Facial Recognition using Eigenfaces and Applications to Breast Cancer Diagnosis

Suyog Soti 106319772
Stevan Maksimovic 105071205
Andrew Levitt 106408347
APPM 3310-001

August 23, 2018

Abstract

Facial recognition technology has become commonplace in recent years with the goal of selecting the appropriate image from an image library that is most closely related to a given input image. One of the most common techniques for doing so utilizes Principal Component Analysis (PCA) to reduce extremely large images to easily comparable weight vectors. The program then 'recognizes' the image from the library with the closest associated weight vector to that of the input image. The ability to select the closest image with similar lighting, angle, and other photographic conditions is well documented. We then attempted to use the algorithm to predict cancer malignancy. We found that when using a dataset of breast cancer mammograms separated into benign and malignant categories, the facial recognition algorithm was unable to accurately predict whether an input mammogram was in fact benign or malignant due to the high variance across these images.

1 Introduction

Facial recognition technology has become commonplace in recent years as a method of analyzing and comparing the vast amounts of facial image data being continuously created. Our method of choice utilizes Principal Component Analysis (PCA) to pull distinctive features from the image library in the form of eigenface images, where the eigenface images are the eigenvectors of the associated covariance matrix. Then each image in the library can be represented as a weighted sum of the features (eigenfaces). In this way, large images can be reduced to smaller weight vectors and quickly compared.[1] The program then selects the image from the library with the closest associated weight vector to that of the input image. The error in the input and the closest selected image can be quantified by the difference between their weight vectors. If the error is small enough, we know the two faces are likely the same. If it is larger than an acceptable amount then we can assume two faces are not the same, and if the error surpasses a certain threshold, we may assume then the input image may not be a face at all. Since this algorithm can be used to compare any set of data represented by vectors, we studied its effectiveness in recognizing malignant breast cancers. Using a publicly available collection of mammograms divided into categories 'malignant' and 'benign', we attempted to use facial recognition to match an input image to an image in the dataset, then using the label of the matched image, predict the malignancy of the input.

2 Mathematical Formulation

Given a database of images, we begin by reading each image as a vector by appending each row of values to each other. For example, in our database [2] of 112 by 92 pixel images, each image can be converted into a vector of length 112*92=10,304 where the first 92 values represent the first row of the image, the second 92 values represent the second row of the image, and so on and so forth. The mean image vector is calculated by taking the mean value of each element of every vector, and is then subtracted from each respective image vector. The covariance matrix of this set of mean-subtracted vectors is calculated, and its eigenvectors are found.[3] Since each eigenvector has the same dimension as the images, it can be denoted an "eigenface" and be thought of as certain facial image structures that have large deviations from the average face. Each image can thus be represented as a linear combination of eigenvectors added to the average face.[1] Principal component analysis can be used to find the most important (read: farthest from mean) eigenvectors, reducing the number of vectors needed to represent an image. Finally, each image is projected into the space spanned by the eigenvectors, and since each mean-subtracted image can be represented by a weighted sum of eigenfaces, a vector formed by the weights can represent the image instead. This essentially reduces the dimension of each image from 10,304 pixels to approximately 50 weight vectors. The weight vector of an input image is then compared to each weight vector of the image database, and the closest weight vector represents the closest image.

2.1 Preprocessing

We begin with the library of p images that make up the potential matches for recognition. In our case, we have 400 images from the AT&T Laboratories Cambridge Face Database [2] and 3 of our own images, for a total of 403 images. It should be noted that each image must be of identical pixel resolution (in the database of faces we use, each image is 112 by 92 pixels). Each image is taken into the python program as a matrix of size mxn with each entry containing 3 RGB values. Each image is converted from color to grayscale by taking the average of the 3 RGB values assigned to each pixel in the associated matrix. Each grayscale image is thus represented by an mxn matrix with each entry corresponding to a pixel containing a grayscale value between 0 and 255 (0 pure black and 255 pure white). Each image matrix is then converted into a single vector existing in $\mathbb{R}^{mn}$ by concatenating the rows. For example, 112 by 92 pixel images are converted into a 10,304-dimensional vector. These image vectors then placed arbitrarily into a matrix of size $mnxp$ where the columns are the grayscale image vectors. An 'average image vector,' an element of $\mathbb{R}^{mn}$, is then created by taking the average grayscale value of each vector value (read: pixel) across all images in the library. This average image vector is then subtracted from each image vector in the image vector matrix. For explanation purposes, the mean-subtracted image vector matrix will be called A. The goal is to now compute the eigenvectors of the covariance matrix AA^T of size $mnxmn$ that we can use to represent images and reduce dimensionality later on.

