Image Processing Covid-CT Scans

Jalen Lee, Collin Chan, Madelyn Joe

Cover Letter

In this paper, we perform experimental studies and develop machine learning-based diagnosis models of COVID-19. Using our dataset of 746 COVID-CT scans from real-life patients, we create diagnosis methods based on supervised learning, specifically Linear Discriminant Analysis (LDA), Quadratic Discriminant Analysis (QDA), and Logistic Regression, and achieve an accuracy of 0.685, 0.78, 0.689. In the process of collecting these results, we utilized Cholesky Decomposition and Parallel Computing to optimize our classification methods, so we could obtain computation times of 10 min, 12 min, and 3 min. This is important because CT scans are often used to diagnose COVID-19, especially in cases where the results from tests like PCR are inconclusive. We are aiming to improve the accuracy and speed of COVID-19 diagnosis, which is crucial in managing the pandemic. We can also help identify patterns and characteristics of COVID-19 on CT scans that may not be visible to the human eye. Given the global increase in Covid-19 cases, the findings presented in our paper will appeal to healthcare professionals, COVID-19 patients, and those who are interested in image processing or machine learning methods in health. Our findings will allow readers to understand the factors and processes that go into classifying COVID-19 and develop more cost-effective procedures.

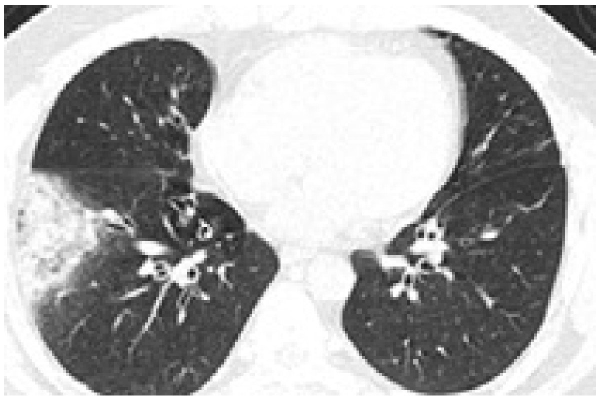
Introduction

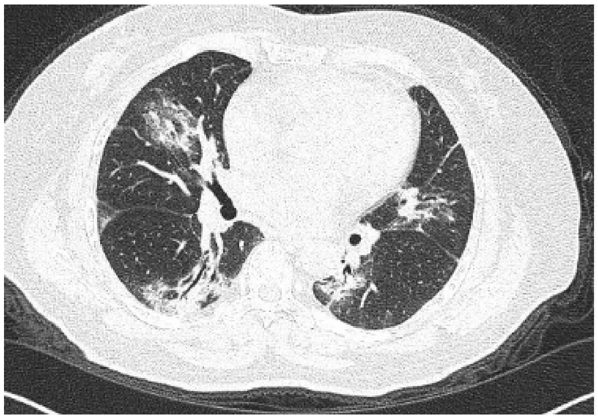
The World Health Organization has defined COVID-19 as a contagious, communicable, and fast-spreading disease that has largely affected the entire world in health, mental, and even lifestyle aspects. Since it has made such a large impact on our lives, it is important that we try to take control of this disease as there have already been 761,071,826 confirmed cases worldwide and 6,879,677 total deaths. In addition to wearing masks, social distancing, taking PCR tests, and getting vaccines, CT scans are an alternative solution to diagnosing the disease and saving lives. A Covid-CT scan is a specialized type of computed tomography (CT) scan used to identify and treat Covid-19, the illness brought on by the recent coronavirus (SARS-CoV-2). It is a medical imaging technique that uses X-rays to produce detailed cross-sectional images of the body. COVID-CT scans are crucial because they can quickly and accurately identify COVID-19 in individuals who are displaying symptoms like coughing, fever, and breathing difficulties. The scans can show lung abnormalities, like ground-glass opacities and consolidation, that is indicative of COVID-19.

Description of the Dataset

The dataset used in this project was found in a UCSD Github where these COVID-CT scans were from COVID-19-related papers from medRxiv, bioRxiv, NEJM, JAMA, and Lancet ranging from 2009-2021. The dataset includes 349 Covid-positive CT scans and 397 Covid-negative CT scans from patients from a variety of hospitals which were selected by reading figure captions in the papers. The images consist of chest scans that mainly picture the heart and lungs. Within the lung regions, the Covid containing lung regions occupy a small proportion with a focus on a pattern of thickened lung tissue with interlacing lines of increased density and ground glass opacities, hazy areas, consolidation, or distribution of abnormalities that could all indicate severe inflammation or damage to lung tissue. Some possible discrepancies include scans with different-sized hearts or lungs or people with larger body masses that could account for larger outer regions of the scan. This is why additional information like age, gender, weight, BMI, and past health history could be helpful factors to consider in our binary classification process. Also, since the images were taken from paper instead of our data being the original images, the quality of the original CT scans degrade as they are printed, which may reduce the precision of the diagnosis. The accuracy and bits per pixel are decreased, and the image resolution is decreased as a result of the quality reduction. Though this could be a concern to our accuracy, it should be noted that experienced radiologists are able to make an accurate diagnosis from low-quality images like smartphones. The difference in picture quality between original CT scans and those in papers should not significantly reduce the accuracy of diagnosis, but there could always be room for improvement.

Covid Positive CT Scans



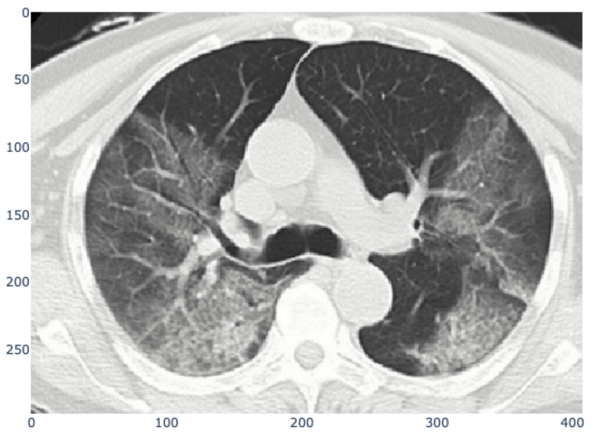


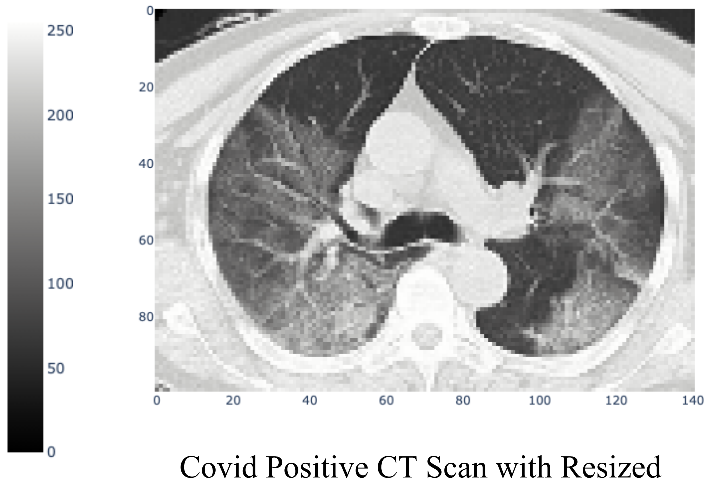
Covid Negative CT Scans

Data Processing

With our total of 746 images, there was an issue of inconsistency within sizing as some images would be smaller or larger than others and some would be rectangular or square. To address this issue and save RAM space, we used the OpenCV package to scale the images to (141, 100). This value was based on a percentage of the median of all image sizes. While decreasing our image dimensions did unfortunately decrease the quality of the picture and may contribute to an accuracy decrease, we had to consider the number of images needed to run through our model and overall computation time. We also decided to grayscale our images since CT scans are already black and white and for consistency reasons because random images had colored writing or arrows. Grayscaling the images also made it easier to process and analyze compared to colored images which reduce the complexity of processing algorithms, helping with data compression especially because we are working with such large data, noise reduction, and compatibility. We then got all the pixel values of the images in a matrix and looked at the dimensions of the images. Then, we reshaped the values into one large column, so each of the 746 images would be represented by a single column. These columns were further used as inputs to our classification models.

