Introduction to Machine Learning

KSINMAL1KU

Amir Souri
asou@itu.dk

IT University of Copenhagen
May 22, 2020

Contents

Chapter 1

Assignment One

1.1 Design of the library

I did not utilize any discovered design-pattern in my library. The reason is that python does not force the developer to create any class in order to run and use the script. It is a nice feature of python and you cannot do the same in java, c#, c++, etc. The design of my library is providing the user a set of useful functions to perform the points or image transformation without instantiating an object. The user directly can call all the functions from the library by importing the file, *transformations.py*. It is a sort of functional programming approach but suffers from side effects. I believe it is fine since it is not necessary to run functions in parallel as well as there is not a huge amount of data at least in this application. The library has a few functions to perform transformations such as rotating, translating, scaling, etc. Imagine the user wants to rotate an image 90 degrees and then scale its height and width by 0.5. To do so, the user should first create those required transformation matrices. Let's do it:

```
from transformations import *
R = rotating(90)
S = scaling(.5, .5)
```

Now the user has to combine them using *'combine'*:

```
RS = combine(R, S)
```

And to apply it on an image, calls *'transform_image'*:

```
transform_image(image, RS)
```

Of course, the user should load the image using cv2

```
image = cv2.imread('path/image.jpg')
```

If the user wants another transformation, let say this time a translate transformation:

```
T = translating(42, 42)
```

Then

```
RST = combine(RS, T)
```

And now perform it to a set of point just for fun by calling *'transform_points'*:

```
points = np.array([ [10, 20], [100, 200], [21, 21] ])
transform_points(image, RS)
```

1.1.1 Implementation details

I implemented a *combine* function that takes as many as arbitrary numbers of transformation matrices as its argument. The function returns the combined transformation matrix. As you know, the order is opposite when you manually multiply the transformation matrices. That is if you want to transform a point first by translating or shifting it by 10 (say T matrix) and then scale it by 2 (say S matrix), you have to multiply S by T (S • T). But I hided this inconvenience for the user, simply by first reversing the order of transformations and then combine them with a built-in function called *reduce*. Therefore, the user can first translate or shift a point (say p) by 10 and then scale it by 2, using the *'combine'* function as follows.

```
com = combine(T, S)
```

Note that T and S can be achieved by calling *translating* and *scaling* function from the library as I did before. And to apply the combined transformation on the point, p, the user can utilize the *transform_points* function as previously demonstrated. Note that the user is allowed to pass a *numpy array* or even a *list* for a single point to the function like this, [5, 5].
The above discussion applies to transform an image as well. The only difference is that the user should use the *transform_image* function instead of *transform_points*. For more details please inspect the actual implementations in the file, *transformations.py* .

1.1.2 Backward mapping

Backward mapping is a technique to map backwardly an image using an *affine matrix*. To make it clear that why it is used, let's assume we want to scale an image by 1.7 in direction of the x-axis and to be more simple by 1

in direction of the y-axis using forward mapping[1]. Here by forward mapping I mean to transform an image f(x,y) to g(x',y') by multiplying the scale transformation matrix by each points in the input image f(x,y) to obtain the output image g(x',y'). If the size of the image is 300 x 200, it will end up 510 x 200. We restrict our attention to the x-axis since the problem will appear there, in this example. Let's see how forward mapping works:

$$\begin{bmatrix} 1.7 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} x'_i \\ y'_i \end{bmatrix}$$

x	0	1	2	3	4	5	6	7	8	...	300
x'	0	1.7	3.4	5.1	6.8	8.5	10.2	11.9	13.6	...	510

The x and y coordinate must be an integer (the position of pixels) , but as you can see the values of x-axis in the output are floating point. I can round them off to the nearest value e.g. 10.2 to 11 and 11.9 to 12. But now x'=11 has no value, hence a hole in the image output. The problem is the same when using rotation transformation. To tackle this problem, backward mapping comes to play. It goes through all the pixels in the **output image** g(x',y'), one pixel at a time. And for each position (x',y') it uses the inverse transformation matrix to calculate the position (x,y) in the **input image**. In this way, it knows where in the input image a pixel must com from in order to be mapped to (x',y'). Lets see how the backward mapping works in general:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ 0 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

You may argue the backward mapping might result in a value of (x,y) that does not exist e.g. f(42.7, 85.4). Well, I simply can round it off to f(43, 85). But there is also a few better and more computational demanding approaches.