2.2 Covariance and Covariance Matrices

Covariance is a method of determining the extent of the correlation between two random sets. A simple model calculating the covariance between two set, X and Y, is shown below.

$$cov(X,Y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - E(X))(y_i - E(Y)) \tag{1}$$

x_i is the ith element of X, y_i is the ith element of Y, $E(X)$ is the mean value of X, $E(Y)$ is the mean value of Y, and n is the number of elements in each set.It should be noted that a positive covariance implies a direct correlation between the two variables while a negative covariance implies indirect correlation. For a set of n-dimensional vectors, the covariance matrix is given by

$$\Sigma = \begin{bmatrix} E[(X_1-\mu_1)(X_1-\mu_1)] & E[(X_1-\mu_1)(X_2-\mu_2)] & \dots & E[(X_1-\mu_1)(X_n-\mu_n)] \\ E[(X_2-\mu_2)(X_1-\mu_1)] & E[(X_2-\mu_2)(X_2-\mu_2)] & \dots & E[(X_2-\mu_2)(X_n-\mu_n)] \\ \vdots & \vdots & \ddots & \vdots \\ E[(X_n-\mu_n)(X_1-\mu_1)] & E[(X_n-\mu_n)(X_2-\mu_2)] & \dots & E[(X_n-\mu_n)(X_n-\mu_n)] \end{bmatrix} [1] \tag{2}$$

X_i is the set of values of the ith position in each vector, and μ_i is the mean value of X_i. Given this, the ij-th value of the covariance matrix is the covariance between the ith and the jth pixels for every image. On the diagonal, each value is the variance of each pixel. Instead of relating two different sets, variance measures the average distance each pixel is from the mean. By using the covariance matrix, we are able to represent each image's deviation from the mean in a square matrix. For example, we convert the $mnxp$ mean subtracted matrix A into the square covariance matrix $\Sigma = AA^T$. We can then find the eigenvectors of this square covariance matrix. The significance of the associated eigenvectors will be addressed shortly.

2.3 Eigenfaces

Unfortunately, the computation of AA^T is computationally intensive to the point of failure from insufficient memory and extended processing time. In our case, this matrix would be 10,304 by 10,304 and would take approximately one gigabyte of memory.[1] In our simulation, the program simply crashed. The following was derived as a workaround using the much smaller matrix, A^TA:

$$\begin{gathered} A^TA\vec{v} = \lambda\vec{v} \\ AA^TA\vec{v} = \lambda A\vec{v} \\ let\ \vec{u} = A\vec{v} \\ AA^T\vec{u} = \lambda\vec{u} \end{gathered} \tag{3}$$

Thus, the eigenvalues of AA^T and A^TA are the same, and the eigenvectors of AA^T are the eigenvectors of A^TA multiplied by A. In essence, the eigenvalues and the associated eigenvectors of the more manageable pxp matrix can be computed, then the eigenvectors of the pxp matrix can be transformed into the space spanned by the eigenvectors of the AA^T. It should be noted that we are not losing any potential eigenvectors by utilizing this workaround as we know that A^TA and AA^T share the same potential eigenvalues shown by (3) and will thus have the same number of associated eigenvectors.

We would ultimately like to find a way to represent all images in our face library as a linear combination of images called eigenfaces. The question is why the eigenvectors of the covariance matrix form the eigenfaces. We know from above discussion on covariance matrices that off-diagonal elements represent the covariance between two pixels while the diagonal elements represent the variance of a single pixel. Illustratively, consider the 2-dimensional datasets in Figure 1.

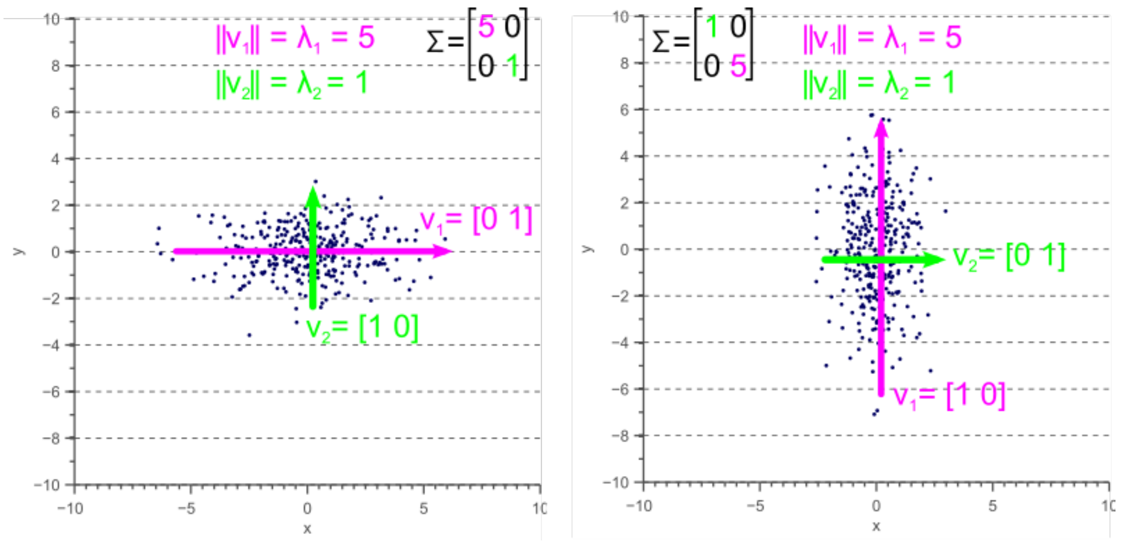


Figure 1: Eigenvectors and Eigenvalues for the covariance matrix of different data sets without covariance [5]

With each respective covariance matrix denoted Σ, we can see how each set has nonzero variances (diagonal elements of Σ), but zero covariances (off diagonal elements of Σ). Graphically, we can see that that pure variance corresponds to the the variation in the data along the standard basis vectors ($\vec{e_1}$, $\vec{e_2}$). Interestingly, the eigenvalues of the covariance matrix correspond to the magnitude of the variance of the data in the standard basis directions.

