was reshaped to the required input dimensions, reconstructed using the autoencoder, and compared to the original image to compute the MSE. These values were stored in `mse_tot_test_notumor`. Similarly, we calculated the MSE for the simulated outliers in the test set (`X_test_outlier_notumor`). Each simulated outlier image was processed in the same manner, and the resulting MSE values were stored in `mse_simulation_test_notumor`. We then plotted histograms of the MSE values for both sets. The histogram for the "notumor" data is shown in blue, while the histogram for the simulated outliers is shown in red. We also included a vertical line representing the chosen threshold of 0.013, which was determined during the validation phase.

```
mse_simulation_test_notumor = []

for i  in range(len(X_test_outlier_notumor)):
  print(i)
  observation = X_test_outlier_notumor[i]
  processed_sample = observation.reshape(1,224,224,3)
  pred = autoencoder.predict(processed_sample)
  mse = mean_squared_error(pred.flatten(), observation.flatten())
  mse_simulation_test_notumor.append(mse)

mse_tot_test_notumor = []

for i in range(len(X_test_notumor)):
   print(i)
   real = X_test_notumor[i]
   preprocessed_real = real.reshape(1,224,224,3)
   pred = autoencoder.predict(preprocessed_real)
   mse = mean_squared_error(pred.flatten(), real.flatten())
   mse_tot_test_notumor.append(mse)

plt.hist(mse_tot_test_notumor, bins=50, edgecolor='k', alpha=0.7, label='Notumor
    Data')
plt.hist(mse_simulation_test_notumor, bins=50, edgecolor='k', color='red', alpha
    =0.7, label='Simulated outlier ')
plt.xlabel('MSE')
plt.ylabel('Frequency')
plt.title('Histogram of MSE for Autoencoder Predictions')
threshold = 0.013
```

```
plt.axvline(threshold, color='blue', linestyle='--', linewidth=2, label=f'
    Threshold = {threshold}')
plt.legend()
plt.show()
```



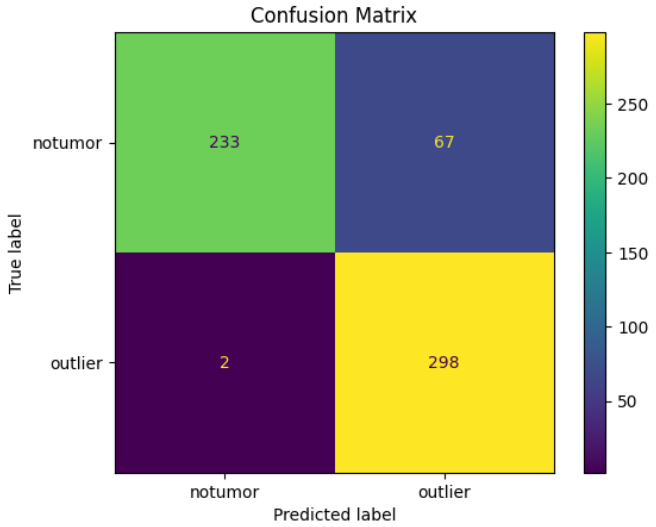
This visualization clearly demonstrates the separation between the MSE distributions of normal and anomalous images, reinforcing the effectiveness of our chosen threshold. From the graph, we can see that the MSE values for "notumor" data are primarily concentrated between 0 and 0.02. The MSE values for simulated outliers are more dispersed and can reach higher values, up to 0.08 or beyond. The chosen threshold of 0.013 effectively separates the two distributions, ensuring reliable detection of outliers.

To test the performance of our autoencoder-based classifier on the test set, we first created the ground truth labels. The test set consisted of images from the "no tumor" category and simulated outliers. We labeled the "no tumor" images as "notumor" and the simulated outliers as "outlier". We then used the previously determined threshold of 0.013 to classify the test images based on their Mean Squared Error (MSE) values. For each image in the "no tumor" test set, if the MSE was below the threshold, it was classified as "notumor"; otherwise, it was classified as "outlier". This process was repeated for the simulated outliers. The predictions for both "no tumor" and simulated outliers were combined into a single list (`y_pred_test`) and compared to the ground truth labels (`y_test`). We evaluated the performance of our classifier using metrics such as the confusion matrix and the classification report, which includes precision, recall, and F1-score for both classes.

```
list1 = ["notumor" for _ in range(len(mse_tot_test_notumor))]
list2 = ["outlier" for _ in range(len(mse_simulation_test_notumor))]

y_test = list1 + list2

preds_1 = []
preds_2 = []

for error in mse_tot_test_notumor:
    if error < threshold:
        pred = "notumor"
    else:
        pred = "outlier"
    preds_1.append(pred)

for error in mse_simulation_test_notumor:
    if error < threshold:
        pred = "notumor"
    else:
        pred = "outlier"
    preds_2.append(pred)

```

```
y_pred_test = preds_1 + preds_2

evaluate_metrics(y_pred_test, y_test)
```

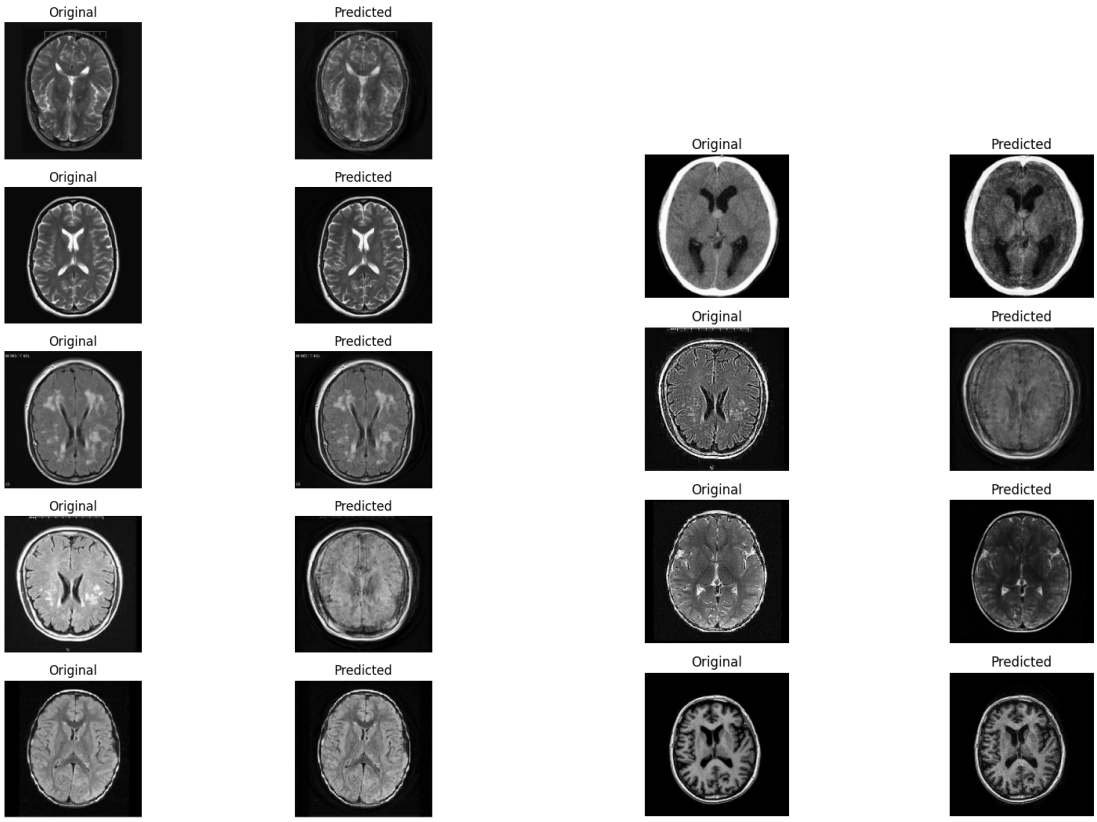
```
              precision    recall  f1-score   support

     notumor       0.99      0.78      0.87       300
     outlier       0.82      0.99      0.90       300

    accuracy                           0.89       600
   macro avg       0.90      0.89      0.88       600
weighted avg       0.90      0.89      0.88       600
```



The results indicated a robust performance of the autoencoder in distinguishing between normal and anomalous images, with an overall accuracy of 0.89. The recall for "notumor" images was 0.78, while the recall for "outlier" images was 0.99. This means that the model correctly identified 78% of the normal "notumor" images but mistakenly classified 22% of them as outliers, indicating a conservative approach that favors detecting potential anomalies over missing them. On the other hand, the model demonstrated a high effectiveness in detecting outliers, correctly identifying 99% of the anomalous images and missing only 1%. The confusion matrix revealed that out of 300 "no tumor" images, 233 were correctly classified, and 67 were misclassified as outliers. Conversely, out of 300 simulated outlier images, 298 were correctly identified as outliers, and only 2 were misclassified as "notumor".
To further evaluate the performance of our trained autoencoder model, we visualized a set of original and reconstructed images from the test set. We selected 9 images from the "notumor" test set and displayed them alongside their corresponding reconstructions produced by the autoencoder. The visualization is organized in a 2x9 grid, where each row displays an original image on the left and the corresponding reconstructed image on the right. The original images are presented in their true form, while the reconstructed images show how well the autoencoder was able to recreate the input images. By examining these visualizations, we can qualitatively assess the reconstruction quality of the autoencoder. Good reconstructions indicate that the model has effectively learned the features of the "notumor" images, while significant discrepancies between the original and reconstructed images might highlight areas where the model could be improved.

```
plt.figure(figsize=(10, 20))

for i in range(9):

    plt.subplot(9, 2, 2 * i + 1)
    real = X_test_notumor[i].astype(np.float32)
    plt.imshow(real)
    plt.title('Original')
    plt.axis('off')
```

```
    plt.subplot(9, 2, 2 * i + 2)
    z = real.reshape(1, 224, 224, 3)
    k = autoencoder.predict(z)
    k_new = k[0, :, :, :]
    plt.imshow(k_new)
    plt.title('Predicted')
    plt.axis('off')

plt.tight_layout()
plt.show()
```

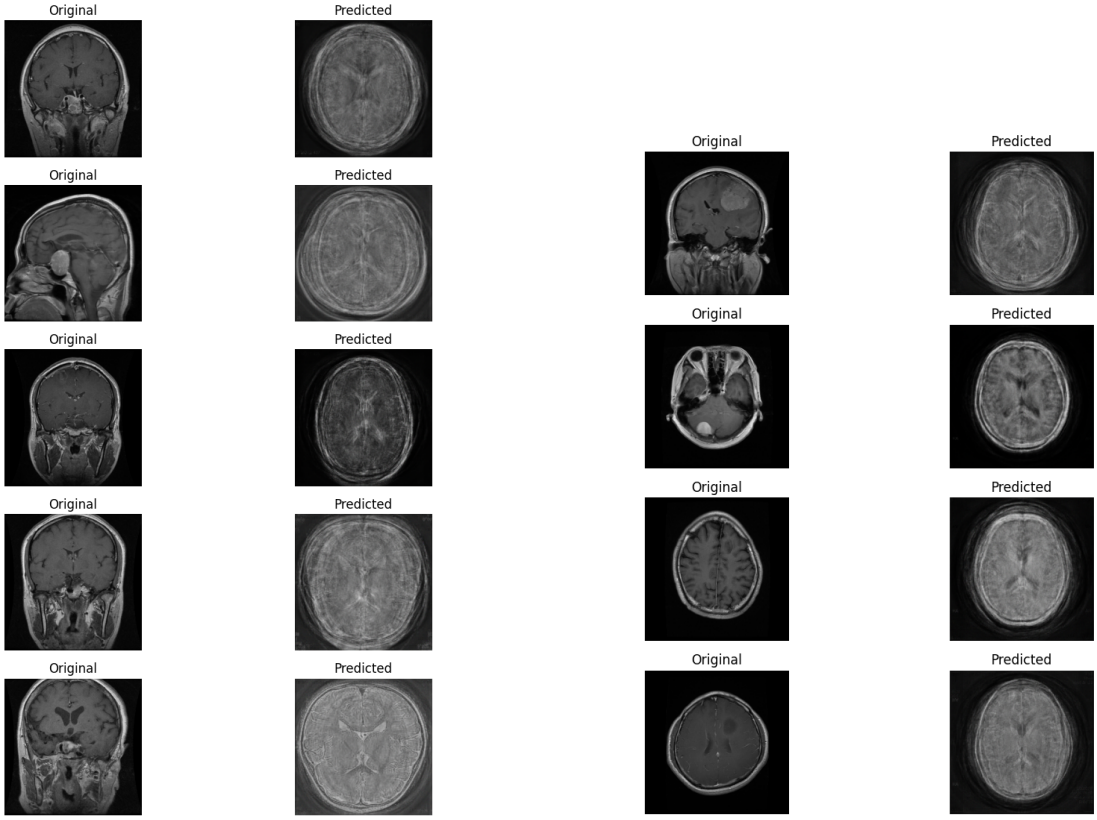


The visual comparison between the original and reconstructed images from the "notumor" test set reveals that the autoencoder has effectively learned to capture the essential features of normal brain scans. In most cases, the reconstructed images closely resemble the original ones, indicating high reconstruction accuracy.

To further evaluate the performance of our trained autoencoder model on outlier detection, we visualized a set of original and reconstructed images from the outlier test set. We selected 9 images from the simulated outliers and displayed them alongside their corresponding reconstructions produced by the autoencoder. The visualization is organized in a 2x9 grid, where each row displays an original image on the left and the corresponding reconstructed image on the right. The original images are presented in their true form, while the reconstructed images show how the autoencoder attempted to recreate the input images. By examining these visualizations, we can qualitatively assess the reconstruction quality of the autoencoder for outlier images. Typically, the reconstructions of outlier images should be poorer compared to the "notumor" images, indicating that the autoencoder has learned to distinguish normal from anomalous patterns.

```
plt.figure(figsize=(10, 20))

for i in range(9):

    plt.subplot(9, 2, 2 * i + 1)
    real = X_test_outlier_notumor[i].astype(np.float32)
    plt.imshow(real)
    plt.title('Original')
```

```
    plt.axis('off')

    plt.subplot(9, 2, 2 * i + 2)
    z = real.reshape(1, 224, 224, 3)
    k = autoencoder.predict(z)
    k_new = k[0, :, :, :]
    plt.imshow(k_new)
    plt.title('Predicted')
    plt.axis('off')

plt.tight_layout()
plt.show()
```

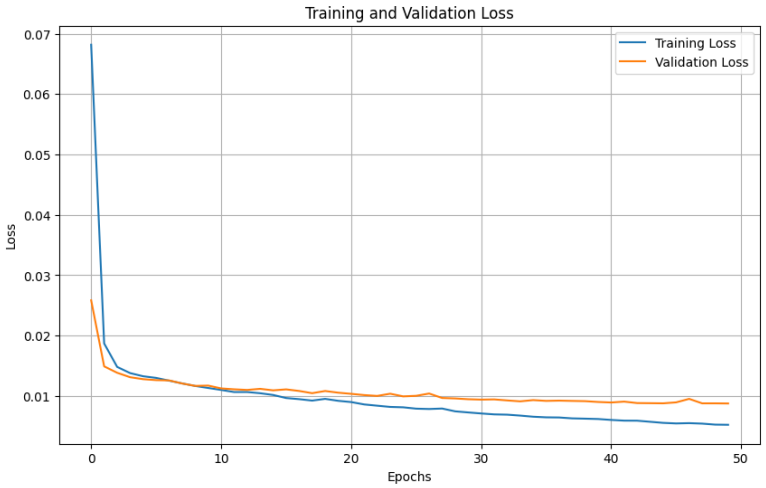


The visual comparison between the original and reconstructed images from the "outlier" test set reveals that the autoencoder has effectively learned to distinguish normal from anomalous patterns. The original images are clear and detailed, while the reconstructed images are of noticeably lower quality. This significant discrepancy indicates that the autoencoder struggles to accurately reconstruct outlier images, which is a desired outcome.

4.1.5 Training, tuning and external validation for glioma

We repeated the same process for outlier detection for the "glioma" category.
To preprocess the glioma images for our autoencoder model, we normalized the pixel values by scaling them to the range [0, 1] and reshaped each image to the dimensions 224×224×3. These reshaped images were then aggregated into a single NumPy array and converted to the float16 data type to optimize memory usage and computational efficiency.
The dataset was subsequently divided into training and testing sets (`X_train_glioma` and `X_test_glioma`) using an 80-20 split, ensuring reproducibility by setting a fixed random state. This preprocessing pipeline was essential for preparing the data for effective training and evaluation of our autoencoder model.
To train our autoencoder model for glioma image reconstruction, we compiled the model using the Adam optimizer with a learning rate of 0.0005 and the mean squared error (MSE) loss function. The model was trained on the glioma training dataset (`X_train_glioma`). We set the

training parameters to 50 epochs, a batch size of 32, and enabled shuffling to ensure the training data was well mixed. Additionally, we used a validation split of 20% from the training data to monitor the model's performance on unseen data during training. The training process was monitored by plotting the training and validation loss.

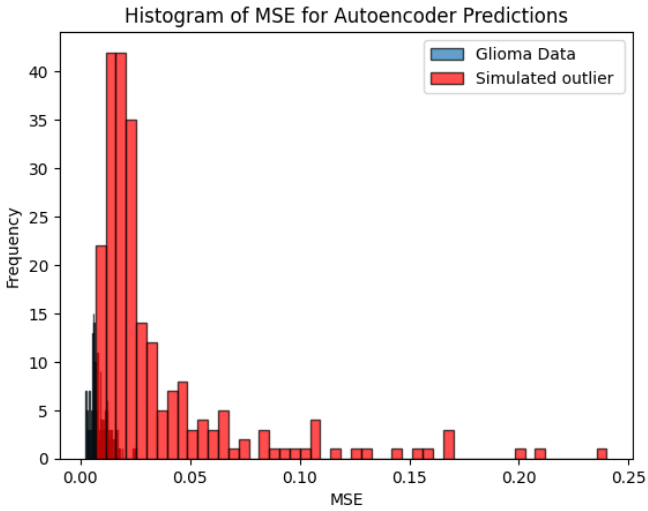


The plot revealed a decrease in both losses over time, indicating effective learning and good model performance.

For outlier detection simulation in the "glioma" category, we generated a set of images from the meningioma, notumor, and pituitary tumor categories to act as outliers. We randomly selected 514 images from these datasets, normalized them by scaling the pixel values to the range [0, 1], and collected them into a list. This list was then split into two subsets: the first 228 images were designated for validation (`X_val_outlier_glioma`), and the remaining images were reserved for testing (`X_test_outlier_glioma`). Additionally, a portion of the glioma training set was set aside for validation (`X_val_glioma`), specifically the images from indices 972 to 1200. The `X_test_glioma` set, defined earlier, remained unchanged.

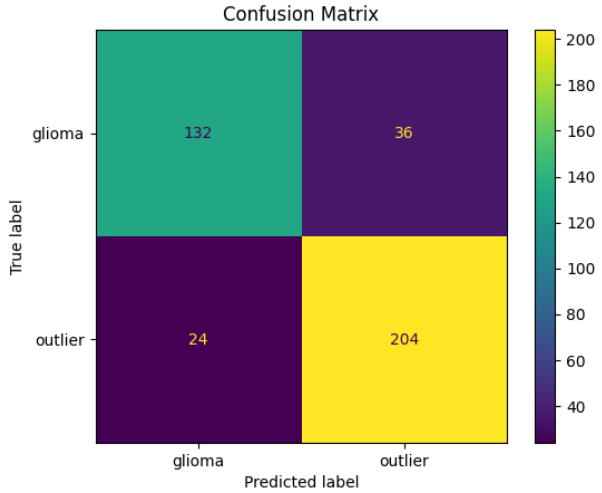
As before, to detect outliers using our trained autoencoder, we established a method based on the mean squared error (MSE) between the original and reconstructed images. If the MSE exceeds a certain threshold, the image is considered an outlier.

First, we calculated the Mean Squared Error (MSE) for both validation glioma images and simulated outliers. For the simulated outliers (`X_val_outlier_glioma`), we processed each image, predicted it using the autoencoder, and computed the MSE between the predicted and original images, storing these values in a list (`mse_simulation_validation_glioma`). Similarly, we computed the MSE for the glioma validation images (`X_val_glioma`), storing them in a list (`mse_tot_validation_glioma`). We then plotted histograms of these MSE values: one for the glioma validation set and one for the simulated outliers. By comparing the MSE distributions of the glioma images and the simulated outliers, we could visualize the range of MSE values and identify an appropriate threshold to distinguish between normal and anomalous images.



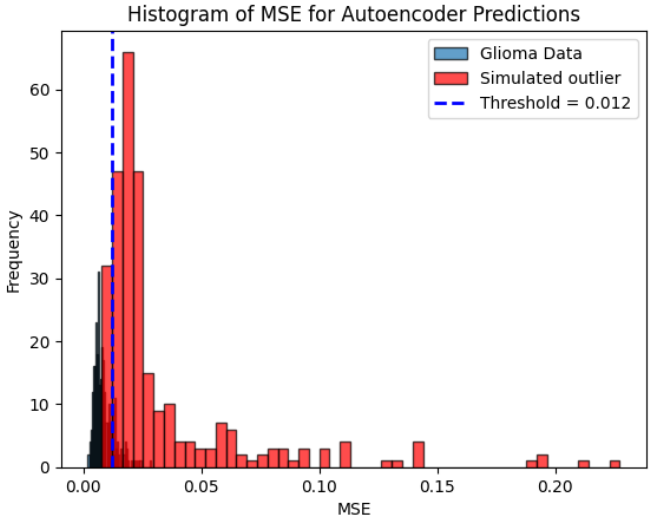
The histogram helped identify a range of thresholds to test. To determine the optimal threshold for distinguishing between glioma images and outliers, we conducted an analysis using these threshold values. We first created ground truth labels for the validation set, with "glioma" labels for the true glioma images and "outlier" labels for the simulated outliers. We then defined a range of thresholds from 0.001 to 0.025, evenly spaced in 10 steps. For each threshold, we classified the images based on their MSE values: if the MSE was below the threshold, the image was labeled as "glioma"; otherwise, it was labeled as "outlier". We iterated over each threshold, generating predictions for both the glioma and simulated outlier validation sets. The combined predictions were compared against the ground truth labels to evaluate the performance of each threshold using the evaluation metrics. By iterating over the range of thresholds, we identified the optimal threshold value: 0.012. Below are the evaluation metrics for this threshold.

	precision	recall	f1-score	support
glioma	0.85	0.79	0.81	168
outlier	0.85	0.89	0.87	228
accuracy			0.85	396
macro avg	0.85	0.84	0.84	396
weighted avg	0.85	0.85	0.85	396



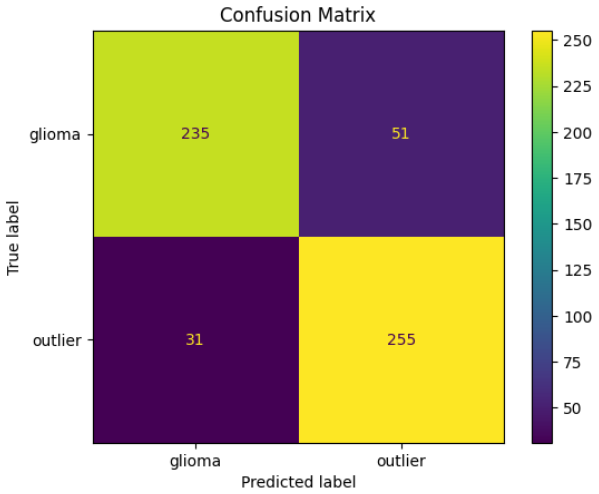
The confusion matrix for the threshold value of 0.012 reveals that out of 168 glioma images, 132 were correctly classified, and 36 were misclassified as outliers. Conversely, out of 228 simulated outlier images, 204 were correctly identified as outliers, and 24 were misclassified as glioma.
Subsequently, to evaluate the performance of the autoencoder on the test set, we calculated the Mean Squared Error (MSE) for both glioma and simulated outlier images. For the sim-

ulated outliers (`X_test_outlier_glioma`), each image was reshaped and processed by the autoencoder, and the MSE between the predicted and original images was computed and stored in `mse_simulation_test_glioma`. Similarly, for the glioma test images (`X_test_glioma`), each image was processed by the autoencoder, and the resulting MSE values were stored in `mse_tot_test_glioma`. We then plotted histograms of the MSE values for both sets. The histogram for the glioma data is shown in blue, while the histogram for the simulated outliers is shown in red. We also included a vertical line representing the chosen threshold of 0.012, which was determined during the validation phase.



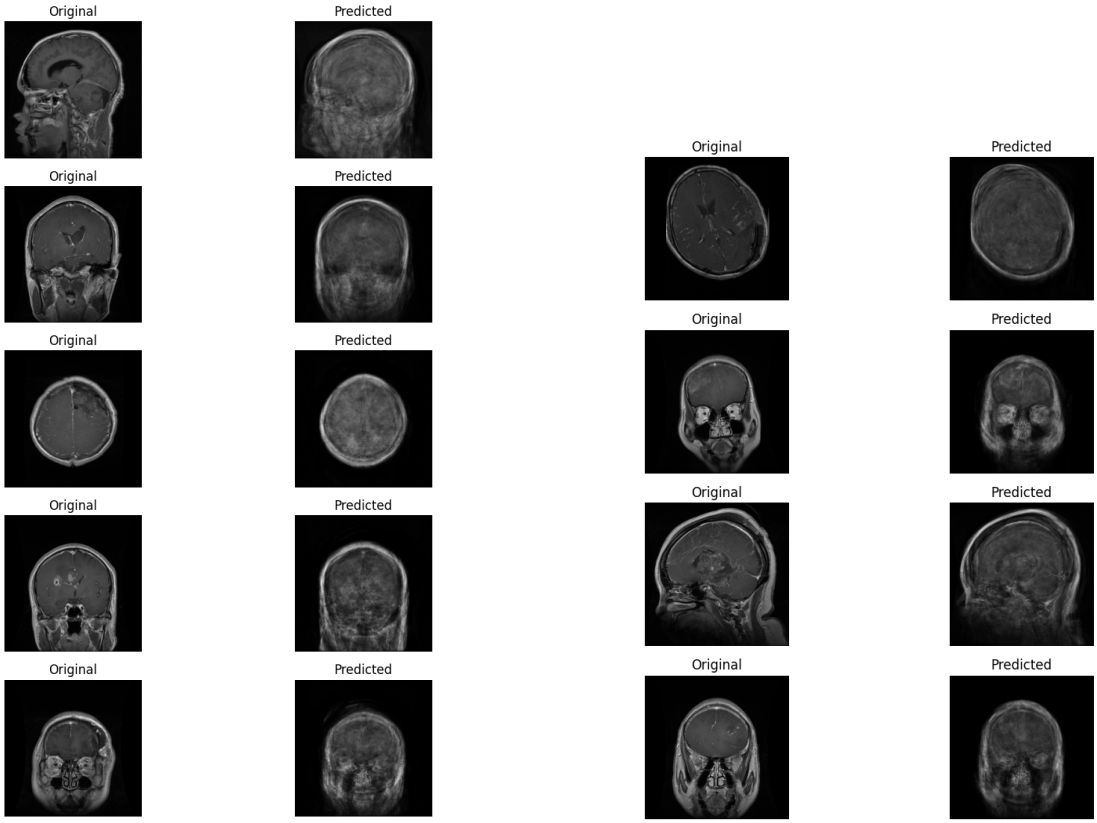
Finally, to test the performance of our autoencoder-based classifier on the test set, we first created the ground truth labels. The test set consisted of images from the "glioma" category and simulated outliers. We labeled the glioma images as "glioma" and the simulated outliers as "outlier". We then used the previously determined threshold of 0.012 to classify the test images based on their Mean Squared Error (MSE) values. For each image in the glioma test set, if the MSE was below the threshold, it was classified as "glioma"; otherwise, it was classified as "outlier". This process was repeated for the simulated outliers. The predictions for both glioma and simulated outliers were combined into a single list (`y_pred_test`) and compared to the ground truth labels (`y_test`). We evaluated the performance of our classifier using metrics such as the confusion matrix and the classification report, which includes precision, recall, and F1-score for both classes.

```
              precision    recall  f1-score   support

      glioma       0.88      0.82      0.85       286
     outlier       0.83      0.89      0.86       286

    accuracy                           0.86       572
   macro avg       0.86      0.86      0.86       572
weighted avg       0.86      0.86      0.86       572
```

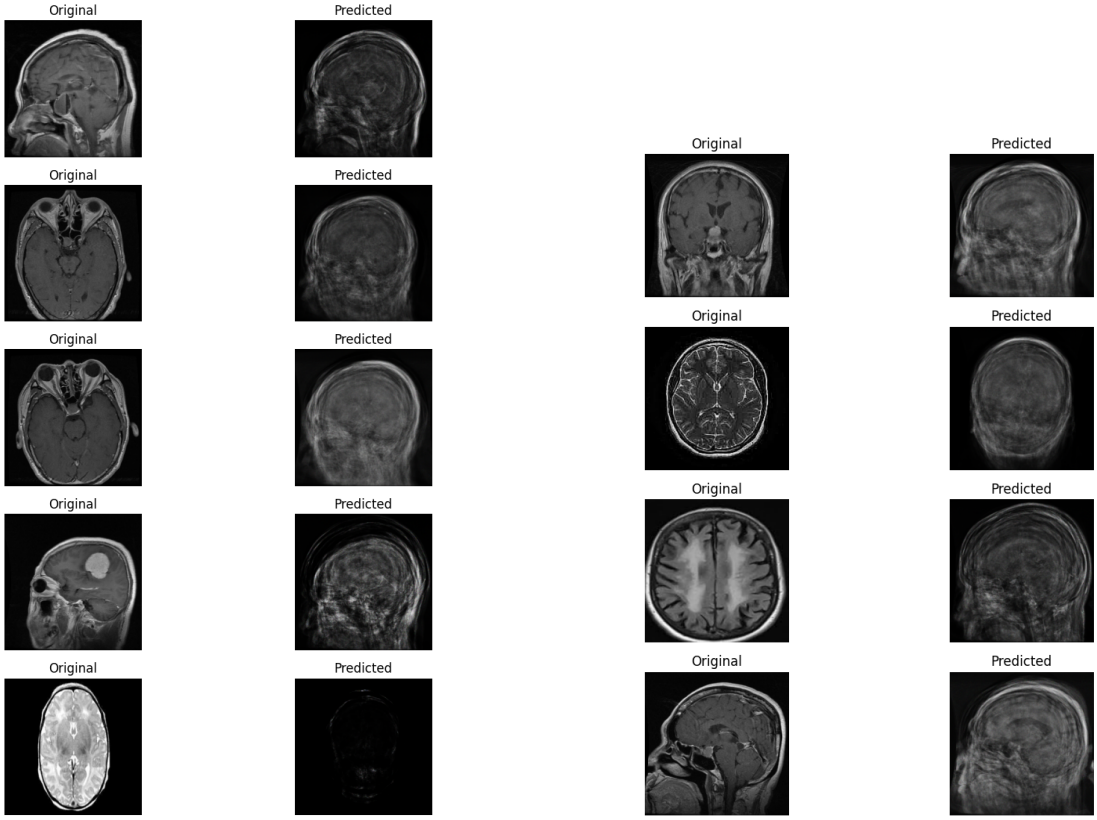


The results indicated a strong performance of the autoencoder in distinguishing between normal and anomalous images, with an overall accuracy of 0.86. The recall for glioma images was 0.82, while the recall for outlier images was 0.89. This means that the model correctly identified 82% of the normal glioma images but mistakenly classified 18% of them as outliers, indicating a conservative approach that favors detecting potential anomalies over missing them. On the other hand, the model demonstrated high effectiveness in detecting outliers, correctly identifying 89% of the anomalous images and missing only 11%. The confusion matrix revealed that out of 286 glioma images, 235 were correctly classified, and 51 were misclassified as outliers. Conversely, out of 286 simulated outlier images, 255 were correctly identified as outliers, and 31 were misclassified as glioma.

