EEE 485 Statistical Learning and Data Analysis
Project Final Report: Chest Cancer Classification Using CT Scan Image Dataset

21801124 Talha Naeem, 21801356 Fahad Waseem Butt
Bilkent University, 06800, Ankara, Turkey

1.0 Introduction and Problem Description

For this project, the goal is to make use of a chest CT scan image dataset to make a classifier to 4 different types of chest cancer. The image data was processed into arrays through our methods and then used in later steps. For this purpose, multiple methods are tested, those being Logistic Regression, k-Nearest Neighbours, and the Convolutional Neural Network. All of the methods were made in this project from scratch, without the use of packages such as scikit-learn. As the methods were implemented from scratch, it can be argued that there is room for improvement, however, the theory followed was based on the correct information and the results are appropriate.

2.0 Dataset Description

We used a chest CT scan image dataset from Kaggle [1]. The images are in png or jpg format. The image data was divided manually into three directories: Train (70 % of the total images), Validation (15 % of the total images), and Test (15 % of the total images). There are 4 classifications: Normal, Adenocarcinoma, Large Cell Carcinoma, and Squamous Cell Carcinoma. Of these 4 classes, 1 represents a normal lung, and 3 represent 3 different types of lung cancer. The dataset has the following folder and path distribution:

```
Cancer Cell Data
|_  Train   700 Images
          |_  Adenocarcinoma              236 Images
          |_  Large Cell Carcinoma        131 Images
          |_  Normal                      151 Images
          |_  Squamous Cell Carcinoma     182 Images
|_  Valid   150 Image
          |_  Adenocarcinoma              51 Images
          |_  Large Cell Carcinoma        28 Images
          |_  Normal                      32 Images
          |_  Squamous Cell Carcinoma     39 Images
|_  Test    150 Images
          |_  Adenocarcinoma              51 Images
          |_  Large Cell Carcinoma        28 Images
          |_   Normal                     32 Images
          |_  Squamous Cell Carcinoma     39 Images
```

To traverse through all the images in the specific folder we have to use the glob library and PIL library to read the image pixels. We have our label associated with each image (row). The number of columns represents the number of features which is 4096 pixels per image because we have reshaped each image to (64x64).
But still, we have Red, Green, and Blue in the range of [0, 255]. The next step of preprocessing is to use min-max normalization on the data so the data is in the range [0, 1]. This is necessary as we build a convolutional neural network, and without normalization, the neural network explodes in a multitude of values, so it is essential to normalize to ensure that all pixels are close to each other.
Now the last part of preprocessing is to encode the labels. Our labels are in the text format (Normal, Squamous Cell Carcinoma, Large Cell Carcinoma, Adenocarcinoma), but raw pixels cannot feed the text input to statistical models. We need to encode them. There are different ways to encode the labels, but we used feature encoding [0,1,2,3].

3.0 Machine Learning Methods and Results

3.1 Principal Component Analysis

Principal Component Analysis works as follows: the data is normalized (in this project min-max normalization was used) to reduce the disparity in weights of large (e.g. 200) and small (e.g. 2) feature values; the features in the data are mean centered; the covariance matrix is found; eigenvalues, and

then eigenvectors, are found using the covariance matrix; the eigenvectors are normalized, giving Principal Components (the normalized eigenvectors). The Principal Components may be used to check the Proportion of Variance Explained, to see how much information the Principal Components capture [2].

Image data is being used in this project, which has three channels, RGB: Red, Green, and Blue. In standard Machine Learning methods, it is not good practice to use raw pixel data to train models. So Principal Component Analysis was needed to extract important information from the RGB channels.

For this purpose, the data was loaded as *64x64* images, which is 4096 pixels. The image data was in the form of a 3D array, where each image was in the form *64x64x3*, meaning that for each image, the RGB channel information was separated (this meant that the entire data was in the form *1000x64x64x3*). The RGB information was separated and the pixels were flattened, hence splitting the data into 3 separate parts for each channel R, G, and B, and putting it in the form *1000x4096*.

The data was separated into splits, so instead of three *1000x4096* arrays, the data had three arrays of dimension *(number of images in the class in the data split (training, validation, or test))x4096*.

Principal Component Analysis was applied to reduce the number of features, which was 4096. The goal was to have each class for each split of the dataset has the same number of Principal Components while having a minimum Proportion of Variance Explained of 75%. Through some trial and error, it was found that 25 Principal Components met this requirement. The Proportion of Variance Explained for each Class for each split of the dataset is shown in Figure 1 as follows.

Class [Number] (Data Split)	PVE for R Channel for 25 PCs / %	PVE for G Channel for 25 PCs / %	PVE for B Channel for 25 PCs / %
Normal [0] (Training)	97.65322479302134	97.65282690809904	97.65221764453241
Squamous Cell Carcinoma [1] (Training)	80.89206448525901	80.89206448525901	80.89206448525901
Large Cell Carcinoma [2] (Training)	81.6279657146654	81.62846685899767	81.62245790204224
Adenocarcinoma [3] (Training)	76.24013526381984	76.2386589550493	76.23878100726542
Normal [0] (Validation)	99.99689960331688	99.99677514402659	99.99206938568119
Squamous Cell Carcinoma [[1] (Validation)	94.26547102365959	94.27143854304035	94.27195099427269
Large Cell Carcinoma [2] (Validation)	99.5286345235554	99.5286345235554	99.5286345235554
Adenocarcinoma [3] (Validation)	90.90884960503752	90.90884960503752	90.90884960503752
Normal [0] (Testing)	99.99999999999999	99.99999999999999	100.00000000000000
Squamous Cell Carcinoma [[1] (Testing)	95.49637985063929	95.49637985063929	95.49637985063929
Large Cell Carcinoma [2] (Testing)	99.34917929241999	99.34917929241999	99.34917929241999
Adenocarcinoma [3] (Testing)	90.34865444482301	90.35056211648755	90.35035650682522

Figure 1: Table of The Proportion of Variance Explained for Each Class for Each Split of the Dataset

It can be seen that, overall, the Proportion of Variance explained is well above the minimum threshold of 75%. The data was now separated into RGB channels and needed to be recombined which was done so by stacking the R, G, and B channels per class. This way, the channel information was also included as data points (basically multiplying the data points by 3), and the features for each image channel were also reduced. This means that the entire data, which was in the form of *1000x4096x3* had its features reduced to *3000x25*. So at the end of the Principal Component Analysis, the data splits and their sizes were as follows: training (*2100x25*), validation (*450x25*), and testing (*450x25*). This method takes a very small amount of time, roughly 10 seconds.

3.2 One-vs-All Logistic Regression

One-vs-All Logistic Regression is a form of Multi-Class Logistic Regression, wherein the multi-class data is taken as a binary classification problem, and each binary classification is made for each pair of classes [3].
For this implementation, the sigmoid activation function was used so that it can convert the outputs into a categorical output in the range [0,1], which helps for binary classification, which is the case here as we are using the One-vs-All method. Log Loss is used instead of something like Cross-Entropy Loss, as log loss is better for binary models [4], which is the case in our implementation.
The One-vs-All Logistic Regression implementation had two hyperparameters that needed to be tuned: the number of epochs for which the model was trained on, and the learning rate used. For optimizing the hyperparameters, different numbers of each hyperparameter were tested on the validation data and the results of this testing are shown in the loss plots in Figures 2 and 3.

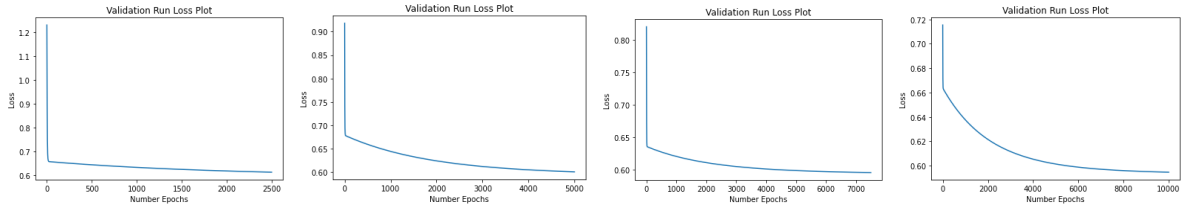

Figure 2: Loss for Logistic Regression with Epoch Counts [2500, 5000, 7500, 10000]

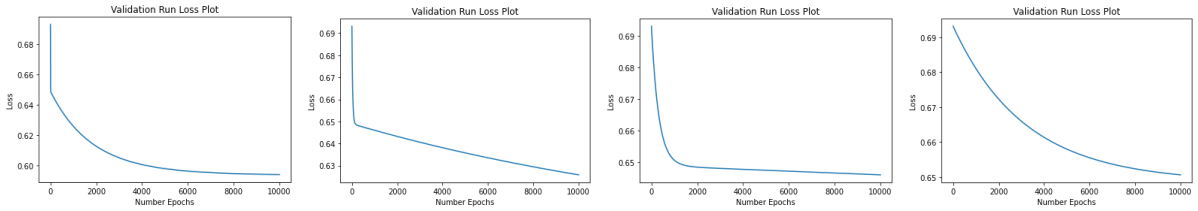

Figure 3: Loss for Logistic Regression with Learning Rates [0.1, 0.01, 0.001, 0.0001]