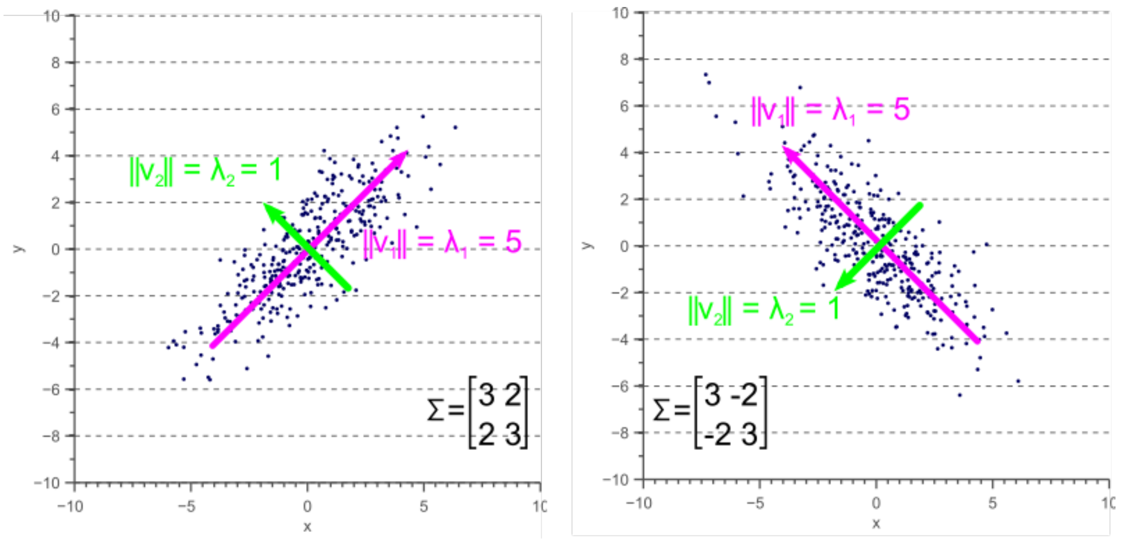


Figure 2: Eigenvectors and Eigenvalues for the covariance matrix of different data sets with covariance [5]

Note how the off-diagonal elements in Figure 2 are nonzero, we observe that the greatest variation in the datasets no longer lie along the standard basis vectors. However, the eigenvalues remain identical to their counterparts in the previous datasets. Moreover, the greatest variation in the data lies in the directions of the respective eigenvectors! The covariance matrix is taken to be a linear transformation on eigenvector $\vec{v}$, scaling it by eigenvalue λ of the form:

$$\Sigma\vec{v} = \lambda\vec{v} \text{ [4]} \tag{4}$$

Recognizing their orthogonality, these eigenvectors are the directions of purely linear transformations, corresponding to a new set of orthogonal basis vectors that form axes from which we can represent our data. We now have a way of representing images in our original facial database as a linear combination of the eigenvectors! Since each eigenvector has the same dimension as each of the image vectors, it can also represent an image, denoted an eigenface. The unique part about these basis vectors however, is they now contain a quantifiable 'importance' in the dataset, measured by their respective eigenvalues. This is of critical importances when applying principal component analysis (PCA) to ultimately achieve dimensional reduction as explained below.

2.4 Principal Component Analysis

Principal component analysis is, trivially, a way of finding the principal components of a dataset, with the principal components being a set of axes along which there is a large deviation in the dataset.

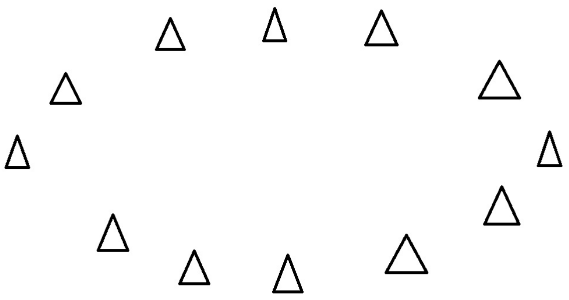
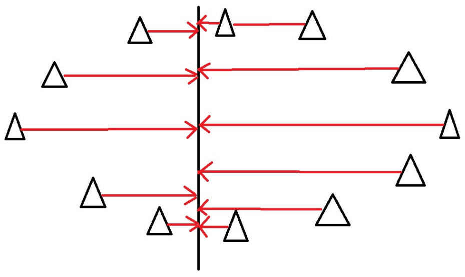
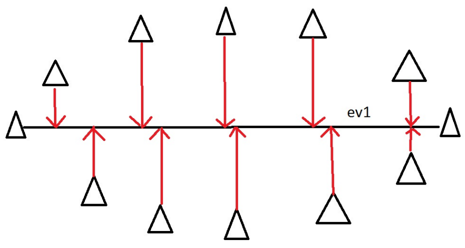


Figure 3: An example set of data represented by triangles



(a) Vertical component of dataset



(b) Horizontal component of dataset

Figure 4: Vertical and horizontal components of the dataset in Figure 3 [6]

For example, given the dataset of triangles in Figure 3, we can draw many different axes. One possible axis is displayed in Figure 4a. The variation in the dataset is the highlighted distance along the axis. Another possible axis is highlighted in Figure 4b. We can see that the axis in Figure 4b represents much more deviation in the data than the axis in Figure 4a, or in any other possible axis for this dataset. Thus, we would consider the axis in Figure 4b the principal component. It should be noted that to represent the dataset in its entirety, both axes (4a and 4b) are necessary as they represent all components necessary to describe each point in the 2 dimensional dataset. One may also select multiple principal components outside of a threshold of data variation assuming a dataset of dimension greater than 2.