To visually assess the performance of our trained autoencoder model, we displayed 9 original and reconstructed glioma test images side-by-side in a 2x9 grid. Each row shows the original image on the left and its autoencoder reconstruction on the right. This comparison allowed us to evaluate the model's reconstruction capabilities.



The visual comparison between the original and reconstructed images from the glioma test set shows that the autoencoder captures some essential features of glioma brain scans but struggles with finer details. Some reconstructed images exhibit noticeable blurring and lack of detail, leading to higher reconstruction errors.
Similarly, to visually assess the performance of our autoencoder model on the outlier test set, we displayed 9 original and reconstructed outlier images side-by-side in a 2x9 grid. Each row shows the original image on the left and its autoencoder reconstruction on the right. This comparison allowed us to evaluate the model's ability to detect outliers by highlighting discrepancies between the original and predicted images.

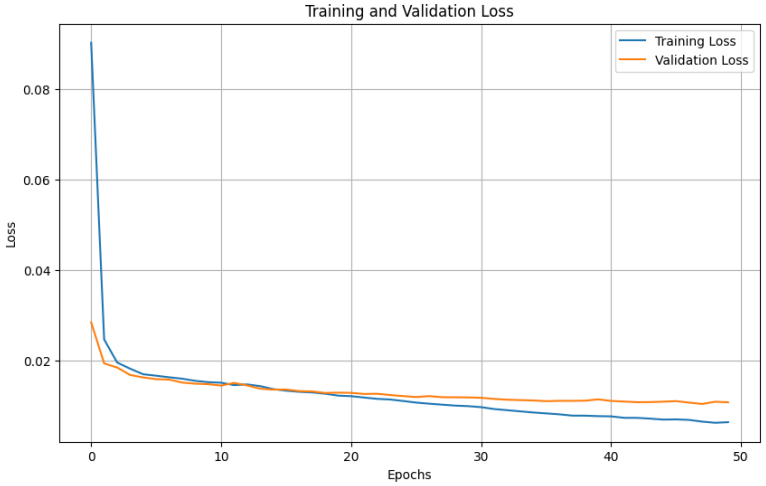


The visual comparison between the original and reconstructed images from the outlier test set reveals that the autoencoder has effectively learned to distinguish normal from anomalous patterns. The original images are clear and detailed, while the reconstructed images are of noticeably lower quality. This significant discrepancy indicates that the autoencoder struggles to accurately reconstruct outlier images, which is a desired outcome. This high reconstruction error helps the model effectively identify anomalies, confirming its utility in distinguishing glioma images from other types.

4.1.6 Training, tuning and external validation for meningioma

We repeated the same process for outlier detection for the "meningioma" category.
To preprocess the meningioma images for our autoencoder model, we normalized the pixel values by scaling them to the range [0, 1] and reshaped each image to the dimensions $224 \times 224 \times 3$. These reshaped images were then aggregated into a single NumPy array and converted to the float16 data type to optimize memory usage and computational efficiency.
The dataset was subsequently divided into training and testing sets (`X_train_meningioma` and `X_test_meningioma`) using an 80-20 split, ensuring reproducibility by setting a fixed random state. This preprocessing pipeline was essential for preparing the data for effective training and evaluation of our autoencoder model.
To train our autoencoder model for meningioma image reconstruction, we compiled the model using the Adam optimizer with a learning rate of 0.0005 and the mean squared error (MSE) loss

function. The model was trained on the meningioma training dataset (`X_train_meningioma`). We set the training parameters to 50 epochs, a batch size of 32, and enabled shuffling to ensure the training data was well mixed. Additionally, we used a validation split of 20% from the training data to monitor the model's performance on unseen data during training. The training process was monitored by plotting the training and validation loss.

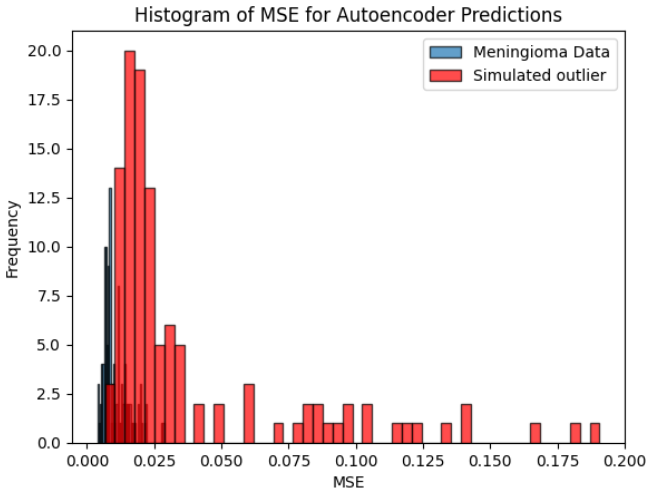


The plot revealed a decrease in both losses over time, indicating effective learning and good model performance.

For outlier detection simulation in the meningioma category, we generated a set of images from the glioma, notumor, and pituitary tumor categories to act as outliers. We randomly selected 255 images from these datasets, normalized them by scaling the pixel values to the range [0, 1], and collected them into a list. This list was then split into two subsets: the first 113 images were designated for validation (`X_val_outlier_meningioma`), and the remaining images were reserved for testing (`X_test_outlier_meningioma`). Additionally, a portion of the meningioma training set was set aside for validation (`X_val_meningioma`), specifically the images from indices 453 to 566. The `X_test_meningioma` set, defined earlier, remained unchanged.

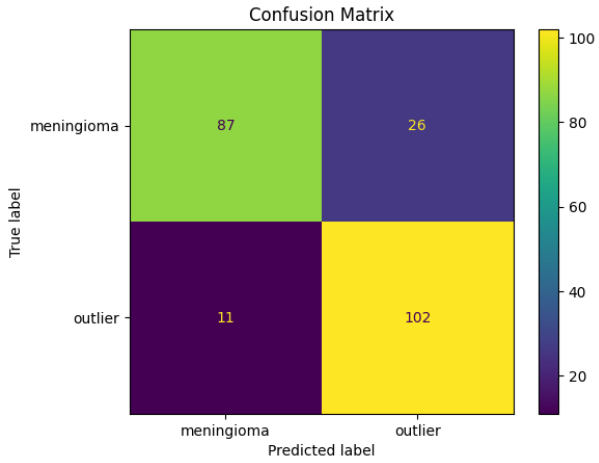
As before, to detect outliers using our trained autoencoder, we established a method based on the mean squared error (MSE) between the original and reconstructed images. If the MSE exceeds a certain threshold, the image is considered an outlier.

First, we calculated the Mean Squared Error (MSE) for both validation meningioma images and simulated outliers. For the simulated outliers (`X_val_outlier_meningioma`), we processed each image, predicted it using the autoencoder, and computed the MSE between the predicted and original images, storing these values in a list (`mse_simulation_validation_meningioma`). Similarly, we computed the MSE for the meningioma validation images (`X_val_meningioma`), storing them in a list (`mse_tot_validation_meningioma`). We then plotted histograms of these MSE values: one for the meningioma validation set and one for the simulated outliers. By comparing the MSE distributions of the meningioma images and the simulated outliers, we could visualize the range of MSE values and identify an appropriate threshold to distinguish between normal and anomalous images.



The histogram helped identify a range of thresholds to test. To determine the optimal threshold for distinguishing between meningioma images and outliers, we conducted an analysis using these threshold values. We first created ground truth labels for the validation set, with "meningioma" labels for the true meningioma images and "outlier" labels for the simulated outliers. We then defined a range of thresholds from 0.001 to 0.025, evenly spaced in 10 steps. For each threshold, we classified the images based on their MSE values: if the MSE was below the threshold, the image was labeled as "meningioma"; otherwise, it was labeled as "outlier". We iterated over each threshold, generating predictions for both the meningioma and simulated outlier validation sets. The combined predictions were compared against the ground truth labels to evaluate the performance of each threshold using the evaluation metrics. By iterating over the range of thresholds, we identified the optimal threshold value: 0.013. Below are the evaluation metrics for this threshold.

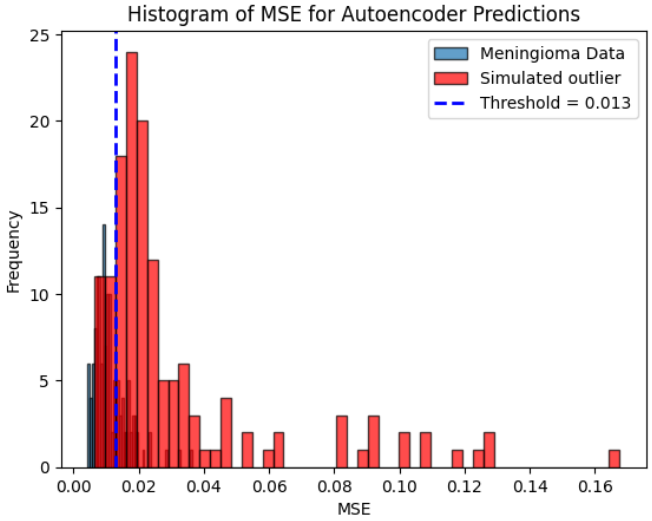
	precision	recall	f1-score	support
meningioma	0.89	0.77	0.82	113
outlier	0.80	0.90	0.85	113
accuracy			0.84	226
macro avg	0.84	0.84	0.84	226
weighted avg	0.84	0.84	0.84	226



The confusion matrix for the threshold value of 0.013 reveals that out of 113 meningioma images, 87 were correctly classified, and 26 were misclassified as outliers. Conversely, out of 113 simulated outlier images, 102 were correctly identified as outliers, and 11 were misclassified as meningioma.

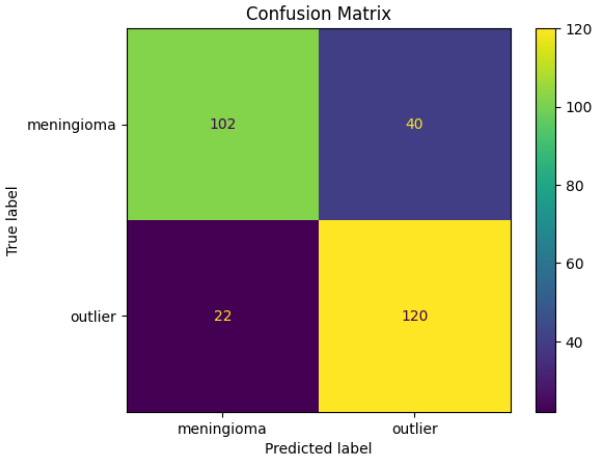
Subsequently, to evaluate the performance of the autoencoder on the test set, we calculated the Mean Squared Error (MSE) for both meningioma and simulated outlier images. For

the simulated outliers (`X_test_outlier_meningioma`), each image was reshaped and processed by the autoencoder, and the MSE between the predicted and original images was computed and stored in `mse_simulation_test_meningioma`. Similarly, for the meningioma test images (`X_test_meningioma`), each image was processed by the autoencoder, and the resulting MSE values were stored in `mse_tot_test_meningioma`. We then plotted histograms of the MSE values for both sets. The histogram for the meningioma data is shown in blue, while the histogram for the simulated outliers is shown in red. We also included a vertical line representing the chosen threshold of 0.013, which was determined during the validation phase.

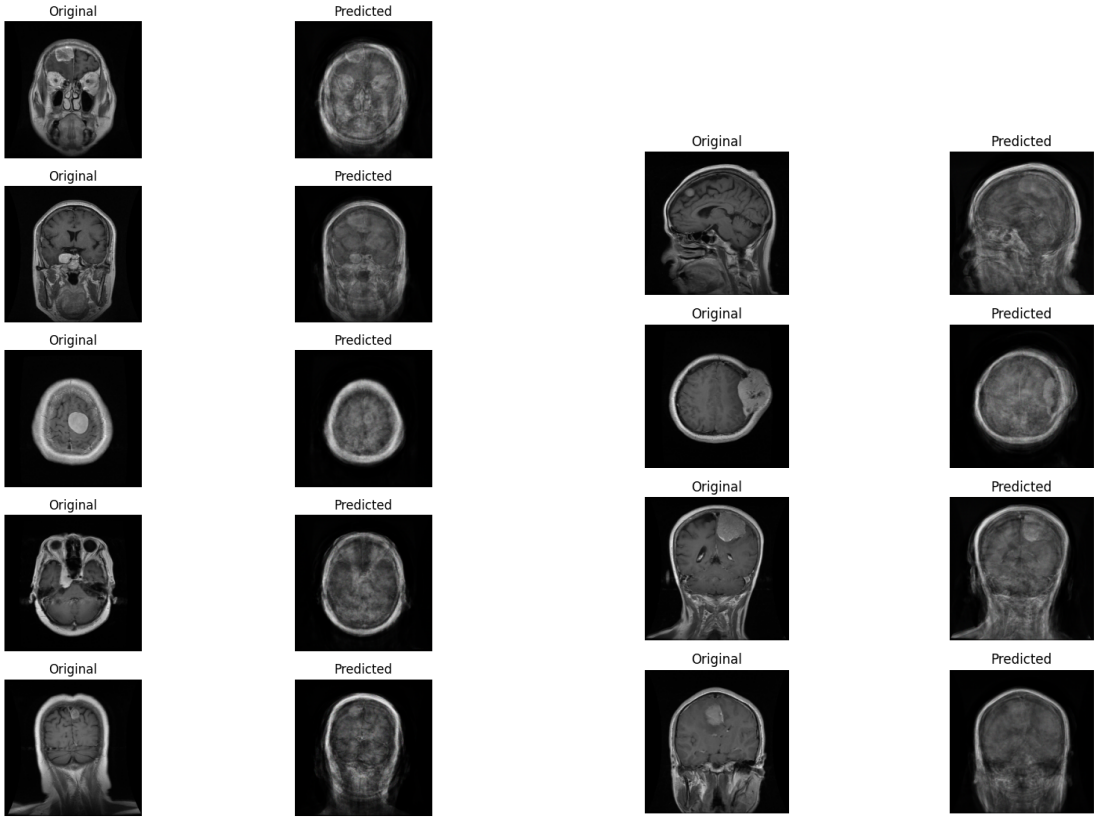


Finally, to test the performance of our autoencoder-based classifier on the test set, we first created the ground truth labels. The test set consisted of images from the "meningioma" category and simulated outliers. We labeled the meningioma images as "meningioma" and the simulated outliers as "outlier". We then used the previously determined threshold of 0.013 to classify the test images based on their Mean Squared Error (MSE) values. For each image in the meningioma test set, if the MSE was below the threshold, it was classified as "meningioma"; otherwise, it was classified as "outlier". This process was repeated for the simulated outliers. The predictions for both meningioma and simulated outliers were combined into a single list (`y_pred_test`) and compared to the ground truth labels (`y_test`). We evaluated the performance of our classifier using metrics such as the confusion matrix and the classification report, which includes precision, recall, and F1-score for both classes.

	precision	recall	f1-score	support
meningioma	0.82	0.72	0.77	142
outlier	0.75	0.85	0.79	142
accuracy			0.78	284
macro avg	0.79	0.78	0.78	284
weighted avg	0.79	0.78	0.78	284

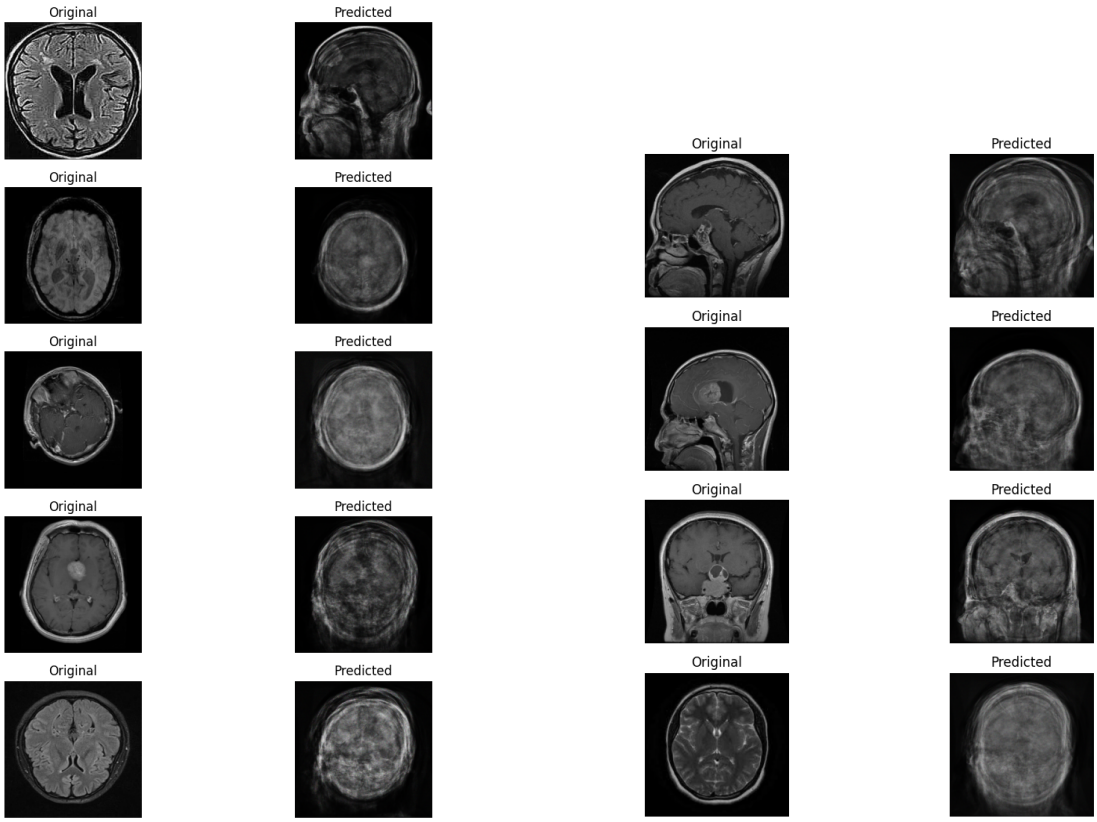


The results indicated a good performance of the autoencoder in distinguishing between normal and anomalous images, with an overall accuracy of 0.78. The recall for meningioma images was 0.72, while the recall for outlier images was 0.85. This means that the model correctly identified 72% of the normal meningioma images but mistakenly classified 28% of them as outliers, indicating a conservative approach that favors detecting potential anomalies over missing them. On the other hand, the model demonstrated high effectiveness in detecting outliers, correctly identifying 85% of the anomalous images and missing only 15%. The confusion matrix revealed that out of 142 meningioma images, 102 were correctly classified, and 40 were misclassified as outliers. Conversely, out of 142 simulated outlier images, 120 were correctly identified as outliers, and 22 were misclassified as meningioma.
To visually assess the performance of our trained autoencoder model, we displayed 9 original and reconstructed meningioma test images side-by-side in a 2x9 grid. Each row shows the original image on the left and its autoencoder reconstruction on the right. This comparison allowed us to evaluate the model's reconstruction capabilities.



The visual comparison between the original and reconstructed images from the meningioma test

set shows that the autoencoder captures some essential features of meningioma brain scans but struggles with finer details. Some reconstructed images exhibit noticeable blurring and lack of detail, leading to higher reconstruction errors.
Similarly, to visually assess the performance of our autoencoder model on the outlier test set, we displayed 9 original and reconstructed outlier images side-by-side in a 2x9 grid. Each row shows the original image on the left and its autoencoder reconstruction on the right. This comparison allowed us to evaluate the model's ability to detect outliers by highlighting discrepancies between the original and predicted images.

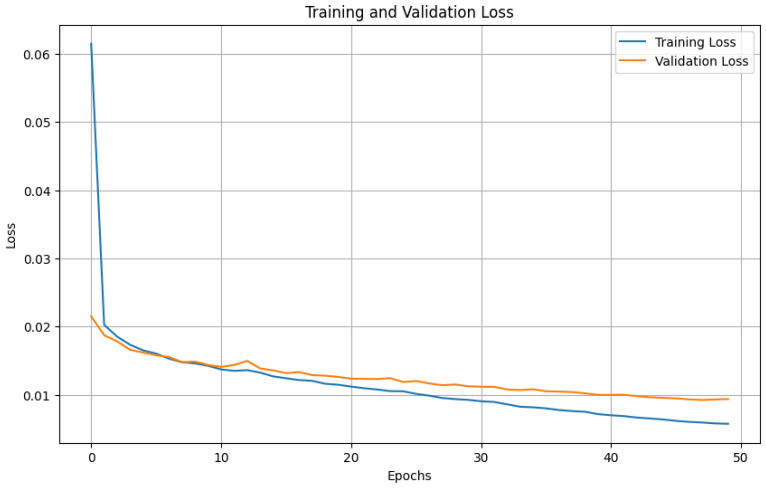


The visual comparison between the original and reconstructed images from the outlier test set reveals that the autoencoder has effectively learned to distinguish normal from anomalous patterns. The original images are clear and detailed, while the reconstructed images are of noticeably lower quality. This significant discrepancy indicates that the autoencoder struggles to accurately reconstruct outlier images, which is a desired outcome. This high reconstruction error helps the model effectively identify anomalies, confirming its utility in distinguishing meningioma images from other types.

4.1.7 Training, tuning and external validation for pituitary tumor

We repeated the same process for outlier detection for the "pituitary tumor" category.
To preprocess the pituitary tumor images for our autoencoder model, we normalized the pixel values by scaling them to the range [0, 1] and reshaped each image to the dimensions $224 \times 224 \times 3$. These reshaped images were then aggregated into a single NumPy array and converted to the float16 data type to optimize memory usage and computational efficiency.
The dataset was subsequently divided into training and testing sets (`X_train_pituitary_tumor` and `X_test_pituitary_tumor`) using an 80-20 split, ensuring reproducibility by setting a fixed random state. This preprocessing pipeline was essential for preparing the data for effective training and evaluation of our autoencoder model.
To train our autoencoder model for pituitary tumor image reconstruction, we compiled the model using the Adam optimizer with a learning rate of 0.0005 and the mean squared error (MSE) loss function. The model was trained on the pituitary tumor training dataset

(`X_train_pituitary_tumor`). We set the training parameters to 50 epochs, a batch size of 32, and enabled shuffling to ensure the training data was well mixed. Additionally, we used a validation split of 20% from the training data to monitor the model's performance on unseen data during training. The training process was monitored by plotting the training and validation loss.

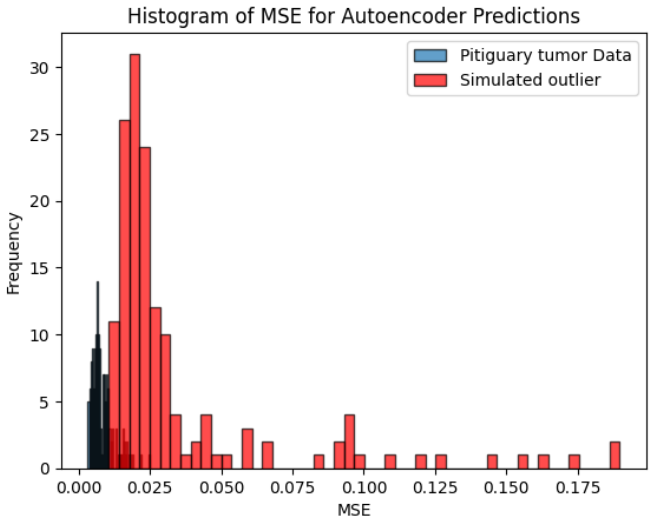


The plot revealed a decrease in both losses over time, indicating effective learning and good model performance.

For outlier detection simulation in the pituitary tumor category, we generated a set of images from the glioma, notumor, and meningioma categories to act as outliers. We randomly selected 335 images from these datasets, normalized them by scaling the pixel values to the range [0, 1], and collected them into a list. This list was then split into two subsets: the first 149 images were designated for validation (`X_val_outlier_pituitary_tumor`), and the remaining images were reserved for testing (`X_test_outlier_pituitary_tumor`). Additionally, a portion of the pituitary tumor training set was set aside for validation (`X_val_pituitary_tumor`). The `X_test_pituitary_tumor` set, defined earlier, remained unchanged.

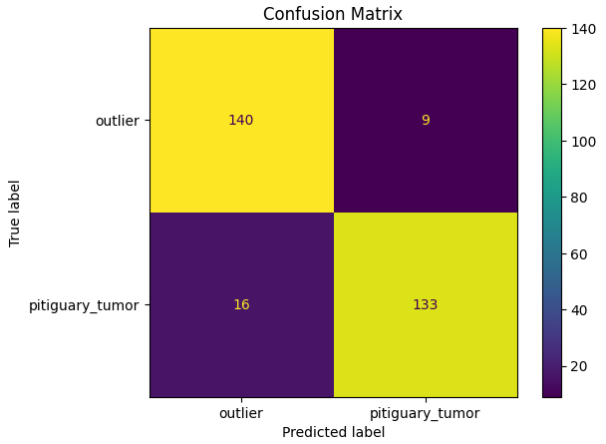
As before, to detect outliers using our trained autoencoder, we established a method based on the mean squared error (MSE) between the original and reconstructed images. If the MSE exceeds a certain threshold, the image is considered an outlier.

First, we calculated the Mean Squared Error (MSE) for both validation pituitary tumor images and simulated outliers. For the simulated outliers (`X_val_outlier_pituitary_tumor`), we processed each image, predicted it using the autoencoder, and computed the MSE between the predicted and original images, storing these values in a list (`mse_simulation_validation_pituitary_tumor`). Similarly, we computed the MSE for the pituitary tumor validation images (`X_val_pituitary_tumor`), storing them in a list (`mse_tot_validation_pituitary_tumor`). We then plotted histograms of these MSE values: one for the pituitary tumor validation set and one for the simulated outliers. By comparing the MSE distributions of the pituitary tumor images and the simulated outliers, we could visualize the range of MSE values and identify an appropriate threshold to distinguish between normal and anomalous images.



The histogram helped identify a range of thresholds to test. To determine the optimal threshold for distinguishing between pituitary tumor images and outliers, we conducted an analysis using these threshold values. We first created ground truth labels for the validation set, with "pituitary tumor" labels for the true pituitary tumor images and "outlier" labels for the simulated outliers. We then defined a range of thresholds from 0.001 to 0.025, evenly spaced in 10 steps. For each threshold, we classified the images based on their MSE values: if the MSE was below the threshold, the image was labeled as "pituitary tumor"; otherwise, it was labeled as "outlier". We iterated over each threshold, generating predictions for both the pituitary tumor and simulated outlier validation sets. The combined predictions were compared against the ground truth labels to evaluate the performance of each threshold using the evaluation metrics. By iterating over the range of thresholds, we identified the optimal threshold value: 0.013. Below are the evaluation metrics for this threshold.