It can be seen in Figure 2 that the model with the best loss curve is the fourth model with 10,000 epochs used, and from Figure 3 it is observed that the model with the learning rate 0.1 was the best. This conclusion is based on how the loss plot is decreasing, where the smoothness of the decrease and the amount of the reduction are considered. The plot of 10,000 epochs and the plot for learning rate 0.1 both have a smooth reduction in the loss as the plot reaches convergence, and have the best loss drop when compared to the initial value.
Using these hyperparameters the accuracies found were as:
training=54.285714285714285%, validation=39.111111111111114%, testing=35.77777777777777%.
The training accuracy is of an acceptably high value, with the validation and testing accuracies being close to each other as being on the low side. While unexpected, it makes sense that the logistic regression accuracy is not too high as the model was made from scratch. A state-of-the-art model, such as one from external libraries, may give better results due to many different optimization techniques, but our implementation was limited to what can be built from scratch in a limited time frame. The overall time taken to train each different model was about 15 to 20 seconds.

3.3 k-Nearest Neighbours

The kNN algorithm works as follows: the training data distribution is observed; then, one by one, for each point of the validation or testing data, the distance of the point is calculated from the points in the training data; finally, based on a ‘k’ number of closest data points to the test data point being checked, the point is classified to the label with the highest number of points closest to the test point [5]. The distance metric used was the Euclidean Distance [6] which is calculated as follows.

$$d(x, y) = \sqrt{\Sigma_{i=1}^{n}(y_i - x_i)^2}$$

The kNN model may perform very differently for different values of k, but there is no exact way to determine a correct value for k, where k is the number of nearest neighbors to a point being checked. Having a k that is too low can result in a high variance but low bias and a k that is too high results in a high bias but low variance [6]. Overall, the kNN method is relatively simple to implement in most cases, requires simple calculations, and is easier to follow than other methods. However, issues arise when the dataset being used is too large, and as a consequence of the curse of dimensionality, the kNN method performs worse on larger datasets. Furthermore, while this algorithm requires simpler calculations, it requires too many and takes up a lot of memory; the kNN method is also prone to overfitting [6].
The kNN algorithm was chosen to test how it would perform on our large dataset and set it as a point of reference for a weak learner. Figure 4 shows the Accuracy of the kNN model as the number of k Nearest Neighbours increases, with this run being done on the validation data.

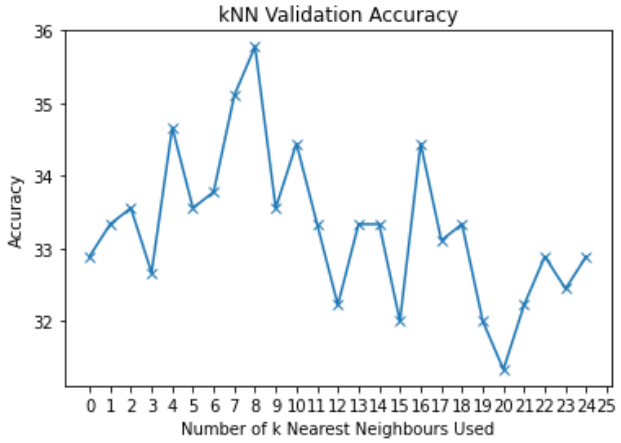

Figure 4: Accuracy of the kNN Model as the Number of k Nearest Neighbours Increases
It can be seen that the best k is 9, and so the testing data was used to find the testing accuracy of kNN. The testing accuracy of the kNN model with k as 9 was 35.77777777777777%.
The accuracy gathered from the kNN model is relatively low, for both the validation and testing runs, but this was expected. This is because when the number of data points is high, a higher k value is needed to find better accuracy. But a high k value leads to high bias and low variance, which causes the performance of the model to drop. Even though the number of features was greatly reduced from 4096 pixels to 25 Principal Components after the PCA method, it seems that for this data, even a feature space of 25 elements is too high, which causes the kNN to perform poorly. The time taken per run of the kNN method was roughly 2 minutes per run, with lower k values giving quicker results whereas higher k values gave slower results. Hence this proves that the kNN method is not appropriate for larger datasets.

3.4 Convolutional Neural Network [8]

We used three layers in the CNN: (1) Convolutional layer (2) Maxpooling Layer (3) Softmax Layer.
Before feeding the input into the CNN, it is essential that we normalize it. The best normalization that we found turned out to be min-max normalization. Averaging and Mean Centered and Standardization reduced accuracy and blew away the loss to NAN.

Layers' Input & Output Shapes

In the convolutional layer, we used a total of 16, 3x3 filters that output the shape (62, 62, 16). This is because our input was of shape (64x64), given an f x f filter, the formula is h - f + 1 for height and w - f + 1 for width. For the Maxpooling layer, we are using a 2x2 pooling layer, and its output shape is expected to be (31, 31, 16). The reason is we are taking the maximum of the convolution layer output. For the Softmax Layer, the input shape will be (31*31*16, 4) and the output shape is expected to be 4, which will be our final output. This output has the same shape as our labels. In other words, it takes the Maxpooling layer output as input and outputs our prediction. Each layer has forward and

backpropagation. It is assumed that the reader is familiar with how forward and back propagation works for each layer.

Hyperparameters

The loss that we used in the forward layer was a cross-entropy loss. No optimizers were used. We did not use minibatch gradient descent in the backpropagation, but just simply stochastic gradient descent. Hence there will be stochasticity in finding the minimum, and also it takes a longer time to find it as compared to minibatch. The reason why the minibatch SGD was not implemented was due to project deadline constraints. Although the results for SGD were not bad either as we will see shortly. We first start with the training and validation process to tune the hyperparameters. First, we should specify our hyperparameters. To make things simpler and time efficient we predecided the most hyperparameters like loss function, not using any optimizer, and fixing the number of epochs to 4 because we reach the minimum at around 4 epochs. The reason for preselecting most of the hyperparameters is that we need to save time. Because it takes a total of approximately 5 hours to train, validate and test the CNN. Hence it made the most sense to us to preselect most of the hyperparameters. Since it takes a lot of time to run. We have to ensure that we run a minimum number of times to optimize the hyperparameters. As a result, the only hyperparameter left to be optimized is now our learning rate. We are testing these learning rates: [0.0001, 0.001, 0.01, 0.1]. We will validate each learning rate on the validation set, we will choose the best one, and then put it on the test.