Heatmaps for Before and After Resizing Images Pixel Intensity Values


Covid Positive CT Scan with Original Dimensions (408, 298)


Covid Positive CT Scan with Resized Dimensions (141, 100)

Algorithms

Cholesky Decomposition can be applied to a matrix A if the matrix is symmetrical and positive definite. When A is decomposed via Cholesky, it results in $L \times L^T$ where L means a lower triangular matrix. This is useful since we can apply triangular solve for inverting which saves half the time compared to a regular inversion of a matrix. In this project, Cholesky will be used to decompose the covariance matrix in order to compute the LDA and QDA formulas. A covariance matrix, in theory, should always be symmetrical and positive definite so it would be safe to apply Cholesky decomposition.

Parallel computing was also used to take multiple different sets of training and testing data and to classify CT images for Covid Positive and Negative simultaneously. However, due to memory issues, the number of samples taken was not a huge number. Also based on how our code worked, the samples of training and testing data were not the same for all different methods of classification. This means that the data is not consistent with all the methods.

Proposed Methods

Linear Discriminant Analysis

Assumptions for LDA are independent measurements taken, normality in our data, and equal covariance for both classification groups. For independence, we can't tell for sure since the images were captured by captions on the research paper. We can deduce that it was randomly sampled since the gathering of this data can not be controlled if a certain paper had certain results which would lean towards independent measurements being taken. As for normality, this assumption was violated since this spread of variables is not normally distributed which makes sense as image borders would mostly be all black whereas the center is typically all white from the individual's body parts making the different predictors lower or higher peaks. We know that the equal covariance between the two groups is violated since QDA and LDA do lead to significantly different results, but LDA was done here for the purpose of applying Cholesky decomposition and serving as a baseline of results. Violations of these assumptions will lead to a lower accuracy rate, but we will go on and compare with logistic regression to see how our models performed.

Both LDA and QDA are considered generative models. This means that the label of Covid positive or negative is needed in order to build a classification model. The formula for calculating classification by LDA is given as

$$\text{sign}\left(2(\mu_1 - \mu_{-1})^T \Sigma^{-1} \mathbf{X} + \mu_{-1}^T \Sigma^{-1} \mu_{-1} - \mu^T \Sigma^{-1} \mu + 2 \log\left(\frac{P(Y=1)}{P(Y=-1)} \right) \right)$$

Where sign in this context would classify the CT scan to be positive or negative with covid. If the value inside sign( ) is over 0 then the image would be classified as positive. If the value inside sign( ) is under 0 then the image would be classified as negative. μ_1 and μ_{-1} would mean the row averages of every image column of the covid CT scans for the positive and negative groups respectively. Σ would mean the covariance matrix of all image columns. **X** would be the input or the test value for the equation, in this context the image column, which is reshaping the matrix into one long column by stacking all columns of the matrix on top of each other, would be inserted as **X.** The probability of Y = 1 and Y = -1 are the number of training samples from the different labels

As mentioned before, the covariance matrix should, in theory, be symmetrical and positive definite. I say this in theory because as the matrix column gets bigger in value as the image matrix gets increased, this means that we have significantly more predictors than sample size in our model. How we remedied this problem was we added a little bias by adding a penalty of λ times the trace of the matrix over the size of the matrix to our covariance matrix to keep it positive and definite in order to apply Cholesky decomposition. This way we can have a bigger covariance matrix and increasing the size of the image matrix leads to better accuracy.

This leads us to how we optimized this classification model. The two parts we can speed up in LDA are $\mu_{-1}^T \Sigma^{-1} \mu_{-1}$ and $\mu^T \Sigma^{-1} \mu$. Without loss of generality, let's denote the two different means as M. After applying Cholesky on the covariance matrix we have

$$M^T (L L^T)^{-1} M$$

$$M^T (L^T)^{-1} L^{-1} M$$

$(L^{-1} M)^T L^{-1} M$When the inverse is distributed and transposed is factored this is the equation we are left with. Notice that instead of solving for all four components we can solve the ones on the right of L^{-1} M and then take the transpose of it and multiply them together and it would yield the same result. Moreover, L has the special property of being a lower triangular matrix. Compared to the speeds of a regular matrix, solving this takes only half the time.

Another portion where Cholesky decomposition was used was $2(\mu_1 - \mu_{-1})^T \Sigma^{-1} \mathbf{X}$. Since Cholesky already was taken for the optimizations of the parts prior it won't take additional time to decompose it. Again substituting the inverse of the covariance matrix with $(L L^T)^{-1}$ gives

$$2(\mu_1 - \mu_{-1})^T (L L^T)^{-1} \mathbf{X}$$

$$2(\mu_1 - \mu_{-1})^T (L^T)^{-1} L^{-1} \mathbf{X}$$

When a lower triangular matrix is transposed, this gives an upper triangular matrix. This part can then be solved with two triangular solutions, one lower and one upper.

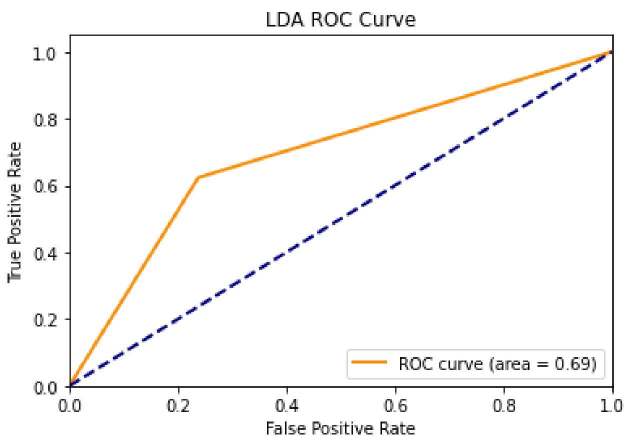
	Actual Positive	Actual Negative
Predicted Positive	0.5942857142857143	.235
Predicted Negative	0.40571428571	0.765