1.1.3 Image tests

I opted the *laa_laa* image that is shown in Figure 1.1 from the input directory and apply two different transformations on it. Each of which comprising combination of three transformations. The first one translates the image in both x and y direction 21 pixels and then scales it in both x and y direction

[1]The example is take from PM (P. 143)

2 times and at the end rotates it 45 degrees. This is shown below in Figure 1.2. The second one rotates the image 20 degree and then translates it in both x and y direction 7 pixels and at the end scales down it by 1/2. This is shown in the Figure 1.3 in the next page.

Figure 1.1: Input Image

Figure 1.2: After shifting, scaling, rotating

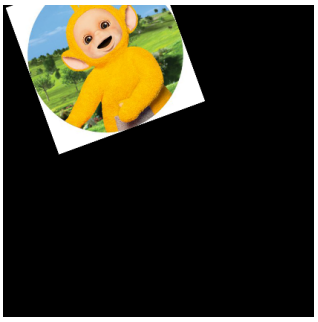


Figure 1.3: After rotating, shifting, scaling

Let's take a look at the code to apply those transformation:

```
t, s, r = translating(21, 21), scaling(2, 2), rotating(45)
r2, t2, s2 = rotating(20), translating(7, 7), scaling(.5, .5)
com_str = combine(s, t, r)
com_rts = combine(r2, t2, s2)
tsr = transform_image(img_laa, com_str)
rts = transform_image(img_laa, com_rts)
cv2.imshow('Before', img_laa)
cv2.imshow('After_21translated_2scaled_45rotated', str)
cv2.imshow('After_20rotated_7translated_0.5scaled', rts)
```

1.1.4 Manual tests

$$R = \begin{bmatrix} cos(\theta) & sin(\theta) & 0 \\ -sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.93969262 & 0.34202014 & 0 \\ -0.34202014 & 0.93969262 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$T = \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 7 \\ 0 & 1 & 7 \\ 0 & 0 & 1 \end{bmatrix}$$

$$S = \begin{bmatrix} x & 0 & 0 \\ 0 & y & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.5 & 0 & 0 \\ 0 & 0.5 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$S \bullet T \bullet R = \begin{bmatrix} 0.46984631 & 0.17101007 & 3.5 \\ -0.17101007 & 0.46984631 & 3.5 \\ 0 & 0 & 1 \end{bmatrix}$$

$$combine(R, T, S) = \begin{bmatrix} 0.46984631 & 0.17101007 & 3.5 \\ -0.17101007 & 0.46984631 & 3.5 \\ 0 & 0 & 1 \end{bmatrix}$$

1.2 Tests

I defined 3 arbitrary (random) points whose values are between 1 and 500. A scaling matrix that scales the points in the direction of the x-axis and the y-axis by two quantities, which are randomly chosen between -11 and 10. A translating matrix that shifts the points in the direction of the x-axis and the y-axis by two quantities, which are randomly chosen between -400 and 400. A rotating matrix that rotates the points θ degrees which is a random number between -360 and 360. I combined them using the combine function to use it to transform the points. Then I used that transformed points and the points to find the *affine* transformation using the *learn_affine* function. Finally, I compared the resulting transformation matrix (*affine transformation*) with the one I created (combined matrices). I replicated the above test 100 times and all of them passed.