Results for training and validation

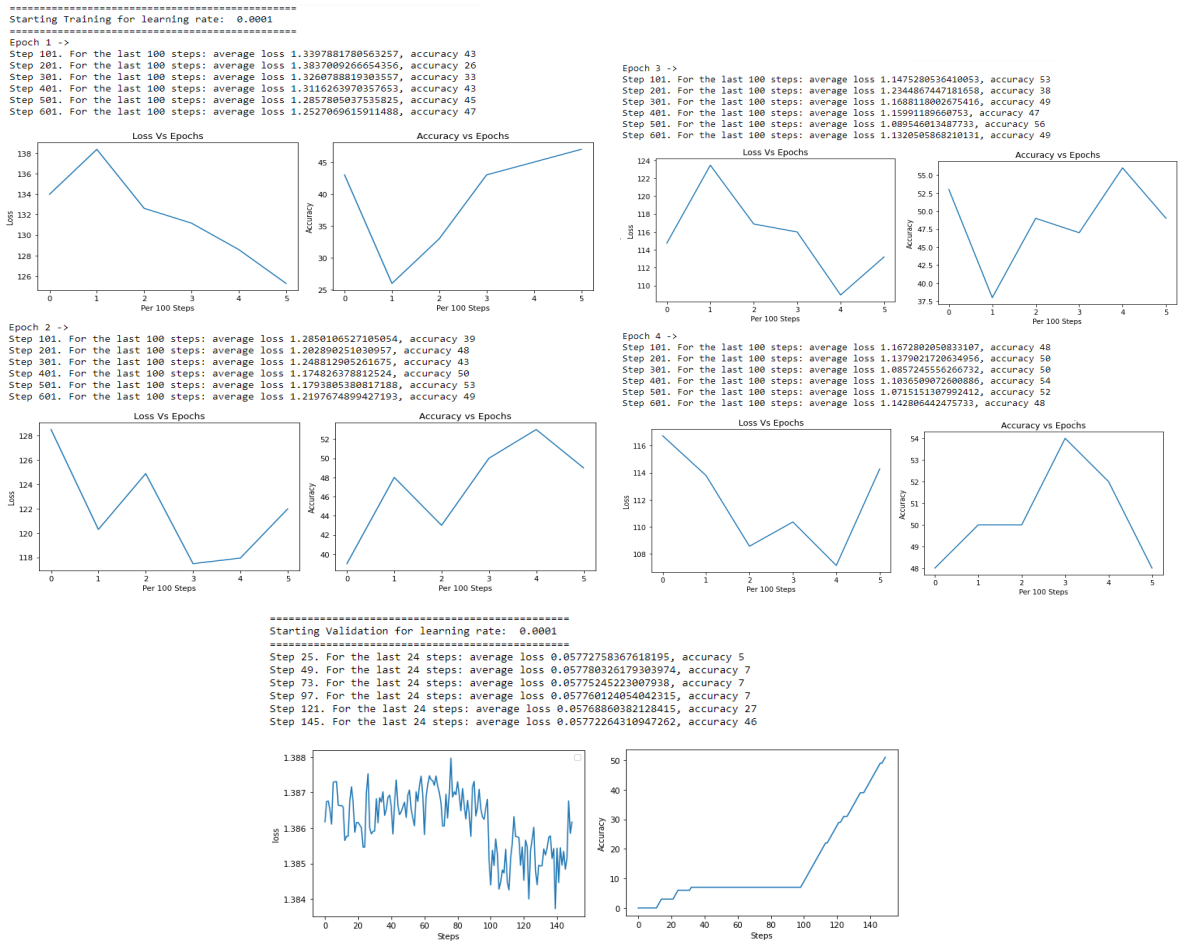

Figure 5: Loss and Accuracy Plots for Learning Rate 0.0001

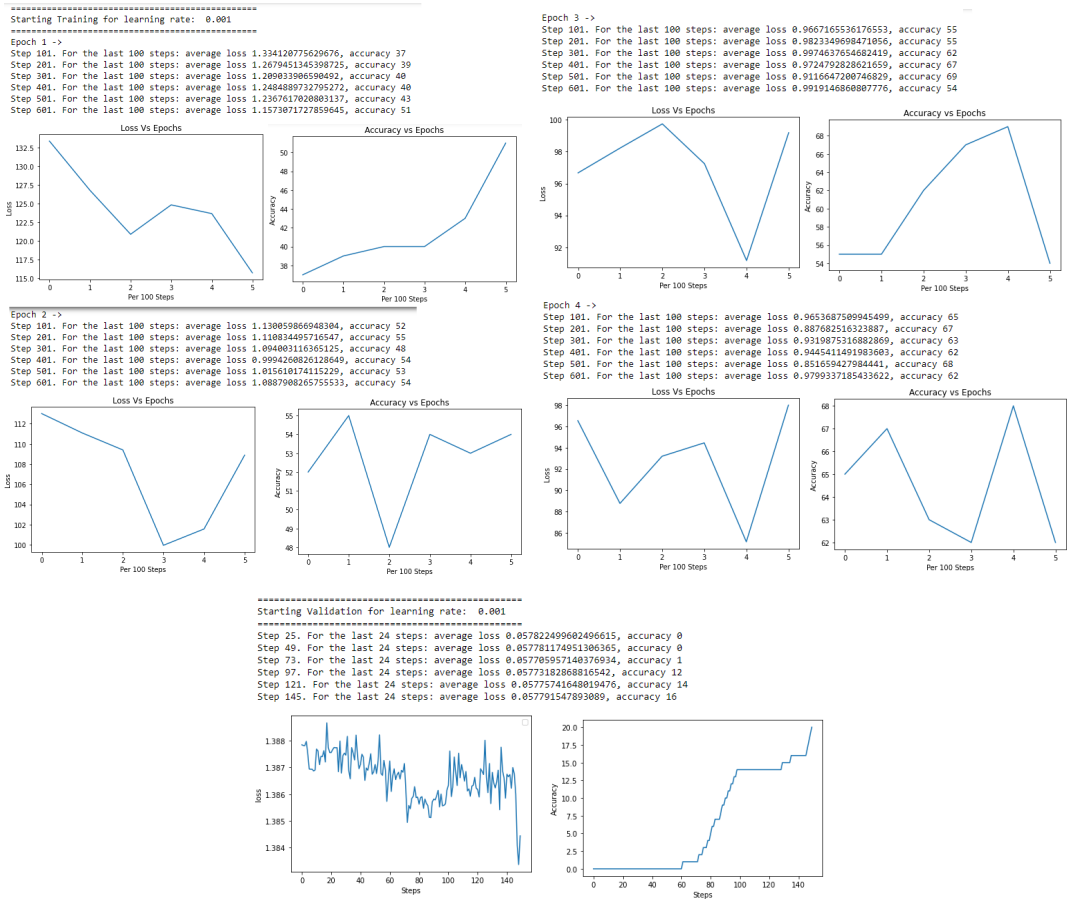

Figure 6: Loss and Accuracy Plots for Learning Rate 0.001

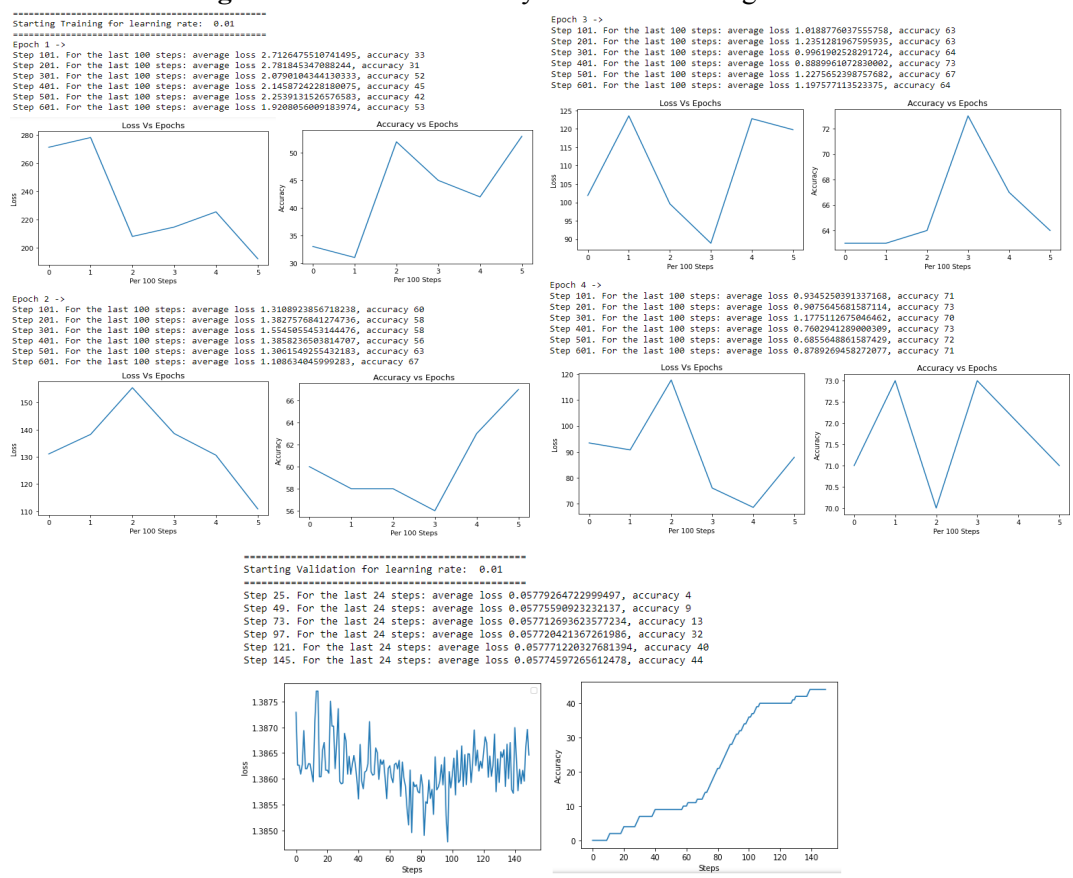

Figure 7: Loss and Accuracy Plots for Learning Rate 0.01

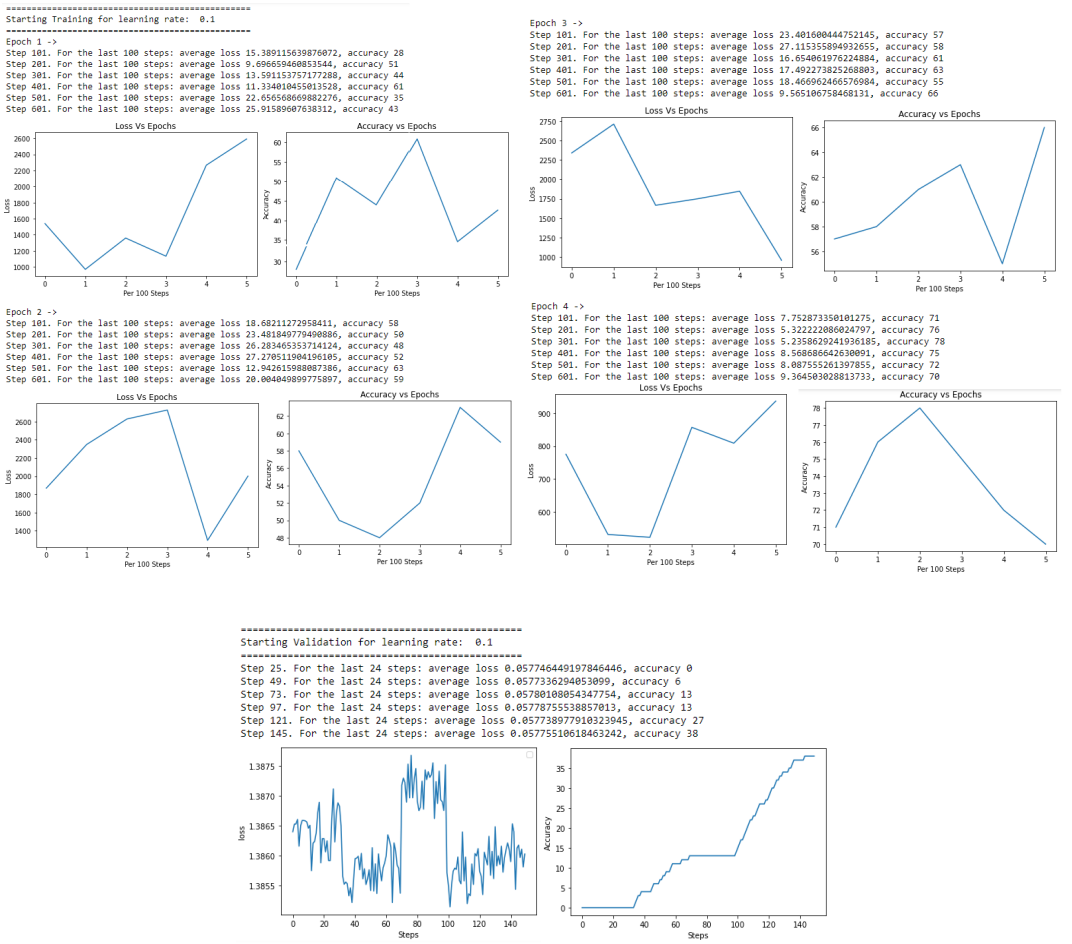

Figure 8: Loss and Accuracy Plots for Learning Rate 0.1

As can be seen from the series of results mentioned above, the best validation results took place when the learning rate was 0.01 and the corresponding validation accuracy was around 45. We will now use this on our test dataset and see the results.

Testing on the test data with optimized hyperparameters

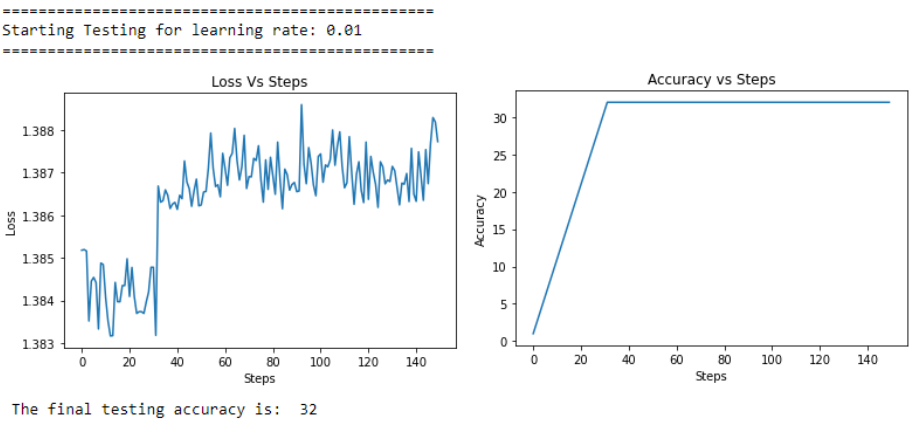

Figure 10: Testing Loss and Accuracy Plots

As we can see that the test loss and validation loss are very similar in fact less than the training loss except for the fact they are presented differently. In training loss, we are presenting loss and accuracy sampled and averaged after every 100 samples of images, while in validation and testing we are plotting after every sample of the image.

Also, the overall accuracy seems to be low which is 32. The maximum validation accuracy we got was around 52. It makes sense because we are not using an optimizer. That is the reason it does not

perform like state-of-the-art models. Another reason can be that even though we get a good score at the 4th epoch, there is no guarantee that we have found the optimal minimum. Also if we have used minibatch gradient loss and update, the accuracy would have improved.

4.0 Challenges

For PCA, the biggest challenge was addressing how to stay mathematically consistent when dealing with reduction of multiple image channels, but once that was understood, the rest was the standard process. When working on Logistic Regression, the challenge was in figuring out how to handle the multi-class classification aspect of the problem, and then in understanding how to implement the method from scratch. The kNN implementation was not too challenging, as after the PCA method, the code worked as normal, and ran as expected, so there was no issue in it. For CNN the challenge was tuning the hyperparameters because it took 2 hours to run the training process. The total time for all training, validating, and testing was 5 hours. Hence, we had to preselect some hyperparameters.

5.0 Gantt Chart of Work Packages

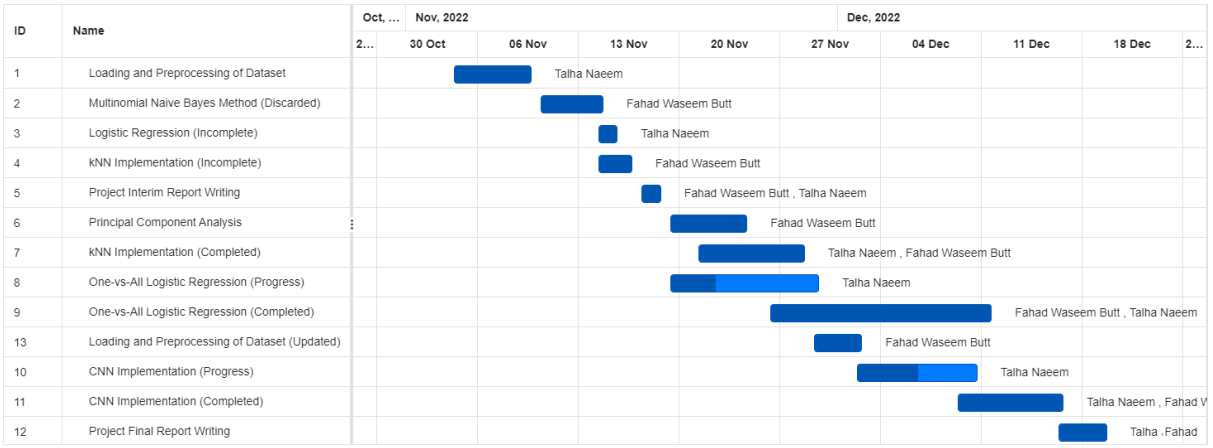

Figure 5: Gantt Chart Showing Project Timeline [9]

6.0 Conclusion