Total Accuracy: 0.6853333333333333

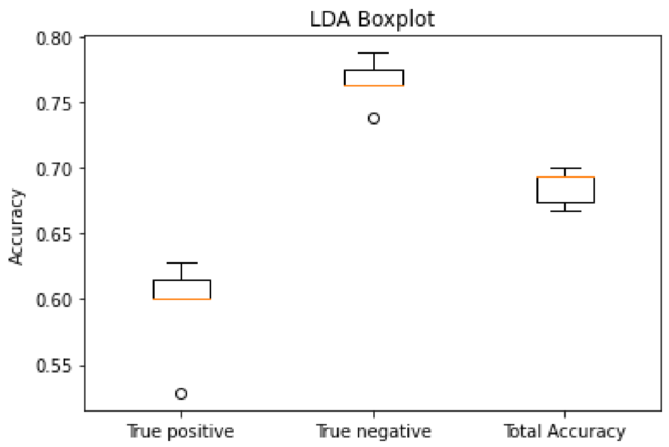
True Positive Variance: 0.0011918367346938775

True Negative Variance: 0.0002749999999999995

True Accuracy Variance: 0.0001671111111111126



LDA ROC Curve results were from 1 trial with AUC of 0.69



LDA Boxplots for Accuracy of True Positive, True Negative, and Total Accuracy Rates

After taking a few samples of different training and testing data we averaged the different accuracy rates and found how much variance there was between all the samples. In this batch, there seems to be a low variance of the accuracy of rates In LDA there is a low true positive rate. In this context, a higher true positive rate is preferred over a true negative rate since it is more important to determine if an individual actually has Covid or not. If an individual is determined to not have Covid by LDA and actually has Covid, in reality, will potentially lead to the spreading of the disease whereas if an individual is determined to have Covid by LDA and in reality not have it, it might potentially cause panic to the individual but not physically hurt the rest of the population. The ROC curve is a useful tool for comparing different binary classification models as it helps visualize their trade-offs between true positives and false positives at different threshold values. From looking at the LDA ROC curve, we can assume our model does not perform the best because the curve is not hugging the top left corner of the plot, but it adequately correctly classifies all positive and negative instances. It performs decently, but it could be better. With an AUC of 0.5 indicating random guessing and an AUC of 1 indicating perfect performance, our AUC of 0.69 means it is relatively successful at diagnosing CT scans accurately.

Quadratic Discriminant Analysis

The assumptions for QDA are independent measurements taken and normality in our data. These are the same assumptions that LDA had and had already talked about these issues above.

The formula to calculating QDA is given by

$$\text{sign}\left(\log\left(\frac{|\Sigma_{-1}|}{|\Sigma_{1}|}\right) + (\mathbf{X} - \mu_{-1})^T \Sigma_{-1}^{-1} (\mathbf{X} - \mu_{-1}) - (\mathbf{X} - \mu_{1})^T \Sigma_{1}^{-1} (\mathbf{X} - \mu_{1}) + 2\log\left(\frac{P(Y=1)}{P(Y=-1)}\right)\right)$$

This formula is very similar to the previous formula with some slight tweaks. Σ_{-1} and Σ_1 take the place of the single covariance matrix in LDA. These symbols represent the covariance of the two different groups of negative CT and positive CT respectively. Additionally, the new symbol introduced $|\Sigma_{-1}|$ and $|\Sigma_1|$ means to the determinant of the two covariance matrices. Other than that the different symbols retain their old meaning in the previous equation for LDA.

Similar to LDA, QDA also runs into problems of needing to regularize the covariance matrices in order to increase the image matrix and image column. For both positive and negative covariance matrices, we remedied this by adding a little bias by adding a penalty of λ times the trace of the matrix over the size of the matrix to our covariance matrix to keep it positive and definite.

A new problem unique to QDA which LDA does not have was overflow issues when calculating the log of the quotients of determinants since this isn't calculated in LDA when the model assumes that both covariances are equivalent. When taking the inverse of the covariance and then the determinant of that often leads to the result being infinity along with a long computation time.

$$\log\left(\frac{|\Sigma_{-1}|}{|\Sigma_{1}|}\right)$$

A clever way to solve this problem was to use the Cholesky decomposition as this is already used for the purpose of optimization. As a reminder, we can split up the determinant of a product as the following

$$\det(A) = \det(L\, L^T) = \det(L)\, \det(L^T)$$

Then we recognize that the determinant of a triangular matrix is simply the product of the diagonal values. Taking the transpose of a lower triangular matrix does not affect the diagonal values. This gives

$$\det(L) = \det(L^T) = \boldsymbol{\Pi}\, (\mathrm{diag}(L_{(i)}))$$

Substituting this knowledge our updated formula for the log of determinants parts will look like

$$\log(\, \boldsymbol{\Pi}\, (\mathrm{diag}(L_{(i)}^{(-1)}))^2 \,/\, \boldsymbol{\Pi}\, (\mathrm{diag}(L_{(i)}^{(1)}))^2)$$

Where the numerator $L^{(-1)}$ represents the lower triangular matrix of the negative covariance and the denominator $L^{(1)}$ being the positive covariance. Then the three different following log properties

$$\log(a \times b) = \log(a) + \log(b)$$

$$\log(\tfrac{a}{b}) = \log(a) - \log(b)$$

$$\log(a^b) = b\, \log(a)$$

can all be applied to $\log(\, \boldsymbol{\Pi}\, (\mathrm{diag}(L_{(i)}^{(-1)}))^2 \,/\, \boldsymbol{\Pi}\, (\mathrm{diag}(L_{(i)}^{(1)}))^2)$ giving

$$\boldsymbol{\Sigma}\, 2 \log(\, \mathrm{diag}(L_{(i)}^{(-1)})) - \boldsymbol{\Sigma}\, 2 \log(\, \mathrm{diag}(L_{(i)}^{(1)}))$$

This equation means to take the sum of 2 times the log of the ith diagonal value of the lower triangular matrix of the negative covariance minus the sum of 2 times the log of the ith diagonal value of the lower triangular matrix of the positive covariance.

Instead of taking many cofactors and multiplying many times leading to an overflow error resulting in infinity in our formula which would in turn classify everything as positive for Covid, this mathematical identity takes the sum of log values effectively solving the overflow issue and speeding up computation speed.

Again, in QDA there is a similar term $(\mathbf{X} - \mu_{-1})^T\, \Sigma_{-1}^{-1}\, (\mathbf{X} - \mu_{-1})$ and $(\mathbf{X} - \mu_1)^T\, \Sigma_1^{-1}\, (\mathbf{X} - \mu_1)$ compared to LDA. Without loss of generality, denote M as the input minus the respective group means M

$$M = (\mathbf{X} - \mu_{-1}) \text{ and } (\mathbf{X} - \mu_1)$$

Once again we will have a similar result as before

$$M^T\, (L\, L^T)^{-1}\, M$$

$$M^T (L^T)^{-1} L^{-1} M$$

$$(L^{-1} M)^T L^{-1} M$$

Again we can solve the two rightmost terms with lower triangular solve and multiply itself with the transpose to speed up computation.