1.2.1 Number of solutions

I believe that not all the combinations of points produce a valid solution. Consider the pints [[2, 2], [3, 3], [4, 4]]. If you write it as a linear system of equation and try to solve it you will find out that the equations of this linear system are dependent. Hence it has either infinitely solutions or no solution. You can verify the linearly dependence by observing the last equation that can be derived from the first equation or vice versa. Moreover, this set of points does not span R^2. Note also the determinant is 0. However, the *Moore–Penrose* inverse A^+ can find the inverse even for a singular matrix. *OpenCv* provides a function to find the *Moore–Penrose* inverse namely *linalg.pinv*. The *Moore–Penrose* inverse is also called the *pseudoinverse* of matrix.

1.3 Evaluation of initial implementation

My implementation works awesome. It is fast and smooth. The only issue i countered was, there could be a transformation matrix which is singular. Therefore it throws an exception from *learn_affine* function (I implemented it). The main point is to know that all the square matrices do not have necessarily an inverse form. I recorded an example and you can find it in the output directory, *noninvertible.webm*.

1.3.1 Improvement

I opted the Caching improvement and implemented it. In the *_update_patches* method I iterated over input *indices*. And inside the for loop inspected the *cached* list to determine whether it is *None*. If so, I call the *_mask_patch* and store its returned value to the *patches* and *cached*. Therefore, I could use the *cached* values next time to avoid the redundant computations. Otherwise, I leveraged the existing cached values to save time. At the end, I utilized the *update* method for updating the already cached variable. So that if the input *indices* were *None*, it assigns the same length as the patches (the number of triangles) to the *cached* . Otherwise, it assigns all the *caches* values to *None*. Although this improvement improves the application behind the scene, but human cannot sense this improvement in practice, maybe because it was fast enough before! You can observe what I mean by watching the video *vid_1* and the improved one *vid_2* in the output directory or by running the application without this improvement. I commented out the code snip before improvement code snip inside the *_update_patches* method. Nevertheless, a more precise way of testing the difference would be to calculate the time it take to the recalculations. Last but not least I did not encounter any issue besides those I mentioned before.

Chapter 2

Assignment Two

2.1 Implementation details (Detectors)

The *find_glints* function masks the iris on the input image using *bitwise_and* and *circle* method. The center coordinate of the circle is the center of the given pupil. After converting the image to grayscale, the function applies the threshold method on that. It applies the opening approach [1] using *erode* and *dilate* methods as follows. First erode image one time and then dilate it two times with the same 3x3 kernel. The next step is to find contours and compute the center of each contour. Afterward, it computes all the distances between the detected glints and the center of the pupil. Finally, it returns at most four glints which have the smallest distances from the center of the pupil. That is, it is possible the function returns less than four glints but it throws an exception if it won't return any glint. Both *find_pupil and find_glints* functions throw an exception if the *findContours* method does not return any contour.

2.1.1 Theory

An edge in an image is defined as a position where a significant change in gray-level values occurs[2]. According to the reference in this course, they can be found by employing the Grass-Fire Algorithm[3]. Essentially it starts in the upper-left corner of a binary image and selects the first pixel whose value is one. Then for each its neighboring pixel (up to the given connectivity) it tests if the neighboring pixel value is as well one. If so it also selects that

[1] PM (P. 90)
[2] PM (P. 73)
[3] PM (P. 94)

pixel and repeats the test on the new pixel's neighbors. It repeats this for each new pixel selected until there is no more connected pixel that contains one. Then it labels it as the first object and stores its boundaries as the set of **contours**. There are many different algorithms that can be used such as *Square Tracing*, *Moore-Neighbor Tracing, Radial Sweep, and Theo Pavlidis'* which are beyond the scope of this course.