In this project we formulated the problem which was to classify the cancer images according to their corresponding labels. We decided to use two statistical methods, kNN and logistic regression, and one neural network. We realized that the way we formula the algorithm of the model has the biggest impact on the final result. We were able to make the algorithms work and found very convincing results. These models could not mimic the accuracy of the state-of-the-art models, but still, we were able to provide sufficient evidence to support that our results made sense. It can be concluded that this project was a success. The goal of the project was to gain an in-depth understanding of how machine learning methods work by making them from scratch, so this goal was definitely accomplished. We needed to research and understand the mathematics behind our chosen models to implement them. The results have been discussed in depth in the sections above, and to summarize, the models performed as expected, but we were limited by not having state-of-the-art models.

7.0 References

[1] “Chest CT-Scan images Dataset”, Kaggle, 2020. [Online]. Available: https://www.kaggle.com/datasets/mohamedhanyyy/chest-ctscan-images
[2] “Principal Component Analysis”, Javatpoint. [Online]. Available: https://www.javatpoint.com/principal-component-analysis

[3] J. Brownlee, “One-vs-Rest and One-vs-One for Multi-Class Classification”, Machine Learning Mastery, 2020. [Online] Available: https://machinelearningmastery.com/one-vs-rest-and-one-vs-one-for-multi-class-classification/
[4] J. D. McCaffrey, “Log Loss and Cross Entropy are Almost the Same”, jamesmccaffreywordpress, 2016. [Online]. Available: https://jamesmccaffrey.wordpress.com/2016/09/25/log-loss-and-cross-entropy-are-almost-the-same/
[5] Antony C., “K-Nearest Neighbor. A complete explanation of K-NN”, Medium, 2021. [Online]. Available: https://medium.com/swlh/k-nearest-neighbor-ca2593d7a3c4
[6] “What is the k-nearest neighbors algorithm?”, IBM. [Online] Available: https://www.ibm.com/topics/knn
[8] S. Saha, “A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way”, Towards Data Science, 2018. [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53
[9] “Free Online Gantt Chart Software.”, Onlinegantt. [Online]. Available: https://www.onlinegantt.com/#/gantt

9.0 Appendix I: Revisions Made from Feedback

We were asked to first change the distribution of the dataset. The distribution was supposed to be 70% training and 15% validation and testing. We manually changed the distribution of the Kaggle folder of the image dataset and loaded the images in the python notebook using the Dataloader function in python. We clarified using the shapes of the data in order to ensure that we meet this feedback requirement

Secondly, we were informed that inserting the raw pixels into the statistical models is the wrong way unless it is a CNN. So we performed PCA and hence selected the best features and had the best PVE. Then we inserted these features into the kNN and logistic regression. We directly inserted the raw pixels into the CNN because it performs convolutions in the layers to extract the features themselves.