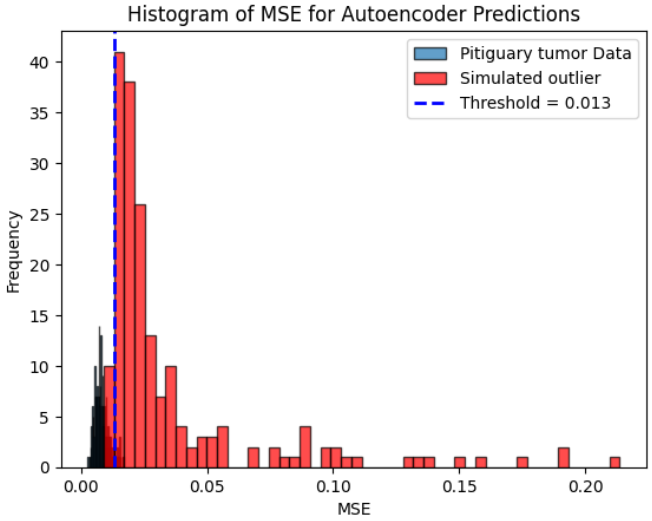
	precision	recall	f1-score	support
outlier	0.90	0.94	0.92	149
pitiguary_tumor	0.94	0.89	0.91	149
accuracy			0.92	298
macro avg	0.92	0.92	0.92	298
weighted avg	0.92	0.92	0.92	298



The confusion matrix for the threshold value of 0.013 reveals that out of 149 pituitary tumor images, 133 were correctly classified, and 16 were misclassified as outliers. Conversely, out of 149 simulated outlier images, 140 were correctly identified as outliers, and 9 were misclassified as pituitary tumors.

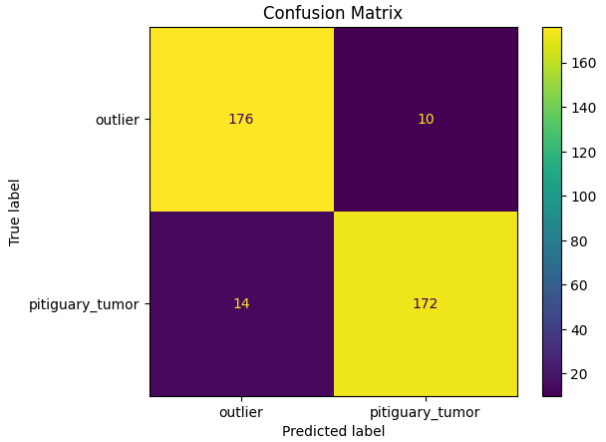
Subsequently, to evaluate the performance of the autoencoder on the test set, we calculated the Mean Squared Error (MSE) for both pituitary tumor and simulated outlier images. For the

simulated outliers (`X_test_outlier_pituitary_tumor`), each image was reshaped and processed by the autoencoder, and the MSE between the predicted and original images was computed and stored in `mse_simulation_test_pituitary_tumor`. Similarly, for the pituitary tumor test images (`X_test_pituitary_tumor`), each image was processed by the autoencoder, and the resulting MSE values were stored in `mse_tot_test_pituitary_tumor`. We then plotted histograms of the MSE values for both sets. The histogram for the pituitary tumor data is shown in blue, while the histogram for the simulated outliers is shown in red. We also included a vertical line representing the chosen threshold of 0.013, which was determined during the validation phase.



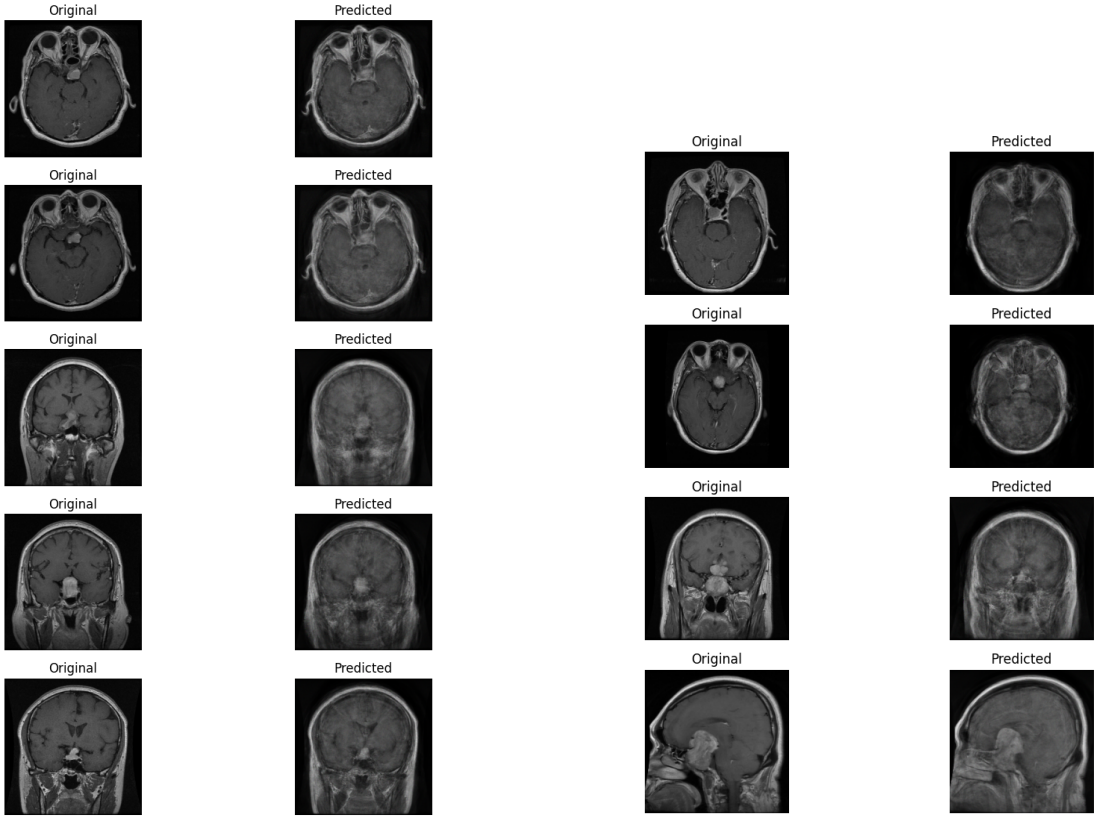
Finally, to test the performance of our autoencoder-based classifier on the test set, we first created the ground truth labels. The test set consisted of images from the "pituitary tumor" category and simulated outliers. We labeled the pituitary tumor images as "pituitary tumor" and the simulated outliers as "outlier". We then used the previously determined threshold of 0.013 to classify the test images based on their Mean Squared Error (MSE) values. For each image in the pituitary tumor test set, if the MSE was below the threshold, it was classified as "pituitary tumor"; otherwise, it was classified as "outlier". This process was repeated for the simulated outliers. The predictions for both pituitary tumor and simulated outliers were combined into a single list (`y_pred_test`) and compared to the ground truth labels (`y_test`). We evaluated the performance of our classifier using metrics such as the confusion matrix and the classification report, which includes precision, recall, and F1-score for both classes.

	precision	recall	f1-score	support
outlier	0.93	0.95	0.94	186
pitiguary_tumor	0.95	0.92	0.93	186
accuracy			0.94	372
macro avg	0.94	0.94	0.94	372
weighted avg	0.94	0.94	0.94	372



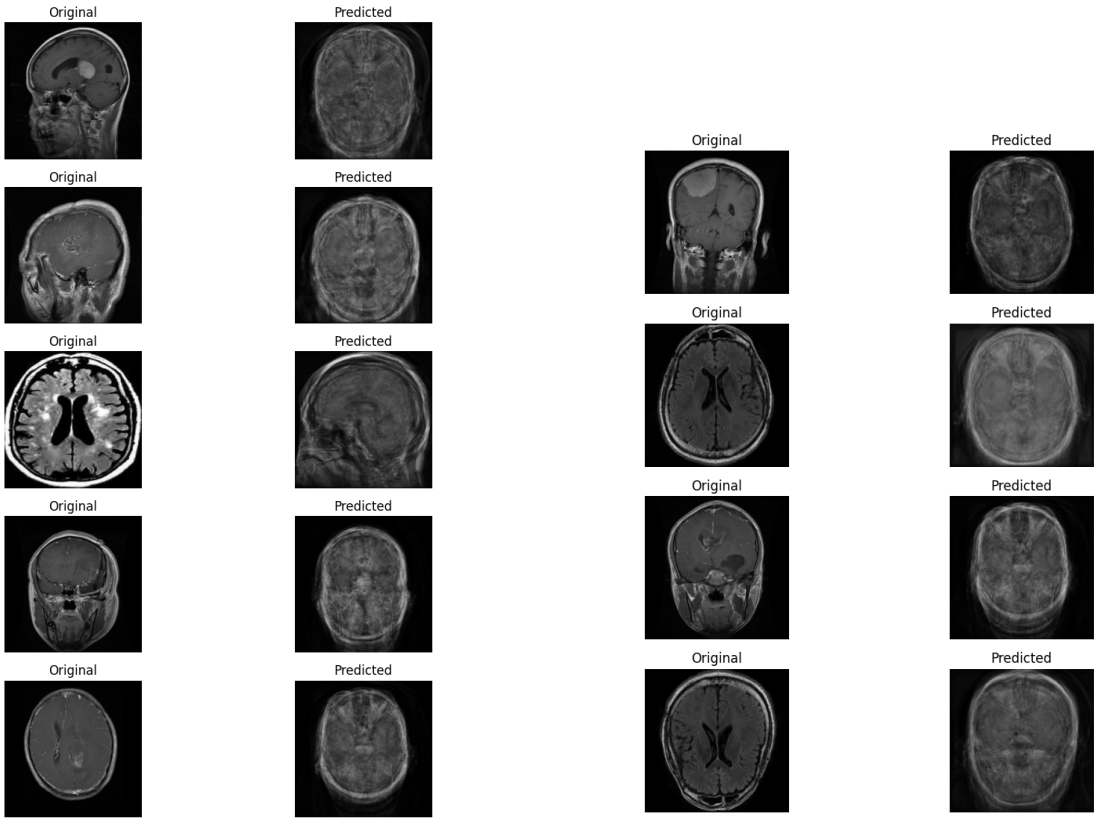
The results indicated a robust performance of the autoencoder in distinguishing between normal and anomalous images, with an overall accuracy of 0.94. The recall for pituitary tumor images was 0.92, while the recall for outlier images was 0.95. This means that the model correctly identified 92% of the normal pituitary tumor images but mistakenly classified 8% of them as outliers. The model also demonstrated high effectiveness in detecting outliers, correctly identifying 95% of the anomalous images and missing only 5%. The confusion matrix revealed that out of 186 pituitary tumor images, 172 were correctly classified, and 14 were misclassified as outliers. Conversely, out of 186 simulated outlier images, 176 were correctly identified as outliers, and 10 were misclassified as pituitary tumor.

To visually assess the performance of our trained autoencoder model, we displayed 9 original and reconstructed pituitary tumor test images side-by-side in a 2x9 grid. Each row shows the original image on the left and its autoencoder reconstruction on the right. This comparison allowed us to evaluate the model's reconstruction capabilities.



The visual comparison between the original and reconstructed images from the pituitary tumor test set reveals that the autoencoder has effectively learned to capture the essential features of

normal brain scans. In most cases, the reconstructed images closely resemble the original ones, indicating high reconstruction accuracy.
Similarly, to visually assess the performance of our autoencoder model on the outlier test set, we displayed 9 original and reconstructed outlier images side-by-side in a 2x9 grid. Each row shows the original image on the left and its autoencoder reconstruction on the right. This comparison allowed us to evaluate the model's ability to detect outliers by highlighting discrepancies between the original and predicted images.



The visual comparison between the original and reconstructed images from the outlier test set reveals that the autoencoder has effectively learned to distinguish normal from anomalous patterns. The original images are clear and detailed, while the reconstructed images are of noticeably lower quality. This significant discrepancy indicates that the autoencoder struggles to accurately reconstruct outlier images, which is a desired outcome.

4.1.8 Outlier Detection

Having trained the autoencoder models for all our four classes, we idetified the outliers.
We loaded the pre-trained autoencoder models for each tumor type. The autoencoder compresses the input image to a lower-dimensional representation and then reconstructs it. The mean squared error (MSE) between the original and reconstructed image is calculated for each image. Images with an MSE above a specified threshold were considered outliers. This process is repeated for each tumor type.

```
from tensorflow.keras.models import load_model, save_model
import tensorflow as tf

df_glioma = df[df['Label'] == 'glioma']
df_notumor = df[df['Label'] == 'notumor']
df_pituitary_tumor = df[df['Label'] == 'pituitary tumor']
df_meningioma = df[df['Label'] == 'meningioma']

Notumor
autoencoder =
tf.keras.models.load_model('/content/drive/MyDrive/ProgettoDataMining/autoencd
```

```
er_notumor.keras')

X = []
for i, x in enumerate(df_notumor["Image"].values):
x = x / 255
new = x.reshape(1, 224, 224, 3)
X.append(new)

X = np.vstack(X).astype(np.float16)

mse_tot = []
for i in range(len(X)):
real = X[i]
preprocessed_real = real.reshape(1, 224, 224, 3)
pred = autoencoder.predict(preprocessed_real)
mse = mean_squared_error(pred.flatten(), real.flatten())
mse_tot.append(mse)

threshold = 0.013
pred_notumor = [value > threshold for value in mse_tot]
indices_notumor = [index for index, value in enumerate(pred_notumor) if value]
print(indices_notumor)
print("the percentage of outlier is", len(indices_notumor) /
len(pred_notumor))
del autoencoder
```

4.2 Outlier Detection with Isolation Forest and NMF

After applying an autoencoder to detect outliers in tumor images, we explored less computationally expensive approaches like Non-negative Matrix Factorization (NMF) and Isolation Forest. These methods were chosen to compare the effectiveness of simpler models in detecting outliers. Isolation Forest is an unsupervised learning algorithm that detects anomalies by isolating observations. It works by creating random splits in the data, and anomalies are expected to be isolated quickly in fewer splits. This method is particularly suitable for outlier detection due to its efficiency and ability to handle high-dimensional datasets.Nmf approach should help uncover hidden structures in the data, facilitating the identification of outliers based on their deviations from typical patterns.
Non-negative Matrix Factorization (NMF) is used to reduce the dimensionality of the image data. NMF decomposes the original image matrix into two non- negative matrices, which helps in uncovering hidden structures. The images are resized to 150×150 pixels, normalized, and transformed using the NMF model. The resulting encoded features are then used to create a DataFrame, which is split into different tumor types.

```
from sklearn.decomposition import NMF
import joblib

Load NMF model
nmf = joblib.load('/content/drive/MyDrive/ProgettoDataMining/nmf_model1.pkl')
size = (150, 150)

for i in range(len(df)):
df.at[i, "Image"] = cv2.resize(df.at[i, "Image"], size)

tot_image_matrix = create_matrix_of_rows_images_normalized_and_resized(df)
Encoded_matrix = nmf.transform(tot_image_matrix)
del tot_image_matrix

df_encoded = pd.DataFrame(Encoded_matrix)
df_encoded["Label"] = df["Label"]

df_glioma = df_encoded[df_encoded['Label'] == 'glioma']
df_notumor = df_encoded[df_encoded['Label'] == 'notumor']
df_pituitary_tumor = df_encoded[df_encoded['Label'] == 'pituitary tumor']
df_meningioma = df_encoded[df_encoded['Label'] == 'meningioma']
del df_encoded
```

Isolation Forest is applied to each tumor type separately. The `contamination` parameter is set based on the estimated percentage of outliers. The model is trained using the encoded features (excluding labels), and predictions are made to identify outliers. The number of detected outliers and their indices are printed for each tumor type.

```
clf = IsolationForest(contamination=0.075, random_state=42)
X_train = df_notumor.drop(columns=["Label"])
clf.fit(X_train)
pred_notumor = clf.predict(X_train)
print(len(df_notumor[pred_notumor == -1].index))
df_notumor[pred_notumor == -1].index
```

```
clf = IsolationForest(contamination=0.07, random_state=42)
X_train = df_glioma.drop(columns=["Label"])
clf.fit(X_train)
pred_glioma = clf.predict(X_train)
print(len(df_glioma[pred_glioma == -1].index))
df_glioma[pred_glioma[pred_glioma == -1].index
```

```
clf = IsolationForest(contamination=0.10, random_state=42)
X_train = df_meningioma.drop(columns=["Label"])
clf.fit(X_train)
pred_meningioma = clf.predict(X_train)
print(len(df_meningioma[pred_meningioma == -1].index))
df_meningioma[pred_meningioma == -1].index
```

```
clf = IsolationForest(contamination=0.035, random_state=42)
X_train = df_pituitary_tumor.drop(columns=["Label"])
clf.fit(X_train)
pred_pituitary_tumor = clf.predict(X_train)
print(len(df_pituitary_tumor[pred_pituitary_tumor == -1].index))
df_pituitary_tumor[pred_pituitary_tumor == -1].index
```

After applying an autoencoder to detect outliers, we tried simpler models like NMF and Isolation Forest to compare their effectiveness. Isolation Forest isolates observations by creating random splits in the data, making it suitable for anomaly detection due to its efficiency and scalability. However, our results indicated that these simpler methods did not converge with the autoencoder's results, suggesting that they might not be sufficient for accurately detecting outliers in this context. Further exploration and tuning of these models may be required to improve their performance.

5. Machine Learing Models

In this section, we experimented with various techniques to classify the MRI images. The techniques explored include LASSO Logistic Regression (both LASSO OVR Logistic Regression and LASSO Multinomial Logistic Regression), Decision Tree Classifier, Random Forest, and Support Vector Classifier. Our objective was to identify the best model for accurately classifying the images and to use this model to extract the most critical features.

5.1 Libraries

These imports collectively set up the environment for efficient data handling, model training, evaluation, and results visualization.

```
import pandas as pd
import numpy as np
import joblib
import gc
import matplotlib.pyplot as plt
from sklearn import tree
from sklearn.linear_model import LogisticRegressionCV
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.utils.multiclass  import unique_labels
from sklearn.metrics import  accuracy_score, f1_score, recall_score,
    precision_score, confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay, classification_report
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import KFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.feature_selection import RFECV
from sklearn.feature_selection import RFE
from sklearn.svm import SVC
```

5.2 DataFrame

First, we created the `df` object to represent the DataFrame contained in the pickle file named "df_statistic2.pkl". To gain insight into the dataset's class distribution, we computed and displayed the frequency of each unique label within the "Label" column of the DataFrame.

```
from google.colab import drive
drive.mount('/content/drive')

df = pd.read_pickle("/content/drive/MyDrive/ProgettoDataMining/df_statistic2.pkl"
    )
df["Label"].value_counts()
```

```
Label
notumor            1500
glioma             1426
pituitary tumor     930
meningioma          708
Name: count, dtype: int64
```

The dataset consists of four different classes: "notumor" with 1500 instances, "glioma" with 1426 instances, "pituitary tumor" with 930 instances, and "meningioma" with 708 instances. These counts reveal the class imbalance within the dataset, where "notumor" has the highest representation followed by "glioma", "pituitary tumor", and "meningioma" classes, respectively. To address this imbalance, we created a balanced dataset by ensuring an equal number of samples for each class label in the original DataFrame `df`. We set the desired number of samples per class to 700 and initialized an empty DataFrame `new_df` to store the balanced data. We then iterated through each unique class label in the original DataFrame, filtering to include only rows with the current label and randomly selecting 700 samples from these filtered rows, ensuring reproducibility with a fixed random state of 42. The selected samples for each class were then concatenated into `new_df`, resulting in a balanced dataset with an equal number of samples for each class, ready for subsequent analysis.

```
num_for_each_class = 700
new_df = pd.DataFrame()

for i in df["Label"].unique():
  print(i)
  sample = df[df["Label"] == i]
  sample = sample.sample(n=num_for_each_class, random_state=42)
  new_df = pd.concat([new_df, sample])

new_df
```

After creating the balanced dataset, we proceeded with data preparation tasks to set up the data for subsequent modeling. First, we created a new DataFrame `X` to serve as the feature matrix, excluding the `Label` column. Next, we created a series `y` to hold the class labels extracted from the `Label` column of `new_df` and stored the unique class labels present in `y` in the array `labels`. We then split the balanced dataset into training and test sets, allocating 25% of the data to the test set and the remaining 75% to the training set. To maintain the class distribution in both sets and avoid bias, we used the `stratify=y` argument, and ensured reproducibility of the split with `random_state=42`. After splitting the data, we verified if the class distributions were indeed balanced between the training and test sets by printing the proportion of each class, expressed as fractions of the total samples.
Next, we converted the column names of the DataFrame `X` into a new array `features_name`, where each column name is represented as a string. These names are essential for identifying and referencing each feature in subsequent data processing steps, such as model fitting, evaluation, or interpretation.
Additionally, we prepared indices for splitting the dataset into training and test sets. We reset the index to create a sequence of integers from 0 to the length of the dataset minus one, stored in the `index` array. Based on the `test_size` parameter, we calculated the `split_index` to determine the boundary between the training and test sets. We then divided the `index` array into two parts: `train_index` for the training set and `test_index` for the test set. This approach allowed us to separate the dataset into training and testing subsets using these indices. Finally, we reassigned the indices of `X_train` and `X_test` to `train_index` and `test_index`, respectively, ensuring the feature matrices for the training and test sets reflected their respective positions in the original dataset. Similarly, we set the indices of `y_train` and `y_test` to `train_index` and `test_index`, ensuring the target labels for the training and test sets corresponded to the correct observations. This step is crucial for maintaining the integrity and traceability of the data throughout the model training and evaluation process.

```
X = new_df.drop('Label', axis=1, inplace=False)

y = new_df["Label"].copy()

labels = y.unique()

test_size = 0.25
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size,
    stratify=y, random_state=42)

print("Proportion of training classes:")
```

```
print(y_train.value_counts(normalize=True))
print("\nProportion of test classes:")
print(y_test.value_counts(normalize=True))

features_name = X.columns.astype(str)

index = np.arange(len(X))
split_index = int(len(index) * (1-test_size))

train_index = index[:split_index]
test_index = index[split_index :]

X_train.index = train_index
X_test.index = test_index
y_train.index = train_index
y_test.index = test_index
```

```
Proportion of training classes:
Label
pituitary tumor    0.25
notumor            0.25
glioma             0.25
meningioma         0.25
Name: proportion, dtype: float64

Proportion of test classes:
Label
glioma             0.25
meningioma         0.25
pituitary tumor    0.25
notumor            0.25
Name: proportion, dtype: float64
```

5.3 Performance evaluation function

Before discussing the models, it is essential to introduce a fundamental function that we applied to evaluate their performance. To ensure our models are effective, we needed a comprehensive method to measure their accuracy and reliability. This was achieved through the `evaluate_metrics` function. This function not only provides the model's accuracy, which measures the overall correctness of the model by calculating the ratio of correctly predicted instances to the total instances, but also computes and presents essential classification metrics such as precision, recall, f1-score, and support for each class. These metrics offer insights into how well the model's predictions match the actual labels (`y_test`) and provide a detailed evaluation of the model's performance for each class:

- `precision` measures the proportion of instances predicted as positive that are correctly classified as positive. Mathematically, it is calculated as the ratio of true positives (TP) to the sum of true positives and false positives (FP);
- `recall` measures the proportion of actual positive instances in the dataset that the model correctly identifies. Mathematically, it is calculated as the ratio of true positives (TP) to the sum of true positives and false negatives (FN);
- `f1-score` is the harmonic mean of precision and recall, providing a single score that balances precision and recall. It is particularly useful as a single metric to compare model performances. Mathematically, it is calculated as $2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$;
- `support` indicates the number of actual instances of each class in the test dataset.

High values of precision and recall are fundamental for our models to effectively assist in medical diagnoses. Precision measures the model's ability to correctly identify true positives among all positive predictions, thereby reducing the risk of false positives. High precision reduces the chances of incorrectly diagnosing healthy patients with a tumor, preventing unnecessary stress, additional tests, and unwarranted treatments. It also minimizes the likelihood of misdiagnosing

a patient with the wrong type of tumor, which could lead to inappropriate treatments, adverse side effects, and delays in receiving the correct treatment. Such misdiagnoses not only would affect the well-being of patients but also place a significant burden on healthcare resources. Recall, meanwhile, measures the model's ability to correctly identify positive cases, thereby reducing the risk of false negatives. High recall ensures that patients with the disease are accurately identified, enabling them to receive necessary treatments promptly. This is vital for serious conditions like tumors, where early intervention can significantly impact outcomes. Additionally, high recall helps ensure that patients without the disease are not misdiagnosed, providing a comprehensive evaluation of the model's performance.
Additionally, the `evaluate_metrics` function generates a confusion matrix, where each row represents the true labels, while each column represents the predicted labels. This matrix visually presents the true positives, true negatives, false positives, and false negatives, offering a detailed understanding of the model's predictive accuracy and its ability to correctly classify unseen data. This evaluation step is crucial for validating how well the model generalizes to new observations and provides insights into its real-world applicability.

```
def evaluate_metrics(y_pred, y_test):
    labels = unique_labels(y_test, y_pred)
    print(labels)
    report = classification_report(y_test, y_pred, target_names=labels)
    print(report)
    print("\n" * 5)
    cm = confusion_matrix(y_test, y_pred, labels=labels)
    cm_display = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=
    labels)
    plt.figure(figsize=(8, 6))
    cm_display.plot()
    plt.title('Confusion Matrix')
    plt.show()
```

5.4 Logistic Regression

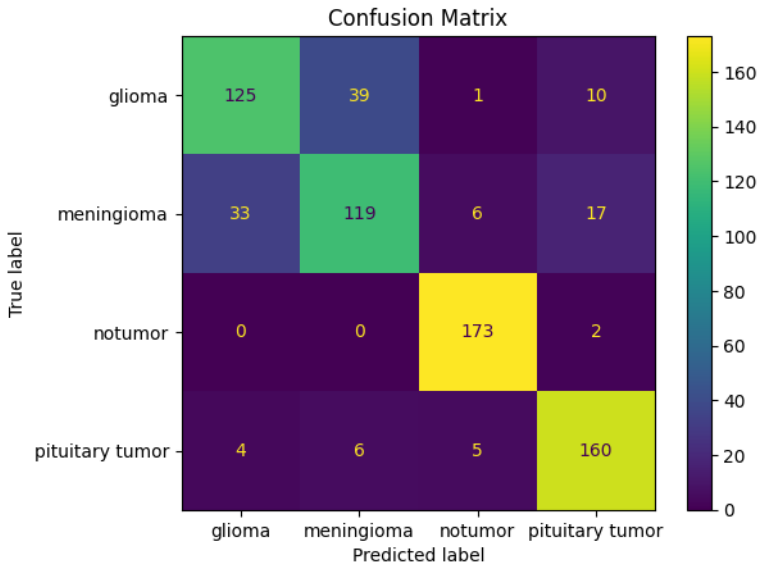
5.4.1 OvR LASSO Logistic Regression

This section details the training and evaluation of a LASSO Logistic Regression model using the One-vs-Rest (OvR) strategy alongside automated cross-validation.
Logistic Regression was chosen for its fundamental ability to predict the probability of an observation belonging to one of several possible classes. Traditionally used for binary classification tasks, Logistic Regression was not directly applicable to our dataset, which includes four distinct classes (notumor, glioma, pituitary tumor, and meningioma). To adapt Logistic Regression to multi-class classification, we employed the One-vs-Rest (OvR) strategy. This approach trains individual binary classifiers, where each classifier identifies one class as the positive class against all others as negatives. By leveraging Logistic Regression's strength in binary classification, OvR effectively addresses multi-class problems. In addition, we incorporated LASSO (L1 regularization) to enhance model performance by mitigating overfitting. LASSO penalizes large coefficients, shrinking some to zero and performing feature selection, thereby simplifying the model and highlighting the most influential features. This regularization technique is particularly advantageous in datasets with many features, ensuring that only the most relevant variables contribute to the model's predictions. Moreover, To ensure the robustness and reliability of our model, we implemented 5-fold cross-validation (CV). This involved dividing the training data into five subsets, training the model on four subsets and validating it on the fifth, and repeating this process five times. Each subset was used exactly once for validation, providing a more reliable estimate of the model's performance by reducing the variance associated with a single train-test split and ensuring good generalization to unseen data.
The LASSO Logistic Regression model was instantiated using the `LogisticRegressionCV` class from `scikit-learn`, which incorporates cross-validation to optimize model parameters. Configured to utilize all available CPU cores for parallel processing, the model was set with the following parameters: L1 penalty for regularization, a fixed random state for reproducibility, refitting enabled to retain the best model, `ovr` for handling multiple classes, the `liblinear`

solver for optimization, 5-fold cross-validation, and accuracy as the scoring metric. The Logistic Regression model was then trained on the training data: `X_train` and `y_train`. After training, we printed the range of regularization parameters tested during cross-validation (`logitic_regression_ovr.Cs_`) and the best regularization strengths identified for each model (`logitic_regression_ovr.C_`). Predictions of class labels on the test data (`X_test`) were generated using the trained model, and these predictions were stored in `y_pred`. These predictions represent the model's classifications of observations into one of several classes based on their features. To evaluate the model's performance on the test data, the `evaluate_metrics` function was applied, providing a comprehensive assessment of the model's accuracy and effectiveness in classifying the test samples. This function computes and presents essential classification metrics, such as precision, recall, f1-score, and support for each class, providing insights into how well the model's predictions match the actual labels (`y_test`). Additionally, it generates a confusion matrix that visually presents the true positives, true negatives, false positives, and false negatives, offering a detailed understanding of the model's predictive accuracy.

```
common_params = {
    'penalty': 'l1',
    'random_state': 42,
    'refit': True,
    'multi_class': 'ovr',
    'solver': 'liblinear',
    'cv': 5,
    'scoring': 'accuracy'
}
logitic_regression_ovr = LogisticRegressionCV(n_jobs=-1, **common_params)

logitic_regression_ovr.fit(X_train, y_train)

print("The penalties evaluated are:", logitic_regression_ovr.Cs_)
print("\n", "The bests for every models are:", logitic_regression_ovr.C_)

y_pred = logitic_regression_ovr.predict(X_test)

print("Test")
evaluate_metrics(y_pred,y_test)
```

```
The penalties evaluated are:
 [1.00000000e-04 7.74263683e-04 5.99484250e-03 4.64158883e-02
 3.59381366e-01 2.78255940e+00 2.15443469e+01 1.66810054e+02
 1.29154967e+03 1.00000000e+04]

The bests for every models are:
 [0.35938137 2.7825594  2.7825594  2.7825594]

Test
['glioma' 'meningioma' 'notumor' 'pituitary tumor']
                precision    recall  f1-score   support

         glioma       0.77      0.71      0.74       175
     meningioma       0.73      0.68      0.70       175
        notumor       0.94      0.99      0.96       175
pituitary tumor       0.85      0.91      0.88       175

       accuracy                           0.82       700
      macro avg       0.82      0.82      0.82       700
   weighted avg       0.82      0.82      0.82       700
```



These results show the performance metrics of the LASSO Logistic Regression model evaluated on the test dataset consisting of 700 instances across the four classes glioma, meningioma, notumor, and pituitary tumor. The model achieved an overall accuracy of 82%, indicating that it correctly classified 82% of the observations in the test set. The precision, recall, and f1-score metrics provide insights into the model's performance for each class. Precision measures the accuracy of positive predictions for each class; specifically, the model achieved 77% precision for glioma, 73% for meningioma, 94% for notumor, and 85% for pituitary tumor, signifying the proportion of correctly predicted instances within each class. Recall measures the model's ability to correctly identify instances of each class. Here, the model correctly identified 71% of all glioma instances, 68% of all meningioma instances, 99% of all notumor instances, and 91% of all pituitary tumor instances. The F1-score, which harmonizes precision and recall, revealed a robust balance for the notumor class, achieving the highest F1-score of 0.96 among all classes. These metrics collectively indicate that the model's predictions are reliable across most classes, particularly excelling in distinguishing "notumor" and "pituitary tumor". The macro-average and weighted-average metrics, both at 82%, reinforce the overall robustness and balanced performance of the model across all classes in the test dataset. The confusion matrix illustrates the performance of the model across the four classes. Each row represents the true labels, while each column represents the predicted labels. For glioma, the model correctly predicted 125 instances (true positives), but also misclassified 39 meningiomas, 1 notumor, and 10 pituitary tumors as glioma (false positives). Similarly, for meningioma, the model correctly identified 119 instances, but also confused 33 gliomas, 6 notumors, and 17 pituitary tumors as meningioma. Notumor was predicted accurately with 173 instances correctly identified, but 2 pituitary tumors were misclassified as notumor. For pituitary tumor, the model correctly predicted 160 instances, while misclassifying 4 gliomas, 6 meningiomas, and 5 notumors as pituitary tumor. This confusion matrix provides a detailed breakdown of the model's classification performance, highlighting areas where it excels and areas where it can improve in accurately distinguishing between different types of brain tumors.