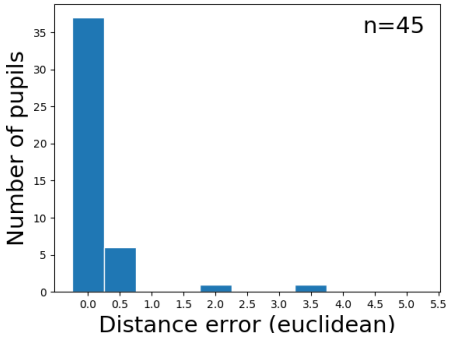
2.1.2 Evaluation

I start with pupil detection. Most of the time the distance error is less than two. The precision is reasonable for all datasets but pattern3. It can be observed by looking at Figure 2.1. However, it is more error-prone when detecting pupils form pattern2 and pattern3 images, especially in pattern3. It can be verified in Table 2.1. The reason is that the ground-trust pupils in that datasets were not correctly detected for a few images. It applies to other datasets but less. An expert might have caused the mistake when detecting the pupils. I stored all of them in a sub-directory in output, called *mistake*. Looking at Table 2.1, the precision of glint detection is not satisfying although it detects most of the glints correctly. It is subtle in Figure 2.2. It gets worse when the person is moving his head. But the most important issue is the data is noisy. i.e the reflection of some light source in the eyes. The source could be the Sun, a lamp, or even a monitor!

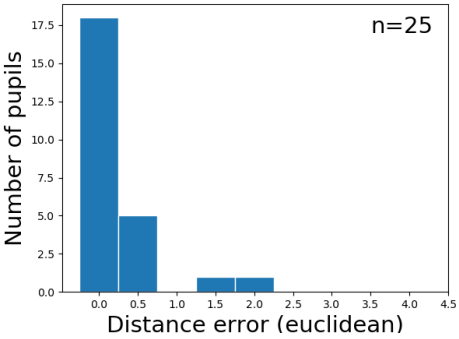
	pupil			glint	
Dataset	Mean	Median	RMSE	Mean	Median
pattern0	0.42	0.27	0.52	2.32	0.56
pattern1	0.50	0.34	0.52	3.14	0.51
pattern2	0.55	0.31	0.68	1.70	0.56
pattern3	2.38	0.50	2.58	1.55	0.56
moving_medium	0.57	0.31	0.68	2.87	0.51
moving_hard	0.52	0.30	0.91	2.88	0.54

Table 2.1: Metric for pupils and glints

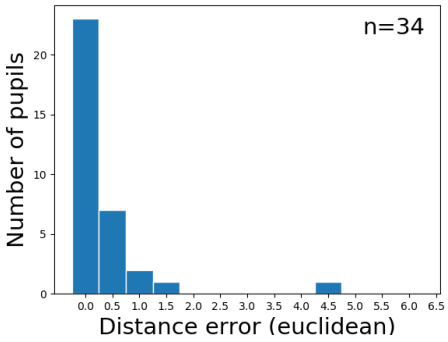
Figure 2.1: **Histograms of distance error for pupil**