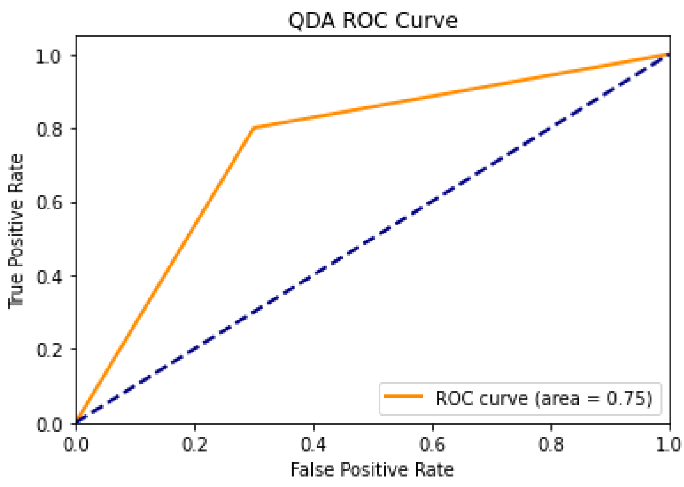
	Actual Positive	Actual Negative
Predicted Positive	0.8028571428571428	0.24
Predicted Negative	0.19714285714	0.76

Total Accuracy: 0.78

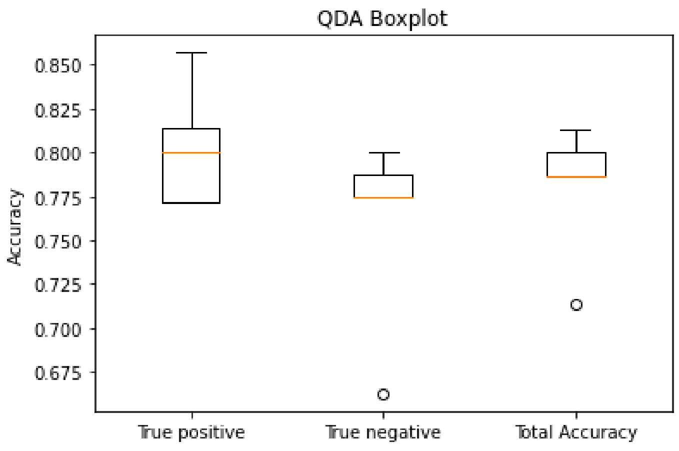
True Positive Variance: 0.0010122448979591819

True Negative Variance: 0.0024625000000000016

Total Accuracy Variance: 0.0012088888888888883



QDA ROC Curve results were from 1 trial with AUC of 0.75



QDA Boxplots for Accuracy of True Positive, True Negative, and Total Accuracy Rates

Again multiple samples were taken and values were averaged. Compared to LDA the true positive rate has risen by a sizable amount. This means a lower amount of misclassifying individuals that might lead to the spreading of the virus. However, QDA does seem to have a bigger variance in terms of these accuracy values. For the QDA ROC Curve, our AUC is equal to 0.75 whereas the LDA ROC Curve is 0.69 which is less than our QDA, so we can conclude that our QDA model classifies our CT scans more accurately.

Logistic Regression

On the other spectrum, Logistic regression is known as a discriminative approach compared to the LDA and QDA being generative. This means that in order to build a logistic regression model it is not necessary to use the different class types. Logistic regression is used here to serve as a way to compare the results for LDA and QDA. Logistic regression has the formula

$$\frac{1}{1+e^{-(x^T B)}}$$

Which classifies how likely someone would be classified as positive for covid. X here serves as the input values for the different predictors B. In order to find the best B value, the maximum likelihood estimator needs to be found which means the most likely value for B to take. This can be done with gradient descent, an iterative method that finds the minimum of specific values with the help of gradient values for B. Due to the lack of time for this project, a package was just used here to compare results.

The assumption of no multicollinearity will be violated since there are too many predictors since every pixel serves as a predictor. Neighboring pixels will have similar values which would cause predictors to be related to each other. Independence is assumed to be satisfied since the images are randomly sampled. As for no influential outliers, this is satisfied since the range of pixel values is from 0 to 255 so there won't be any extreme outliers.

When logistic regression was performed from the package it seems that the gradient descent algorithm was stopped early as the maximum iterations counter was exceeded, this potentially had an effect on the calculations

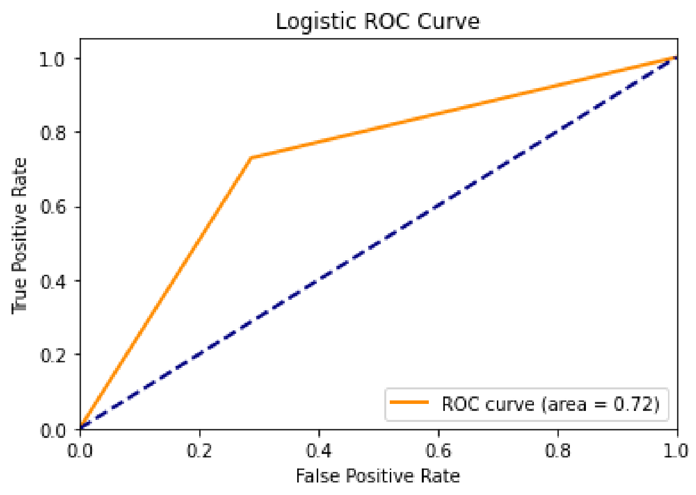
	Actual Positive	Actual Negative
Predicted Positive	0.6411428571428571	0.26975
Predicted Negative	0.35885714285	0.73025

Total Accuracy: 0.6886666666666668

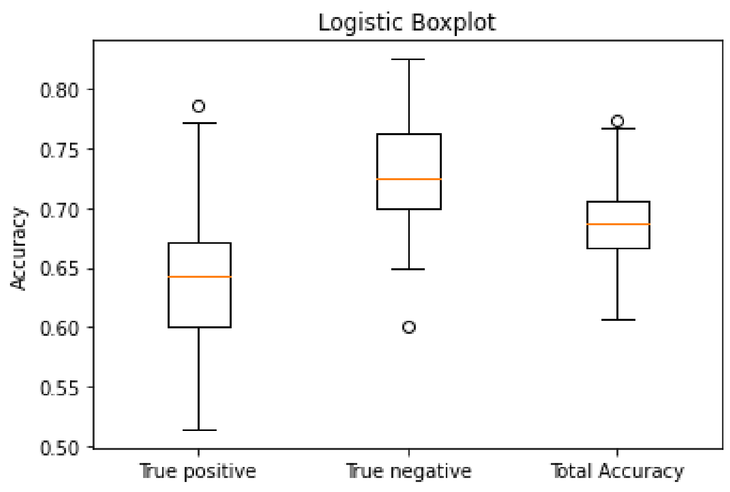
True Positive Variance: 0.003217469387755101

True Negative Variance: 0.0019286874999999998

Total Accuracy Variance: 0.0010955555555555556



Logistic ROC Curve results were from 1 trial with AUC of 0.72



Logistic Boxplots for Accuracy of True Positive, True Negative, and Total Accuracy Rates

Serving as a comparison for LDA and QDA, logistic regression does seem to have a bigger variance. Additionally, these accuracy rates do seem comparable to the other two models. For logistic regression's ROC Curve, the AUC is equal to 0.72 which is slightly larger than LDA's and slightly smaller than QDA, sitting right in the middle of both models. Again, this AUC is not the best but also not the worst, so we can say logistic regression also decently classifies the CT scans since the AUC values don't vary significantly.

Computation Time

Parallel Computing

We implement parallel computing in LDA, QDA, and Logistic functions. With those computations using mostly matrix multiplication, computation time unoptimized reached approximately over 30 minutes to compute, if not crashing our programs due to ram overload for one single testing. With parallel computing, we were able to drastically reduce our computation time to approximately 10 min, 12 min, and 3 min respectively for 5 different tests based on different training samples. Due to these restraints, we had to limit our functions to 5 instances and get the averages from those 5 accuracy values.