After training and evaluating the model's performance, we leveraged the trained LASSO Logistic Regression model to analyze its coefficients and identify features penalized to zero by L1 regularization. This process helps in understanding which features were deemed insignificant by the model, aiding in feature selection and model interpretability. To achieve this, we first extracted the unique classes on which the model was trained. Subsequently, we constructed a DataFrame (`df_logitic_regression_ovr`) where each row corresponds to a feature, and each column represents a class label by transposing the coefficients obtained from the model. We then filtered the rows of this DataFrame to identify features that were penalized to zero by LASSO, indicating their insignificance in predicting the target classes. This was achieved by creating `selected_rows`, which stores only those rows where all coefficients are zero. Finally, these rows

were printed, providing clear insights into which features were considered less important by the model during training.

```
classes = logitic_regression_ovr.classes_
df_logitic_regression_ovr = pd.DataFrame(logitic_regression_ovr.coef_.T, columns=
    classes)
selected_rows = df_logitic_regression_ovr[df_logitic_regression_ovr.isin([0]).all
    (axis=1)]
selected_rows
```

```
glioma  meningioma  notumor pituitary tumor
```

The output, showing only the class names without associated numerical values, indicates that none of the features were penalized to zero by the LASSO regularization process. This suggests that all features in the trained model are considered important for predicting the target classes.

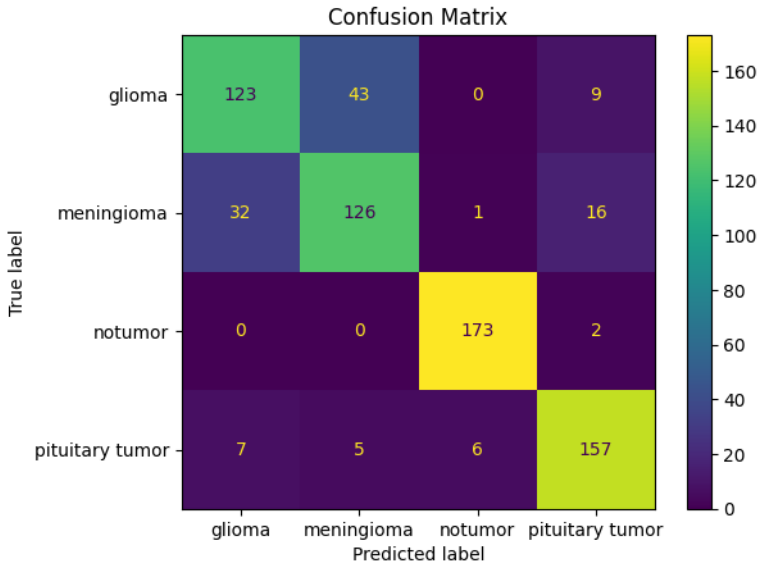
5.4.2 Multinomial LASSO Logistic Regression

This section details the training and evaluation of a Multinomial Logistic Regression model with LASSO regularization alongside automated cross-validation.
In the previous section, we utilized the One-vs-Rest (OvR) strategy to adapt Logistic Regression for multi-class classification. Now, we've opted for a more direct and intuitive approach with Multinomial Logistic Regression. Both OvR and Multinomial Logistic Regression are effective techniques for handling multi-class classification tasks, but they differ significantly in their approach. OvR tackles the problem by training multiple binary logistic regression models, each focusing on distinguishing one class from the rest. On the other hand, Multinomial Logistic Regression employs a unified model that directly predicts the probabilities of an observation belonging to each class. This model considers all classes simultaneously, capturing their inherent relationships and dependencies. This approach provides a comprehensive perspective on the data distribution and the nuanced interactions between different classes.
The Multinomial Logistic Regression model was instantiated using the `LogisticRegressionCV` class from `scikit-learn`, which incorporates cross-validation to optimize model parameters. Configured to utilize all available CPU cores for parallel processing, the model was set with the following parameters: L1 penalty for regularization, a fixed random state for reproducibility, refitting enabled to retain the best model, `multinomial` for handling multiple classes, the `saga` solver for optimization, 5-fold cross-validation, and a maximum of 50,000 iterations. The Multinomial Logistic Regression model was then trained on the training data: `X_train` and `y_train`. After training, we printed the range of regularization parameters tested during cross-validation (`logistic_regression_multinomial.Cs_`) and the best regularization strengths identified for each model (`logistic_regression_multinomial.C_`). Predictions of class labels on the test data (`X_test`) were generated using the trained model, and these predictions were stored in `y_pred`. To evaluate the model's performance on the test data, the `evaluate_metrics` function was applied, providing a comprehensive assessment of the model's accuracy and effectiveness in classifying the test samples.

```
common_params = {
    'penalty': 'l1',
    'random_state': 42,
    'refit': True,
    'multi_class': 'multinomial',
    'solver': "saga",
    'cv': 5,
    'max_iter': 50000
}
logistic_regression_multinomial = LogisticRegressionCV(n_jobs=-1, **common_params
    )

logistic_regression_multinomial.fit(X_train, y_train)

print("The penalties evaluated are:", "\n", logistic_regression_multinomial.Cs_)
print("\n", "The bests for every models are:", "\n",
    logistic_regression_multinomial.C_)

```

```
y_pred = logistic_regression_multinomial.predict(X_test)

print("Test")
evaluate_metrics(y_pred,y_test)
```

```
The penalties evaluated are:
 [1.00000000e-04 7.74263683e-04 5.99484250e-03 4.64158883e-02
 3.59381366e-01 2.78255940e+00 2.15443469e+01 1.66810054e+02
 1.29154967e+03 1.00000000e+04]

 The bests for every models are:
 [2.7825594 2.7825594 2.7825594 2.7825594]

Test
['glioma' 'meningioma' 'notumor' 'pituitary tumor']
                precision    recall  f1-score   support

        glioma       0.76      0.70      0.73       175
    meningioma       0.72      0.72      0.72       175
       notumor       0.96      0.99      0.97       175
pituitary tumor       0.85      0.90      0.87       175

      accuracy                           0.83       700
     macro avg       0.82      0.83      0.83       700
  weighted avg       0.82      0.83      0.83       700
```



The output shows that the Multinomial LASSO Logistic Regression model achieved an overall accuracy of 83%, indicating that it correctly classified 83% of the observations in the test set. The model achieved 76% precision for glioma, 72% for meningioma, 96% for notumor, and 85% for pituitary tumor, signifying the proportion of correctly predicted instances within each class. The model correctly identified 70% of all glioma instances, 72% of all meningioma instances, 99% of all notumor instances, and 90% of all pituitary tumor instances. The F1-score, which harmonizes precision and recall, revealed a robust balance for the notumor class, achieving the highest F1-score of 0.97 among all classes. These metrics collectively indicate that the model's predictions are reliable across most classes, particularly excelling in distinguishing "notumor" and "pituitary tumor". The macro-average and weighted-average metrics reinforce the overall robustness and balanced performance of the model across all classes in the test dataset. The confusion matrix illustrates that the model correctly predicted 123 instances for glioma, but also misclassified 43 meningiomas, and 9 pituitary tumors as glioma. For meningioma, the model correctly identified 126 instances, but also confused 32 gliomas, 1 notumor, and 16 pituitary tumors as meningioma. Notumor was predicted accurately with 173 instances correctly identified, but 2 pituitary tumors

were misclassified as notumor. For pituitary tumor, the model correctly predicted 157 instances, while misclassifying 7 gliomas, 5 meningiomas, and 6 notumors as pituitary tumor.
After training and evaluating the model's performance, we utilized the trained Multinomial Logistic Regression model to analyze its coefficients and filter out features that were penalized to zero by LASSO regularization.

```
classes = logistic_regression_multinomial.classes_
df_logistic_regression_multinomial = pd.DataFrame(logistic_regression_multinomial
    .coef_.T, columns=classes)
selected_rows = df_logistic_regression_multinomial[
    df_logistic_regression_multinomial.isin([0]).all(axis=1)]
selected_rows
```

```
glioma  meningioma  notumor  pituitary  tumor
```

However, since the output showed only the class names without associated numerical values, none of the features were penalized to zero by the LASSO regularization process. This suggests that all features in the trained model were considered important for predicting the target classes. To explore the importance and contribution of each feature in predicting their respective classes, we analyzed the coefficients associated with the features. The analysis involved iterating through each class name in the `classes` list. For each class, we retrieved the coefficients associated with the features from the DataFrame `df_logistic_regression_multinomial`, representing the weight or impact of each feature on predicting the class. By sorting these coefficients in descending order for each class, we identified the top ten features that contribute the most to predicting that particular class. These important features were then visualized using a bar chart, where taller bars indicate features that are more influential in predicting the corresponding class. These visualizations provide insights into which features are most critical for distinguishing each class according to the trained Multinomial Logistic Regression model.

```
num_features = 10

for name in classes:
    x = df_logistic_regression_multinomial[name].sort_values(ascending=False)
    important_features = x.sort_values(ascending=False)
    features = important_features.index.astype(str)
    print("\n" * 5,"The predicted distribution for ", name, "\n"*2)
    plt.figure(figsize=(8, 6))
    plt.bar(features[:num_features], important_features[:num_features], color='
    skyblue')
    plt.xlabel('features')
    plt.ylabel('Features Score')
    plt.title('More important Coefficient  Values')
    plt.show()
```

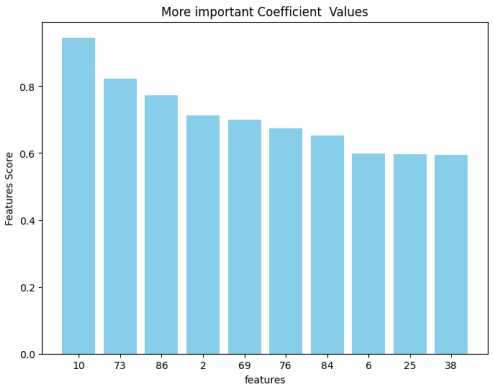


Figure 5.1: Glioma

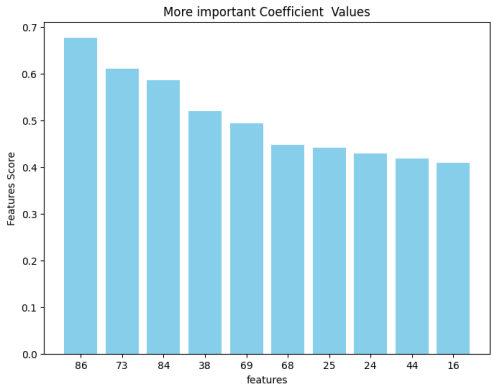


Figure 5.2: Meningioma

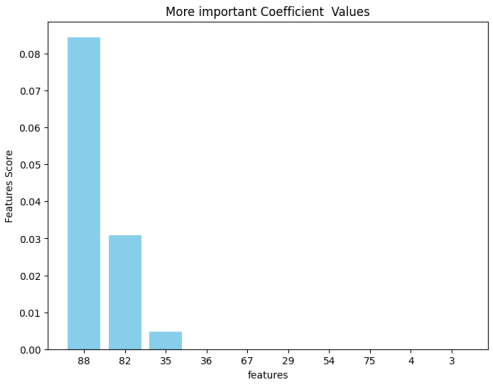


Figure 5.3: NoTumor

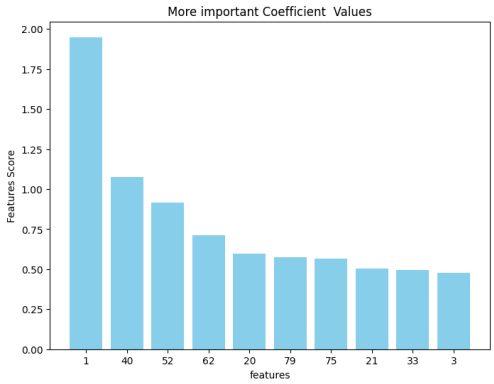


Figure 5.4: Pituitary Tumor

The bar charts showed bars of roughly the same height, indicating that the features have similar levels of importance when using the same coefficient scale on the y-axis. This uniformity in feature importance suggests that no single feature overwhelmingly influences the model's predictions for any given class. Instead, it implies that the model relies on a balanced combination of features to make accurate classifications.

5.5 Decision Tree Classifier

In this section, we detail the training and evaluation of a Decision Tree Classifier. We chose the Decision Tree Classifier (or Classification Tree) not only for its ability to create an effective classification model but especially for its inherent interpretability. This model constructs a decision tree where each node makes decisions based on specific features, classifying observations into different classes. Its rule-based structure helps to understand the decision-making process, making it an excellent tool for communicating results and insights.
To enhance model performance, we conducted hyperparameter tuning on the Decision Tree Classifier by adjusting its parameters, specifically focusing on `ccp_alpha` and `criterion`. We compared the performance of the model using two different splitting criteria: Entropy and Gini. These criteria are crucial as they determine how the decision tree partitions the feature space at each node, impacting the overall structure and accuracy of the model. For each criterion, we performed a cost complexity pruning to identify the optimal alpha value. This hyperparameter tuning process helps to find the best balance between model complexity and predictive performance, ensuring that the decision tree is both accurate and generalizable.
To facilitate this tuning process, we developed a function named `tuning_decision_tree`. This function automates the process of hyperparameter tuning, making it more efficient and systematic. First, the Decision Tree Classifier is initialized with a specified splitting criterion and a fixed random state for reproducibility. The `cost_complexity_pruning_path` method, which reduces the size of the tree by removing less important nodes, is then called on the training data (`X_train` and `y_train`). This method computes the alpha values and the impurities (`ccp_alphas` and `impurities`). The alpha values represent different levels of pruning, indicating the trade-off between tree complexity and accuracy, while impurities measure the total impurity of the nodes in the tree after pruning. Their relationship is plotted to visually assess the impact of pruning on the tree's impurity. Next, the function performs cross-validation on the Decision Tree Classifier using different pruning levels determined by the `ccp_alphas`. It initializes the K-Folds cross-validator to split the data into `num_fold` subsets, ensuring random distribution and reproducibility. Lists are created to store the results of cross-validation, including the number of nodes, tree depth, training accuracy, and validation accuracy for each fold. The code then iterates over each fold, splitting the data into new training and validation sets. Within each fold, for every `ccp_alpha` value, a new Decision Tree Classifier is initialized, trained on the new training set, and evaluated on both the training and validation sets. The results, including the number of nodes, tree depth, and accuracy scores, are recorded for each `ccp_alpha` value.

This process helps identify the optimal pruning level that balances tree complexity and model performance. After converting the lists storing cross-validation results into numpy arrays for easier manipulation, the function computes the mean values of these results across all cross-validation folds to get an overall estimate of the model's performance at each level of pruning. The function then generates two main visualizations: the first plots the number of nodes and tree depth against the alpha values, illustrating how tree complexity changes with pruning; the second shows the accuracy for both training and validation sets against alpha values, helping identify the optimal alpha that balances model complexity and performance.

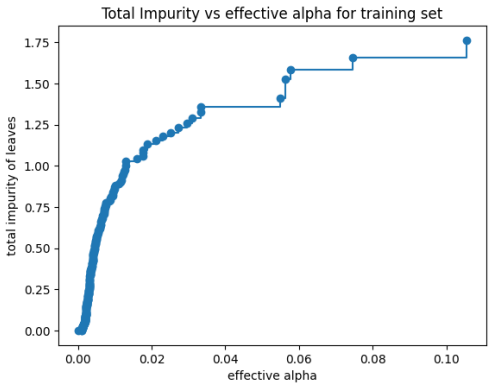
```
def tuning_decision_tree(criterion, X_train, y_train, num_fold):

    clf = DecisionTreeClassifier(random_state=0, criterion=criterion)

    path = clf.cost_complexity_pruning_path(X_train, y_train)
    ccp_alphas, impurities = path.ccp_alphas, path.impurities
    fig, ax = plt.subplots()
    ax.plot(ccp_alphas[:-1], impurities[:-1], marker="o", drawstyle="steps-post")
    ax.set_xlabel("effective alpha")
    ax.set_ylabel("total impurity of leaves")
    ax.set_title("Total Impurity vs effective alpha for training set")

    kf = KFold(n_splits=num_fold, shuffle=True, random_state=42)
    cv_node_counts = []
    cv_depth = []
    cv_train_scores = []
    cv_test_scores = []
    count = 0
    for train_index, validation_index in kf.split(X_train):
        print(count)
        count += 1
        new_X_train, new_y_train = X_train.iloc[train_index], y_train.iloc[
    train_index]
        X_validation, y_validation = X_train.iloc[validation_index], y_train.iloc
    [validation_index]
        node_counts = []
        depth = []
        train_scores = []
        test_scores = []
        for ccp_alpha in ccp_alphas:
            clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
            clf.fit(new_X_train, new_y_train)
            node_counts.append(clf.tree_.node_count)
            depth.append(clf.tree_.max_depth)
            train_scores.append(clf.score(new_X_train, new_y_train))
            test_scores.append(clf.score(X_validation, y_validation))
        cv_node_counts.append(node_counts)
        cv_depth.append(depth)
        cv_train_scores.append(train_scores)
        cv_test_scores.append(test_scores)
    cv_node_counts = np.array(cv_node_counts)
    cv_depth = np.array(cv_depth)
    cv_train_scores = np.array(cv_train_scores)
    cv_test_scores = np.array(cv_test_scores)
    cv_node_counts = np.mean(cv_node_counts, axis=0)
    cv_depth = np.mean(cv_depth, axis=0)
    cv_train_scores = np.mean(cv_train_scores, axis=0)
    cv_test_scores = np.mean(cv_test_scores, axis=0)

    fig, ax = plt.subplots(2, 1)
    ax[0].plot(ccp_alphas, cv_node_counts, marker="o", drawstyle="steps-post")
    ax[0].set_xlabel("alpha")
    ax[0].set_ylabel("number of nodes")
    ax[0].set_title("Number of nodes vs alpha")
    ax[1].plot(ccp_alphas, cv_depth, marker="o", drawstyle="steps-post")
    ax[1].set_xlabel("alpha")
    ax[1].set_ylabel("depth of tree")
    ax[1].set_title("Depth vs alpha")
    fig.tight_layout()

    fig, ax = plt.subplots()
```

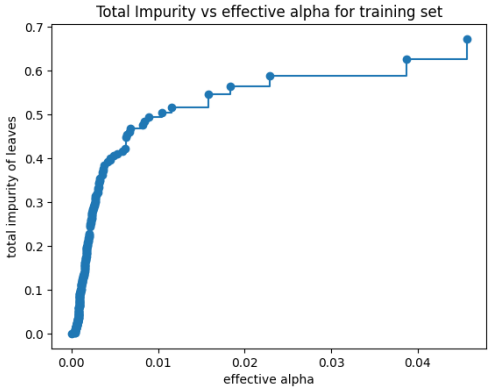
```
    ax.set_xlabel("alpha")
    ax.set_ylabel("accuracy")
    ax.set_title("Accuracy vs alpha for training and testing sets")
    ax.plot(ccp_alphas, cv_train_scores, label="train")
    ax.plot(ccp_alphas, cv_test_scores, label="test")
    ax.legend()
    plt.show()

    return cv_node_counts, cv_depth, cv_train_scores, cv_test_scores, ccp_alphas
```

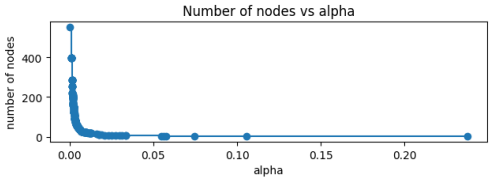
Using this function, we performed the hyperparameter tuning with the two different splitting criteria, Entropy and Gini. Specifically, the `tuning_decision_tree` function was called with the specified criterion ("entropy" or "gini"), training data (`X_train` and `y_train`), and the number of folds for cross-validation. This function returns various metrics that provide insights into the model's complexity and performance at different levels of pruning: the average number of nodes in the tree across folds, the average depth of the tree across folds, training accuracy scores across folds, validation accuracy scores across folds, and the cost complexity pruning alphas evaluated. These outputs provide insights into the model's complexity and performance at different levels of pruning.

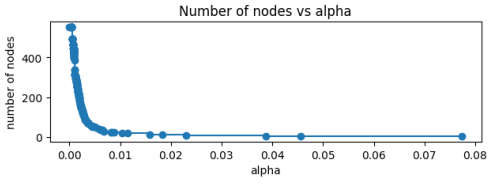
```
criterion = "entropy"
num_fold = 10
cv_node_counts_entropy, cv_depth_entropy, cv_train_scores_entropy,
    cv_test_scores_entropy, ccp_alphas_entropy = tuning_decision_tree(criterion,
    X_train, y_train, num_fold)
```

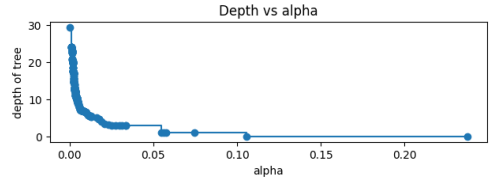
```
criterion = "gini"
num_fold= 10
cv_node_counts_gini, cv_depth_gini, cv_train_scores_gini, cv_test_scores_gini,
    ccp_alphas_gini = fine_tuning_decision_tree(criterion, X_train, y_train,
    num_fold)
```

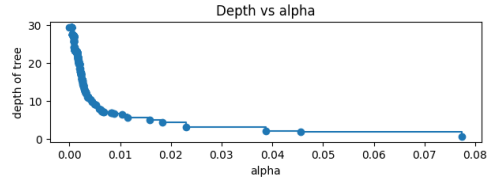


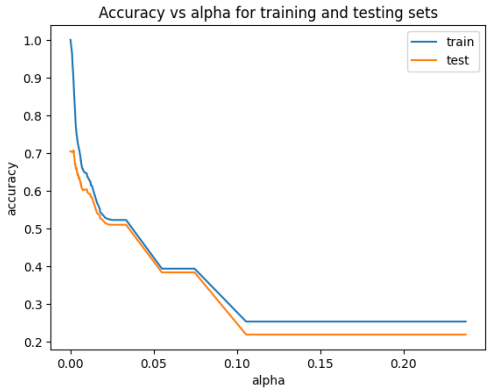


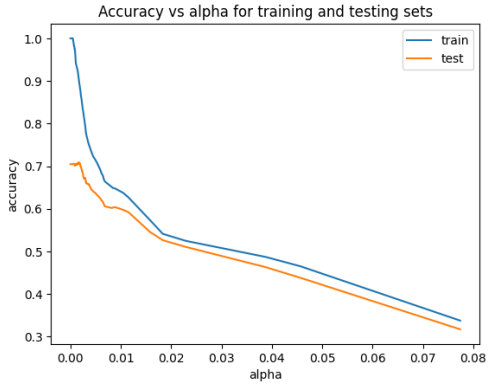












As we can observe from the graphical results, gradually increasing the alpha parameter leads to several notable trends: an increase in total impurity, a decrease in the number of nodes and tree depth, and a reduction in accuracy for both the training and testing sets. These trends illustrate the effects of cost complexity pruning, where higher alpha values simplify the tree by removing less important nodes, thereby increasing impurity but reducing overfitting. These visualizations help in identifying the optimal alpha value, which balances model complexity and performance. This optimal value is effectively pinpointed by selecting the alpha corresponding to the highest accuracy score during cross-validation.

```
max_score = np.max(cv_test_scores_entropy)
print("The best accuracy is", max_score)
index_max = np.argmax(cv_test_scores_entropy)
best_alpha_entropy = ccp_alphas_entropy[index_max]
print("with an alpha value of", ccp_alphas_entropy[index_max])
```

```
The best accuracy is 0.7071428571428572
with an alpha value of 0.001778732139766473
```

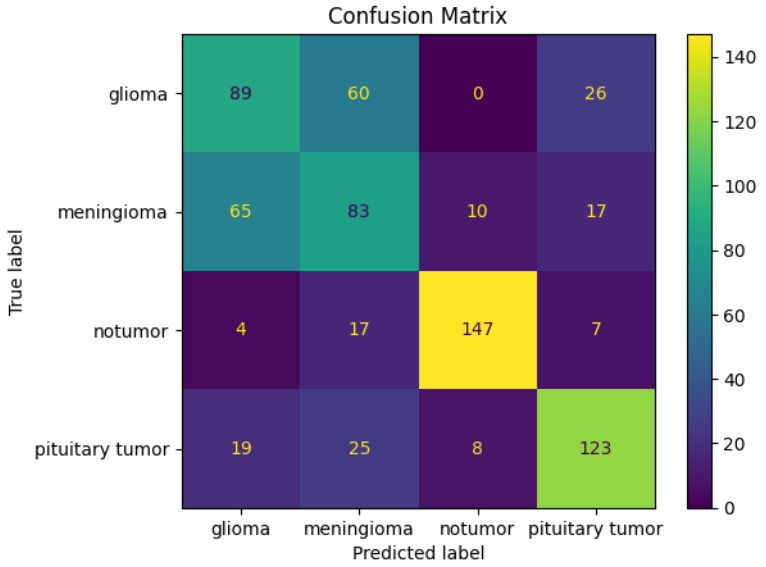
```
max_score = np.max(cv_test_scores_gini)
print("The best accuracy is", max_score)
index_max = np.argmax(cv_test_scores_gini)
best_alpha_gini = ccp_alphas_entropy[index_max]
print("with an alpha value of", best_alpha_gini)
```

```
The best accuracy is 0.7085714285714286
with an alpha value of 0.003935535574519241
```

For the entropy criterion, the best accuracy achieved during cross-validation was approximately 70.71%, with an optimal alpha value of about 0.00178. For the gini criterion, the best accuracy achieved was slightly higher at approximately 70.86%, with an optimal alpha value of around 0.00394. Overall, while both criteria yielded similar results, the gini criterion provided a marginally better accuracy with a slightly larger alpha value compared to entropy.
Based on these results, we decided to train and evaluate a Decision Tree Classifier initialized with the Gini criterion and its optimal alpha value to explore potential improvements. The model was trained on the training data (`X_train` and `y_train`) using the `fit` method, where the classifier learned to map features to their respective target labels. Once trained, the model predicted labels for the test data (`X_test`) using the `predict` method, storing these predictions in `y_pred`. To evaluate the model's performance on the test data, the `evaluate_metrics` function was invoked.

```
alpha = best_alpha_gini
clf = DecisionTreeClassifier(criterion="gini", ccp_alpha=alpha, random_state=8)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

print("Test")
evaluate_metrics(y_pred, y_test)
```

```
Test
['glioma' 'meningioma' 'notumor' 'pituitary tumor']
                precision    recall  f1-score   support

         glioma       0.50      0.51      0.51       175
     meningioma       0.45      0.47      0.46       175
        notumor       0.89      0.84      0.86       175
pituitary tumor       0.71      0.70      0.71       175

       accuracy                           0.63       700
      macro avg       0.64      0.63      0.63       700
   weighted avg       0.64      0.63      0.63       700
```

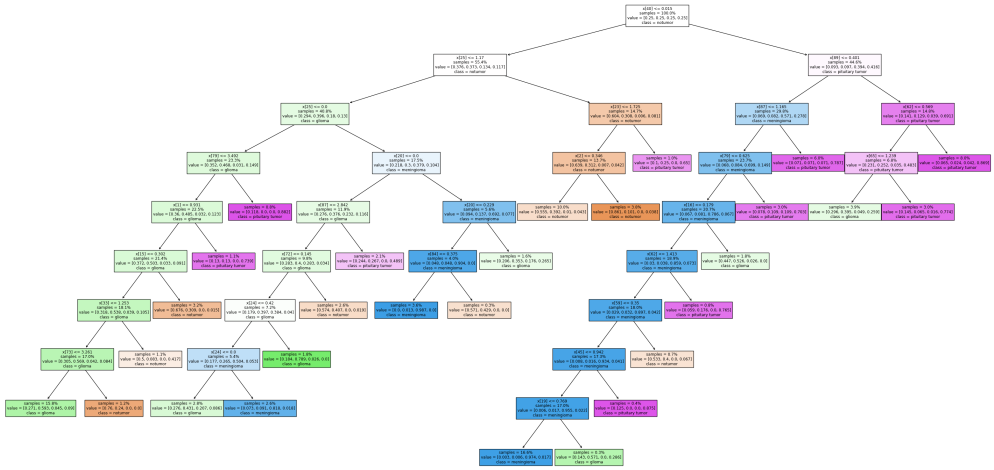


The output shows that the model achieved an overall accuracy of 63%. Breaking down the results by class, the precision, recall, and f1-score for glioma were 0.50, 0.51, and 0.51, respectively, indicating moderate performance in predicting this class. For meningioma, the precision was 0.45, recall was 0.47, and the f1-score was 0.46, suggesting similar moderate performance. The model performed significantly better in predicting the notumor class, with a precision of 0.89, recall of 0.84, and an f1-score of 0.86, showing high reliability in identifying this class. For pituitary tumor, the precision was 0.71, recall was 0.70, and the f1-score was 0.71, indicating good performance. The macro average across all classes for precision, recall, and f1-score was approximately 0.64, highlighting the model's balanced but moderate performance across different classes. The weighted average was also around 0.64, reflecting the overall effectiveness of the model considering the support (number of instances) for each class. The confusion matrix provides further insights into the model's performance across the four classes. For glioma, the model correctly predicted 89 instances (true positives), but also misclassified 60 meningiomas, and 26 pituitary tumors as glioma (false positives). Similarly, for meningioma, the model correctly identified 83 instances, but also confused 65 gliomas, 10 notumors, and 7 pituitary tumors as meningioma. For notumor, the model correctly predicted 147 instances, but also confused 4 gliomas, 17 meningiomas, and 7 pituitary tumors as notumor. For pituitary tumor, the model correctly predicted 123 instances, while misclassifying 19 gliomas, 25 meningiomas, and 8 notumors as pituitary tumor.