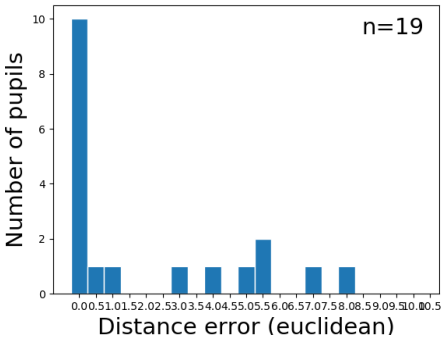
(a) pattern0



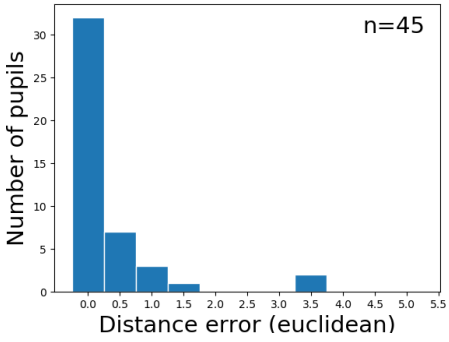
(b) pattern1



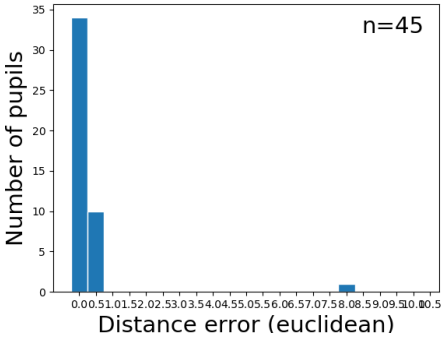
(c) pattern2



(d) pattern3

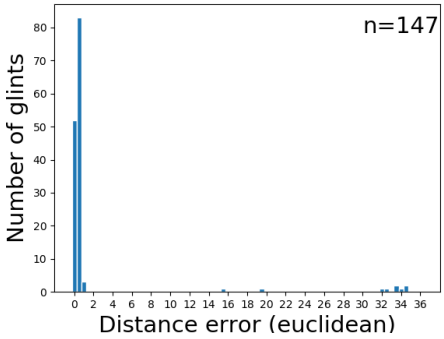


(e) moving medium

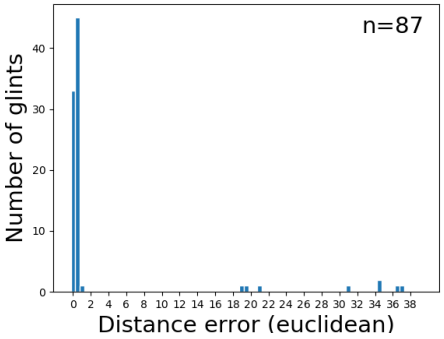


(f) moving hard

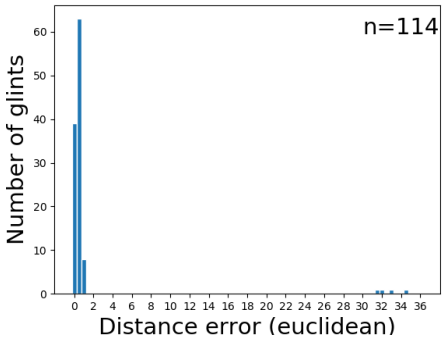
Figure 2.2: **Histogram of distance error for glints**