We were asked to use vision libraries to visualize the results. We used Matplotlib and plotted the results that we thought gave meaningful insight as to how our models are behaving.

We were asked not to write the textbook definition and hence we just explained the thought process as to how we modeled the problem into code.

We were told to write peer contributions in the Gantt Chart, and as can be seen in the Gantt Chart section that we have added peer contribution sections as well.

We were asked to improve the structure of the report, and we tried our best to do that as well.

9.1 Appendix II; Code Used in the Project

```
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)

!unzip gdrive/MyDrive/EEE_485_Project/Data.zip > /dev/null

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time
import os,glob
from PIL import Image

"""# Dataset Loader"""

class Dataset_Loader:
    def __init__(self, path_list, dim):
        self.path = path_list
        self.dim = dim

    def flatten_pixels(self, list_pixels):
        return np.array(list_pixels).flatten().tolist()


    def identify_label(self, img_path):
        if self.path.index(img_path) == 0:
            label = 0     # Normal
        elif self.path.index(img_path) == 1:
            label = 1     # Squamous Cell Carcinoma
        elif self.path.index(img_path) == 2:
            label = 2     # Large Cell Carcinoma
        elif self.path.index(img_path) == 3:
            label = 3     # Adenocarcinoma
        return label

    def update_labels(self, updated_label_list, identified_label):
        return updated_label_list.append(identified_label)

    def load_data_as_dataframe(self):
        all_pixels=[]
        updated_label_list = []
        for img_path in self.path:
            os.chdir(img_path)
            for filename in glob.glob('*'):
                if (filename.endswith('.png')) or (filename.endswith('.jpg')) or (filename.endswith('.jpeg')):
                    img_pixels = []
                    identified_label = self.identify_label(img_path)
                    updated_label_list.append(identified_label)
                    img = Image.open(filename).convert('L').resize((self.dim,self.dim),Image.ANTIALIAS)
                    img_pixels = list(img.getdata())
                    all_pixels.append((self.flatten_pixels(img_pixels)))
        return pd.DataFrame(all_pixels), updated_label_list

"""# Organizing Data"""
```

```
train_path_list=["/content/Data/train/normal",
        "/content/Data/train/squamous.cell.carcinoma_left.hilum_T1_N2_M0_IIIa",
        "/content/Data/train/large.cell.carcinoma_left.hilum_T2_N2_M0_IIIa",
        "/content/Data/train/adenocarcinoma_left.lower.lobe_T2_N0_M0_Ib"]

train_data = Dataset_Loader(train_path_list,28)
df_train, train_labels = train_data.load_data_as_dataframe()

valid_path_list=["/content/Data/valid/normal",
        "/content/Data/valid/squamous.cell.carcinoma_left.hilum_T1_N2_M0_IIIa",
        "/content/Data/valid/large.cell.carcinoma_left.hilum_T2_N2_M0_IIIa",
        "/content/Data/valid/adenocarcinoma_left.lower.lobe_T2_N0_M0_Ib"]

valid_data = Dataset_Loader(valid_path_list,28)
df_valid, valid_labels = valid_data.load_data_as_dataframe()

test_path_list=["/content/Data/test/normal",
        "/content/Data/test/squamous.cell.carcinoma",
        "/content/Data/test/large.cell.carcinoma",
        "/content/Data/test/adenocarcinoma"]

test_data = Dataset_Loader(test_path_list,28)
df_test, test_labels = test_data.load_data_as_dataframe()

df_train['Labels']= train_labels
print('\n', df_train)

df_valid['Labels']= valid_labels
print('\n', df_valid)

df_test['Labels']= test_labels
print('\n', df_test)

DTrain = df_train.to_numpy()

DValid = df_valid.to_numpy()

DTest = df_test.to_numpy()

# Seperate Features (X) and Labels (Y)
trainX = np.delete(DTrain, -1, 1)
trainY = np.delete(DTrain, np.s_[0: -1], 1)
valX = np.delete(DValid, -1, 1)
valY = np.delete(DValid, np.s_[0: -1], 1)
testX = np.delete(DTest, -1, 1)
testY = np.delete(DTest, np.s_[0: -1], 1)
print(trainX.shape)

# Conversion to int for easy usage
trainX = trainX.astype(int)
print(trainX.shape)
trainY = trainY.astype(int)
valX = valX.astype(int)
valY = valY.astype(int)
```

```
testX = testX.astype(int)
testY = testY.astype(int)

"""# PCA

## PCA Functions and Imports
"""

#importing libraries
import os
import sys
import cv2 as cv
import numpy as np
import plotly.io as pio
import plotly.graph_objs as go

from PIL import Image
from skimage import color
from plotly import subplots

def covariance_matrix(X):
  X_mean_centered  = (X.T - np.mean(X,axis=1)).T
  X_cov = np.cov(X_mean_centered.T)
  return X_cov

def eigenvalues(X):
  X_cov = covariance_matrix(X)
  l, u = np.linalg.eig(X_cov)
  l = np.real(l)
  u = np.real(u)
  eigen_d = dict(zip(l,u.T))
  eigen_s  = sorted(eigen_d)[::-1]
  return eigen_d, eigen_s

def first_k_eigenvalues(X, k):
  eigen_d, eigen_s = eigenvalues(X)
  first_10_eigenvalues = eigen_s[:k]
  return first_10_eigenvalues

def PVE(X, k):
  k_eigenvalues = first_k_eigenvalues(X,k)
  eigen_d, eigen_s = eigenvalues(X)
  first_10_eigenvectors = []
  for e in k_eigenvalues:
      first_10_eigenvectors.append(eigen_d[e])
  eigen_sum = np.sum(eigen_s)
  first_10_pve = k_eigenvalues/eigen_sum
  return first_10_eigenvectors, eigen_sum, first_10_pve

def PCA_load_images(img_path, total, size = 64):
  X = []
  X_f = []
  count = 1
  for img in os.listdir(img_path):
      if count == total + 1:
```

```
            break
        sys.stdout.flush()
        img_array = cv.imread(os.path.join(img_path, img))
        img_pil = Image.fromarray(img_array)
        img_64x64 = np.array(img_pil.resize((size, size), Image.BILINEAR))
        X.append(img_64x64)
        img_array = img_64x64.reshape(4096,3)
        X_f.append(img_array)
        count += 1

    X_f = np.asarray(X_f)
    X_f.shape

    X_split = [np.squeeze(subarray) for subarray in np.dsplit(X_f, 3)]

    R = X_split[0]
    G = X_split[1]
    B = X_split[2]

    R_norm = ((R - np.amin(R))/(np.amax(R) - np.amin(R))).T
    G_norm = ((G - np.amin(G))/(np.amax(G) - np.amin(G))).T
    B_norm = ((B - np.amin(B))/(np.amax(B) - np.amin(B))).T

    return R_norm, G_norm, B_norm

def PCA(X, reduced_dim = 25):
    first_eigenvectors, eigen_sum, first_pve = PVE(X, reduced_dim)
    eigen_d, eigen_s = eigenvalues(X)

    first_nparray = np.array(first_eigenvectors)

    print(first_pve*100)
    print("Total PVE:")
    print(np.sum(first_pve*100))

    return first_nparray

def get_reduced_data(R_PC, G_PC, B_PC, total, reduced_dim = 25):
    R_PC_normalized = ((R_PC - np.amin(R_PC))/(np.amax(R_PC) - np.amin(R_PC)))
    G_PC_normalized = ((G_PC - np.amin(G_PC))/(np.amax(G_PC) - np.amin(G_PC)))
    B_PC_normalized = ((B_PC - np.amin(B_PC))/(np.amax(B_PC) - np.amin(B_PC)))

        RGB_PC = (np.stack((R_PC_normalized, G_PC_normalized, B_PC_normalized),
axis=0)).reshape((3*total),reduced_dim)

    return RGB_PC

def processingforplot(image) -> None:  # For image plots
    plt.figure()
    plt.imshow(image)

"""## PCA Training Split

### Class 0: Normal
"""
```

```
trn0_R, trn0_G, trn0_B = PCA_load_images('/content/Data/train/normal', total = 151)

print("\n For Red \n")
trn0_R_PC = (PCA(trn0_R)).T
print("\n For Green \n")
trn0_G_PC = (PCA(trn0_G)).T
print("\n For Blue \n")
trn0_B_PC = (PCA(trn0_B)).T

# Recombination
trn0_RGB_PC = get_reduced_data(trn0_R_PC, trn0_G_PC, trn0_B_PC, total = 151)

"""### Class 1: Squamous Cell Carcinoma"""