After training and evaluating the model's performance, a visual representation of the decision tree was generated using the `plot_tree` function from the `tree` module. This visualization provides a detailed overview of how the Decision Tree Classifier makes decisions by recursively partitioning the feature space based on the selected splitting criteria. Each node in the tree represents a decision rule applied to a specific feature, with branches extending to subsequent nodes or leaf nodes indicating the predicted class labels. This graphical representation aids in

understanding the hierarchical structure of the model's decision-making process, offering insights into the logic behind its predictions.

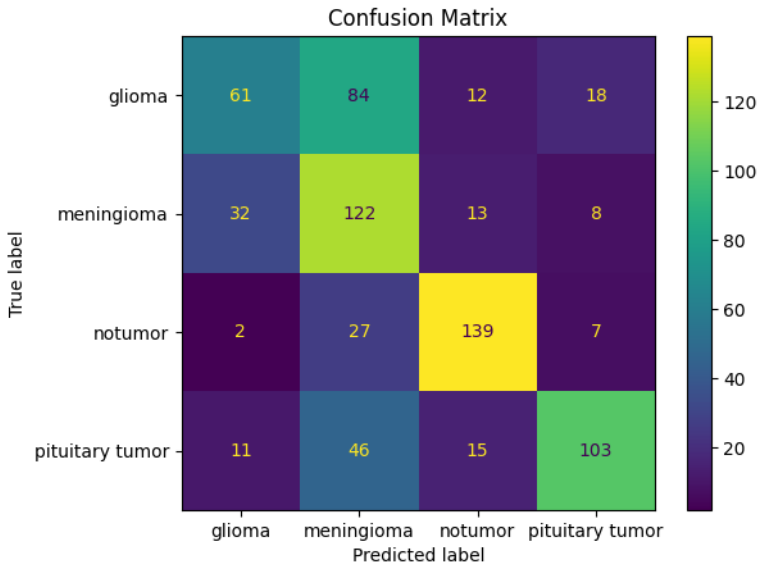
```
plt.figure(figsize=(40, 20))
tree.plot_tree(clf, class_names=labels, proportion=True, filled=True, impurity=
    False)
```



To obtain a smaller and more interpretable tree, we retrained and evaluated the Decision Tree Classifier with a `max_depth` of 4. By limiting the depth, the model is constrained to a smaller size, making it easier to interpret the decision rules while still aiming to maintain reasonable predictive performance.

```
clf = DecisionTreeClassifier(criterion="gini", max_depth=4, random_state=8)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)

print("Test")
evaluate_metrics(y_pred, y_test)
```

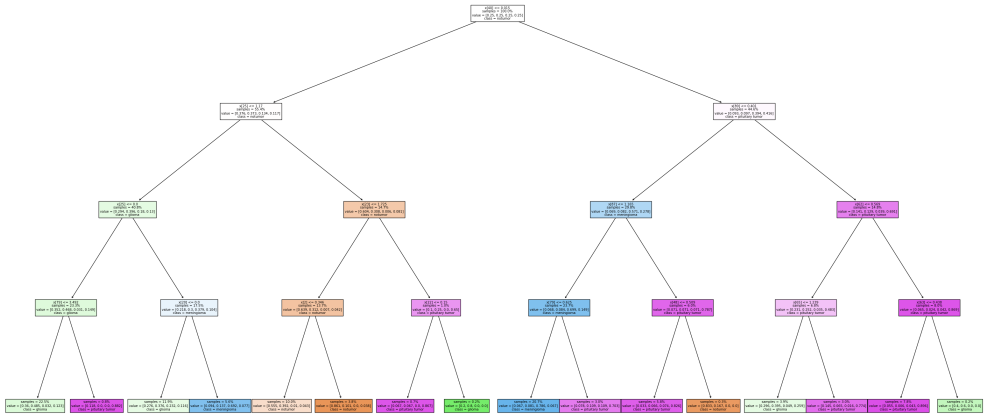
```
Test
['glioma' 'meningioma' 'notumor' 'pituitary tumor']
                precision    recall  f1-score   support

        glioma       0.58      0.35      0.43       175
    meningioma       0.44      0.70      0.54       175
       notumor       0.78      0.79      0.79       175
pituitary tumor       0.76      0.59      0.66       175

      accuracy                           0.61       700
     macro avg       0.64      0.61      0.60       700
  weighted avg       0.64      0.61      0.60       700
```



The output shows that the model achieved an overall accuracy of 61%. Breaking down the results by class, the precision, recall, and f1-score for glioma were 0.58, 0.35, and 0.43, respectively, indicating moderate performance in predicting this class. For meningioma, the precision was 0.44, recall was 0.70, and the f1-score was 0.54, suggesting similar moderate performance. The model performed significantly better in predicting the notumor class, with a precision of 0.78, recall of 0.79, and an f1-score of 0.79, showing high reliability in identifying this class. For pituitary tumor, the precision was 0.76, recall was 0.59, and the f1-score was 0.66, indicating good performance. The confusion matrix provides further insights into the model's performance across the four classes. For glioma, the model correctly predicted 61 instances (true positives), but also misclassified 84 meningiomas, 12 notumors, and 18 pituitary tumors as glioma (false positives). Similarly, for meningioma, the model correctly identified 122 instances, but also confused 32 gliomas, 13 notumors, and 8 pituitary tumors as meningioma. For notumor, the model correctly predicted 139 instances, but also confused 2 gliomas, 27 meningiomas, and 7 pituitary tumors as notumor. For pituitary tumor, the model correctly predicted 103 instances, while misclassifying 11 gliomas, 46 meningiomas, and 15 notumors as pituitary tumor. In summary, while limiting the tree's depth to 4 makes the model simpler and more interpretable, it results in a slight trade-off in performance compared to the deeper tree model.

After training and evaluating the model's performance, a visual representation of the new and smaller decision tree was generated using the `plot_tree` function from the `tree` module.

```
plt.figure(figsize=(40, 20))
tree.plot_tree(clf, class_names=labels, proportion=True, filled=True, impurity=
    False)
```



Our model exhibited fairly low recall values for the glioma and pituitary tumor classes, indicating that a significant proportion of patients with these conditions were not correctly identified. Consequently, it is prudent not to rely on the model for diagnosing these conditions. However, our model demonstrated relatively high precision and recall values for the notumor class, indicating its effectiveness in reducing both false positives and false negatives. Therefore, physicians could consider using the model as a supportive tool while still relying on additional diagnostic tests and clinical evaluations to confirm the absence of a tumor. Specifically, the interpretation of the decision tree revealed that a combination of a high value for feature 2, a low value for feature 23, a high value for feature 25, and a low value for feature 40 was highly indicative of the absence of a tumor. In the case of the meningioma class, our model demonstrated moderate precision and relatively high recall values, indicating its effectiveness in reducing false negatives, though it still produced a significant number of false positives. Physicians could consider using the model as a supportive tool while still relying on additional diagnostic tests and clinical evaluations to confirm the presence of a tumor. Specifically, the decision tree showed that a low value for features 79, 87, and 89, combined with a high value for feature 40, strongly indicated the presence of a meningioma tumor.

5.6 Random Forest

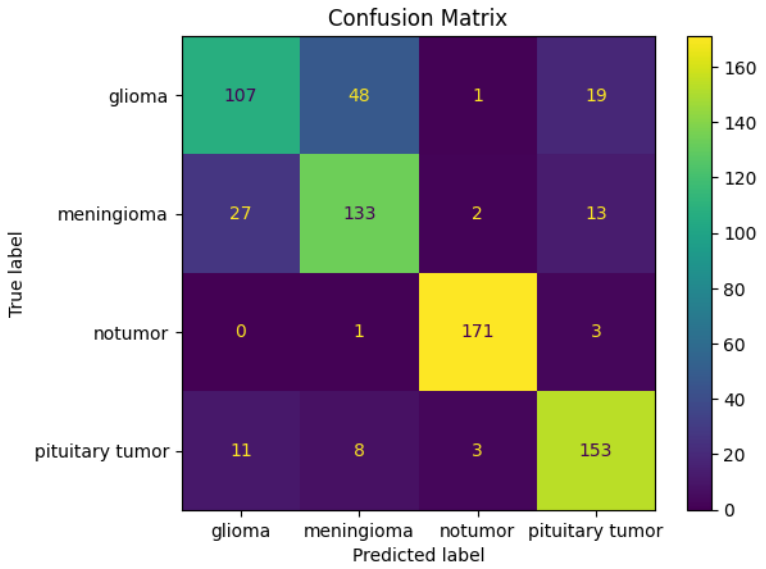
This section outlines the training and evaluation of a Random Forest Classifier. We selected the Random Forest Classifier due to its ability to aggregate multiple decision trees, thereby enhancing performance compared to a single decision tree. This ensemble approach harnesses the strengths of individual trees, resulting in more accurate and reliable classifications.
To enhance model performance, we conducted hyperparameter tuning on the Random Forest Classifier using `GridSearchCV` from the `scikit-learn` library. The Random Forest Classifier (`rf`) was instantiated with a fixed random state for reproducibility and configured to utilize all available CPU cores for parallel processing. We defined a parameter grid specifying various values for the maximum depth of the trees, the minimum number of samples required at a leaf node, the number of trees in the forest, the criterion for measuring the quality of splits ("gini"), and the number of features to consider when looking for the best split. `GridSearchCV` was configured with this parameter grid, a 10-fold cross-validation, and set to use all available cores for computation, measure performance based on accuracy, and refit the best model on the entire dataset once the optimal parameters are found. Executing `grid_search.fit(X_train, y_train)` systematically tested all the possible combinations of hyperparameters specified in the parameter grid by training the `rf` on the training data. For each combination, it performed a 10-fold cross-validation, involving splitting the training data into 10 subsets, training the model on 9 subsets, and validating it on the remaining subset. This process was repeated 10 times, with each subset used once as the validation set. The average performance across these folds was calculated for each hyperparameter combination. Once the combination that yielded the

highest accuracy was identified, the model was refitted on the entire training dataset using this optimal set of hyperparameters, ensuring that the final model was trained with the best possible parameter configuration found during the search. The highest average accuracy obtained from testing all hyperparameter combinations was printed. The best model, trained with the optimal hyperparameters, was stored in the variable `model`, and its parameters were stored in the variable `par`. Finally, predictions on the test data `X_test` were made using the `predict` method, and these predictions were stored in the variable `y_pred`. To evaluate the model's performance on the test data, the `evaluate_metrics` function was invoked.

```
rf = RandomForestClassifier(random_state=42, n_jobs=-1)

params = {
    'max_depth': [10, 12, 15, 20],
    'min_samples_leaf': [5, 10, 20, 50],
    'n_estimators': [200, 400, 600],
    'criterion': ["gini"],
    'max_features': ["sqrt", "log2", None]
}
grid_search = GridSearchCV(estimator=rf,
                           param_grid=params,
                           cv=10,
                           n_jobs=-1,
                           verbose=1,
                           scoring="accuracy",
                           refit=True)

grid_search.fit(X_train, y_train)

print("Best score", grid_search.best_score_)

model = grid_search.best_estimator_

model.get_params()
par = model.get_params()

y_pred = model.predict(X_test)

print("Test")
evaluate_metrics(y_pred,y_test)
```

```
Best score 0.8047619047619048

{'bootstrap': True,
 'ccp_alpha': 0.0,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': 12,
 'max_features': 'sqrt',
 'max_leaf_nodes': None,
 'max_samples': None,
 'min_impurity_decrease': 0.0,
 'min_samples_leaf': 5,
 'min_samples_split': 2,
 'min_weight_fraction_leaf': 0.0,
 'n_estimators': 200,
 'n_jobs': -1,
 'oob_score': False,
 'random_state': 42,
 'verbose': 0,
 'warm_start': False}

Test
['glioma' 'meningioma' 'notumor' 'pituitary tumor']
                precision    recall  f1-score   support

         glioma       0.74      0.61      0.67       175
     meningioma       0.70      0.76      0.73       175
        notumor       0.97      0.98      0.97       175
pituitary tumor       0.81      0.87      0.84       175
```

```
      accuracy                           0.81       700
     macro avg       0.80      0.81      0.80       700
  weighted avg       0.80      0.81      0.80       700
```

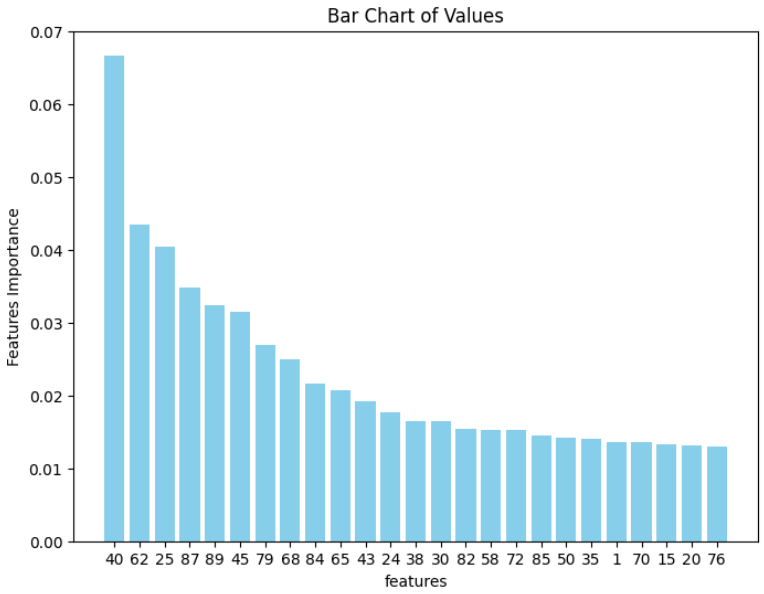


The performance evaluation of the Random Forest Classifier on the test data indicates that an overall accuracy of 81% was achieved, with the model demonstrating strong performance across most classes. Specifically, the classifier achieved a precision of 0.74 and a recall of 0.61 for glioma, a precision of 0.70 and a recall of 0.76 for meningioma, an impressive precision and recall of 0.97 and 0.98 respectively for notumor, and a precision of 0.81 and a recall of 0.87 for pituitary tumor. The high F1-scores, particularly for notumor (0.97) and pituitary tumor (0.84), further emphasize the model's reliability in these categories. The macro and weighted averages of precision, recall, and F1-score hover around 0.80 to 0.81, indicating balanced performance across all classes. These results highlight the model's strong capability in correctly classifying no tumor cases and a generally good performance in identifying glioma, meningioma, and pituitary tumors, making it a robust classifier for this multi-class medical classification task. The confusion matrix provides further insights into the model's performance across the four classes. It shows that for glioma, the model correctly predicted 107 instances but misclassified 48 meningiomas, 1 notumor, and 19 pituitary tumors. For meningioma, it correctly identified 133 instances but confused 27 gliomas, 2 notumors, and 13 pituitary tumors. For notumor, the model accurately predicted 171 instances, misclassifying 1 meningioma and 3 pituitary tumors. For pituitary tumor, the model correctly predicted 153 instances but misclassified 11 gliomas, 8 meningiomas, and 3 notumors.

To understand which features contributed most significantly to the model's predictions, we calculated and visualized the importance of each feature used in the Random Forest model. First, we extracted the feature importances from the trained model, which quantify the relative contribution of each feature. Then, we retrieved the feature names from the training data, ensuring they were formatted as strings. These names and their corresponding importance scores were stored in a DataFrame, which was sorted in descending order based on the importance values. We selected the top 25 most important features and displayed them in a bar plot. The plot shows the feature names on the x-axis and their importance scores on the y-axis, providing a clear visual representation of the most influential features in the model's decision-making process.

```
importances = model.feature_importances_

feature_names = X_train.columns
feature_names = feature_names.astype(str)

```

```
feature_importance_df = pd.DataFrame({'Feature': feature_names, 'Importance':
    importances})
feature_importance_df = feature_importance_df.sort_values(by='Importance',
    ascending=False)

num_to_show = 25
z = feature_importance_df.iloc[:num_to_show,:]
plt.figure(figsize=(8, 6))
plt.bar(z['Feature'], z['Importance'], color='skyblue')
plt.xlabel('features')
plt.ylabel('Features Importance')
plt.title('Bar Chart of Values')
plt.show()
```



Having identified the model with the optimal set of hyperparameters and determined the feature importances, we proceeded with Recursive Feature Elimination with Cross-Validation to refine the feature selection process.

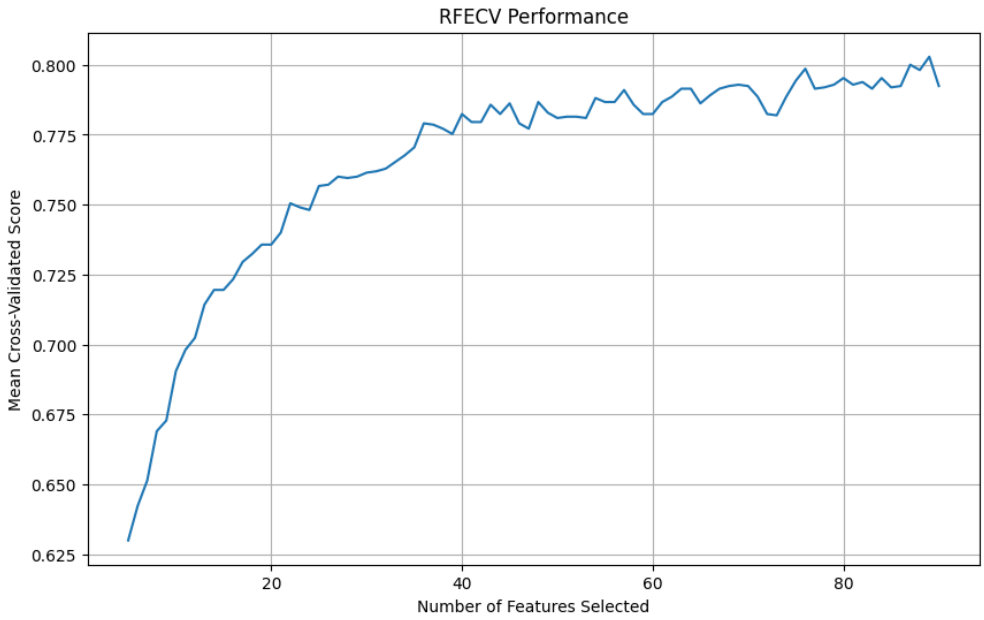
Recursive Feature Elimination (RFE) is a feature selection technique that recursively removes the least important features and builds the model iteratively until the specified number of features is reached. It helps in selecting the most significant features that contribute to the model's predictive power, thereby reducing model complexity and improving performance. Recursive Feature Elimination with Cross-Validation (RFECV) extends RFE by integrating cross-validation into the feature elimination process. It evaluates the model performance for each feature subset through cross-validation, ensuring that the optimal number of features is selected based on cross-validated accuracy. This approach provides a robust mechanism for feature selection by considering both feature importance and model performance across multiple folds.

The `RFECV` was initialized with the Random Forest best model as the estimator. It was configured to use 5-fold cross-validation, constrained to select at least 5 features, set to remove one feature at a time, and utilize all available CPU cores for computation. The RFECV algorithm recursively eliminated the least important features, fitting the model on the training data (`X_train` and `y_train`), and evaluating its performance through cross-validation at each iteration. After the fitting process, the optimal number of features was determined and printed. The cross-validation results, including the mean test scores for each feature set, were extracted and plotted against the number of features selected. This plot visualizes how the model's performance changes with varying numbers of features, helping to identify the optimal feature set size that maximizes the model's cross-validated score.

```
from sklearn.feature_selection import RFECV

```

```
rfe = RFECV(model, cv=5, min_features_to_select = 5, step=1, n_jobs= -1)
rfe.fit(X_train, y_train)

best_number = rfe.n_features_
print('Optimal number of features: %d' % rfe.n_features_)

result = rfe.cv_results_
r = result['mean_test_score']

plt.figure(figsize=(10, 6))
plt.plot(range(5, len(r) + 5), r)
min_features_to_select
plt.xlabel('Number of Features Selected')
plt.ylabel('Mean Cross-Validated Score')
plt.title('RFECV Performance')
plt.grid(True)
plt.show()
```

```
Optimal number of features: 89
```

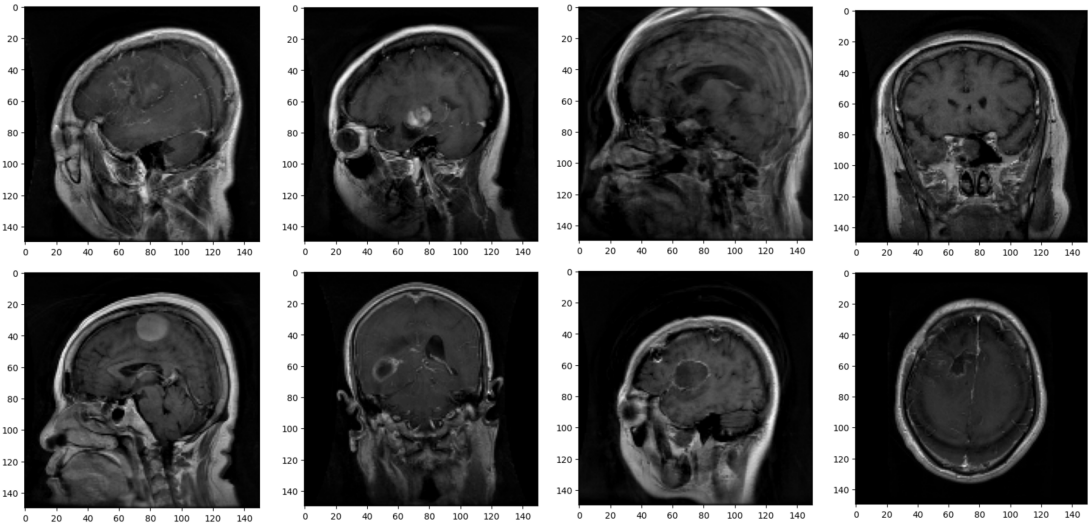


Based on the RFECV performance plot, it is evident that satisfactory accuracy, around 70%, can be achieved even with significantly fewer features. Therefore, we employed Recursive Feature Elimination (RFE) to further refine our feature selection process and identify the 8 most important variables. This approach ensures that the final model leverages the key features that contribute most significantly to its predictive power, enhancing both efficiency and effectiveness by reducing the complexity of the model while maintaining a high level of performance. After the RFE process, the selected feature names were extracted, converted to integers, and stored in a list (`col`). This list of selected feature indices was then printed, providing a clear view of the features deemed most important by the RFE process. To further analyze these selected features, we reshaped each selected feature vector into a 150x150 pixel image in grayscale, allowing us to inspect the spatial structure of the most important features.

```
rfe_new = RFE(model, n_features_to_select=8)
rfe_new.fit(X_train, y_train)

col = feature_names[rfe_new.support_]
col = list(col.astype(int))
print(col)

features[col].shape

```

```
size = (150,150)
for i in col:
  image = features[i,:]
  image = image.reshape(size)
  plt.imshow(image, cmap="gray")
  plt.show()
```

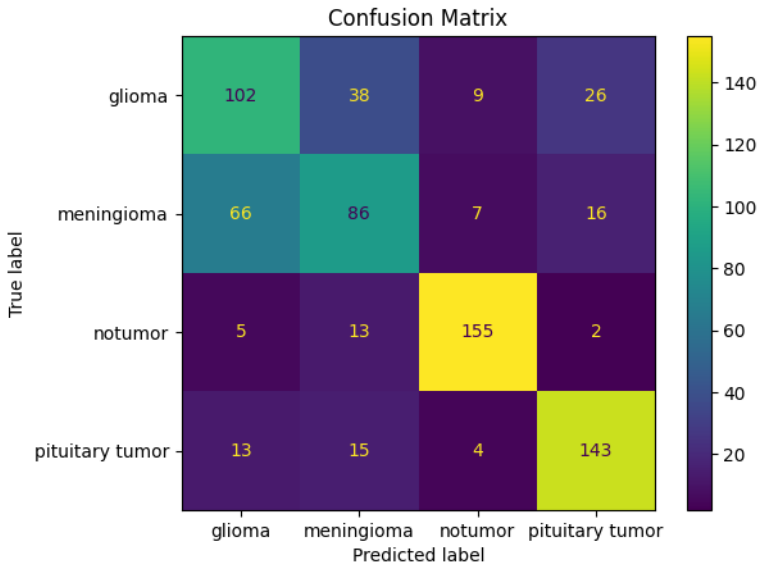
```
[25, 40, 45, 62, 68, 79, 87, 89]

(8, 22500)
```



After identifying the 8 features that contribute most significantly to the predictive power, we created a new features matrix `X_new` containing only these selected features. We then split this new dataset into training and test sets, ensuring that the class distribution is maintained in both sets through stratification. Using this refined dataset, we performed the Random Forest classification again. By training the Random Forest classifier on the new training data and making predictions on the test data, we were able to evaluate the model's performance using the selected features. The `evaluate_metrics` function was used to compute and print essential classification metrics and displaying the confusion matrix, thereby providing insights into the effectiveness of the model on the refined feature set.

```
X_new = X.iloc[:, col]
test_size = 0.25
X_train_new, X_test_new, y_train_new, y_test_new = train_test_split(X_new, y,
    test_size=test_size, stratify=y, random_state=42)

rf_model_new = RandomForestClassifier(random_state=40)
rf_model_new.fit(X_train_new, y_train_new)
y_pred_new = rf_model_new.predict(X_test_new)
print("Test")
evaluate_metrics(y_pred_new,y_test_new)
```

```
Test
['glioma' 'meningioma' 'notumor' 'pituitary tumor']
                precision    recall  f1-score   support

         glioma       0.55      0.58      0.57       175
     meningioma       0.57      0.49      0.53       175
        notumor       0.89      0.89      0.89       175
pituitary tumor       0.76      0.82      0.79       175

       accuracy                           0.69       700
      macro avg       0.69      0.69      0.69       700
   weighted avg       0.69      0.69      0.69       700
```



The performance evaluation shows varying levels of effectiveness across different classes. The model achieved a precision of 0.55 and a recall of 0.58 for glioma, and a precision of 0.57 and a recall of 0.49 for meningioma, indicating moderate performance in identifying these classes. For notumor, the model performed excellently with a precision and recall of 0.89, resulting in a high F1-score of 0.89. Similarly, for pituitary tumor, the model achieved a precision of 0.76 and a recall of 0.82, reflecting strong performance with an F1-score of 0.79. The model achieved an overall accuracy of 69%, suggesting that it correctly classified 69% of the instances in the test set. The macro and weighted averages for precision, recall, and F1-score are all 0.69, indicating balanced performance across all classes. In particular, the confusion matrix illustrates that for glioma, the model correctly predicted 102 instances but misclassified 38 meningiomas, 9 notumor, and 26 pituitary tumors. For meningioma, it correctly identified 86 instances but confused 66 gliomas, 7 notumors, and 16 pituitary tumors. For notumor, the model accurately predicted 155 instances, misclassifying 5 gliomas, 13 meningiomas, and 2 pituitary tumors. For pituitary tumor, the model correctly predicted 143 instances but misclassified 13 gliomas, 15 meningiomas, and 4 notumors.