Given n eigenvalues of the covariance matrix, $\lambda_i, i = 1, 2, \ldots, n$, the total variance, v is given by

$$v = n(\lambda_1 + \lambda_2 + \cdots + \lambda_n) \tag{5}$$

and k principal components is determined by the smallest k that satisfies

$$\frac{n(\lambda_1 + \lambda_2 + \cdots + \lambda_k)}{v} > \epsilon. \text{ [1]} \tag{6}$$

Thus, the k principal components are the eigenvectors corresponding to the largest k eigenvalues that satisfy the inequality above where ϵ is a number between 0 and 1 decided by the user.

2.5 Dimensionality Reduction and Facial Recognition

After PCA, we have k principal components/eigenvectors of the covariance matrix. These eigenvectors can form a $mnxk$ matrix V. Eigenvectors/eigenfaces represent a "standard facial component," so each mean-subtracted image vector, $vecx$ can be projected onto a k-dimensional space. [3]

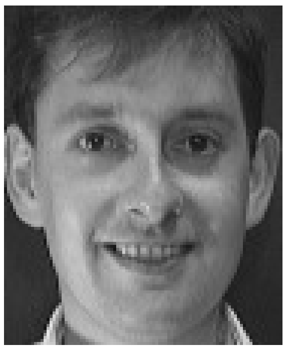
$$\vec{w} = V^T \vec{x} \tag{7}$$

Each mean-subtracted image is a weighted sum of eigenfaces, and the k-dimensional vector, $\vec{w}$, represents the weights. Instead of each picture being a mn-dimensional vector, where mn is 10,304 with our set of images, it can instead be represented by a k-dimensional vector, where $k \leq$400. This makes comparing images much faster for facial recognition. However, when k is increased the accuracy of the facial recognition increases, but computation time also increases. If $\vec{x}$ is a weight vector representing the input image and $\vec{w}_i$, $i = 1, 2, \ldots, k$, is a weight vector representing the ith image in the image data set, the error between the two images is calculated by

$$e_i = ||\vec{x} - \vec{w}_i||^2 \tag{8}$$

If e_j is the smallest of all values of e_i, then the jth image of the dataset is the closest image to the input image.

(a) Input face


(b) Output image with 9 principal components [2]

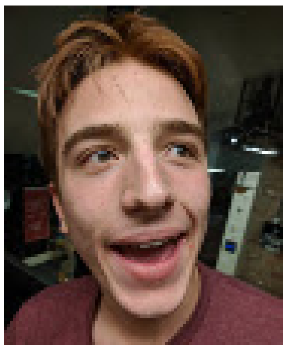

(c) Output image with 10 principal components

Figure 5: Comparing an input face and output faces with different number of principal components

As seen in Figure 5, more principal components leads to greater accuracy in facial recognition. Using this technique, 45 principal components is enough to obtain 100 percent accuracy in recognizing faces in the database we used. It can also be seen in the figure below that error seems relatively stabilized by that time.