Conclusion

The accuracy rates do seem comparable to what logistic regression is resulting in. However, in terms of computation speed, the logistic regression package is drastically faster compared to LDA and QDA. From our results, we do value the true positive results more, as we value if someone has covid or not in comparison to true negative. With that, we achieved through QDA an 80% true positive accuracy and 60% true positive accuracy from LDA. Analyzing these values we conclude that one, QDA is a better method for processing the Covid-CT images, and secondly that CT scans are a viable way to determine if someone has covid or not. Looking initially at CT images of covid positive and covid negative images subjectively there is not much of a difference, especially since some of the images varied due to the scan as well as the anatomy of the person being scanned. However, achieving 60-80% accuracy proves that through image processing we can determine signs of covid.

With restraints like Google Colab Ram Space, and our computer's computation time being slow, we can very likely improve our accuracies if we can process more images as well as a loop over LDA and QDA more. Another way our accuracies could improve was to tune our hyperparameters through cross-validation. LDA and QDA had three and four values that should be tuned respectively. LDA had two dimensions for the image, where changing these values would change the shape of the image. Altering the shape of the image might give a differing accuracy rate, but we kept that consistent for all methods. Then there is the regularization parameter for keeping the covariance positive definite, where LDA had one and QDA had two. Adjusting how much bias to add would also change the accuracies in classification. Given that our ram was limited and computation time limited us to 5 iterations of each method, we achieved our hypothesized results, the hypothesis being a slightly positive accuracy given the images are hard to differentiate to the naked eye. All in all, image processing is a viable method as a source to detect if someone has covid or not and as an additional covid testing option for when other tests or inconclusive.