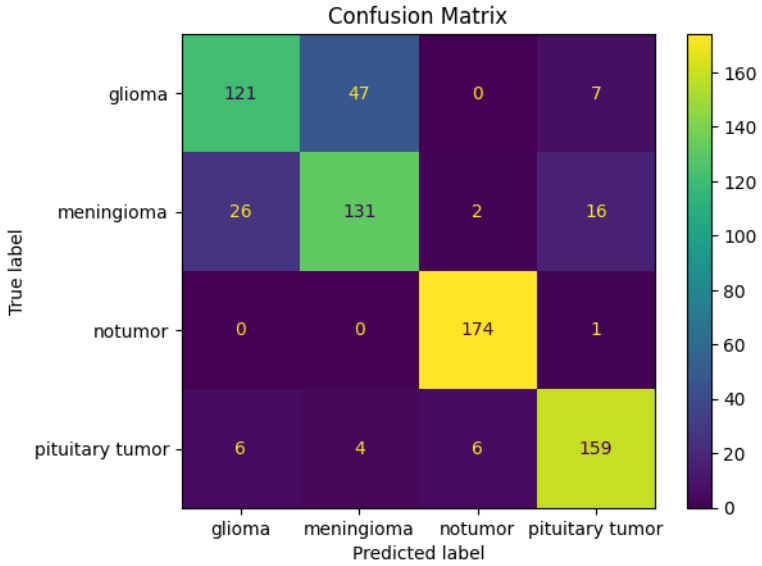
5.7 Support Vector Classifier

This paragraph details the training and evaluation of a Support Vector Classifier (SVC). To enhance model performance, we conducted hyperparameter tuning on the Support Vector Classifier using `RandomizedSearchCV` from the `scikit-learn` library. The Support Vector Classifier (`svc`) was instantiated with `probability=True` to enable probability estimates. We defined a parameter grid, specifying various values for the penalty parameter `C`, the kernel coefficient `gamma`, the kernel type, the polynomial degree, and the kernel coefficient `coef0`. `RandomizedSearchCV` was configured with this parameter grid, set to perform 10 iterations, use 5-fold cross-validation, utilize all available cores for computation, and refit the best model on the entire dataset once the optimal parameters are found. Executing `random_search.fit(X_train, y_train)` systematically tested all the possible combinations of hyperparameters specified in the parameter grid by training `svc` on the training data. For each combination, it performed a 5-fold cross-validation, involving splitting the training data into 5 subsets, training the model on 4 subsets, and validating it on the remaining subset. This process was repeated 5 times, with each subset used once as the validation set. The average performance across these 5 folds was calculated for each hyperparameter combination. Once the combination that yielded the highest accuracy was identified, the model was refitted on the entire training dataset using this optimal set of hyperparameters, ensuring that the final model was trained with the best possible parameter configuration found during the search. The best model trained with the optimal hyperparameters found during the

search (`svc`) and its parameters (`best_params`) were stored and printed. Finally, predictions on the test data `X_test` were made using the `predict` method, storing these predictions in the variable `y_pred`. To evaluate the model's performance on the test data, the `evaluate_metrics` function was invoked.

```
svc = SVC(probability=True)

par = {
    'C': [0.1, 1, 10, 100],
    'gamma': [0.0001, 0.001, 0.1, 1],
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
    'degree': [2, 3, 4],
    'coef0': [0, 1]
}

random_search = RandomizedSearchCV(svc,
                                   par,
                                   n_iter=10,
                                   cv=5,
                                   n_jobs=-1,
                                   refit=True,
                                   random_state=42)

random_search.fit(X_train, y_train)

best_params = random_search.best_params_
print(best_params)

svc_best_model = random_search.best_estimator_
print(random_search.best_estimator_)

y_pred = svc_best_model.predict(X_test)

print("Test")
evaluate_metrics(y_pred, y_test)

del y_pred, random_search
gc.collect()

print("Train")
evaluate_metrics(svc_best_model.predict(X_train), y_train)
```

```
{'kernel':'linear','gamma':0.1,'degree':2,'coef0':1,'C':0.1}

Test
['glioma' 'meningioma' 'notumor' 'pituitary tumor']
                precision    recall  f1-score   support

         glioma       0.79      0.69      0.74       175
     meningioma       0.72      0.75      0.73       175
        notumor       0.96      0.99      0.97       175
pituitary tumor       0.87      0.91      0.89       175

       accuracy                           0.84       700
      macro avg       0.83      0.84      0.83       700
   weighted avg       0.83      0.84      0.83       700
```



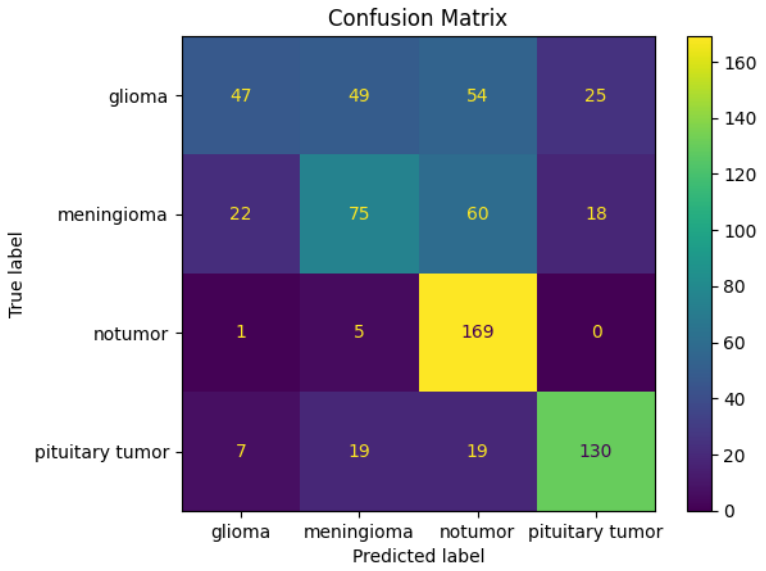
The performance evaluation of the Support Vector Classifier on the test data indicates that an overall accuracy of 84% was achieved, with the model demonstrating strong performance across the four classes. For glioma, the model attained a precision of 0.79, recall of 0.69, and f1-score of 0.74. For meningioma, the model attained a precision of 0.72, recall of 0.75, and f1-score of 0.73. The model excelled in predicting the notumor class with a precision of 0.96, recall of 0.99, and f1-score of 0.97. For pituitary tumor, the precision was 0.87, recall was 0.91, and f1-score was 0.89, demonstrating good performance. The macro and weighted averages for precision, recall, and f1-score were approximately 0.83, highlighting the model's balanced and effective performance across different classes. These results suggest that the SVC is a reliable tool for identifying different tumor types and the absence of tumors, thus supporting medical diagnostics. The confusion matrix provides further insights into the model's performance across the four classes. It shows that for glioma, the model correctly predicted 121 instances but misclassified 47 meningiomas, and 7 pituitary tumors. For meningioma, it correctly identified 131 instances but confused 26 gliomas, 2 notumors, and 16 pituitary tumors. For notumor, the model accurately predicted 174 instances, misclassifying only 1 pituitary tumor. For pituitary tumor, the model correctly predicted 159 instances but misclassified 6 gliomas, 4 meningiomas, and 6 notumors.

Using the refined dataset obtained selecting the 8 features, we performed the Support Vector Classifier again. By training the Support Vector Classifier on the new training data and making predictions on the test data, we were able to evaluate the model's performance using the selected features. The `evaluate_metrics` function was used to compute and print essential classification metrics and displaying the confusion matrix, thereby providing insights into the effectiveness of the model on the refined feature set.

```
svc_new = SVC(**best_params, random_state=40)
svc_new.fit(X_train_new, y_train_new)
y_pred_new = svc_new.predict(X_test_new)
print("Test")
evaluate_metrics(y_pred_new, y_test_new)
```

```
Test
['glioma' 'meningioma' 'notumor' 'pituitary tumor']
                precision    recall  f1-score   support

        glioma       0.61      0.27      0.37       175
    meningioma       0.51      0.43      0.46       175
       notumor       0.56      0.97      0.71       175
pituitary tumor       0.75      0.74      0.75       175

      accuracy                           0.60       700
     macro avg       0.61      0.60      0.57       700
```

```
weighted avg       0.61      0.60      0.57       700
```



The performance evaluation shows varying levels of effectiveness across different classes. The model achieved a precision of 0.61 and a recall of 0.27 for glioma, and a precision of 0.51 and a recall of 0.43 for meningioma. For notumor, the model achieved a precision of 0.56 and a recall of 0.97. For pituitary tumor, the model achieved a precision of 0.75 and a recall of 0.74. The model achieved an overall accuracy of 60%, suggesting that it correctly classified 60% of the instances in the test set. In particular, the confusion matrix illustrates that for glioma, the model correctly predicted 47 instances but misclassified 49 meningiomas, 54 notumor, and 25 pituitary tumors. For meningioma, it correctly identified 75 instances but confused 22 gliomas, 60 notumors, and 18 pituitary tumors. For notumor, the model accurately predicted 169 instances, misclassifying 1 glioma, and 5 meningiomas. For pituitary tumor, the model correctly predicted 130 instances but misclassified 7 gliomas, 19 meningiomas, and 19 notumors.

6. Pairplot among key features

In this section, we visualized the pairwise relationships between the eight most significant features identified during the Recursive Feature Elimination (RFE) process. This analysis aims to uncover patterns and assess the potential for further feature reduction.

6.1 Libraries

These imports collectively set up the environment to perform data manipulation, visualization, and analysis.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

6.2 DataFrame

First, we created the `df` object to represent the DataFrame contained in the pickle file named "selected_df.pkl". This DataFrame includes the eight most important features ([25, 40, 45, 62, 68, 79, 87, 89]) identified during the Recursive Feature Elimination process for Random Forest, which contribute most significantly to its predictive power, along with the column label. The dataset was then reset to ensure a clean index, and the column names were converted to strings for consistency. To create a representative subset for subsequent analysis, we randomly sampled 500 entries from the dataset without replacement. The distribution of the labels in this new subset `df_new` was then verified.

```
from google.colab import drive
drive.mount('/content/drive')

path = "/content/drive/MyDrive/ProgettoDataMining/selected_df.npy"
df = pd.read_pickle(path)

df = df.reset_index(drop=True)
df.columns = df.columns.astype(str)

df_new = df.sample(n=500, replace=False, random_state=46)
df_new["Label"].value_counts()
```

```
Label
notumor            142
pituitary tumor    131
glioma             115
meningioma         112
Name: count, dtype: int64
```

The resulting subset comprised 142 samples of notumor, 131 samples of pituitary tumor, 115 samples of glioma, and 112 samples of meningioma.

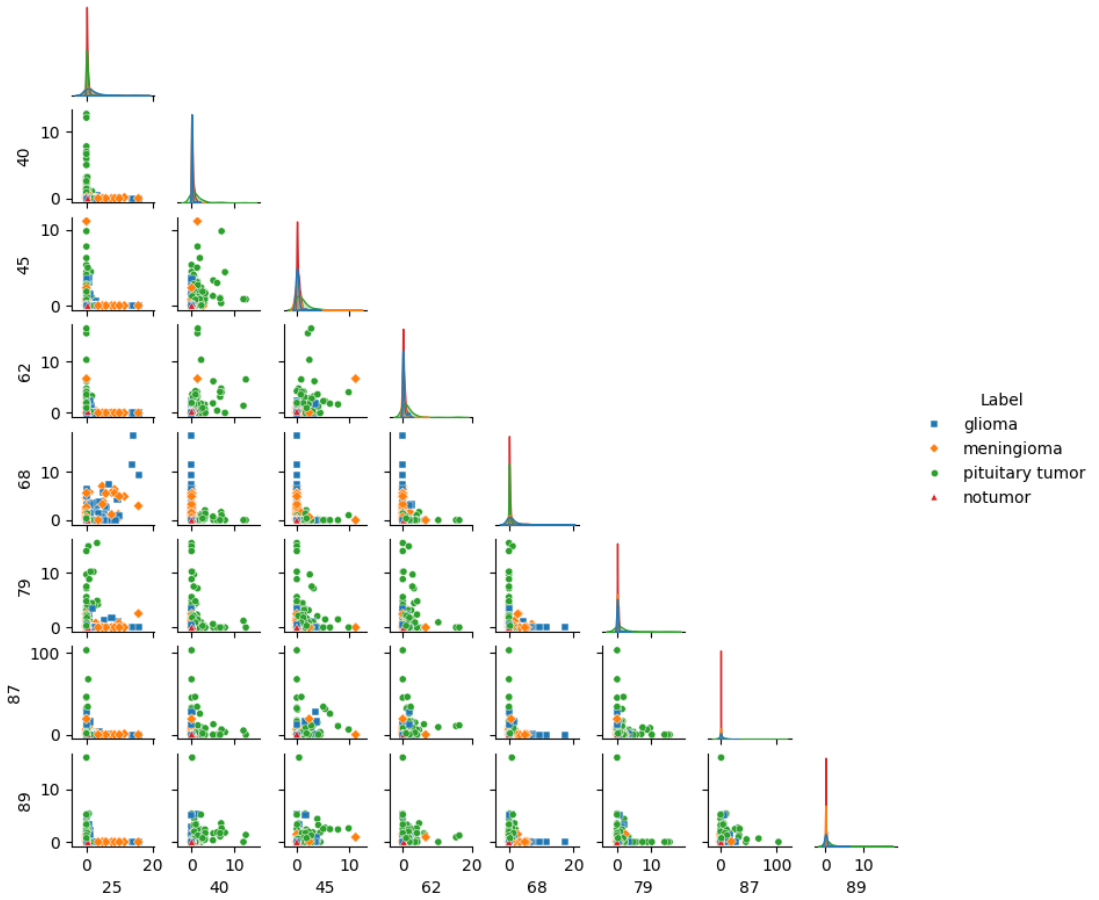
6.3 Data Visualization

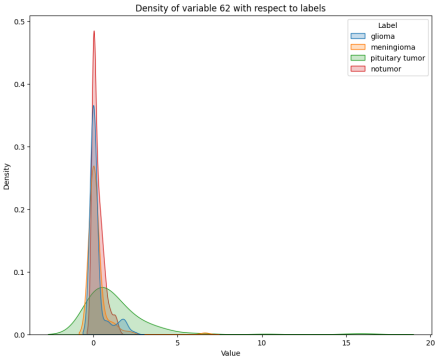
To visualize the relationships between the features in our subset `df_new`, we created a pairplot using the `seaborn` library. We defined a dictionary to map each tumor label to a specific marker shape: squares for glioma, diamonds for meningioma, circles for pituitary tumor, and triangles for notumor. We set the marker size to 20 and specified the order of the hue categories to maintain consistency in the color coding. By plotting the pairplot with these configurations, we could effectively examine the interactions between features across different tumor labels.
To analyze the pairplot graphs in detail, we zoomed in on specific scatter plots and diagonal density plots. Below is an example of a zoomed-in scatter plot showing the relationship between variables 68 and 89, providing a more detailed view of how different tumor types are distributed across these two features. Additionally, below is a zoomed-in example of a KDE plot for variable 62, offering a focused view of the distribution of this single feature across different labels, allowing us to understand how well this feature separates the various tumor types.

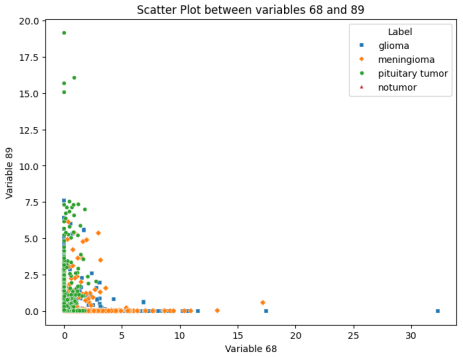
```
markers = {
    "glioma": "s",
    "meningioma": "D",
    "pituitary tumor": "o",
    "notumor": "^"
    }
value = 20
hue_order = ["glioma", "meningioma", "pituitary tumor", "notumor"]
sns.pairplot(df_new, hue="Label", height=1, aspect = 1, hue_order=hue_order,
    markers=markers, plot_kws={"s": value}, corner=True)
```

```
plt.figure(figsize=(10, 8))
sns.kdeplot(data=df_new, x='62', hue='Label', hue_order = hue_order, fill=True)
plt.title('Density of variable 62 with respect to labels')
plt.xlabel('Value')
plt.ylabel('Density')
plt.show()
```

```
plt.figure(figsize=(8, 6))
value = 20
sns.scatterplot(
    data=df,
    x='68',
    y='89',
    hue='Label',
    hue_order = hue_order,
    style='Label',
    markers=markers,
    s=value
    )
plt.title('Scatter Plot between variables 68 and 89')
plt.xlabel('Variable 68')
plt.ylabel('Variable 89')
plt.legend(title='Label')
plt.show()
```







The pairplot visualizes the pairwise relationships between the selected features (25, 40, 45, 62, 68, 79, 87, 89) for the different classes: glioma, meningioma, pituitary tumor, and notumor. Each subplot shows a scatter plot of one feature against another, colored by the tumor label, while the diagonal subplots show the distribution of each feature for the different labels.
There is significant overlap among the data points in the scatter plots, making them difficult to interpret. In particular, the red points representing the notumor class are largely obscured, with only one clearly visible. Despite this, we can observe that meningioma and glioma show considerable overlap in most feature combinations, indicating that these two types might be more challenging to distinguish from each other based on the selected features alone. This observation is consistent with the results obtained from our models, which also struggled to differentiate between these classes. Similarly, the distributions in the diagonal subplots also overlap considerably, complicating their interpretation. Overall, the analysis indicated poor separability among the classes, suggesting that no single feature or pair of features is sufficient to distinguish the tumor types effectively, necessitating the use of all features.

7. Preprocessing Deep Learning

Given the limited amount of data available for training our deep learning models, augmenting our dataset was essential to enhance the model's performance and accuracy. Various data augmentation techniques can be employed to generate additional training samples. In our approach, we applied thresholding, filtering, cropping, and geometric transformations to expand and diversify our dataset.

7.1 Libraries

These imports collectively set up the environment to perform data manipulation, image processing, and visualization.

```
import random
import imutils
import cv2
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from keras.preprocessing.image import ImageDataGenerator
```

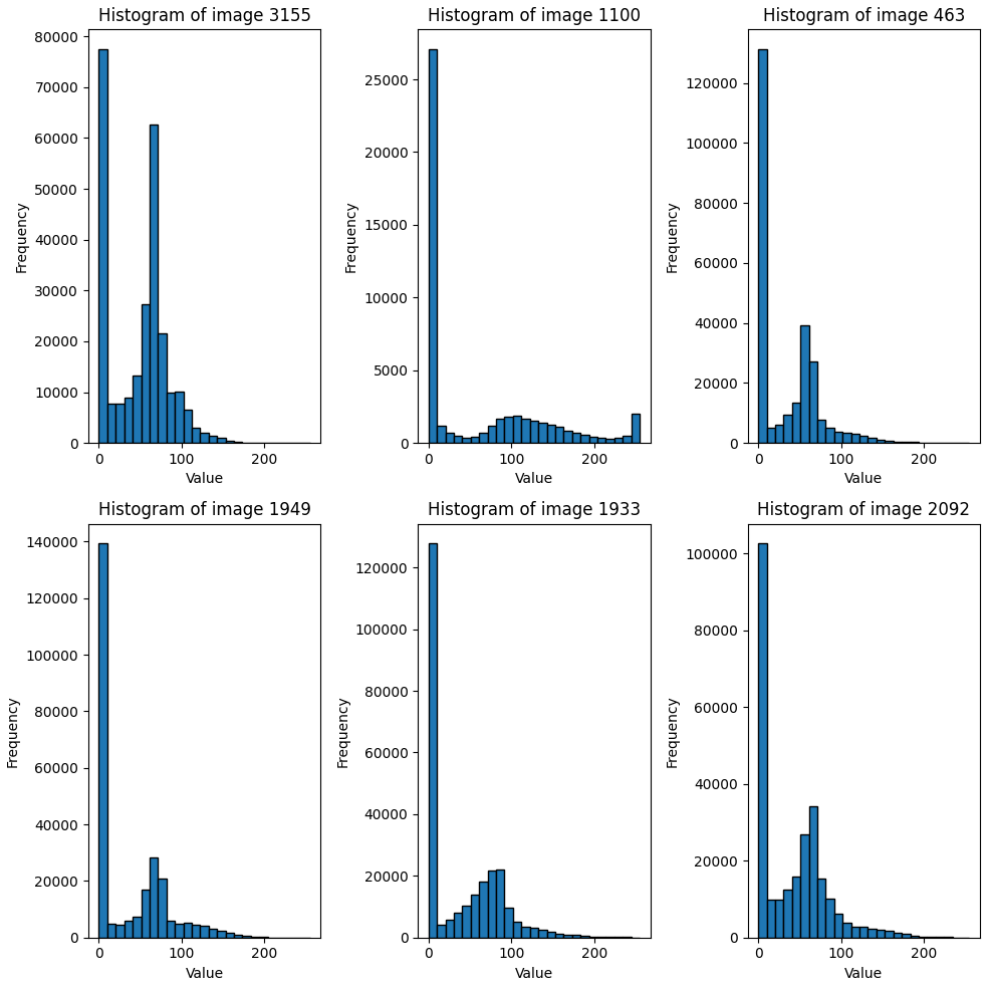
7.2 Data Analysis

First, we created the `df` object to represent the DataFrame contained in the pickle file named "complete_df.pkl".

```
from google.colab import drive
drive.mount('/content/drive')

df = pd.read_pickle("/content/drive/MyDrive/ProgettoDataMining/complete_df.pkl")
```

To analyze the distribution of pixel values in the images in our dataset, we generated histograms of six randomly selected images. We set up a subplot grid with three columns and two rows and created a 10x10 inch figure to host the histograms. This process allowed us to observe the distributional characteristics of the images in our dataset, providing an important preliminary understanding of the data before proceeding with further pre-processing and analysis.

```
n_example = 6
num_col = 3
num_row = n_example // num_col
plt.figure(figsize=(10, 10))

for i in range(n_example):
    random.seed(i)
    index = random.randint(0, len(df) - 1)
    image = df.at[index, "Image"].flatten()
    plt.subplot(num_row, num_col, i + 1)
    plt.hist(image, bins=25, edgecolor='black')
    plt.xlabel('Value')
    plt.ylabel('Frequency')
    plt.title(f'Histogram of image {index}')
plt.tight_layout()
plt.show()
```



From the histograms, we observed the presence of two peaks: one representing the background and the other representing the head. This distinction is crucial for our data pre-processing steps, as it indicates the separability of the foreground and background regions in the images.

7.3 Cropping

From the histograms, we observed two peaks: one indicating the background and the other representing the head. To segment these two parts effectively, we tested various thresholds to find the optimal separation between them.
To achieve this, we developed the `evaluate_thresholds` function. This function systematically tests and visualizes the effects of different threshold and blur kernel values on a given image. Threshold values are used to convert a grayscale image into a binary image by setting pixels below a certain value to 0 (black) and those above to 255 (white), thus separating the foreground from the background. Blur kernel values refer to the dimensions of the matrix used for blurring, a process that reduces noise and detail in an image by averaging pixel values within the kernel size, making certain features more distinguishable. The function creates a grid of subplots, each representing the image processed with a specific combination of threshold and blur values. It calculates the required number of rows and columns based on the ranges of thresholds and blur kernel dimensions provided. For each combination, it applies thresholding either directly or after blurring the image. The processed images are then displayed in a 10x10 inch figure with appropriate labels and titles for easy comparison.

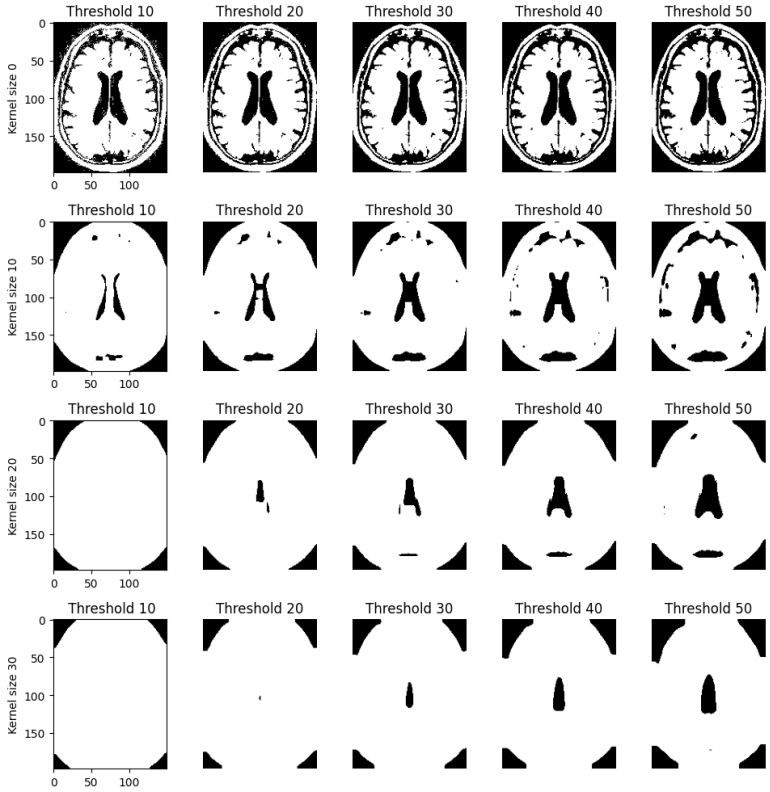
```
1 def evaluate_thresholds(image, range_thresholds, range_blur_kernel_dimension):
2
```

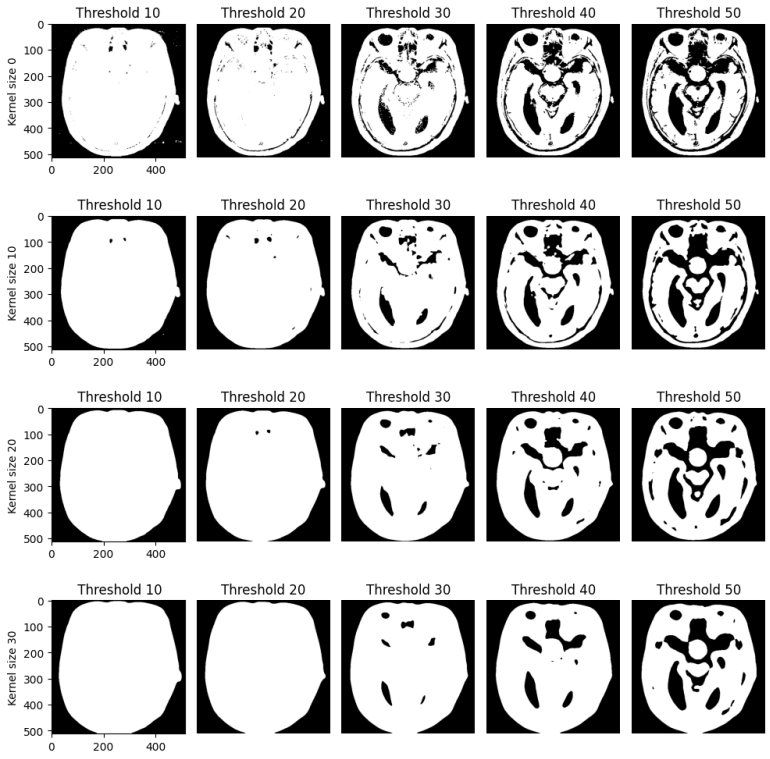
```
    num_rows = len(range_blur_kernel_dimension)
    num_cols = len(range_thresholds)
    plt.figure(figsize=(10, 10))

    for i, ksize in enumerate(range_blur_kernel_dimension):
        for j, threshold in enumerate(range_thresholds):
            plt.subplot(num_rows, num_cols, i * num_cols + j + 1)

            if j == 0:
                if ksize == 0:
                    new_image = (image > threshold).astype(np.uint8) * 255
                    plt.imshow(new_image, cmap="gray")
                    plt.title(f'Threshold {threshold}')
                else:
                    blurred_image = cv2.blur(image, (ksize, ksize))
                    thresholded_image = (blurred_image > threshold).astype(np.
uint8) * 255
                    plt.imshow(thresholded_image, cmap="gray")
                    plt.title(f' Threshold {threshold}')
                plt.ylabel(f'Kernel size {ksize}', fontsize=10)
            else:
                if ksize == 0:
                    new_image = (image > threshold).astype(np.uint8) * 255
                    plt.imshow(new_image, cmap="gray")
                    plt.title(f'Threshold {threshold}')
                else:
                    blurred_image = cv2.blur(image, (ksize, ksize))
                    thresholded_image = (blurred_image > threshold).astype(np.
uint8) * 255
                    plt.imshow(thresholded_image, cmap="gray")
                    plt.title(f'Threshold {threshold}')
                plt.axis('off')

    plt.tight_layout()
    plt.show()
```

To effectively evaluate different threshold and blur kernel values, we selected two random images from `df`, specifically choosing the images at index 100 and 5. We defined a range of threshold values ([10, 20, 30, 40, 50]) and a range of blur kernel dimensions ([0, 10, 20, 30]), and utilized the `evaluate_thresholds` function to systematically test these parameters on the selected images. This evaluation process helps us determine the optimal threshold and blur values that best enhance the image quality for subsequent analysis and effective cropping.

```
image = df.at[100,"Image"]
range_thresholds = [10, 20, 30, 40, 50]
range_blur_kernel_dimension = [0, 10, 20, 30]
evaluate_thresholds(image, range_thresholds, range_blur_kernel_dimension)
```

```
image = df.at[5,"Image"]
range_thresholds = [10, 20, 30, 40, 50]
range_blur_kernel_dimension = [0, 10, 20, 30]
evaluate_thresholds(image, range_thresholds, range_blur_kernel_dimension)
```



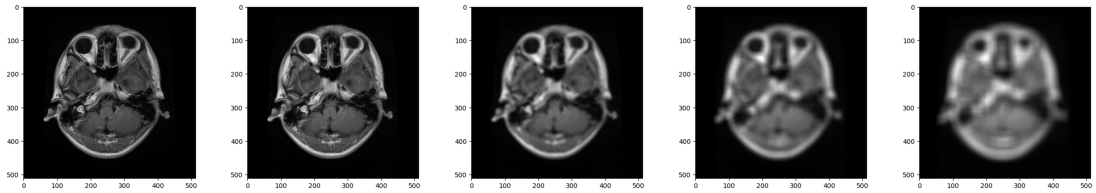


Observing the results, we concluded that the best values are a threshold of 20 and a kernel size of 30x30.