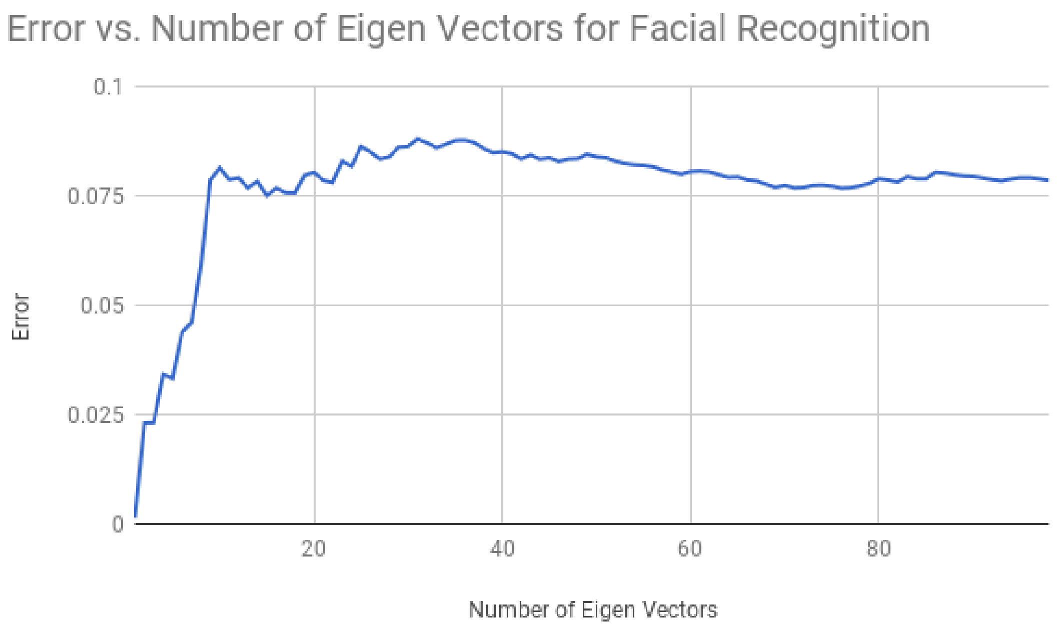


Figure 6: Error of facial recognition

3 Application to Breast Cancer Diagnosis

This technique can be applied to any other dataset where the data can be represented as a vector. For example, we decided to examine a dataset of mammograms of breast cancer and determine whether the tumor is benign or malignant. The code as shown in the appendix takes in an input mammogram, and returns the mammogram in the database that looks the most like the input mammogram. Based on whether the the return mammogram had cancer or not, the program will then return whether input mammogram is malignant or not. As we delve into the results of the application, it is important to note that the only known difference between a malignant tumor and a benign tumor is the growth rate of the malignant tumor so it should be impossible for a doctor to judge the tumor just based on a mammogram of it.[7] The goal of this experiment is to try to find out if there is a pattern in the appearance of tumor and its type. The database contained 1700 distinct images of tumors that were zoomed in by a factor of 400x. [9] Each image was 460x700 and was around 0.5 megabytes each. The first hurdle to work with the dataset was the scalability issue. Even with the dimension reduction methods mentioned above, the ram consumption in itself would make this software not work with regular computers. Each image in the database was resized to 100x152 so that the matrix calculations become possible.

Apart from the minor changes necessary to solve the image resolution problem, the mathematics for this application remained the same. 15 benign and 30 malignant tumors were given as inputs at the beginning. Only 50 principal components were used to describe the transformed space. Only 9/15 and 22/30 were accurately guessed which comes to about 68% accuracy. This number is not statistically significant enough for the us to conclude that the program recognizes malignant tumors well. With the current method being ineffective, we considered how our error changes with a differing number of eigenvectors. To find the margin of error for PCA we picked an image and observed how the error changed for the image based on the number of eigenvectors used. A graph depicting the information is shown below.

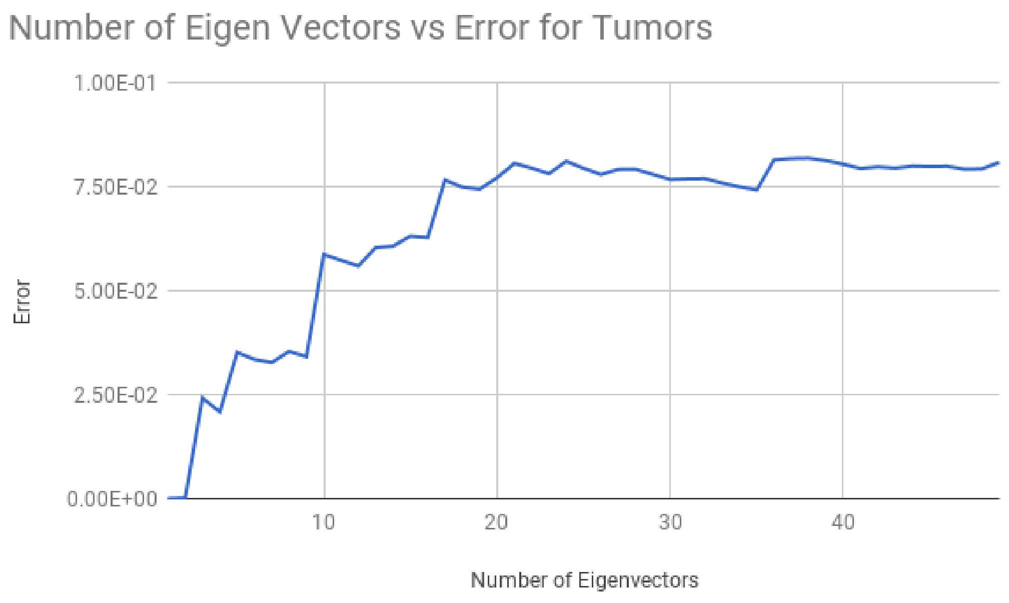


Figure 7: Error for cancer recognition

As seen by the graph above, the error seems to be increasing for the program until it uses at least 20 eigenvectors. It must be noted at this point that the program only starts to return accurately that the tumor is cancerous after using at least 21 eigenvectors, and then the error neither goes up or down. It is important to remember that the program is not accurate as the appearance of the tumors do not have a strong relation to whether they are malignant or benign. It can also be observed that error seems relatively stabilized by that time. Also, considering Figure 6 and Figure 7, it is safe to conclude that the number of eigenvectors used have no correlation to error. This also makes sense with the math described in the previous sections. As the eigenvectors are the basis of the space, ignoring the least impactful components forms a lower dimensional space. Forming a complete space might not necessarily maps to the image closer to the original face when compared to an incomplete space. This shows the power and usefulness of principal component analysis. The incomplete space with fewer eigenvectors will most likely work for most faces. Using more eigenvectors and wasting computational power does not seem feasible if the overall results are not going to change for most faces.