References

https://github.com/UCSD-AI4H/COVID-CT

Appendix

```
import cv2
from skimage import color
from PIL import Image
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow
import numpy as np
from scipy.linalg import solve_triangular
from sklearn.model_selection import train_test_split

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

from PIL import Image
import os


# folder path for CT_COVID
```

```
dir_path = r'/content/drive/MyDrive/Project COVID-CT/data/CT_COVID'    #put file path to CT_COVID

count = 0

# Iterate directory

for path in os.listdir(dir_path):

    # check if current path is a file

    if os.path.isfile(os.path.join(dir_path, path)):

        count += 1


# folder path for CT_NonCOVID

dir_path = r'/content/drive/MyDrive/Project COVID-CT/data/CT_NonCOVID'      #put file path to CT_NonCOVID

count_1 = 0

# Iterate directory

for path in os.listdir(dir_path):

    # check if current path is a file

    if os.path.isfile(os.path.join(dir_path, path)):

        count_1 += 1

print('File count CT_COVID:', count)

print('File count CT_NonCOVID:', count_1)

print('Total files combined:', count + count_1)

# 1st part of path and 2nd part of path

#assigning the file paths for the images to lists

#Covid positive list

CT_COVID_path = r'/content/drive/MyDrive/Project COVID-CT/data/CT_COVID'    #enter file path for google drive folder with CT_COVID images

list_CT_COVID = []

for path in os.listdir(CT_COVID_path):

  #print(path)
  list_CT_COVID.append(path)


print(list_CT_COVID)


#covid negative list

CT_NonCOVID_path = r'/content/drive/MyDrive/Project COVID-CT/data/CT_NonCOVID'
#enter file path for google drive folder with CT_NonCOVID images

list_CT_NonCOVID = []

for path in os.listdir(CT_NonCOVID_path):

  #print(path)

  list_CT_NonCOVID.append(path)


print(list_CT_NonCOVID)

# length and image path for covid negative pics

# CT_NonCOVID images

image_list_COVID_NEGATIVE = []

concat_neg = []

for i in range(0, 397):

  try:

    temp = str(CT_NonCOVID_path) + str("/") + list_CT_NonCOVID[i]

    concat_neg.append(temp)

    img  = Image.open(temp)     #change input of list depending on image u want to see

    image_list_COVID_NEGATIVE.append(img)

  except IOError:

    pass


print(len(image_list_COVID_NEGATIVE))

print(concat_neg)
```

```
# length length and image path for covid positive pics

#CT_COVID images

image_list_COVID_POSITIVE = []

concat_pos = []

for i in range(0, 349):

  try:

    temp = str(CT_COVID_path) + str("/") + list_CT_COVID[i]

    concat_pos.append(temp)

    #img  = Image.open(temp)

    image_list_COVID_POSITIVE.append(img)

  except IOError:

    pass


print(len(image_list_COVID_POSITIVE))

print(concat_pos)

hyperparameter = .345 #multiplier for the median pixel matrix size can't go past .4 or
else there will be ram issues the bigger the size the slower it is

size = (round(410 * hyperparameter), round(290 * hyperparameter))

size

#concatenates all the negative image columns together into a singular matrix

for i in range(len(concat_neg)):

  gray = cv2.imread(concat_neg[i],0)

  if i == 0:

    neg = cv2.resize(gray, dsize = size, interpolation=cv2.INTER_CUBIC).reshape(-1,1)

  else:

    neg2 = cv2.resize(gray, dsize = size, interpolation=cv2.INTER_CUBIC).reshape(-1,1)

    neg = np.concatenate((neg, neg2), axis = 1)
```

```
neg

#concatenates all the positive image columns together into a singular matrix

for i in range(len(concat_pos)):

  gray = cv2.imread(concat_pos[i],0)

  if i == 0:

    pos = cv2.resize(gray, dsize = size, interpolation=cv2.INTER_CUBIC).reshape(-1,1)

  else:

    pos2 = cv2.resize(gray, dsize = size, interpolation=cv2.INTER_CUBIC).reshape(-1,1)

    pos = np.concatenate((pos, pos2), axis = 1)


pos

#for parallel computing I think this might have to be in the function


pos_train, pos_test = train_test_split( pos.T, test_size=0.2, shuffle=True) #split train/test
data


pos_train = pos_train.T #279 postive case training


pos_test = pos_test.T #70 positive case testing


neg_train, neg_test = train_test_split(neg.T, test_size=0.2, shuffle=True)


neg_train = neg_train.T #317 negative case training


neg_test = neg_test.T #80 negative case testing



pos_test #shows positive test
```

```
#run this one for LDA

negmean = np.mean(neg_train, axis = 1) #mean for negative cases
posmean = np.mean(pos_train, axis = 1) #mean for positive cases
whole = np.concatenate((neg_train, pos_train), axis = 1) #combines the training data into one big array
cov = np.cov(whole) #covariance of the training data for LDA

reg_param = 0.1 * np.trace(cov) / cov.shape[0] #Regularization allows to use a bigger sample size for better accuracy
covreg = cov + reg_param * np.identity(cov.shape[0]) #regularized covariance

#rewritten formula for optimization
l = np.linalg.cholesky(covreg) #Cholesky decomposition for covariance
# negmean.T @ np.linalg.inv(np.cov(whole)) @ negmean

negchol = solve_triangular(l, negmean, lower=True) #optimized cholesky decomposition
negopt = negchol.T @ negchol #optimized cholesky decomposition for negative mean part for LDA
poschol = solve_triangular(l, posmean, lower=True) #optimized cholesky decomposition
posopt = poschol.T @ poschol #optimized cholesky decomposition for positive mean part for LDA
TP = []
for i in range(pos_test.shape[1]): #loops through all positive testing data
  front = 2*(posmean - negmean) @ solve_triangular(l.T,solve_triangular(l, pos_test[:,i], lower = True), lower = False)
  if front + negopt - posopt + 2 * np.log(pos_train.shape[1]/neg_train.shape[1]) > 0: #LDA formula optimized
    TP.append(1)
  else:
    TP.append(0)

#run this one for QDA


negmean = np.mean(neg_train, axis = 1) #mean for negative training data

posmean = np.mean(pos_train, axis = 1) #mean for positive training data


poscov = np.cov(pos_train) #covariance for positive training data

negcov = np.cov(neg_train) #covariance for negative training data


reg_param = 0.1 * np.trace(poscov) / poscov.shape[0] #regularizing parameter

poscovreg = poscov + reg_param * np.identity(poscov.shape[0]) #regularizing positive
covariance to allow for a bigger pixal matrix

reg_param = 0.1 * np.trace(negcov) / negcov.shape[0]

negcovreg = negcov + reg_param * np.identity(negcov.shape[0]) #regularizing negative
covariance to allow for a bigger pixal matrix


pL = np.linalg.cholesky(poscovreg) #cholesky decomposition on the positive regularized
covariance

nL = np.linalg.cholesky(negcovreg) #cholesky decomposition on the negative regularized
covariance


lead = np.sum(2*np.log(np.diag(nL))) - np.sum(2*np.log(np.diag(pL))) + 2 *
np.log(pos_train.shape[1]/neg_train.shape[1]) #log of negative determinant/positive
determinant

TP = []

for i in range(pos_test.shape[1]): #loops through all positive testing data

  posopt = solve_triangular(pL, pos_test[:,i] - posmean, lower=True) #optimized way to
solve for inverse

  poseq = posopt.T @ posopt #faster way to solve equation

  negopt = solve_triangular(nL, pos_test[:,i] - negmean, lower=True) #optimized way to
solve for inverse
```

```
  negeq = negopt.T @ negopt #faster way to solve equation
  if lead - poseq + negeq > 0: #classify
    TP.append(1)
  else:
    TP.append(0)
#below is the code for LDA with parallel computing
#under that is the code for QDA with parallel computing
#LDA Collin run this one for LDA
#parallel computing
import multiprocessing as mp
from multiprocessing import Pool
from multiprocessing import Process
import time
start = time.time()

TP_avg = [] #5
TN_avg = [] #5
TP_TN_avg = [] #5

for a in range(0,5):
  pos_train, pos_test = train_test_split( pos.T, test_size=0.2, shuffle=True) #split train/test
data

  pos_train = pos_train.T #279 positive case training

  pos_test = pos_test.T #70 positive case testing

  neg_train, neg_test = train_test_split(neg.T, test_size=0.2, shuffle=True)
```

```
 neg_train = neg_train.T #317 negative case training

 neg_test = neg_test.T #80 negative case testing

 pos_test #shows positive test

 def LDA_math(n):
   global TP_avg
   global TN_avg
   global TP_TN_avg
   global neg_train
   global pos_train
   global neg_test
   global pos_test