To show how varying degrees of blurring impact the clarity and details of the image, we tested a range of blur kernel sizes: [1, 3, 10, 20, 30]. For each kernel size, we applied a blur effect to

the image at index 2 in our dataset. The blurred images were then displayed using a grayscale color map for visual assessment.

```
ksizes =  [1, 3, 10, 20, 30]
for ksize  in  ksizes:
  image = df.at[2, "Image"].copy()
  image = cv2.blur(image, (ksize, ksize))
  plt.imshow(image, cmap = "gray")
  plt.show()
```



Having determined that the best values to create a clear separation between the foreground and background are a threshold of 20 and a kernel size of 30x30, we can accurately crop the images. By using these optimal values, we can focus only on the relevant parts and effectively remove unnecessary background information.

To isolate the region of interest in our images, we developed a function called `crop_image`. This function first applies a blur effect with a kernel size of 30x30 to reduce noise and detail. It then converts the blurred image into a binary image using a threshold value of 20, which helps to distinguish the foreground from the background. Next, the function checks if the image is of type uint8 and converts it if necessary. It then finds the contours that envelop the zero values in the binary image. From the detected contours, it selects the one with the largest area, which presumably corresponds to the primary object of interest. The function extracts the coordinates of the contour points and calculates the minimum and maximum x and y coordinates to determine the bounding box for cropping. It then crops the image using these coordinates, effectively isolating the region of interest while discarding unnecessary background information.

```
def crop_image(image):

    blurred_image = cv2.blur(image, (30, 30))
    thresholded_image = (blurred_image > 20).astype(np.uint8) * 255

    if image.dtype != np.uint8:
        image = image.astype(np.uint8)

    contours = cv2.findContours(image.copy(), cv2.RETR_EXTERNAL, cv2.
    CHAIN_APPROX_SIMPLE)
    cnts = imutils.grab_contours(contours)

    if cnts:
        c = max(cnts, key=cv2.contourArea)
        tot_point = []
        for i in range(len(c)):
            x = c[i, 0, 0]
            y = c[i, 0, 1]
            point = [x, y]
            tot_point.append(point)

        matrix_points = np.array(tot_point)
        x_max = np.max(matrix_points[:, 0])
        x_min = np.min(matrix_points[:, 0])
        y_max = np.max(matrix_points[:, 1])
        y_min = np.min(matrix_points[:, 1])

        print("X range", x_min, x_max)
        print("y range", y_min, y_max)

        cropped_image = image[x_min:x_max, y_min:y_max]
        print("Before", image.shape, "After", cropped_image.shape, "\n" * 6)
```

```
    else:
        print("No contours found in the image..")
        return image

    return cropped_image
```

To add the cropped images to our DataFrame, we introduced a new column, `cropped_image`, and initialized it with `None`. We then iterated through each row of the DataFrame to apply the `crop_image` function to the images. For each image, the function retrieved the image from the DataFrame, applied the cropping process, and stored the resulting cropped image back into the DataFrame. During the iteration, we printed the index of the current image and the shape of the cropped image to monitor the process. This ensured that each image was processed correctly and that the dimensions of the cropped images were as expected.

```
df["cropped_image"] = None

for i in range(len(df)):
    print(i)
    image = df.at[i, "Image"]
    cropped = crop_image(image)
    print(cropped.shape)

    df.at[i, "cropped_image"] = cropped
```

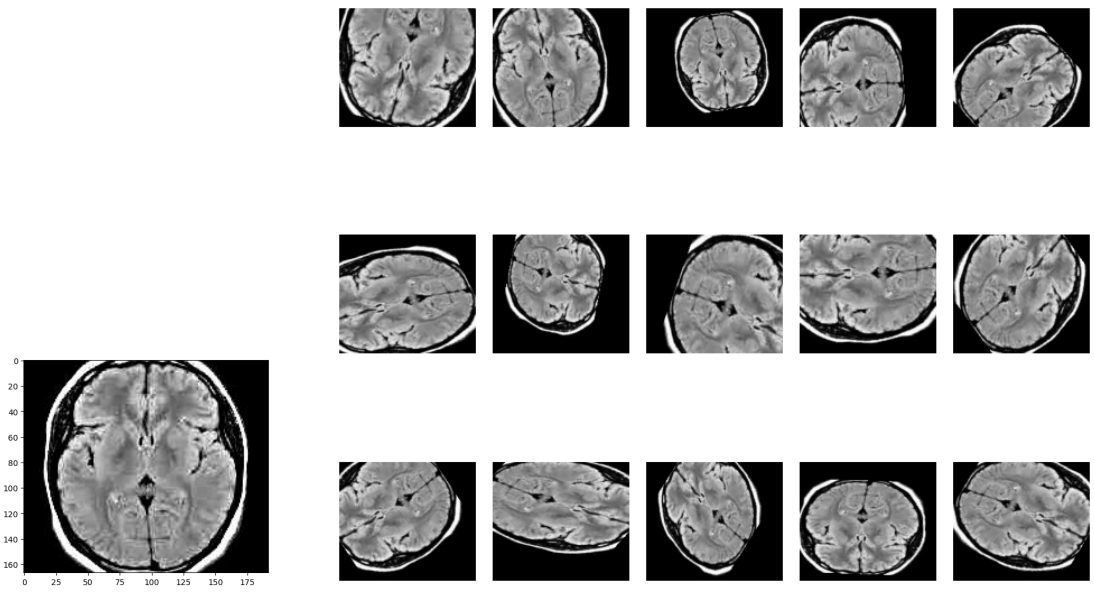
7.4 Geometric transformation

At this point, our DataFrame `df` contained a column with all the cropped images. We aimed to create a new balanced DataFrame, ensuring that it contained the same number of cropped images for each label.
To accomplish this, we developed a function named `geometric_transformation`. This function applies a series of transformations including rotation, shifting, shearing, zooming, and flipping to generate augmented versions of the input images. By specifying a random seed, we ensure reproducibility of the transformations. The `ImageDataGenerator` class from the `Keras` library is used to perform these operations, with parameters set to rotate the images up to 90 degrees, shift them horizontally and vertically by 10%, shear by 40%, and zoom in/out by 30%. Additionally, the images are randomly flipped both vertically and horizontally. These transformations help to increase the variability of the dataset, improving the robustness and performance of our deep learning models by exposing them to a wider range of image variations.

```
def geometric_transformation(image, random_seed=14):
    size = image.shape
    datagen = ImageDataGenerator(
        rotation_range=90,
        width_shift_range=0.1,
        height_shift_range=0.1,
        shear_range=0.4,
        zoom_range=0.3,
        vertical_flip=True,
        horizontal_flip=True,
        fill_mode='constant',
        cval=0
    )
    tensor = image.reshape(size[0], size[1], 1)
    new_tensor = datagen.random_transform(tensor, seed=random_seed)
    new_image = new_tensor[:, :, 0]
    return new_image
```

To visualize the effects of the geometric transformations implemented by the function, we selected a cropped image from the DataFrame, specifically the image at index 47, and displayed it in grayscale. By applying the `geometric_transformation` function, we generated a grid showcasing multiple transformed versions of the original image. This visualization provides a comprehensive examination of how the specific geometric transformations, such as rotation, shifting, shearing, zooming, and flipping, impact the original image.

```
image = df.at[47, "cropped_image"]
plt.imshow(image, cmap="gray")
plt.show()

num_row = 3
num_col = 5
num_example = num_row * num_col
plt.figure(figsize=(7, 7))
for i in range(num_row):
    for j in range(num_col):
        index = i * num_col + j + 1
        random_seed = index
        new_image = geometric_transformation(image, random_seed=random_seed)
        plt.subplot(num_row, num_col, index)
        plt.imshow(new_image, cmap="gray")
        plt.axis('off')
plt.tight_layout()
plt.show()
```



After visualizing the effects of the geometric transformations on a single image, we proceeded to create a balanced DataFrame of cropped images. This step ensured that each label was equally represented, thereby enhancing the quality and fairness of our dataset.
First, we extracted and organized the cropped images from the DataFrame based on their labels, storing these categorized images into separate DataFrames within a dictionary.

```
df["Label"].unique()
dict = {}

for label in df["Label"].unique():
    subselected = df[df["Label"] == label]["cropped_image"]
    subselected_df = subselected.to_frame().reset_index(drop=True)
    dict[label] = subselected_df
```

Next, we calculated the number of observations needed to balance each subset of cropped images in the dictionary, ensuring all categories had the same number of samples. Initially, the code iterated through each label in the dictionary, calculating and printing the number of observations (cropped images) for each label. These counts were stored in a list. After determining the maximum length among all subsets, the code calculated how many additional observations were needed for each label to reach this maximum length. This information was stored in a new dictionary, where the keys were the labels and the values were the required number of additional samples.

```
length = []
keys = dict.keys()
for key in keys:
    l = len(dict[key])
    print("for the key", key, "the number of observations are:", l)
    length.append(l)

print("\n" * 10)

max_len = max(length)
dict_number_sample = {}
for key in keys:
    l = len(dict[key])
    observation_to_sample = max_len - l
    dict_number_sample[key] = observation_to_sample

dict_number_sample
```

```
for the key glioma the number of observations are: 1426
for the key notumor the number of observations are: 1500
for the key pituitary tumor the number of observations are: 930
for the key meningioma the number of observations are: 708


{'glioma': 74, 'notumor': 0, 'pituitary tumor': 570, 'meningioma': 792}
```

After calculating the number of observations needed to balance each subset of cropped images, we generated additional samples for each category to balance the dataset. We initialized an empty dictionary, `generated`, to store the newly created DataFrames. For each key (label) in the `dict_number_sample` dictionary, the corresponding DataFrame is retrieved, and an empty list, container, is initialized to store the generated images. Using a loop, the required number of samples is created by randomly selecting an image from the existing DataFrame and applying the `geometric_transformation` function with a seed for reproducibility. The transformed images are then appended to the container list. Finally, the list is converted into a DataFrame, which is stored in the generated dictionary under the respective label.

```
generated = {}

for key in dict_number_sample.keys():
    mini_df = dict[key]
    container = []
    for i in range(dict_number_sample[key]):
        random.seed(i)
        index = random.randint(0, len(mini_df) - 1)
        image = mini_df.at[index, 'cropped_image']
        new_image = geometric_transformation(image, random_seed=i)
        container.append([new_image])
    generated[key] = pd.DataFrame(container, columns=['cropped_image'])
```

Finally, we combined the original and newly generated samples into a single DataFrame, ensuring a balanced and shuffled dataset. An empty DataFrame, `df_transformation`, was created to store the combined results. For each label in the `generated` dictionary, the corresponding DataFrames from both the original and generated dictionaries were retrieved. These DataFrames were concatenated, and their indices were reset to ensure sequential ordering. A new column, "Label," was added to the concatenated DataFrame to denote the label of the samples. The resulting DataFrame for each label was then appended to `df_transformation`. To eliminate any order bias, the entire DataFrame was randomly shuffled, and the index was reset again. Finally, the counts of each label within `df_transformation` were outputted to verify the balance of the dataset.

```
df_transformation = pd.DataFrame()

for key in generated.keys():
    df_old = dict[key]
    df_new = generated[key]
    df_class = pd.concat([df_old, df_new]).reset_index(drop=True)
    print(len(df_class))
```

```
    df_class["Label"] = key
    df_transformation = pd.concat([df_transformation, df_class]).reset_index(drop
    =True)

df_transformation = df_transformation.sample(frac=1).reset_index(drop=True).
    reset_index(drop=True)

df_transformation["Label"].value_counts()
```

```
Label
glioma             1500
pituitary tumor    1500
notumor            1500
meningioma         1500
Name: count, dtype: int64
```

This process resulted in a comprehensive and balanced DataFrame that includes both the original and augmented samples. Each label of `df_transformation` now contains 1500 observations, ensuring a well-rounded and robust dataset.

8. Convolutional Neural Network (CNN)

We continued our analysis by experimenting Convolutional Neural Networks (CNNs) for classifying tumor images. Using CNNs is highly advantageous due to several key factors:

- `Spatial Hierarchies of Features` → CNNs exploit the spatial structure of images by using local connections (filters/kernels) to detect features such as edges, textures, and patterns. These local features are crucial for identifying tumor characteristics.
- `Hierarchical Feature Learning` → CNNs learn a hierarchy of features, from low-level features (e.g., edges and textures) in the initial layers to high-level features (e.g., shapes and objects) in the deeper layers. This hierarchical learning is beneficial for accurately detecting and classifying tumors, which may have complex patterns.
- `Parameter Sharing Convolutional` → Convolutional layers use the same set of weights (filters) across different parts of the image. This parameter sharing reduces the number of parameters, making the model more efficient and less prone to overfitting, especially important in medical imaging where datasets can be limited.
- `Translation Invariance` → CNNs are inherently robust to translations and distortions. This means that a tumor can be detected regardless of its position within the image, which is vital for medical images where tumors can appear in various locations and orientations.
- `Handling High-Dimensional Data` → Pooling layers in CNNs progressively reduce the spatial dimensions of the data while preserving important features. This helps in managing high-dimensional image data effectively, ensuring that the model focuses on the most salient features of the tumors.
- `Data Augmentation to improve generalization` → CNNs work well with data augmentation techniques (such as rotation, flipping, scaling), which can simulate the variability in medical images. This improves the model's ability to generalize to new, unseen images.
- `State-of-the-Art Performance` → CNNs have consistently demonstrated superior performance in various image classification tasks, including medical imaging. They have been shown to achieve high accuracy in detecting and classifying tumors, sometimes even surpassing human experts in certain cases.

8.1 ResNet50

After experimenting with different architectures and parameters, we opted for the ResNet50 model, which stood out in terms of performance and efficiency.
ResNet50, or Residual Network with 50 layers, is a deep convolutional neural network part of the ResNet family of models introduced by Kaiming He and colleagues in their 2015 paper, "Deep Residual Learning for Image Recognition". This model is widely used for various computer vision tasks, including image classification (recognizing and categorizing objects within images), object detection (identifying and locating objects within images), and image segmentation (dividing an image into different segments or regions). The core innovation of ResNet50 is the introduction of residual learning, achieved through skip connections (also known as shortcut connections) that allow the network to bypass one or more layers. These connections help mitigate the vanishing gradient problem, making it easier to train very deep networks. ResNet50's

architecture comprises 50 layers, including convolutional, pooling, batch normalization, and fully connected layers. The network is composed of multiple residual blocks, each containing several convolutional layers with identity mapping. The bottleneck design used in these residual blocks involves a stack of three layers: 1x1, 3x3, and 1x1 convolutions. The 1x1 convolutions reduce and then restore the number of channels, enhancing computational efficiency. This architecture includes a 7x7 convolution with 64 filters and a stride of 2 in the Conv1 layer, followed by a max-pooling layer. It then has 3 bottleneck blocks in Conv2_x, 4 bottleneck blocks in Conv3_x, 6 bottleneck blocks in Conv4_x, and 3 bottleneck blocks in Conv5_x. This is followed by an average pooling layer, a fully connected layer, and a softmax layer for classification. ResNet50 is recognized for its strong performance on benchmarks like ImageNet, achieving high accuracy while being computationally efficient compared to other deep networks with a similar number of layers. This efficiency and powerful architecture make ResNet50 suitable for a wide range of computer vision applications.

Below is a comprehensive explanation of each part of the code to implement our custom ResNet50, including the theoretical framework and the rationale behind using this model structure.

To adapt the ResNet50 model for our custom tumor classification task, we implemented several modifications to the pre-trained architecture. We started by loading the pre-trained ResNet50 model with weights from the ImageNet dataset. The `include_top=False` parameter was specified to exclude the fully connected layer at the top of the network, allowing us to append our own custom layers. The `input_shape` was set to `(image_size, image_size, 3)`, indicating the model expects RGB images of the specified size. After loading the base ResNet50 model, we modify its output by adding several custom layers. First, we added a `GlobalAveragePooling2D` layer to reduce the spatial dimensions of the feature maps, resulting in a single vector for each feature map. Next, we included a `Dropout` layer with a 0.4 dropout rate to mitigate overfitting by randomly setting 40% of the input units to zero during training. We added a `Dense` layer with softmax activation to output the probab The final model was constructed by specifying the inputs and outputs, linking the input of the base ResNet50 model to the output of our custom layers, effectively tailoring the ResNet50 architecture for our specific classification needs.

```
net = ResNet50(weights='imagenet', include_top=False, input_shape=(image_size,
    image_size, 3))
model = net.output
model = GlobalAveragePooling2D()(model)
model = Dropout(0.4)(model)
model = Dense(4, activation="softmax")(model)
model = Model(inputs=net.input, outputs=model)
```

To prepare the customized ResNet50 model for training, we compiled the model using the Adam optimizer with a learning rate of 0.0001. This optimizer is known for its efficiency and adaptive learning rate capabilities. We set the loss function to categorical cross-entropy, which is suitable for multi-class classification tasks, and included accuracy as a metric to evaluate the model's performance during training. The model summary was then printed to provide an overview of the architecture and the total number of parameters to be trained.

```
adam = tf.keras.optimizers.Adam(learning_rate=0.0001)
model.compile(optimizer=adam, loss='categorical_crossentropy', metrics=['accuracy
    '])
model.summary()
```

To enhance the training process of our custom ResNet50 model, we set up several callbacks. These callbacks are essential for monitoring the model's performance, saving the best model, and adjusting the learning rate dynamically. First, we initialized a `TensorBoard` callback to log the training metrics, enabling us to visualize them using TensorBoard. The `ModelCheckpoint` callback was configured to save the model's weights whenever there was an improvement in validation loss, ensuring that the best model based on validation performance was saved. The `EarlyStopping` callback was set to monitor the validation loss and stop training if there was no improvement for five consecutive epochs, restoring the best weights observed during training. Finally, the `ReduceLROnPlateau` callback was included to reduce the learning rate by a factor of 0.3 if the validation loss did not improve for five epochs, facilitating better convergence during training. All these callbacks were combined into a list for easy implementation during model

training.

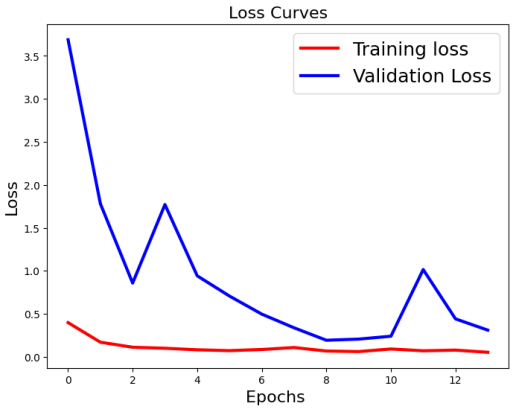
```
logdir = "logs"
tensorboard = TensorBoard(log_dir=logdir, histogram_freq=1)
checkpoint = ModelCheckpoint(filepath='model-{epoch:02d}-{val_accuracy:.2f}-{
    val_loss:.2f}.h5', monitor='val_loss', verbose=1, save_best_only=True, mode='
    min')
es = EarlyStopping(monitor='val_loss', min_delta=0.001, patience=5, mode='min',
    restore_best_weights=True, verbose=1)
rl = ReduceLROnPlateau(monitor='val_loss', factor=0.3, patience=5, verbose=1,
    mode='min')

callbacks = [es, rl, tensorboard, checkpoint]
```

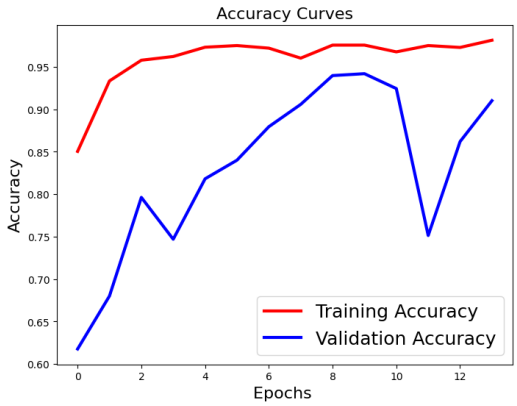
To augment the training data and enhance the model's generalization capability, we used the `ImageDataGenerator` from `Keras`. This generator performs data augmentation by applying random transformations such as horizontal and vertical flipping and rotating the images by up to 90 degrees. These transformations help simulate the variability in the dataset, making the model more robust to different image variations. The augmented data was then fit to the training data using the `datagen.fit(x_train)` method. We also defined the batch size and the number of epochs for training. The batch size was set to 64, meaning that the model would be updated after every 64 samples. The number of epochs was set to 50, indicating that the model would be trained for 50 complete passes over the entire training dataset. The model was trained using the `fit` method, which takes the augmented training data generated by `datagen.flow(x_train, y_train, batch_size=BATCH_SIZE)` and validates the performance on a separate validation set (`x_val, y_val`). The training process uses the defined callbacks to monitor and adjust training as needed.

```
datagen = ImageDataGenerator(horizontal_flip=True, vertical_flip=True,
    rotation_range=90)
datagen.fit(x_train)

BATCH_SIZE = 64
EPOCHS = 50

history = model.fit(
    datagen.flow(x_train, y_train, batch_size=BATCH_SIZE),
    validation_data=(x_val, y_val),
    epochs=EPOCHS,
    callbacks=callbacks
)
```

To evaluate the training process, we plotted the loss and accuracy curves using `Matplotlib`. The first plot visualizes the training and validation loss over the epochs. The training loss curve is plotted in red, while the validation loss curve is in blue. The plot is labeled appropriately with legends, axis labels, and a title to clearly show how the loss values change during the training process. The second plot illustrates the training and validation accuracy, following a similar format with the training accuracy curve in red and the validation accuracy curve in blue. These visualizations help in understanding the model's performance over time, showing trends in how well the model is learning and generalizing to unseen validation data.

```
plt.figure(figsize=[8,6])
plt.plot(history.history['loss'],'r',linewidth=3.0)
plt.plot(history.history['val_loss'],'b',linewidth=3.0)
plt.legend(['Training loss', 'Validation Loss'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Loss',fontsize=16)
plt.title('Loss Curves',fontsize=16)
plt.show()

plt.figure(figsize=[8,6])
plt.plot(history.history['accuracy'],'r',linewidth=3.0)
plt.plot(history.history['val_accuracy'],'b',linewidth=3.0)
plt.legend(['Training Accuracy', 'Validation Accuracy'],fontsize=18)
plt.xlabel('Epochs ',fontsize=16)
plt.ylabel('Accuracy',fontsize=16)
plt.title('Accuracy Curves',fontsize=16)
plt.show()
```

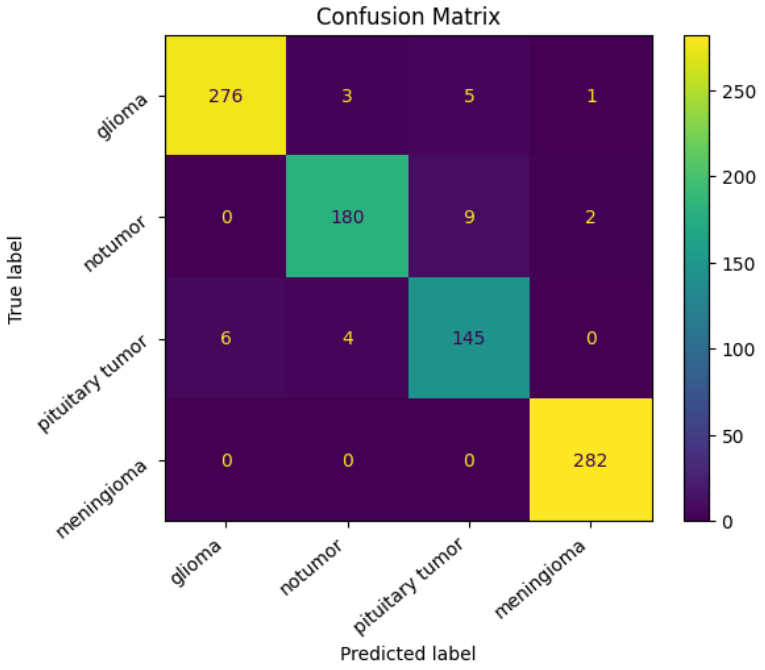




Overall, the training and validation curves provide a comprehensive view of the model's learning dynamics. The first plot shows that the training loss steadily decreased, indicating that the model was learning effectively over time. The validation loss, while generally decreasing, exhibited some fluctuations, which can be attributed to the inherent variability in the validation data. The second plot shows that the training accuracy increased significantly, demonstrating that the model was improving its performance on the training dataset. The validation accuracy also showed a general upward trend, suggesting that the model was able to generalize to unseen data, though with some variability. We observed a discrepancy between the training and validation sets in both accuracy and loss, approximately around 5%, likely due to a drop in the model fit during the last few epochs. This drop could be caused by several factors, including the learning rate being too high initially, leading to instability in the optimization process, and the shuffling of training data resulting in batches that are harder to learn from. Additionally, regularization techniques like dropout might cause temporary drops in accuracy but ultimately lead to better generalization. The optimization algorithm might experience fluctuations as it navigates the loss landscape, especially if the landscape has many local minima or saddle points. Furthermore, the model might be adjusting to different features or patterns in the data, causing temporary drops before improving again. Despite these fluctuations, the overall trends in the loss and accuracy curves indicate that the model is learning and generalizing well.
After training and testing the model, predictions were made. We converted the predicted probabilities to class labels by taking the index of the maximum value in each prediction. Similarly, we converted the actual validation labels to class labels. By mapping these numeric classes to the original labels, we ensured that the evaluation metrics accurately reflected the true classes. To evaluate the performance of our classification model, we utilized the `evaluate_metrics` function with the predicted and actual labels, along with the original label names.

```
original_labels = resizer.new_df['Label'].unique()

y_pred = model.predict(x_val)
y_pred_classes = np.argmax(y_pred, axis=1)
y_val_classes = np.argmax(y_val, axis=1)

y_pred_labels = [original_labels[i] for i in y_pred_classes]
y_val_labels = [original_labels[i] for i in y_val_classes]

evaluate_metrics(y_pred_labels, y_val_labels, original_labels)
```

```
                  precision    recall  f1-score   support

         glioma       0.98      0.97      0.97       285
        notumor       0.96      0.94      0.95       191
pituitary tumor       0.91      0.94      0.92       155
     meningioma       0.99      1.00      0.99       282

       accuracy                           0.97       913
      macro avg       0.96      0.96      0.96       913
   weighted avg       0.97      0.97      0.97       913
```



The classification results indicate that the model achieved an overall accuracy of 97%. The precision, recall, and F1-scores for the individual classes demonstrate strong performance: glioma cases were identified with a precision of 0.98 and recall of 0.97, notumor cases with a precision of 0.96 and recall of 0.94, pituitary tumors with a precision of 0.91 and recall of 0.94, and meningioma cases with a near-perfect precision of 0.99 and recall of 1.00. The macro average and weighted average scores for precision, recall, and F1-score, all at 0.96 or higher, suggest the model's consistent ability to accurately classify different tumor types. These results highlight the model's robustness and reliability in distinguishing between the four classes of tumors.
The confusion matrix allows us to evaluate the model's accuracy by comparing the predicted labels against the true labels. We observed an extremely accurate fit of the model, with no particular pattern of errors, unlike those found in the Visual Transformer. Specifically, for glioma, the model correctly predicted 276 instances, misclassifying 3 notumors, 5 pituitary tumors, and only 1 meningioma. For notumor, the model accurately predicted 180 instances, misclassifying 9 pituitary tumors and 2 meningiomas. For pituitary tumor, the model correctly predicted 145 instances, but misclassified 6 gliomas and 4 notumors. For meningioma, the model achieved perfect classification accuracy with all 282 instances correctly identified. These results confirm the high precision, recall, and F1-scores from the classification report, illustrating the model's robust performance in distinguishing between the four tumor types.