4 Conclusion

Based on the results of using eigenvectors for image recognition in both recognizing faces and determining whether a tumor is benign or malignant, we found this technique is computationally superior to other methods of image recognition. This efficiency comes from the result of reducing the dimension of each image for faster comparison. However, this process requires that all images be the same resolution and that key features be in the same area of each image. For example, the dataset of faces must have the eyes, mouth, nose, etc. in approximately the same place in each image, and lighting conditions must also be generally consistent. This is a likley reason behind the inaccuracy of cancer recognition. Most images of both benign and malignant cancers are very different, and there are many inconsistencies across images. Regarding facial recognition, a possibility for an extension of this study would be to use a larger dataset of faces. We only used 41 unique faces with multiple images of each person, and it would be interesting to see if accuracy diminishes with a greater number of different images. Regarding cancer detection, it is possible that image recognition could be used to determine a potential tumor regardless of its growth rate as the basis for image recognition lies in variation from the average. A mammogram far from the average could be correlated to a potential tumor.

Generally, it would also be possible to apply this process on other vector-represented datasets. These concepts are not exclusive to images, so other applications could include genome analysis, signal processing, and anything involving high-dimensional vector comparison.

References

[1] *Face Recognition with Python.* [Scholarly project]. In Bytefish.de. Retrieved December 10, 2017. https://www.bytefish.de/pdf/facerec_python.pdf

[2] AT&T Laboratories Cambridge Face Database. http://www.cl.cam.ac.uk/research/dtg /attarchive/facedatabase.html

[3] Zabrauskas, Manfred. *Eigenfaces Tutorial.* Retrieved December 3, 2017. http://blog.manfredas.com /eigenfaces-tutorial/

[4] Olver, Peter J. and Chehrzad Shakiban. *Applied Linear Algebra.* 1st ed. Published January 20, 2005. Pearson.

[5] Spruyt, V. (2015, March 22). A geometric interpretation of the covariance matrix. Retrieved December 11, 2017. http://www.visiondummy.com/2014/04/geometric-interpretation-covariance-matrix/

[6] Dallas, G. (2017, November 29). Principal Component Analysis 4 Dummies: Eigenvectors, Eigenvalues and Dimension Reduction. Retrieved December 11, 2017, from https://georgemdallas.wordpress.com/2013/10/30/principal-component-analysis-4-dummies-eigenvectors-eigenvalues-and-dimension-reduction/

[7] Hruban, R., M.D. (2016). What are Tumors? Retrieved December 11, 2017. http://pathology.jhu.edu/pc/BasicTypes1.php?area=ba

[8] Kakade, S. D. (2016). A Review Paper on Face Recognition Techniques. *International Journal for Research in Engineering Application & Management (IJREAM), 02* (02). doi:10.18411/a-2017-023

[9] Spanhol, F., Oliveira, L. S., Petitjean, C., Heutte, L., A Dataset for Breast Cancer Histopathological Image Classification, IEEE Transactions on Biomedical Engineering (TBME), 63(7):1455-1462, 2016. https://web.inf.ufpr.br/vri/databases/breast-cancer-histopathological-database-breakhis/

Appendix

A covar_eigen_face_lib.py

Python library for calculating covariance matrix and eigenvectors for face dataset. Includes other helper functions

```
import numpy as np
from PIL import Image
import os


def read_image(picture):
    img = Image.open(picture)
    img = img.convert("L")
    img_matrix = np.asarray(img, dtype=np.uint8)
    img_vec = np.array([])
    for row in img_matrix:
        img_vec = np.append(img_vec, row)
    return img_vec

def normalize(X, low, high):
    minX, maxX = np.min(X), np.max(X)
    X = X-minX
    X = X/(maxX-minX)
    X = X*(high-low)
    X = X + low
    return X

def covariance(path):
    vectors = []
    folders = os.listdir(path)
    folders.remove("README")
    folders.remove("newFaces")
    # folders.remove("avery")
    for folder in folders:
        newFolder = os.listdir(path + '/' + folder)
        for picture in newFolder:
            picture = path + '/' + folder + '/' + picture
            vectors.append(read_image(picture))

    vectors = np.vstack(vectors)
    avg = np.mean(vectors, axis=0)

    for index in range(len(vectors)):
        vectors[index] = vectors[index] - avg

    covar = np.dot(vectors, vectors.T)
    return vectors.T, covar, avg

def eigenStuff(vectors, covar, k):
    evals, evecs = np.linalg.eigh(covar)
    edict = {}
    for index in range(len(evals)):
        edict[evals[index]] = evecs[index]

    evals = sorted(evals, reverse=True)
    principle_components = evals[:k]
    newEvecs = []
    print("Number␣of␣eigen␣vectors:␣" + str(len(principle_components)))
    for val in principle_components:
        mult = np.dot(vectors, edict[val])
        newEvecs.append(normalize(mult, 0, 255))

    newEvecs = np.vstack(newEvecs).T

    np.savetxt("eigen_matrix.csv", newEvecs, delimiter=',')
    return principle_components, newEvecs

def find_weight(evecs, x, mean=0):
    weight = np.dot(evecs.T, x - mean)
    return weight/np.linalg.norm(weight)


def reconstruct(evecs, weights, mean):
    og = normalize(np.dot(evecs, weights) + mean, 0 ,255)
    return og


def vectorToImage(vector):
    """This function will convert the vector to images"""
    vec = []
    for index in range(112):
        vec.append(vector[index * 92:index * 92 + 92])
    vec = np.vstack(vec)
    img = Image.fromarray(vec)
    img.show()

def count_faces():
    files = os.walk('../faces/')
    count = 0
    for f in files:
        if '/s' in f[0]:
            count += len(f[2])
    return int(count)
```