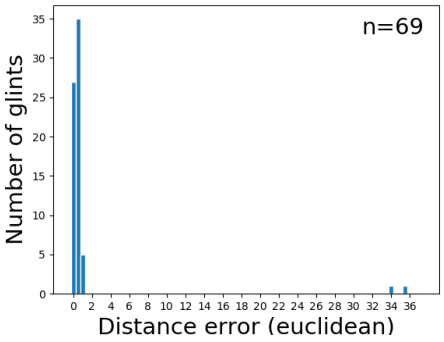
(a) pattern0



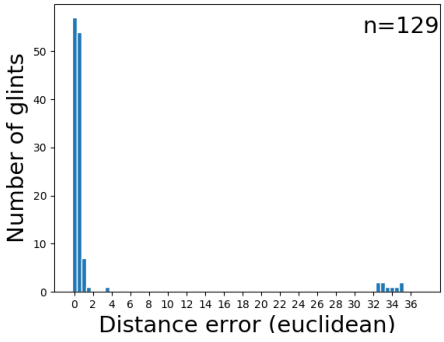
(b) pattern1



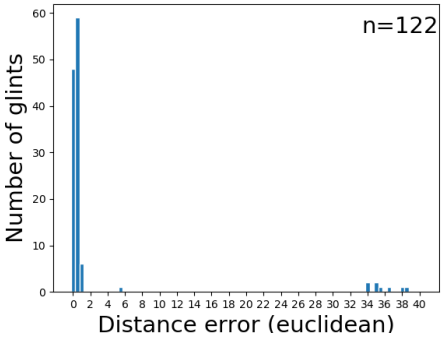
(c) pattern2



(d) pattern3



(e) moving medium



(f) moving hard

2.2 Implementation details(Gaze Estimation)

Inside the *calibrate* method, I used the *find_pupil* function from the detector module to get the pupils from the passed calibration images. Having the pupils, I found their centers to define the design matrix. The centers were float but I rounded them. Since the gaze coordinates were int (in pixel coordinates). Afterward, I passed the design matrix and the gaze positions to *numpy.linalg.lstsq*, to get parameters of the linear model. I did that two times, one for x and the other one for y separately, using the same design matrix. Eventually, it returns the parameters of the linear model.
Inside the *estimate* method, I called the *calibrate* method to get the parameters of the linear model. Then I multiplied them by the created design matrix from the untrained image to estimate the screen coordinates.

2.2.1 Evaluation

The gaze estimator does a good prediction on the *no head movement* image sequences. Though looking at Table 2.2 it seems a terrible model with the huge error metrics, it is an acceptable estimation of the gaze position on the screen with respect to the assumption about a linear relationship between pupil position in the image and gaze. Figure 2.3 shows the distance errors between the real gaze positions and the predicted gaze positions. Again it looks a poor model. However, when visualizing the ground-truth gaze positions and the predicted gaze positions utilizing the *position_vis.py* script, one can observe a sensible prediction. Of course, the estimation is not accurate and there is a long way to estimate the exact gaze position (pixel) on the screen based on the pupil position. Nonetheless, it is the best linear model can do. The model works with *no head movement* image sequences, except for 5 images in *pattern0*. Figure 2.4 and Figure 2.5 show those cases regarding the screen and image respectively. It actually estimates a gaze position but that is out of the screen. I append an example where gaze estimation is inaccurate. 2.4f 2.5f.

Prediction on *head movement* and rotation image sequences is another issue. The model cannot predict any gaze position in most cases. It predicts only around 7 out of 36 untrained gaze positions on each *moving_medium and moving_hard* datasets. The reason is obvious. The model trained on the *no head movement* image sequences but is exposed to estimated the gaze positions given the *head movement* and rotation image sequences.
Table 2.3 shows the correlation between gaze distance error and pupil distance error. The minus sign indicates a negative correlation. As you can see

the pupil distance error has an impact on the gaze distance error. e.g. in the row *pattern3*, the correlation is 0.59 which is quite high. That is, when the variance of pupil detection goes up, the variance of gaze detection goes up with 0.59 degrees. I believe it is better to consider one square around the ground-truth pixel and then calculate the distance from the predicted gaze point to that square on the screen. It is also fair to measure the distance from the whole pupil to the ground-truth pixel instead of the center of the pupil.

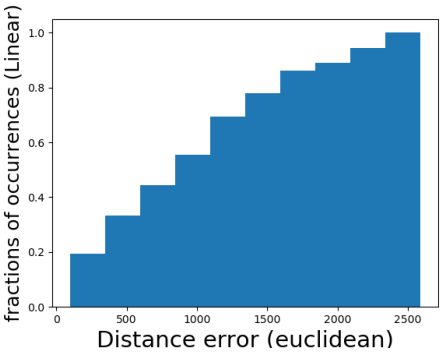
correlation	(Linear)			(Polynomial degree=2)		
Dataset	Mean	Median	RMSE	Mean	Median	RMSE
pattern0	1030.04	916.70	875.68	1022.56	904.35	880.27
pattern1	721.49	609.23	647.58	686.42	555.21	614.94
pattern2	744.16	647.09	684.44	719.99	555.37	669.48
pattern3	320.24	269.52	272.05	277.61	249.99	229.52
moving_medium	1804.45	1686.46	1345.12	1370.48	1293.28	1016.77
moving_hard	2109.26	2117.88	1677.19	1913.81	1560.39	1651.98