8.2 VGG16

Trying improving accuracy, we utilized the VGG16 model. VGG16 is a highly regarded Convolutional Neural Network (CNN) model in the field of computer vision, known for its robust performance and ease of use in transfer learning. Developed by the Visual Geometry Group at the University of Oxford, VGG16 improved upon previous models by employing an architecture with very small (3×3) convolution filters, which significantly enhanced accuracy. The model's architecture was deepened to 16–19 weight layers, resulting in approximately 138 million trainable parameters. VGG16 is capable of object detection and classification, achieving an impressive 92.7% accuracy on the ImageNet dataset, which consists of 1000 categories. The "16" in VGG16 denotes the number of layers with learnable weights. Specifically, the architecture includes thirteen convolutional layers, five max pooling layers, and three dense layers, summing to 21 layers in total, but only 16 layers with weights. The input tensor size for VGG16 is 224 x 224 pixels with 3 RGB channels. A unique aspect of VGG16 is its consistent use of 3x3 convolution

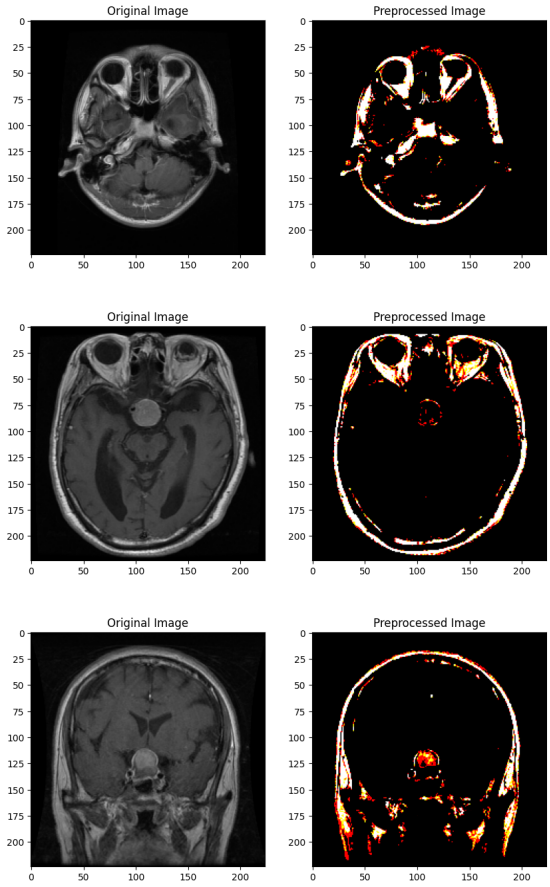
filters with a stride of 1 and the same padding, alongside 2x2 max pooling filters with a stride of 2. This consistent arrangement is maintained throughout the network. The convolutional layers are structured as follows: the Conv-1 layer has 64 filters, Conv-2 has 128 filters, Conv-3 has 256 filters, and both Conv-4 and Conv-5 have 512 filters. Following the convolutional layers are three Fully-Connected (FC) layers: the first two FC layers each have 4096 channels, while the third FC layer performs 1000-way classification for the ILSVRC competition, containing 1000 channels (one for each class). The final layer is a softmax layer, which provides the output probabilities for each class. This architecture makes VGG16 a powerful model for various image classification tasks.

Before creating our VGG16 model, it was necessary to preprocess the images in the dataset to ensure they were suitable for input. To achieve this, we followed several essential steps. Initially, we defined a target size of (224, 224) pixels, which is the standard requirement for the VGG16 model. We then added a new column to the DataFrame named `Preprocessed_vgg16` to store the preprocessed images. In a loop iterating over all the rows in the DataFrame, each grayscale image was loaded and converted into a `numpy` array of type `uint8`. Subsequently, the images were resized to the target size using the `cv2.resize` function. Since the VGG16 model requires RGB images as input, the resized images were converted from grayscale to RGB with `cv2.cvtColor`. Finally, the RGB images were further preprocessed using the `preprocess_input` function from the `keras.applications.vgg16` library. This step is crucial for normalizing the pixel values of the images into the format expected by the VGG16 model. The preprocessed images were then stored in the new DataFrame column, while the original images were updated with the RGB version. This preprocessing phase is essential to ensure that the images are in the correct format and appropriately normalized, thus facilitating the subsequent training and inference process with the VGG16 model.

```
size = (224, 224)
df["Preprocessed_vgg16"] = None

for i in range(len(df)):
    gray_image = df.at[i, "Image"].astype(np.uint8)
    resized_image = cv2.resize(gray_image, size)
    rgb_image = cv2.cvtColor(resized_image, cv2.COLOR_GRAY2RGB)
    preprocessed_image = preprocess_input(rgb_image)
    df.at[i, "Image"] = rgb_image
    df.at[i, "Preprocessed_vgg16"] = preprocessed_image
```

To visually confirm the effectiveness of these preprocessing steps, we selected three sample images from the dataset (indices 2, 5, and 48) and displayed both their original and preprocessed versions side by side. The original images retain their grayscale format, while the preprocessed images are converted to RGB and normalized, ready for feature extraction using VGG16.

```
images = [2, 5, 48]

for i in images:
    plt.figure(figsize=(10, 5))

    plt.subplot(1, 2, 1)
    plt.imshow(df.at[i, "Image"])
    plt.title("Original Image")

    plt.subplot(1, 2, 2)
    plt.imshow(df.at[i, "Preprocessed_vgg16"])
    plt.title("Preprocessed Image")

    plt.show()
```

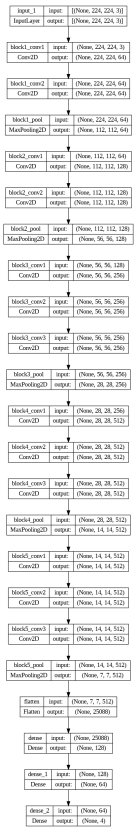


Having preprocessed the images, we proceeded to create a VGG16 model based on the pre-trained ImageNet dataset. The VGG16 model was loaded with its top fully connected layers included and an input shape specified as 224x224 pixels with 3 color channels.

We then modified the model by taking the output from the fourth-to-last layer and adding custom dense layers to tailor the model to our specific classification problem. Specifically, we added a fully connected layer with 128 units and ReLU activation, followed by another fully connected layer with 64 units and ReLU activation. The final output layer was configured with 4 units and softmax activation to classify our images into four categories.

To visualize the architecture of our modified VGG16 model, we utilized the `plot_model` function from `Keras`. This function generated a graphical representation of the model, including both the original VGG16 layers and the custom layers we added. The plot, saved as `model_plot.png`, displays the structure of the network, showing the connections between layers, the shapes of the tensors flowing through the network, and the names of each layer. This visual summary helped in understanding the model's architecture and ensured that our custom modifications integrated seamlessly with the pre-trained VGG16 model.

```
vgg_model = VGG16(weights='imagenet', include_top=True, input_shape=(224, 224, 3)
    )
vgg_model.summary()

x = vgg_model.layers[-4].output
x = Dense(128, activation='relu')(x)
x = Dense(64, activation='relu')(x)
output_layer = Dense(4, activation='softmax')(x)

model = Model(inputs=vgg_model.input, outputs=output_layer)
model.summary()

plot_model(model, to_file='model_plot.png', show_shapes=True, show_layer_names=
    True)
```



We further refined our VGG16 model by freezing all the layers except for the last three custom layers. This step ensures that the pre-trained weights in the original VGG16 layers remain unchanged during training, allowing only the newly added dense layers to be updated. By iterating through the model layers and setting the `trainable` attribute to `False` for all layers except the last three, we effectively limited the training to our custom layers.

```
for layer in model.layers[:-3]:
    layer.trainable = False

model.summary()
```

To prepare our image data for training the VGG16 model, we first iterated through the `Preprocessed_vgg16` column in the DataFrame. For each preprocessed image, we reshaped it to the required input dimensions of (224, 224, 3) and appended it to a list, which we then converted into a NumPy array `X`. This array, representing our input data, was cast to the data type `float16` to save memory. We then extracted the labels from the DataFrame and reshaped them to ensure they were in the correct format for encoding. To handle the categorical nature of the labels, we identified the unique labels and used them to fit a `OneHotEncoder` from `sklearn`, transforming the categorical labels into a binary matrix format. This transformation allowed us to convert the original labels into a one-hot encoded format stored in `y`. Finally, we split the dataset into training and testing sets, with 80% of the data used for training and 20% for testing. This split was performed using `train_test_split` with a fixed random state of 42 to ensure reproducibility. The resulting `X_train`, `X_test`, `y_train`, and `y_test` arrays were prepared for subsequent training and evaluation of the VGG16 model.

```
X = []
for i,x in enumerate(df["Preprocessed_vgg16"].values):
    print(i)
    new = x.reshape(1, 224, 224, 3)
    X.append(new)
X = np.vstack(X)
X = X.astype(np.float16)
print(X.shape)
```

```
y = np.array(df['Label']).reshape(-1, 1)
example = df["Label"].unique()
example = example.reshape(-1,1)
print(example,"\n"*5)
one_hot_encoder = OneHotEncoder(sparse=False)
one_hot_encoder.fit(example)
print(one_hot_encoder.transform(example))
y = one_hot_encoder.transform(y)
y.shape

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
y_train.shape
```

After freeing up memory space by deleting the variables `X` and `df`, we compiled the model using the Adam optimizer with a learning rate of 0.001 and the categorical crossentropy loss function. The Adam optimizer was chosen for its ability to adaptively adjust the learning rate for each parameter, combining the advantages of both AdaGrad and RMSProp to achieve efficient convergence. The categorical crossentropy loss function is suitable for our multi-class classification problem, guiding the model to improve its predictions by minimizing the difference between the true and predicted labels. Accuracy was specified as a performance metric to track the model's classification accuracy during training. Subsequently, we trained the model using the `fit` method, specifying the training data (`X_train` and `y_train`), running the training process for 10 epochs with a batch size of 32. To monitor the model's performance on unseen data during training, we reserved 20% of the training data for validation. The verbose level was set to 2 to provide detailed output for each epoch, allowing us to observe the model's progress and performance improvements over time.

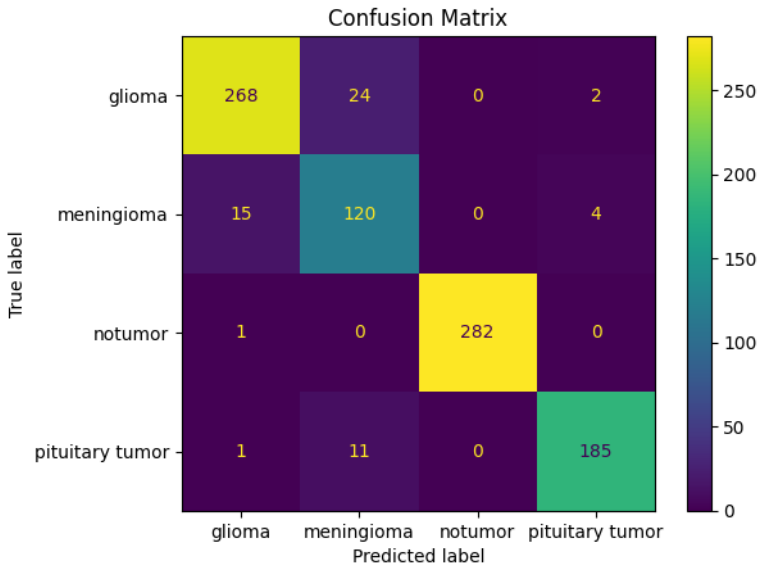
```
del X
del df

model.compile(optimizer=Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(X_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_split=0.2,
                    verbose=2)
```

Following the compiling and training of the VGG16 model, we proceeded to evaluate its performance on the test set. The function `transform_prediction` was defined to convert the model's probability predictions into one-hot encoded vectors by finding the index of the maximum predicted probability and setting this index to 1 while others to 0. This function then uses the `inverse_transform` method of the one-hot encoder to revert these vectors back to their original labels. We applied this function to the test set predictions (`X_test`) to obtain the predicted labels. The true labels (`y_test`) were also inverse transformed to their original labels for comparison. The transformed and flattened predictions and true labels were then passed to the `evaluate_metrics` function to assess the model's performance. This evaluation provided insights into the model's accuracy and effectiveness in classifying the test images into their respective categories.

```
def transform_prediction(preds):
    transformed_preds = []
    for pred in preds:
        max_index = np.argmax(pred)
        one_hot = np.zeros_like(pred)
        one_hot[max_index] = 1
        transformed_preds.append(one_hot)
    trasf= np.array(transformed_preds)
    result = one_hot_encoder.inverse_transform(trasf)
    return result

```

```
new_y_test = one_hot_encoder.inverse_transform(y_test)
new_y_test = new_y_test.flatten()
preds = model.predict(X_test)
preds = transform_prediction(preds)
preds = preds.flatten()
y_pred = preds

print("Test")
evaluate_metrics(new_y_test,y_pred)
```

```
Test
['glioma' 'meningioma' 'notumor' 'pituitary tumor']
                precision    recall  f1-score   support

         glioma       0.94      0.91      0.93       294
     meningioma       0.77      0.86      0.82       139
        notumor       1.00      1.00      1.00       283
pituitary tumor       0.97      0.94      0.95       197

       accuracy                           0.94       913
      macro avg       0.92      0.93      0.92       913
   weighted avg       0.94      0.94      0.94       913
```



The evaluation results showed that the model achieved an overall accuracy of 94%, with the model demonstrating strong performance across the four classes. For glioma, the model attained a precision of 0.94, recall of 0.91, and f1-score of 0.93. For meningioma, the model attained a precision of 0.77, recall of 0.86, and f1-score of 0.82. The model excelled in predicting the notumor class with a precision of 1, recall of 1, and f1-score of 1. For pituitary tumor, the precision was 0.97, recall was 0.94, and f1-score was 0.95. The macro and weighted averages for precision, recall, and F1-scores were all above 0.92, indicating that the model performed very well across all categories.

The confusion matrix provides further insights into the model's performance across the four classes. It shows that for glioma, the model correctly predicted 268 instances but misclassified 24 meningiomas, and 2 pituitary tumors. For meningioma, it correctly identified 120 instances but confused 15 gliomas, and 4 pituitary tumors. For notumor, the model accurately predicted 282 instances, misclassifying only 1 glioma. For pituitary tumor, the model correctly predicted 185 instances but misclassified 1 glioma, and 11 meningiomas.

9. Visual Transformer (ViT)

9.1 Introduction

The Vision Transformer (ViT) model was introduced in a research paper titled "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," presented at ICLR 2021. Developed by Neil Houlsby, Alexey Dosovitskiy, and their colleagues from the Google Research Brain Team, the ViT leverages the transformative power of attention mechanisms. Transformers are deep learning models that apply attention mechanisms to weigh the significance of each part of the input data sequence. Comprising multiple self-attention layers, transformers are predominantly used in natural language processing (NLP) and computer vision (CV). They offer a promising generic learning method applicable across various data modalities. Notably, transformers have achieved state-of-the-art accuracy in computer vision with enhanced parameter efficiency.

In the Vision Transformer, attention mechanisms are applied to image patches to capture relationships between different parts of the image. The process begins with converting image patches into vectors known as embeddings. For each patch embedding, three sets of vectors are created: queries, keys, and values. The similarity between each query and all keys is calculated using the scalar product. These similarity scores are then converted into probabilities using the softmax function, facilitating easier interpretation of their relative importance. The value representations are combined in a weighted manner according to these probabilities, resulting in a new representation for each input position. To enhance the model's performance, multiple attention mechanisms, referred to as heads, are applied in parallel, and their outputs are concatenated in a process known as multi-head attention. This sophisticated approach enables the Vision Transformer to effectively analyze and comprehend images by leveraging the capabilities of transformers, which have proven successful in NLP, and adapting them to the domain of computer vision to achieve superior performance.

The Transformer architecture consists of two main parts: the Encoder and the Decoder. However, for the Vision Transformer (ViT), only the Encoder part is utilized. The primary components of the Vision Transformer are:

1. **Input Embedding**: Inputs, such as words in NLP or image patches in ViT, are transformed into dense vectors. For ViT, the image is divided into non-overlapping patches, each of which is flattened and projected into a vector space.
2. **Positional Encoding**: Because Transformers lack an intrinsic mechanism to account for the position of sequence elements, positional encoding is added to the input vectors. This helps the model retain the order of elements in the sequence.
3. **Encoder**: The Encoder is composed of N identical layers, each containing two main components:
    - **Self-Attention Mechanism**: This mechanism allows the model to focus on different parts of the sequence simultaneously.
    - **Feed-Forward Neural Network**: Each attention layer is followed by a feed-forward neural network applied to each position separately.

As in NLP Transformers, ViTs can process all patches in parallel, making training and inference

more efficient. The ViT architecture can be easily scaled by increasing the number of layers or the size of embedding vectors, allowing it to handle more complex images. Additionally, ViTs are capable of learning spatial relationships and long-range dependencies within an image, which can be challenging for traditional convolutional networks (CNNs) to capture. There are several key differences between Vision Transformers and CNNs:

- **Convolutions vs. Attention**: CNNs use local convolutions to extract features, whereas ViTs use attention to globally consider all patches simultaneously.
- **Positional Encoding**: CNNs inherently understand spatial structure due to the nature of convolutions, whereas ViTs need to add positional information through positional encoding.
- **Computational Efficiency**: CNNs are highly optimized for image processing and are very efficient on dedicated hardware. ViTs, although powerful, may require more computing power to achieve similar performance.

ViTs often require large amounts of training data to perform well because they lack the strong inductive biases of CNNs, which help in handling smaller datasets. For high-resolution images, the number of patches can become very large, increasing computational complexity. This necessitates careful consideration of the balance between model capacity and computational resources.

9.2 Application

Before creating our Vision Transformer (ViT) model, we created three main classes: `PatchEncoder`, `TransformerEncoder`, and `VIT`.
The `PatchEncoder` class is designed to handle the initial step of processing the input images. In this context, the input images are of tumors, and the goal is to segment these images into smaller, more manageable patches. This is crucial because it allows the model to focus on smaller regions of the image, capturing local features that are important for identifying characteristics of the tumor. The constructor (`__init__` method) initializes the layer by setting up a linear projection (`Dense`) to map the patches into a higher-dimensional space suitable for the transformer. It also includes a positional embedding layer (`layers.Embedding`) to maintain spatial information, which is important because the position of a patch within the image can provide context about the tumor's structure. The `call` method processes an input image by extracting patches using tf.image.extract_patches. It divides the image into smaller sections of size `patch_size`. These patches are then reshaped and linearly projected to match the hidden size required for the transformer. Positional embeddings are added to these projected patches to encode their positions within the original image. This step ensures that the model can distinguish between patches from different regions of the image, preserving the spatial relationships that are critical for accurate tumor classification.

```
class PatchEncoder(layers.Layer):

    def __init__(self, patch_size, Hidden_size):
        super(PatchEncoder, self).__init__(name='patch_encoder')
        self.linear_projection = Dense(Hidden_size)
        self.position_embedding = layers.Embedding(
            input_dim=num_patches, output_dim=Hidden_size
        )
        self.num_patches = num_patches

    def call(self, image):
        batch_size = tf.shape(image)[0]  # Retrieve batch size
        patches = tf.image.extract_patches(images=image,
                                           sizes=[1, patch_size, patch_size, 1],
                                           strides=[1, patch_size, patch_size,
    1],
                                           rates=[1, 1, 1, 1],
                                           padding='VALID'
                                           )
```

```
        patch_dims = patches.shape[-1]
        patches = tf.reshape(patches, [batch_size, -1, patches.shape[-1]])
        embedding_input = tf.range(start=0, limit=self.num_patches, delta=1)
        output = self.linear_projection(patches) + self.position_embedding(
    embedding_input)
        return output
```

The `TransformerEncoder` class represents a single transformer encoder layer, a key component of the Vision Transformer architecture. Transformers have been highly effective in capturing long-range dependencies in data due to their self-attention mechanism. For tumor images, this means the model can integrate information from different parts of the image to make more informed decisions. The constructor initializes layer normalization layers (`LayerNormalization`), a multi-head attention mechanism (`MultiHeadAttention`), and dense layers (`Dense`) with GELU activation functions. These components are essential for creating deep, non-linear representations of the input data. In the `call` method of `TransformerEncoder`, layer normalization is first applied to the input to stabilize and accelerate training. The multi-head attention mechanism then allows the model to attend to different parts of the input sequence (the patches) simultaneously, effectively capturing complex relationships between patches. The output of the attention mechanism is added to the original input via a residual connection, which helps in training deeper networks by mitigating the vanishing gradient problem. This is followed by another layer normalization and dense transformation, further refining the learned representations.

```
class TransformerEncoder(layers.Layer):
    def __init__(self, Num_heads, Hidden_size):
        super(TransformerEncoder, self).__init__(name='transformer_encoder')
        self.layer_norm_1 = LayerNormalization()
        self.layer_norm_2 = LayerNormalization()

        self.multi_head_att = MultiHeadAttention(Num_heads, Hidden_size)

        self.dense_1 = Dense(Hidden_size, activation=tf.nn.gelu)
        self.dense_2 = Dense(Hidden_size, activation=tf.nn.gelu)

    def call(self, input):
        x = self.layer_norm_1(input)
        x1 = self.multi_head_att(x, x)

        x1 = Add()([x1, input])
        x2 = self.layer_norm_2(x1)
        x2 = self.dense_1(x2)
        output = self.dense_1(x2)
        output = Add()([output, x1])

        return output
```

The `VIT` class integrates the patch encoding and transformer encoding processes to form the full Vision Transformer model. This class is crucial for constructing a model that can process entire images and produce classification outputs. The constructor initializes the patch encoder, a list of transformer encoder layers, and dense layers for the final classification task. This structure allows the model to first break down the image into patches, encode these patches, and then apply multiple layers of transformer encoding to build a comprehensive understanding of the image content. The `call` method of `VIT` sequentially processes the input through the patch encoder and each transformer encoder layer. This step-by-step processing enables the model to progressively refine its understanding of the image. After the transformer layers, the output is flattened and passed through dense layers, which act as the final classifier. The dense layers use activation functions such as ReLU to introduce non-linearity and enhance the model's capacity to distinguish between different classes of tumors.

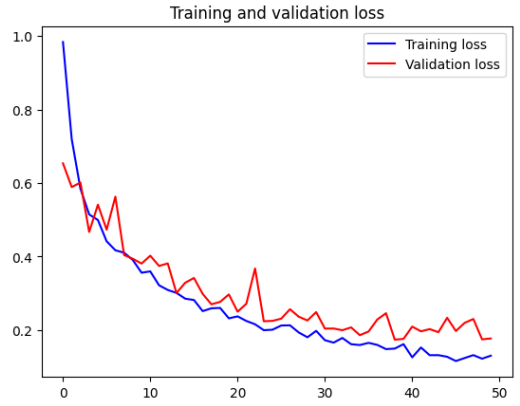
```
class VIT(tf.keras.Model):
    def __init__(self, Num_heads, Hidden_size, num_patches, num_layers,
    mlp_head_units):
        super(VIT, self).__init__(name='vision_transformer')
        self.num_layers = num_layers
        self.patch_encoder = PatchEncoder(patch_size, Hidden_size)
```

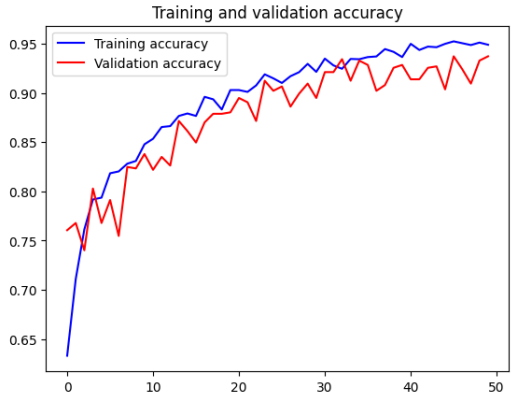
```
        self.trans_encoder = [TransformerEncoder(Num_heads, Hidden_size) for _ in
     range(num_layers)]
        self.flatten = layers.Flatten()
        self.dense_1 = layers.Dense(mlp_head_units, activation='relu')
        self.dense_2 = layers.Dense(mlp_head_units, activation='relu')
        self.dense_3 = layers.Dense(num_classes, activation='softmax')

    def call(self, inputs):
        x = self.patch_encoder(inputs)  # Get encoded patches
        for i in range(self.num_layers):
            x = self.trans_encoder[i](x)  # Apply transformer encoder layers

        x = self.flatten(x)
        x = self.dense_1(x)
        x = self.dense_2(x)
        output = self.dense_3(x)
        return output
```

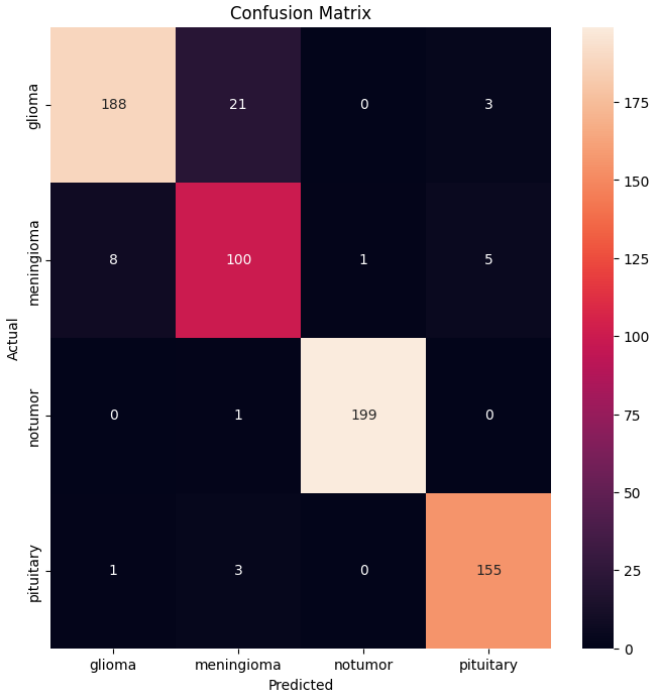
Following the definition of the three main classes, we created and configured the ViT model. We first instantiated the ViT class with the specified parameters. These parameters define the architecture of the transformer, including the number of attention heads, the size of the hidden layers, the number of image patches, the number of transformer layers, and the units in the multi-layer perceptron (MLP) head. We set the loss function to `CategoricalCrossentropy` with `from_logits` set to `False`, which is appropriate for our classification task. To evaluate the model's performance, we included `CategoricalAccuracy` to measure the overall accuracy and `TopKCategoricalAccuracy` with `k=2` to track the accuracy within the top-2 predictions. Then, we compiled the model using the Adam optimizer, configured with a specific learning rate to ensure efficient convergence during training. This combination of optimizer and loss function is well-suited for handling the complexities of training deep neural networks like the Vision Transformer. To ensure that we retain the best-performing model during training, we set up a checkpoint callback. The `ModelCheckpoint` was configured to monitor the validation accuracy and save only the best model weights based on this metric. This approach ensures that the optimal model weights are preserved, facilitating the best possible performance on the validation set. By compiling the ViT model with these settings, we ensured a robust setup for training and validation, leveraging both effective optimization techniques and practical model management strategies.
After setting up the Vision Transformer (ViT) model, we proceeded with the training phase, with validation performed on a separate dataset to monitor and improve generalization.

```
vit = VIT(Num_heads, Hidden_size, num_patches, num_layers, mlp_head_units)
loss_function = tf.keras.losses.CategoricalCrossentropy(
    from_logits=False,
)
metrics = [CategoricalAccuracy(name='accuracy'), TopKCategoricalAccuracy(k=2,
    name='top_k_accuracy')]
vit.compile(
    optimizer=Adam(learning_rate=learning_rate),
    loss=loss_function,
    metrics=metrics
)

checkpoint_filepath = "/content/drive/MyDrive/ProgettoDataMining/best"
checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    checkpoint_filepath,
    monitor="val_accuracy",
    save_best_only=True,
    save_weights_only=True,
)

history = vit.fit(
    training_dataset,
    epochs=num_epochs,
    verbose=1,
    validation_data=vali_dataset,
    callbacks=[checkpoint_callback]
)
```

Following the training phase, we tested the model on an independent test dataset to evaluate its final performance and ensure that it generalizes well to new, unseen data. Below are the results of the model's performance:







The model performed very well, achieving an accuracy of approximately 94% in classifying images correctly. The goodness of fit of the transformer was analyzed through both the evolution of loss and accuracy over the course of training and the confusion matrix. The close alignment of training and validation metrics suggested robust generalization, indicating that the model was not overfitting and could perform well on unseen data. The confusion matrix provided further insights, highlighting specific areas where the model excelled and where it may have needed improvement. For the glioma and meningioma classes, there was a notable misclassification rate of approximately 20%, where images from each category were confused with the other. This indicated a need for further refinement in distinguishing these two classes. In contrast, the model achieved near-perfect classification for pituitary and tumorless images, demonstrating its strong capability in these categories and the absence of systematic errors observed in the glioma and meningioma classes.

10. Conclusions

In this study, we approached the problem using two distinct methodologies: a traditional machine learning approach that leveraged the interpretability of simpler models, and a more intricate deep learning approach focused on predictive accuracy.
Both approaches included an outlier detection phase. Initially, we used an autoencoder, and subsequently, to address the high computational cost of the autoencoder, we tried to reduce complexity by using an NMF + Isolation Forest approach. However, the second approach did not yield the desired results.
For the traditional machine learning approach, we began with feature extraction using linear and easily interpretable models. We initially tried PCA, which resulted in poor outcomes with blurry reconstructed images. Subsequently, NMF allowed us to identify components that we interpreted as the main features of the original images.
Next, we implemented traditional machine learning models to classify images into four classes and select key features. Most models performed well, particularly the Random Forest. Specifically, the combination of Random Forest with Recursive Feature Elimination successfully reduced the features extracted by NMF to just 8, while only slightly decreasing the accuracy from about 80% to 70%, thus ensuring satisfactory results. This feature selection was interpreted as similarities with the base images extracted from NMF, and could be particularly useful for medical professionals when diagnosing non-tumor and pituitary classes, where the models performed exceptionally well.
In the deep learning approach, we started with preprocessing, including image cropping and common geometric transformations to preprocess and augment our dataset. We then implemented deep learning models such as ResNet50 and VGG16, both achieving accuracy levels above 90%. Finally, we tested a Vision Transformer, which also yielded excellent results comparable to other deep models.
In conclusion, the traditional machine learning techniques—LASSO Logistic Regression, Decision Tree Classifier, Random Forest, and Support Vector Classifier—provided a solid foundation for feature selection and preliminary classification. However, the deep learning techniques—Convolutional Neural Networks (ResNet50 and VGG16) and the Transformer-based Vision Transformer—significantly outperformed traditional methods in terms of accuracy and efficiency. This project can serve as a valuable tool for automatic tumor detection and identification, offering results that ensure interpretability for specialists in the field.