B covar_eigen_cancer_lib.py

Python library for calculating covariance matrix and eigenvectors for mammogram dataset. Includes other helper functions

```
import numpy as np
from PIL import Image
import os
import sys


# Print iterations progress
def printProgressBar(iteration,
                     total,
                     prefix='',
                     suffix='',
                     decimals=1,
                     length=100,
                     fill='_'):
    """
    Call in a loop to create terminal progress bar
    @params:
        iteration   - Required  : current iteration (Int)
        total       - Required  : total iterations (Int)
        prefix      - Optional  : prefix string (Str)
        suffix      - Optional  : suffix string (Str)
        decimals    - Optional  : positive number of decimals in percent complete (In
        length      - Optional  : character length of bar (Int)
        fill        - Optional  : bar fill character (Str)
    """
    percent = ("{0:." + str(decimals) + "f}").format(
        100 * (iteration / float(total)))
    filledLength = int(length * iteration // total)
    bar = fill * filledLength + '-' * (length - filledLength)
    print('\r%s |%s| %s%% %s' % (prefix, bar, percent, suffix), end='\r')
    # Print New Line on Complete
    if iteration == total:
        print()

def read_image(picture):
    img = Image.open(picture)
    img = img.convert("L")
    baseheight = 100
    hpercent = (baseheight / float(img.size[1]))
    wsize = int((float(img.size[0]) * float(hpercent)))
    img = img.resize((wsize, baseheight), Image.ANTIALIAS)
    img_matrix = np.asarray(img, dtype=np.uint8)
    img_vec = np.array([])
    for row in img_matrix:
        img_vec = np.append(img_vec, row)
    if len(img_vec) != 15200:
        return None
    return img_vec

def covariance(path):
    vectors = []
    folders = ["benign", "malignant"]
    printProgressBar(0, 1807, prefix='Progress:', suffix='Complete', length=50)
    safe = 0
    for folder in folders:
        newFolder = os.listdir(path + '/' + folder)
        for cancer in newFolder:
            if "." not in cancer:
                cancer = path + '/' + folder + '/' + cancer
                sobs = os.listdir(cancer)
                for sob in sobs:
                    sob = cancer + "/" + sob + "/400X"
                    zooms = [sob + "/" + x for x in os.listdir(sob)]
                    for x in zooms:
                        tmp = read_image(x)
                        if tmp is not None:
                            vectors.append(tmp)
                        printProgressBar(
                            len(vectors),
                            1807,
                            prefix='Progress:',
                            suffix='Complete',
                            length=50)
        if folder is folders[0]:
            safe = len(vectors)
            print(folder, safe)

    vectors = np.vstack(vectors)
    print("Shape of the vector is", vectors.shape)
    avg = np.mean(vectors, axis=0)

    for index in range(len(vectors)):
        vectors[index] = vectors[index] - avg

    covar = np.dot(vectors, vectors.T) / len(vectors)

    return vectors, covar, avg, safe

def eigenStuff(vectors, covar, k):
    evals, evecs = np.linalg.eigh(covar)
    edict = {}
    for index in range(len(evals)):
        edict[evals[index]] = evecs[index]
    principle_components = np.array([evals[0]])
    evals = sorted(evals, reverse=True)
    principle_components = evals[:k]
    newEvecs = []
    print("Number of eigen vectors: " + str(len(principle_components)))
    for val in principle_components:
        mult = np.dot(vectors, edict[val])
        newEvecs.append(mult)

    newEvecs = np.vstack(newEvecs).T

    return principle_components, newEvecs


def find_weight(evecs, x, mean=0):
    weight = np.dot(evecs.T, x - mean)
    return weight


def reconstruct(evecs, weights, mean):
    og = np.dot(evecs, weights) + mean
    return og

def vectorToImage(vector):
    """This function will convert the vector to images"""
    vec = []
    for index in range(100):
        vec.append(vector[index * 152:index * 152 + 152])
    vec = np.vstack(vec)
    img = Image.fromarray(vec)
    img.show()


def count_faces():
    files = os.walk('../breast/')
    count = 0
    for f in files:
        if '/S' in f[0]:
            count += len(f[2])
    return int(count) - 13
```