   negmean = np.mean(neg_train, axis = 1) #mean for negative cases
   posmean = np.mean(pos_train, axis = 1) #mean for positive cases
   whole = np.concatenate((neg_train, pos_train), axis = 1) #combines the training data
into one big array
   cov = np.cov(whole) #covariance of the training data for LDA

   reg_param = 0.1 * np.trace(cov) / cov.shape[0] #Regularization allows to use a bigger
sample size for better accuracy
   covreg = cov + reg_param * np.identity(cov.shape[0]) #regularized covariance

   #rewritten formula for optimization
   l = np.linalg.cholesky(covreg) #Cholesky decomposition for covariance
```

```
  # negmean.T @ np.linalg.inv(np.cov(whole)) @ negmean


  negchol = solve_triangular(l, negmean, lower=True) #optimized cholesky
decomposition

  negopt = negchol.T @ negchol #optimized cholesky decomposition for negative mean
part for LDA

  poschol = solve_triangular(l, posmean, lower=True) #optimized cholesky
decomposition

  posopt = poschol.T @ poschol #optimized cholesky decomposition for positive mean
part for LDA


  #TP, TN, TP_TN computation

  TP = []

  for i in range(pos_test.shape[1]): #loops through all positive testing data

    front = 2*(posmean - negmean) @ solve_triangular(l.T,solve_triangular(l, pos_test[:,i],
lower = True), lower = False)

    if front + negopt - posopt + 2 * np.log(pos_train.shape[1]/neg_train.shape[1]) > 0: #LDA
formula optimized

      TP.append(1)

    else:

      TP.append(0)

  TP_avg = np.mean(TP) #True positive for LDA


  TN = []

  for i in range(neg_test.shape[1]): #loops through all negative testing data

    front = 2*(posmean - negmean) @ solve_triangular(l.T,solve_triangular(l, neg_test[:,i],
lower = True), lower = False)

    if front + negopt - posopt + 2 * np.log(pos_train.shape[1]/neg_train.shape[1]) < 0: #LDA
formula optimized

      TN.append(1)

    else:
      TN.append(0)
  TN_avg = np.mean(TN) #True negative for LDA

  total = np.mean(TP + TN) #total accuracy rate
  TP_TN_avg = total

  return(TP_avg,TN_avg,TP_TN_avg)

if __name__=='__main__':
    pool = Pool(processes=4)
    n = '2' #input value not used, just need something to fill.
    result = pool.map(LDA_math, n)
    pool.close()
    #print(result)
    TP_avg.append(result[0][0])
    TN_avg.append(result[0][1])
    TP_TN_avg.append(result[0][2])


end = time.time()
print(end-start)
print(TP_avg)
print(TN_avg)
print(TP_TN_avg)
#last output lambda: 0.1
#[0.6285714285714286, 0.6, 0.5285714285714286, 0.6142857142857143, 0.6]
#[0.7625, 0.775, 0.7875, 0.7625, 0.7375]
```

```
#[0.7, 0.6933333333333334, 0.6666666666666666, 0.6933333333333334,
0.6733333333333333]

#true pos mean: 0.5942857142857143

#true neg mean: 0.765

#total accuracy mean: 0.6853333333333333

#true pos var: 0.0011918367346938775

#true neg var: 0.00027499999999999995

#total accuracy var: 0.00016711111111111126

#9 min 37 seconds

TP_mean = np.mean(TP_avg)

TN_mean = np.mean(TN_avg)

TP_TN_mean = np.mean(TP_TN_avg)

TP_var = np.var(TP_avg)

TN_var = np.var(TN_avg)

TP_TN_var = np.var(TP_TN_avg)

print("true pos mean: " + str(TP_mean))

print("true neg mean: " + str(TN_mean))

print("total accuracy mean: " + str(TP_TN_mean))

print("true pos var: " + str(TP_var))

print("true neg var: " + str(TN_var))

print("total accuracy var: " + str(TP_TN_var))

#boxplot LDA

import matplotlib.pyplot as plt

import numpy as np


my_dict = {'True positive': TP_avg, 'True negative': TN_avg, 'Total Accuracy': TP_TN_avg}

plt.boxplot(my_dict.values(), labels=my_dict.keys());

plt.ylabel('Accuracy')
plt.title('LDA Boxplot')

TN = []

negativepred = []

for i in range(neg_test.shape[1]): #loops through all testing data

  front = 2*(posmean - negmean) @ solve_triangular(l.T,solve_triangular(l, neg_test[:,i],
lower = True), lower = False) #use cholesky to speed up compuation by inverse by lower
and upper triangular solve

  if front + negopt - posopt + 2 * np.log(349/397) < 0: #combining all values

    TN.append(1) #1 if correct guess

    negativepred.append(0)

  else:

    TN.append(0) #0 if wrong

    negativepred.append(1)

ypred = np.concatenate((TP, negativepred)) # LDA ROC CURVE

ytest = np.concatenate((np.ones(len(TP)), np.zeros(len(negativepred))))

from sklearn.metrics import roc_curve, auc

import matplotlib.pyplot as plt


fpr, tpr, thresholds = roc_curve(ytest, ypred)

roc_auc = auc(fpr, tpr)


plt.figure()

plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')
plt.title('LDA ROC Curve')
plt.legend(loc="lower right")
plt.show()
#QDA Collin run this one for QDA
#parallel computing
import multiprocessing as mp
from multiprocessing import Pool
from multiprocessing import Process
import time
start = time.time()

TP_avg = [] #5
TN_avg = [] #5
TP_TN_avg = [] #5

for a in range(0,5):
  pos_train, pos_test = train_test_split( pos.T, test_size=0.2, shuffle=True) #split train/test
data

  pos_train = pos_train.T #279 postive case training

  pos_test = pos_test.T #70 positive case testing

  neg_train, neg_test = train_test_split(neg.T, test_size=0.2, shuffle=True)

  neg_train = neg_train.T #317 negative case training

  neg_test = neg_test.T #80 negative case testing
```

```
#neg1.shape #397

#print(pos_test) #shows positive test

def QDA_math(n):
  global TP_avg
  global TN_avg
  global TP_TN_avg
  global neg_train
  global pos_train
  global neg_test
  global pos_test
  negmean = np.mean(neg_train, axis = 1) #mean for negative training data
  posmean = np.mean(pos_train, axis = 1) #mean for positive training data

  poscov = np.cov(pos_train) #covariance for positive training data
  negcov = np.cov(neg_train) #covariance for negative training data

  reg_param = 0.1 * np.trace(poscov) / poscov.shape[0] #regularizing parameter
  poscovreg = poscov + reg_param * np.identity(poscov.shape[0]) #regularizing positive
covariance to allow for a bigger pixal matrix
  reg_param = 0.1 * np.trace(negcov) / negcov.shape[0]
  negcovreg = negcov + reg_param * np.identity(negcov.shape[0]) #regularizing negative
covariance to allow for a bigger pixal matrix

  pL = np.linalg.cholesky(poscovreg) #cholesky decomposition on the positive
regularized covariance
  nL = np.linalg.cholesky(negcovreg) #cholesky decomposition on the negative
```

```
regularized covariance

    lead = np.sum(2*np.log(np.diag(nL))) - np.sum(2*np.log(np.diag(pL))) + 2 *
np.log(pos_train.shape[1]/neg_train.shape[1]) #log of negative determinant/positive
determinant

    #TP, TN, TP_TN computation

    TP = []

    for i in range(pos_test.shape[1]): #loops through all positive training data

      posopt = solve_triangular(pL, pos_test[:,i] - posmean, lower=True) #optimized way to
solve for inverse

      poseq = posopt.T @ posopt #faster way to solve equation

      negopt = solve_triangular(nL, pos_test[:,i] - negmean, lower=True) #optimized way to
solve for inverse

      negeq = negopt.T @ negopt #faster way to solve equation

      if lead - poseq + negeq > 0: #classify

        TP.append(1)

      else:

        TP.append(0)

    TP_avg = np.mean(TP) #True positive for QDA

    #print(TP_avg)



    TN = []

    for i in range(neg_test.shape[1]): #loops through all negative training data

      posopt = solve_triangular(pL, neg_test[:,i] - posmean, lower=True) #optimized way to
solve for inverse

      poseq = posopt.T @ posopt #faster way to solve equation

      negopt = solve_triangular(nL, neg_test[:,i] - negmean, lower=True) #faster way to
solve equation
```

```
    negeq = negopt.T @ negopt #faster way to solve equation
    if lead - poseq + negeq < 0: #classify
      TN.append(1)
    else:
      TN.append(0)

  TN_avg = np.mean(TN) #True negative for QDA
  #print(TN_avg)

  total = np.mean(TP + TN) #Accuracy rate for QDA
  TP_TN_avg = total
  #print(TP_TN_avg)

  #comp = [sublist, np.mean(TN), total]
  #return comp
  return(TP_avg,TN_avg,TP_TN_avg)

if __name__=='__main__':
    pool = Pool(processes=4)
    n = '2' #input value not used, just need something to fill.
    result = pool.map(QDA_math, n)
    pool.close()
    #print(result)
    TP_avg.append(result[0][0])
    TN_avg.append(result[0][1])
    TP_TN_avg.append(result[0][2])
```

```
end = time.time()