Table 2.2: Metrics for gaze

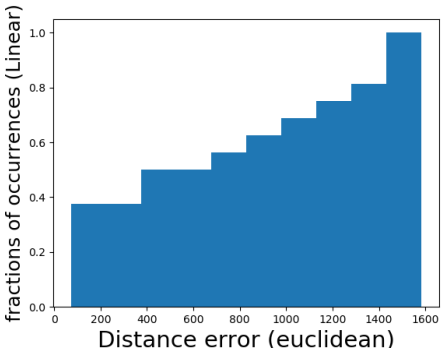
Dataset	correlation (Linear)	correlation (Polynomial degree=2)
pattern0	-0.12433938	-0.10022176
pattern1	-0.43212909	-0.41339488
pattern2	-0.08654306	0.09349133
pattern3	0.5800449	0.48023971
moving_medium	0.04077033	0.00800395
moving_hard	-0.22831533	-0.16972835

Table 2.3: Correlation between gaze distance errors and pupil distance errors

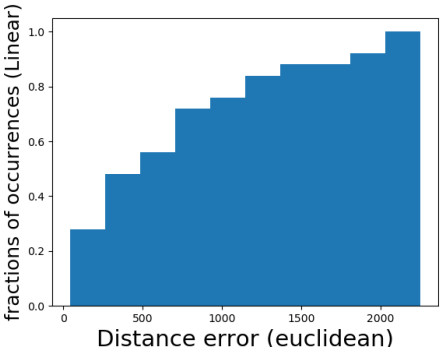
Figure 2.3: **Histograms of distance error for gaze (Linear)**