C facial_recog.py

Python script that takes in an image of a face and returns closest image in database.

```
#!/usr/bin/python3
import numpy as np
from PIL import Image
import covar_eigen_lib as cel
import os

def recognize_face(inputFace, k=100):
    """
        inputFace is a path to a picture
        epsilon is 0.85 by default but user can specify
    """
    path = os.path.realpath(__file__).split("/")
    path = path[0:len(path) - 1]
    path = "/".join(path)
    path += "/../faces"
    vectors, covar, avg = cel.covariance(path)
    transVec = vectors.T
    #pca, newEvecs = cel.eigenStuff(vectors, covar, k)

    currdir = os.listdir()
    curr_k = len(np.genfromtxt("eigen_matrix.csv", delimiter=',').T)
    face_count = cel.count_faces()
    if "eigen_matrix.csv" not in currdir or face_count != len(transVec) or k != curr_
        print("Eigenmatrix is not correct, creating eigenmatrix")
        cel.eigenStuff(vectors, covar, k)
    newEvecs = np.genfromtxt("eigen_matrix.csv", delimiter=',')

    in_vec = cel.read_image(inputFace)
    in_weight = cel.find_weight(newEvecs, in_vec, avg)
    weights = np.array([cel.find_weight(newEvecs, x) for x in transVec])

    err = [np.linalg.norm(in_weight - row) for row in weights]
    index = np.argmin(err)
    min_err = err[index]
    print("Min Err:", min_err)
    if min_err < 0.07:
        print("This face exists in our database")
    elif min_err < 0.14:
        print("This image is most likely a face but does not exist in our database")
    else:
        print("Please make sure the input image is a face")
    cel.vectorToImage(in_vec)
    cel.vectorToImage(transVec[index] + avg)

if __name__ == '__main__':
    path = os.path.realpath(__file__).split("/")
    path = path[0:len(path) - 1]
    path = "/".join(path)
    path += "/../path/to/image"
    recognize_face(path, k=10)
```

D cancer_recog.py

Python script that takes in an image of a face and returns closest image in database.

```
#!/usr/bin/python3
import numpy as np
import covar_eigen_cancer_lib as cel
import os, sys


def recognize_cancer(inputFace, k=100):
    """
        inputFace is a path to a picture
        epsilon is 0.85 by default but user can specify
    """
    path = os.path.realpath(__file__).split("/")
    path = path[0:len(path) - 1]
    path = "/".join(path)
    path += "/../breast"
    transVec, covar, avg, safe = cel.covariance(path)
    myf = open("overall.csv", 'w')
    myf.write("index, predic, real, accurate, err\n")
    for vec_index in range(20):
        in_vec = transVec[vec_index]
        transVec = np.delete(transVec, vec_index, 0)
        vectors = transVec
        avg = np.mean(vectors, axis=0)

        for index in range(len(vectors)):
            vectors[index] = vectors[index] - avg

        covar = np.dot(vectors, vectors.T) / len(vectors)

        vectors = transVec.T
        tempSafe = 0
        if vec_index < safe:
            tempSafe = safe - 1

        principal, newEvecs = cel.eigenStuff(vectors, covar, k)
        in_weight = cel.find_weight(newEvecs, in_vec, avg)
        weights = np.array([cel.find_weight(newEvecs, x) for x in transVec])
        err = [np.linalg.norm(in_weight - row) for row in weights]
        index = np.argmin(err)
        predic = 0
        real = 0
        acc = 0
        if index < tempSafe:
            print("Is Probably Not cancerous")
        else:
            print("Is probably cancerous")
            predic = 1

        if vec_index < safe:
            print("Real: not cancer")
        else:
            print("Real: cancer")
            real = 1
        if real is predic:
            acc = 1
        min_err = err[index]
        mystr = str(vec_index) + ", " + str(predic) + ", " + str(
            real) + ", " + str(acc) + ", " + str(min_err) + "\n"
        myf.write(mystr)
                np.insert(transVec, vec_index, in_vec)

    for vec_index in range(safe+1, safe+21):
        in_vec = transVec[vec_index]
        transVec = np.delete(transVec, vec_index, 0)
        vectors = transVec
        avg = np.mean(vectors, axis=0)

        for index in range(len(vectors)):
            vectors[index] = vectors[index] - avg

        covar = np.dot(vectors, vectors.T) / len(vectors)

        vectors = transVec.T
        tempSafe = 0
        if vec_index < safe:
            tempSafe = safe - 1

        principal, newEvecs = cel.eigenStuff(vectors, covar, k)
        in_weight = cel.find_weight(newEvecs, in_vec, avg)
        weights = np.array([cel.find_weight(newEvecs, x) for x in transVec])

        err = [np.linalg.norm(in_weight - row) for row in weights]
        index = np.argmin(err)
        predic = 0
        real = 0
        acc = 0
        if index < tempSafe:
            print("Is Probably Not cancerous")
        else:
            print("Is probably cancerous")
            predic = 1

        if vec_index < safe:
            print("Real: not cancer")
        else:
                print("Real: cancer")
                real = 1
            if real is predic:
                acc = 1
            min_err = err[index]
            mystr = str(vec_index) + ", " + str(predic) + ", " + str(
                real) + ", " + str(acc) + ", " + str(min_err) + "\n"
            myf.write(mystr)
            np.insert(transVec, vec_index, in_vec)
        myf.close()

if __name__ == '__main__':
    path = os.path.realpath(__file__).split("/")
    path = path[0:len(path) - 1]
    path = "/".join(path)
    path = "/path/to/image"
    recognize_face(path, k=50)
```