trn1_R, trn1_G, trn1_B = PCA_load_images('/content/Data/train/squamous.cell.carcinoma_left.hilum_T1_N2_M0_IIIa', total = 182)

print("\n For Red \n")
trn1_R_PC = (PCA(trn1_R)).T
print("\n For Green \n")
trn1_G_PC = (PCA(trn1_G)).T
print("\n For Blue \n")
trn1_B_PC = (PCA(trn1_B)).T

# Recombination
trn1_RGB_PC = get_reduced_data(trn1_R_PC, trn1_G_PC, trn1_B_PC, total = 182)

"""### Class 2: Large Cell Carcinoma"""

trn2_R, trn2_G, trn2_B = PCA_load_images('/content/Data/train/large.cell.carcinoma_left.hilum_T2_N2_M0_IIIa', total = 131)

print("\n For Red \n")
trn2_R_PC = (PCA(trn2_R)).T
print("\n For Green \n")
trn2_G_PC = (PCA(trn2_G)).T
print("\n For Blue \n")
trn2_B_PC = (PCA(trn2_B)).T

# Recombination
trn2_RGB_PC = get_reduced_data(trn2_R_PC, trn2_G_PC, trn2_B_PC, total = 131)

"""### Class 3: Adenocarcinoma"""

trn3_R, trn3_G, trn3_B = PCA_load_images('/content/Data/train/adenocarcinoma_left.lower.lobe_T2_N0_M0_Ib', total = 236)

print("\n For Red \n")
trn3_R_PC = (PCA(trn3_R)).T
print("\n For Green \n")
trn3_G_PC = (PCA(trn3_G)).T
print("\n For Blue \n")
trn3_B_PC = (PCA(trn3_B)).T
```

```
# Recombination
trn3_RGB_PC = get_reduced_data(trn3_R_PC, trn3_G_PC, trn3_B_PC, total = 236)

"""## PCA Validation Split

### Class 0: Normal
"""

val0_R, val0_G, val0_B = PCA_load_images('/content/Data/valid/normal', total = 32)

print("\n For Red \n")
val0_R_PC = (PCA(val0_R)).T
print("\n For Green \n")
val0_G_PC = (PCA(val0_G)).T
print("\n For Blue \n")
val0_B_PC = (PCA(val0_B)).T

# Recombination
val0_RGB_PC = get_reduced_data(val0_R_PC, val0_G_PC, val0_B_PC, total = 32)

"""### Class 1: Squamous Cell Carcinoma"""

val1_R, val1_G, val1_B = PCA_load_images('/content/Data/valid/squamous.cell.carcinoma_left.hilum_T1_N2_M0_IIIa', total = 39)

print("\n For Red \n")
val1_R_PC = (PCA(val1_R)).T
print("\n For Green \n")
val1_G_PC = (PCA(val1_G)).T
print("\n For Blue \n")
val1_B_PC = (PCA(val1_B)).T

# Recombination
val1_RGB_PC = get_reduced_data(val1_R_PC, val1_G_PC, val1_B_PC, total = 39)

"""### Class 2: Large Cell Carcinoma"""

val2_R, val2_G, val2_B = PCA_load_images('/content/Data/valid/large.cell.carcinoma_left.hilum_T2_N2_M0_IIIa', total = 28)

print("\n For Red \n")
val2_R_PC = (PCA(val2_R)).T
print("\n For Green \n")
val2_G_PC = (PCA(val2_G)).T
print("\n For Blue \n")
val2_B_PC = (PCA(val2_B)).T

# Recombination
val2_RGB_PC = get_reduced_data(val2_R_PC, val2_G_PC, val2_B_PC, total = 28)

"""### Class 3: Adenocarcinoma"""
```

```
val3_R, val3_G, val3_B = PCA_load_images('/content/Data/valid/adenocarcinoma_left.lower.lobe_T2_N0_M0_Ib', total = 51)

print("\n For Red \n")
val3_R_PC = (PCA(val3_R)).T
print("\n For Green \n")
val3_G_PC = (PCA(val3_G)).T
print("\n For Blue \n")
val3_B_PC = (PCA(val3_B)).T

# Recombination
val3_RGB_PC = get_reduced_data(val3_R_PC, val3_G_PC, val3_B_PC, total = 51)

"""## PCA Testing Split

### Class 0: Normal
"""

tst0_R, tst0_G, tst0_B = PCA_load_images('/content/Data/test/normal', total = 32)

print("\n For Red \n")
tst0_R_PC = (PCA(tst0_R)).T
print("\n For Green \n")
tst0_G_PC = (PCA(tst0_G)).T
print("\n For Blue \n")
tst0_B_PC = (PCA(tst0_B)).T

# Recombination
tst0_RGB_PC = get_reduced_data(tst0_R_PC, tst0_G_PC, tst0_B_PC, total = 32)

"""### Class 1: Squamous Cell Carcinoma"""

tst1_R, tst1_G, tst1_B = PCA_load_images('/content/Data/test/squamous.cell.carcinoma', total = 39)

print("\n For Red \n")
tst1_R_PC = (PCA(tst1_R)).T
print("\n For Green \n")
tst1_G_PC = (PCA(tst1_G)).T
print("\n For Blue \n")
tst1_B_PC = (PCA(tst1_B)).T

# Recombination
tst1_RGB_PC = get_reduced_data(tst1_R_PC, tst1_G_PC, tst1_B_PC, total = 39)

"""### Class 2: Large Cell Carcinoma"""

tst2_R, tst2_G, tst2_B = PCA_load_images('/content/Data/test/large.cell.carcinoma', total = 28)

print("\n For Red \n")
tst2_R_PC = (PCA(tst2_R)).T
print("\n For Green \n")
tst2_G_PC = (PCA(tst2_G)).T
print("\n For Blue \n")
tst2_B_PC = (PCA(tst2_B)).T
```

```
# Recombination
tst2_RGB_PC = get_reduced_data(tst2_R_PC, tst2_G_PC, tst2_B_PC, total = 28)

"""### Class 3: Adenocarcinoma"""

tst3_R, tst3_G, tst3_B = PCA_load_images('/content/Data/test/adenocarcinoma', total = 51)

print("\n For Red \n")
tst3_R_PC = (PCA(tst3_R)).T
print("\n For Green \n")
tst3_G_PC = (PCA(tst3_G)).T
print("\n For Blue \n")
tst3_B_PC = (PCA(tst3_B)).T

# Recombination
tst3_RGB_PC = get_reduced_data(tst3_R_PC, tst3_G_PC, tst3_B_PC, total = 51)

"""# Rearranging Data"""

def new_data(D0_RGB_PC, D1_RGB_PC, D2_RGB_PC, D3_RGB_PC):
  return np.vstack((D0_RGB_PC, D1_RGB_PC, D2_RGB_PC, D3_RGB_PC))

def new_class_labels(Dx_RGB_PC, label):
  Y = []
  for x in Dx_RGB_PC:
    Y.append(label)
  Y = np.asarray(Y)

  return Y

def new_labels(D0_Y, D1_Y, D2_Y, D3_Y):
  return np.concatenate((D0_Y, D1_Y, D2_Y, D3_Y))

# PCA Reduced Dataset with Colour Channel Information
trainPCX = new_data(trn0_RGB_PC, trn1_RGB_PC, trn2_RGB_PC, trn3_RGB_PC)
valPCX = new_data(val0_RGB_PC, val1_RGB_PC, val2_RGB_PC, val3_RGB_PC)
testPCX = new_data(tst0_RGB_PC, tst1_RGB_PC, tst2_RGB_PC, tst3_RGB_PC)

# Per Class Labels for Colour Channel Enhanced Data
trn0_Y = new_class_labels(trn0_RGB_PC, 0)
trn1_Y = new_class_labels(trn1_RGB_PC, 1)
trn2_Y = new_class_labels(trn2_RGB_PC, 2)
trn3_Y = new_class_labels(trn3_RGB_PC, 3)

val0_Y = new_class_labels(val0_RGB_PC, 0)
val1_Y = new_class_labels(val1_RGB_PC, 1)
val2_Y = new_class_labels(val2_RGB_PC, 2)
val3_Y = new_class_labels(val3_RGB_PC, 3)

tst0_Y = new_class_labels(tst0_RGB_PC, 0)
tst1_Y = new_class_labels(tst1_RGB_PC, 1)
tst2_Y = new_class_labels(tst2_RGB_PC, 2)
tst3_Y = new_class_labels(tst3_RGB_PC, 3)

# PCA Reduced Labels with Colour Channel Information
```

```
trainPCY = new_labels(trn0_Y, trn1_Y, trn2_Y, trn3_Y)
valPCY = new_labels(val0_Y, val1_Y, val2_Y, val3_Y)
testPCY = new_labels(tst0_Y, tst1_Y, tst2_Y, tst3_Y)

"""# One-vs-All Logistic Regression"""

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

# Log loss
def cost(theta, x, y):
   h = sigmoid(x @ theta)
   m = len(y)
   cost = 1 / m * np.sum(-y * np.log(h) - (1 - y) * np.log(1 - h))
   grad = 1 / m * ((y - h) @ x)
   return cost, grad

def fit(x, y, max_iter=10000, alpha=0.1):
   x = np.insert(x, 0, 1, axis=1)
   thetas = []
   classes = np.unique(y)
   costs = np.zeros(max_iter)

   for c in classes:
      # one vs. rest binary classification
      binary_y = np.where(y == c, 1, 0)

      theta = np.zeros(x.shape[1])
      for epoch in range(max_iter):
         costs[epoch], grad = cost(theta, x, binary_y)
         theta += alpha * grad

      thetas.append(theta)
   return thetas, classes, costs

def predict(classes, thetas, x):
   x = np.insert(x, 0, 1, axis=1)
   preds = [np.argmax(
      [sigmoid(xi @ theta) for theta in thetas]
   ) for xi in x]
   return [classes[p] for p in preds]

def score(classes, theta, x, y):
   return (predict(classes, theta, x) == y).mean()

np.random.seed(34)

"""Finding Optimal Number of Epochs"""

for epoch in range(2500, 10001, 2500):
  print("\n Validating with learning rate 0.1, and with", epoch, "epochs: \n")

  thetas, classes, costs = fit(trainPCX[:, 2:], trainPCY, max_iter=epoch, alpha=0.1)
  plt.plot(costs)
  plt.title("Validation Run Loss Plot")
```

```
  plt.xlabel('Number Epochs'); plt.ylabel('Loss');
  plt.show()

"""Finding Optimal Learning Rate"""

lr = 0.1
while (lr > 0.0001):
  print("\n Validating with learning rate", lr, "and with 10000 epochs \n")

  thetas, classes, costs = fit(trainPCX[:, 2:], trainPCY, max_iter=10000, alpha=lr)