print(end-start)

print(TP_avg)

print(TN_avg)

print(TP_TN_avg)

#last output lambda: 0.1

#[0.8, 0.7714285714285715, 0.8571428571428571, 0.8142857142857143,
0.7714285714285715]

#[0.775, 0.8, 0.775, 0.7875, 0.6625]

#[0.7866666666666666, 0.7866666666666666, 0.8133333333333334, 0.8,
0.7133333333333334]

#true pos mean: 0.8028571428571428

#true neg mean: 0.76

#total accuracy mean: 0.78

#true pos var: 0.0010122448979591819

#true neg var: 0.0024625000000000016

#total accuracy var: 0.0012088888888888883

#12 min 4 seconds

TP_mean = np.mean(TP_avg)

TN_mean = np.mean(TN_avg)

TP_TN_mean = np.mean(TP_TN_avg)

TP_var = np.var(TP_avg)

TN_var = np.var(TN_avg)

TP_TN_var = np.var(TP_TN_avg)

print("true pos mean: " + str(TP_mean))

print("true neg mean: " + str(TN_mean))

print("total accuracy mean: " + str(TP_TN_mean))
print("true pos var: " + str(TP_var))

print("true neg var: " + str(TN_var))

print("total accuracy var: " + str(TP_TN_var))

#boxplot QDA

import matplotlib.pyplot as plt

import numpy as np


my_dict = {'True positive': TP_avg, 'True negative': TN_avg, 'Total Accuracy': TP_TN_avg}

plt.boxplot(my_dict.values(), labels=my_dict.keys());


plt.ylabel('Accuracy')

plt.title('QDA Boxplot')

TN = []

negativepred = []

for i in range(neg_test.shape[1]): #loops through the testing data

  posopt = solve_triangular(pL, neg_test[:,i] - posmean, lower=True) #optimized inverse
part for formula with lower triangular solve

  poseq = posopt.T @ posopt #Since the inverses are just the transposed of the other I can
solve for one and multiply itself transposed

  negopt = solve_triangular(nL, neg_test[:,i] - negmean, lower=True) #optimized inverse
part for formula with lower triangular solve

  negeq = negopt.T @ negopt #Since the inverses are just the transposed of the other I can
solve for one and multiply itself transposed

  if lead - poseq + negeq < 0:

    TN.append(1) #correct guess

    negativepred.append(0)

  else:

    TN.append(0) #incorrect guess

    negativepred.append(1)
```

```
ypred = np.concatenate((TP, negativepred)) # QDA ROC Curve

ytest = np.concatenate((np.ones(len(TP)), np.zeros(len(negative))))

from sklearn.metrics import roc_curve, auc

import matplotlib.pyplot as plt


fpr, tpr, thresholds = roc_curve(ytest, ypred)

roc_auc = auc(fpr, tpr)


plt.figure()

plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('QDA ROC Curve')

plt.legend(loc="lower right")

plt.show()

TN = []

negativepred = []

for i in range(neg_test.shape[1]): #loops through all testing data

  front = 2*(posmean - negmean) @ solve_triangular(l.T,solve_triangular(l, neg_test[:,i],
lower = True), lower = False) #use cholesky to speed up compuation by inverse by lower
and upper triangular solve

  if front + negopt - posopt + 2 * np.log(349/397) < 0: #combining all values

    TN.append(1) #1 if correct guess

    negativepred.append(0)

  else:
    TN.append(0) #0 if wrong
    negativepred.append(1)
pos_label = np.ones(pos_train.shape[1]) #1 for logisitc regression
neg_label = np.zeros(neg_train.shape[1]) #0 for logistic regression
testing = np.concatenate((pos_test, neg_test), axis = 1).T
from sklearn.linear_model import LogisticRegression
logistic = LogisticRegression()
truepos = np.ones(pos_test.shape[1])
trueneg = np.zeros(neg_test.shape[1])
Ytest = np.concatenate((truepos, trueneg))
Ytest

truepos = np.ones(pos_test.shape[1])
trueneg = np.zeros(neg_test.shape[1])
Ytest = np.concatenate((truepos, trueneg))
Ytest

combine = np.concatenate((pos_train, neg_train), axis = 1).T
label = np.concatenate((pos_label, neg_label))

logistic.fit(combine, label)
logisticpred = logistic.predict(testing)

print(sum(logisticpred == Ytest )/(Ytest.shape[0])) #Total Accuracy rate

print(sum(logisticpred[:pos_test.shape[1]] == Ytest[:pos_test.shape[1]])/ pos_test.shape[1])
#true positive
```

```
print(sum(logisticpred[pos_test.shape[1]:] == Ytest[pos_test.shape[1]:])/ neg_test.shape[1])
#true negative


from sklearn.metrics import roc_curve, auc #ROC for logistic

import matplotlib.pyplot as plt


fpr, tpr, thresholds = roc_curve(Ytest, logisticpred)

roc_auc = auc(fpr, tpr)


plt.figure()

plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)

plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')

plt.xlim([0.0, 1.0])

plt.ylim([0.0, 1.05])

plt.xlabel('False Positive Rate')

plt.ylabel('True Positive Rate')

plt.title('Logistic ROC Curve')

plt.legend(loc="lower right")

plt.show()

#logistic packages

#parallel computing

import multiprocessing as mp

from multiprocessing import Pool

from multiprocessing import Process

import time

start = time.time()


TP_avg = [] #5
```

```
TN_avg = [] #5

TP_TN_avg = [] #5


for a in range(0,100):
  pos_train, pos_test = train_test_split( pos.T, test_size=0.2, shuffle=True) #split train/test
data


  pos_train = pos_train.T #682 x 279 postive case training


  pos_test = pos_test.T #682 x 70 positive case testing


  #pos1.shape #682 x 349


  neg_train, neg_test = train_test_split(neg.T, test_size=0.2, shuffle=True)


  neg_train = neg_train.T #682 x 317 negative case training


  neg_test = neg_test.T #682 x 80 negative case testing


  #neg1.shape #682 x 397


  #print(pos_test) #shows positive test


  def logistic(n):
    global TP_avg
    global TN_avg
    global TP_TN_avg
    global neg_train
    global pos_train

    global neg_test

    global pos_test


    pos_label = np.ones(pos_train.shape[1]) #1 for logisitc regression

    neg_label = np.zeros(neg_train.shape[1]) #0 for logistic regression

    testing = np.concatenate((pos_test, neg_test), axis = 1).T

    from sklearn.linear_model import LogisticRegression

    logistic = LogisticRegression()

    truepos = np.ones(pos_test.shape[1])

    trueneg = np.zeros(neg_test.shape[1])

    Ytest = np.concatenate((truepos, trueneg))

    Ytest


    truepos = np.ones(pos_test.shape[1])

    trueneg = np.zeros(neg_test.shape[1])

    Ytest = np.concatenate((truepos, trueneg))

    Ytest


    combine = np.concatenate((pos_train, neg_train), axis = 1).T

    label = np.concatenate((pos_label, neg_label))


    logistic.fit(combine, label)

    logisticpred = logistic.predict(testing)


    #print(sum(logisticpred[:pos_test.shape[1]] == Ytest[:pos_test.shape[1]])/
pos_test.shape[1]) #true positive

    true_pos = (sum(logisticpred[:pos_test.shape[1]] == Ytest[:pos_test.shape[1]])/
pos_test.shape[1])


    #print(sum(logisticpred[pos_test.shape[1]:] == Ytest[pos_test.shape[1]:])/
neg_test.shape[1]) #true negative

    true_neg = (sum(logisticpred[pos_test.shape[1]:] == Ytest[pos_test.shape[1]:])/
neg_test.shape[1])


    #print(sum(logisticpred == Ytest )/(Ytest.shape[0])) #Total Accuracy rate

    total_acc = (sum(logisticpred == Ytest )/(Ytest.shape[0]))


    return(true_pos, true_neg, total_acc)


  if __name__=='__main__':

      pool = Pool(processes=4)

      n = '2' #input value not used, just need something to fill.

      result = pool.map(logistic, n)

      pool.close()

      #print(result)

      TP_avg.append(result[0][0])

      TN_avg.append(result[0][1])

      TP_TN_avg.append(result[0][2])


end = time.time()

print(end-start)

print(TP_avg)

print(TN_avg)

print(TP_TN_avg)

#last output:

#true pos mean: 0.6411428571428571
```

```
#true neg mean: 0.73025
#total accuracy mean: 0.6886666666666668
#true pos var: 0.003217469387755101
#true neg var: 0.0019286874999999998
#total accuracy var: 0.0010955555555555556
#3min 26 seconds
TP_mean = np.mean(TP_avg)
TN_mean = np.mean(TN_avg)
TP_TN_mean = np.mean(TP_TN_avg)
TP_var = np.var(TP_avg)
TN_var = np.var(TN_avg)
TP_TN_var = np.var(TP_TN_avg)
print("true pos mean: " + str(TP_mean))
print("true neg mean: " + str(TN_mean))
print("total accuracy mean: " + str(TP_TN_mean))
print("true pos var: " + str(TP_var))
print("true neg var: " + str(TN_var))
print("total accuracy var: " + str(TP_TN_var))
#boxplot Logistic
import matplotlib.pyplot as plt
import numpy as np

my_dict = {'True positive': TP_avg, 'True negative': TN_avg, 'Total Accuracy': TP_TN_avg}
plt.boxplot(my_dict.values(), labels=my_dict.keys());

plt.ylabel('Accuracy')
plt.title('Logistic Boxplot')
```