  plt.plot(costs)
  plt.title("Validation Run Loss Plot")
  plt.xlabel('Number Epochs'); plt.ylabel('Loss');
  plt.show()

  lr *= 10**-1

thetas, classes, costs = fit(trainPCX, trainPCY, max_iter=10000, alpha=0.1)
print("Train Accuracy:", score(classes, thetas, trainPCX, trainPCY)*100)
print("Validation Accuracy:", score(classes, thetas, valPCX, valPCY)*100)

thetas, classes, costs = fit(trainPCX, trainPCY, max_iter=10000, alpha=0.1)
print("Train Accuracy:", score(classes, thetas, trainPCX, trainPCY)*100)
print("Test Accuracy:", score(classes, thetas, testPCX, testPCY)*100)

"""# kNN"""

def euclidean_dist(x1, x2):
    d = np.square(x1 - x2)  # (ai-bi)**2 for every point in the vectors
    d = np.sum(d) # adds all values
    d = np.sqrt(d)
    return d

def distance_from_all_training(testP):
    dist_array = np.array([])
    for trainP in trainPCX:
        dist = euclidean_dist(testP, trainP)
        dist_array = np.append(dist_array, dist)
    return dist_array

def KNNClassifier(trainX, trainY, testY, k = 5):
    predictions = np.array([])
    trainY = trainY.reshape(-1,1)
    for test_point in testY: # iterating through every test data point
            dist_array = distance_from_all_training(test_point).reshape(-1,1) # calculating distance from
every training data instance
        neighbors = np.concatenate((dist_array, trainY), axis = 1)
            neighbors_sorted = neighbors[neighbors[:, 0].argsort()] # sorts training points on the basis of
distance
        k_neighbors = neighbors_sorted[:k] # selects k-nearest neighbors
        frequency = np.unique(k_neighbors[:, 1], return_counts=True)
        target_class = frequency[0][frequency[1].argmax()] # selects label with highest frequency
        predictions = np.append(predictions, target_class)

    return predictions
```

```
def accuracy(y_test, y_pred):
    total_correct = 0
    for i in range(len(y_test)):
        if int(y_test[i]) == int(y_pred[i]):
            total_correct += 1
    acc = total_correct/len(y_test)
    return acc

def kNN_run(trainingX, trainingY, testingX, testingY, k):
  dist_array = distance_from_all_training(testingX[0])
  test_predictions = KNNClassifier(trainingX, trainingY, testingX, k)
  acc = accuracy(testingY, test_predictions)
  print('kNN accuracy =', acc*100, ', for k =', k)
  return acc*100

def knn_val_run(trainingX, trainingY, testingX, testingY, max_k = 25):
  val_acc = []
  for k in range(max_k):
    k += 1
    k_val_acc = kNN_run(trainingX, trainingY, testingX, testingY, k)
    val_acc.append(k_val_acc)
  best_acc = max(val_acc)
  best_k = val_acc.index(best_acc) + 1
  print("\n Best kNN accuracy =", best_acc, ", for k =", best_k)
  return val_acc, best_k

"""Validation Runs using k = [1,25]"""

val_acc, best_k = knn_val_run(trainPCX, trainPCY, valPCX, valPCY)

plt.plot(val_acc, marker='x')
plt.xticks(range(0,len(val_acc)+1, 1))
plt.ylabel('Accuracy')
plt.xlabel('Number of k Nearest Neighbours Used')
plt.title("kNN Validation Accuracy")
plt.show()

"""Testing Run with best k = 9"""

kNN_test_acc = kNN_run(trainPCX, trainPCY, valPCX, valPCY, best_k)

"""# CNN"""

import numpy as np

class ConvolutionLayer:
    def __init__(self, kernel_num, kernel_size):
        """
          Constructor takes as input the number of kernels and their size. I assume only squared filters of
size kernel_size x kernel_size
        """
        self.kernel_num = kernel_num
        self.kernel_size = kernel_size
```

```
        # Generate random filters of shape (kernel_num, kernel_size, kernel_size). Divide by kernel_size^2 for weight normalization
        self.kernels = np.random.randn(kernel_num, kernel_size, kernel_size) / (kernel_size**2)

    def patches_generator(self, image):
        """
        Divide the input image in patches to be used during convolution.
        Yields the tuples containing the patches and their coordinates.
        """
        # Extract image height and width
        image_h, image_w = image.shape
        self.image = image
        # The number of patches, given a fxf filter is h-f+1 for height and w-f+1 for width
        for h in range(image_h-self.kernel_size+1):
            for w in range(image_w-self.kernel_size+1):
                patch = image[h:(h+self.kernel_size), w:(w+self.kernel_size)]
                yield patch, h, w

    def forward_prop(self, image):
        """
        Perform forward propagation for the convolutional layer.
        """
        # Extract image height and width
        image_h, image_w = image.shape
        # Initialize the convolution output volume of the correct size
        convolution_output = np.zeros((image_h-self.kernel_size+1, image_w-self.kernel_size+1, self.kernel_num))
        # Unpack the generator
        for patch, h, w in self.patches_generator(image):
            # Perform convolution for each patch
            convolution_output[h,w] = np.sum(patch*self.kernels, axis=(1,2))
        return convolution_output

    def back_prop(self, dE_dY, alpha):
        """
        Takes the gradient of the loss function with respect to the output and computes the gradients of the loss function with respect
        to the kernels' weights.
        dE_dY comes from the following layer, typically max pooling layer.
        It updates the kernels' weights
        """
        # Initialize gradient of the loss function with respect to the kernel weights
        dE_dk = np.zeros(self.kernels.shape)
        for patch, h, w in self.patches_generator(self.image):
            for f in range(self.kernel_num):
                dE_dk[f] += patch * dE_dY[h, w, f]
        # Update the parameters
        self.kernels -= alpha*dE_dk
        return dE_dk

class MaxPoolingLayer:
    def __init__(self, kernel_size):
        """
        Constructor takes as input the size of the kernel
        """
```

```
        self.kernel_size = kernel_size

    def patches_generator(self, image):
        """
        Divide the input image in patches to be used during pooling.
        Yields the tuples containing the patches and their coordinates.
        """
        # Compute the ouput size
        output_h = image.shape[0] // self.kernel_size
        output_w = image.shape[1] // self.kernel_size
        self.image = image

        for h in range(output_h):
            for w in range(output_w):
                                patch = image[(h*self.kernel_size):(h*self.kernel_size+self.kernel_size),
(w*self.kernel_size):(w*self.kernel_size+self.kernel_size)]
                yield patch, h, w

    def forward_prop(self, image):
        image_h, image_w, num_kernels = image.shape
                max_pooling_output = np.zeros((image_h//self.kernel_size, image_w//self.kernel_size,
num_kernels))
        for patch, h, w in self.patches_generator(image):
            max_pooling_output[h,w] = np.amax(patch, axis=(0,1))
        return max_pooling_output

    def back_prop(self, dE_dY):
        """
          Takes the gradient of the loss function with respect to the output and computes the gradients of
the loss function with respect
        to the kernels' weights.
        dE_dY comes from the following layer, typically softmax.
              There are no weights to update, but the output is needed to update the weights of the
convolutional layer.
        """
        dE_dk = np.zeros(self.image.shape)
        for patch,h,w in self.patches_generator(self.image):
            image_h, image_w, num_kernels = patch.shape
            max_val = np.amax(patch, axis=(0,1))

            for idx_h in range(image_h):
                for idx_w in range(image_w):
                    for idx_k in range(num_kernels):
                        if patch[idx_h,idx_w,idx_k] == max_val[idx_k]:
                                        dE_dk[h*self.kernel_size+idx_h, w*self.kernel_size+idx_w, idx_k] =
dE_dY[h,w,idx_k]
            return dE_dk

class SoftmaxLayer:
    """
      Takes the volume coming from convolutional & pooling layers. It flattens it and it uses it in the
next layers.
    """
    def __init__(self, input_units, output_units):
        # Initiallize weights and biases
```

```
        self.weight = np.random.randn(input_units, output_units)/input_units
        self.bias = np.zeros(output_units)

    def forward_prop(self, image):
        self.original_shape = image.shape # stored for backprop
        # Flatten the image
        image_flattened = image.flatten()
        self.flattened_input = image_flattened # stored for backprop
        # Perform matrix multiplication and add bias
        first_output = np.dot(image_flattened, self.weight) + self.bias
        self.output = first_output
        # Apply softmax activation
        softmax_output = np.exp(first_output) / np.sum(np.exp(first_output), axis=0)
        return softmax_output

    def back_prop(self, dE_dY, alpha):
        for i, gradient in enumerate(dE_dY):
            if gradient == 0:
                continue
            transformation_eq = np.exp(self.output)
            S_total = np.sum(transformation_eq)

            # Compute gradients with respect to output (Z)
            dY_dZ = -transformation_eq[i]*transformation_eq / (S_total**2)
            dY_dZ[i] = transformation_eq[i]*(S_total - transformation_eq[i]) / (S_total**2)

            # Compute gradients of output Z with respect to weight, bias, input
            dZ_dw = self.flattened_input
            dZ_db = 1
            dZ_dX = self.weight

            # Gradient of loss with respect ot output
            dE_dZ = gradient * dY_dZ

            # Gradient of loss with respect to weight, bias, input
            dE_dw = dZ_dw[np.newaxis].T @ dE_dZ[np.newaxis]
            dE_db = dE_dZ * dZ_db
            dE_dX = dZ_dX @ dE_dZ

            # Update parameters
            self.weight -= alpha*dE_dw
            self.bias -= alpha*dE_db

            return dE_dX.reshape(self.original_shape)

def CNN_forward(image, label, layers):
    image = ((image - np.amin(image))/(np.amax(image) - np.amin(image)))
    output = image
    for layer in layers:
        output = layer.forward_prop(output)
    # Compute loss (cross-entropy) and accuracy
    loss = -np.log(output[label])
    accuracy = 1 if np.argmax(output) == label else 0
    return output, loss, accuracy
```

```
def CNN_backprop(gradient, layers, alpha=0.05):
    grad_back = gradient
    for layer in layers[::-1]:
        if type(layer) in [ConvolutionLayer, SoftmaxLayer]:
            grad_back = layer.back_prop(grad_back, alpha)
        elif type(layer) == MaxPoolingLayer:
            grad_back = layer.back_prop(grad_back)
    return grad_back


def CNN_training(image, label, layers, alpha=0.05):
    # Forward step
    output, loss, accuracy = CNN_forward(image, label, layers)

    # Initial gradient with of length 4 because we have 4 outputs
    gradient = np.zeros(4)
    gradient[label] = -1/output[label]

    # Backprop step
    gradient_back = CNN_backprop(gradient, layers, alpha)

    return loss, accuracy

import numpy as np
from matplotlib import pyplot as plt


def train_main_method(alpha):

    train_loss = []
    train_acc = []

    global trainX
    global trainY

    # Define the network
    layers = [
    ConvolutionLayer(16,3), # layer with 16 3x3 filters, output (62,62,16)
    MaxPoolingLayer(2), # pooling layer 2x2, output (31,31,16)
    SoftmaxLayer(31*31*16, 4) # softmax layer with 31*31*16 input and 4 outputs
    ]
    print("================================================")
    print("Starting Training for learning rate: ", alpha)
    print("================================================")

    for epoch in range(4):
        print('Epoch {} ->'.format(epoch+1))
        # Shuffle training data
        permutation = np.random.permutation(len(trainX))
        trainX = trainX[permutation]
        trainY = trainY[permutation]
        # Training the CNN
        loss = 0
        accuracy = 0
        for i, (image, label) in enumerate(zip(trainX, trainY)):
```

```
        label = label[0]
        image = image.reshape(64, 64)

        if i>0 and i % 100 == 0: # Every 100 instance print and restart loss and accuracy
            print("Step {}. For the last 100 steps: average loss {}, accuracy {}".format(i+1, loss/100,
accuracy))
            train_loss.append(loss)
            train_acc.append(accuracy)
            loss = 0
            accuracy = 0

        loss_1, accuracy_1 = CNN_training(image, label, layers, alpha)
        loss += loss_1
        accuracy += accuracy_1

    plt.figure()
    plt.xlabel("Per 100 Steps")
    plt.ylabel("Loss")
    plt.legend
    plt.title("Loss Vs Epochs")
    plt.plot(train_loss)
    plt.show

    plt.figure()
    plt.xlabel("Per 100 Steps")
    plt.ylabel("Accuracy")
    plt.legend
    plt.title("Accuracy vs Epochs")
    plt.plot(train_acc)
    plt.show()

    train_loss.clear()
    train_acc.clear()

def validate_cnn(alpha):
  accuracy = 0
  test_loss = []
  test_acc = []

  train_main_method(alpha)

  layers = [
  ConvolutionLayer(16,3), # layer with 16 3x3 filters, output (62,62,16)
  MaxPoolingLayer(2), # pooling layer 2x2, output (31,31,16)
  SoftmaxLayer(31*31*16, 4) # softmax layer with 31*31*16 input and 4 outputs
  ]

  print("================================================")
  print("Starting Validation for learning rate: ", alpha)
  print("================================================")

  for i, (image, label) in enumerate(zip(valX, valY)):
    label = label[0]
    image = image.reshape(64,64)
    _ , loss, acc = CNN_forward(image, label, layers)
```

```
        if i>0 and i % 24 == 0: # Every 24 instance print and restart loss and accuracy
                print("Step {}. For the last 24 steps: average loss {}, accuracy {}".format(i+1, loss/24,
accuracy))

        test_loss.append(loss)
        accuracy += acc
        test_acc.append(accuracy)

    plt.figure()
    plt.xlabel("Steps")
    plt.ylabel("Accuracy")
    plt.plot(test_acc)
    plt.show()

    plt.figure()
    plt.xlabel("Steps")
    plt.ylabel("loss")
    plt.legend()
    plt.plot(test_loss)
    plt.show()

alpha_list = [0.0001, 0.001, 0.01, 0.1]

for alpha in alpha_list:
    validate_cnn(alpha)

layers = [
ConvolutionLayer(16,3), # layer with 16 3x3 filters, output (62,62,16)
MaxPoolingLayer(2), # pooling layer 2x2, output (31,31,16)
SoftmaxLayer(31*31*16, 4) # softmax layer with 31*31*16 input and 4 outputs
]
accuracy = 0
test_loss = []
test_acc = []

train_main_method(0.01)

for i, (image, label) in enumerate(zip(testX, testY)):
    label = label[0]
    image = image.reshape(64,64)
    _ , loss, acc = CNN_forward(image, label, layers)

    test_loss.append(loss)
    accuracy += acc
    test_acc.append(accuracy)

plt.figure()
plt.xlabel("Steps")
plt.ylabel("Accuracy")
plt.title("Accuracy vs Steps")
plt.plot(test_acc)
plt.show()

print("The final testing accuracy is: ", test_acc[-1])
```

```
plt.figure
plt.xlabel("Steps")
plt.ylabel("Loss")
plt.title("Loss Vs Steps")
plt.plot(test_loss)
plt.show()
```