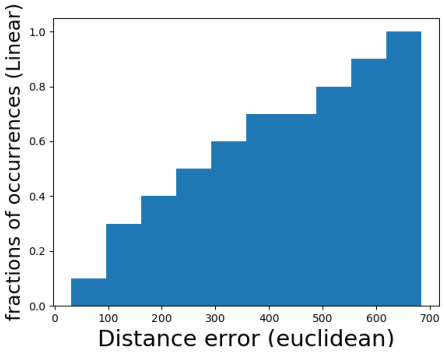
(a) pattern0



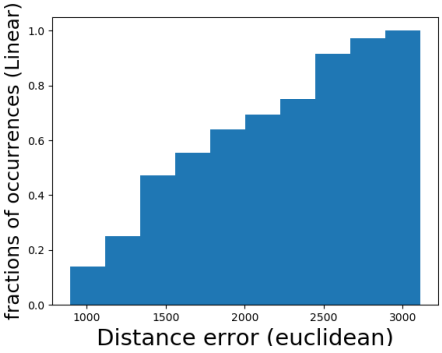
(b) pattern1



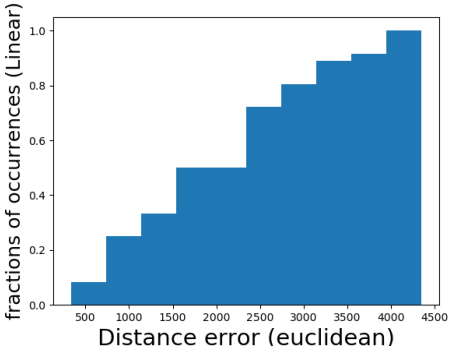
(c) pattern2



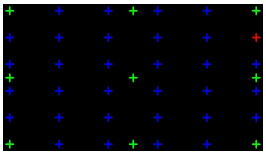
(d) pattern3



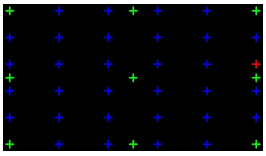
(e) moving medium



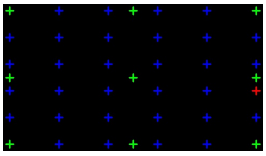
(f) moving hard



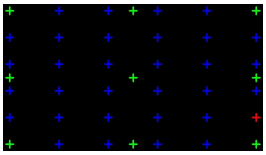
(a) image1



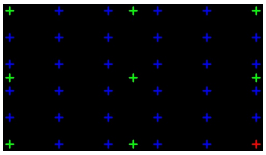
(b) image2



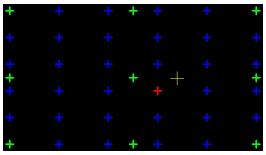
(c) image3



(d) image4

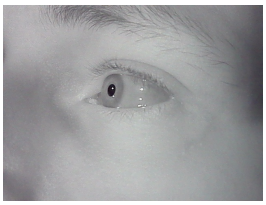


(e) image5

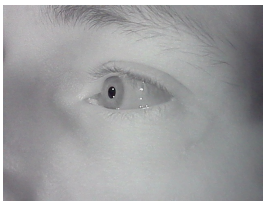


(f) inaccurate

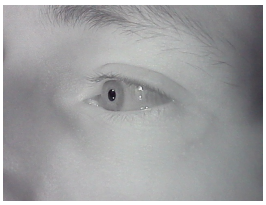
Figure 2.4



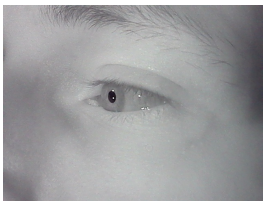
(a) image1



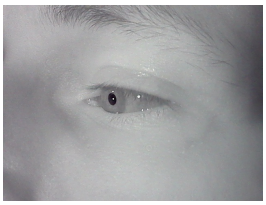
(b) image2



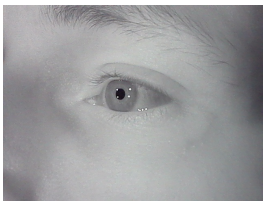
(c) image3



(d) image4



(e) image5



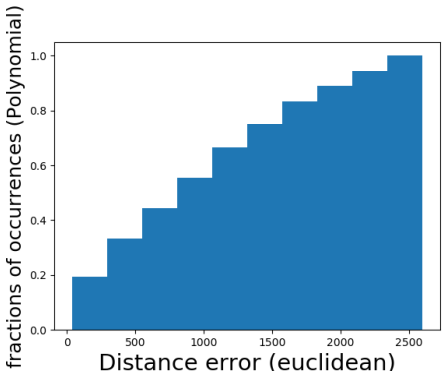
(f) inaccurate

Figure 2.5

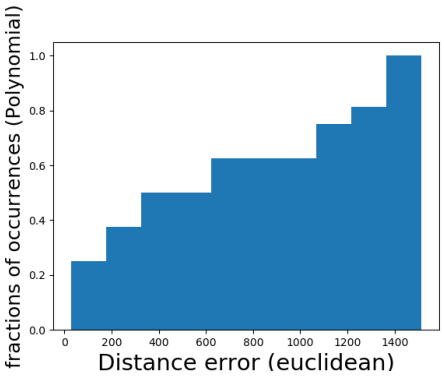
2.3 Improving detection

Inside the *calibrate* method, I found centers of the pupils with the *find_pupil* function and also fetched the ground-truth positions. Utilizing *PolynomialFeatures and fit_transform methods* I created the design matrix. I instantiated the model using *Linear Regression* and trained the data with its *fit* method. Inside the estimate method, I estimated the gaze position, leveraging predict method. The metric are shown in Table 2.2 and the correlation is demonstrated in Table 2.3. Finally, the histograms are shown in Figure 2.6. I experimented with order 2 to 7 and chose order 2 since it seemed that order 2 works best overly for all datasets. The improvement is subtle by looking at Table 2.2 and comparing the linear model with the polynomial model. It is worth noting that one order might predict better given a certain testing dataset and worse on the other ones. When using a linear function, it fits a straight line to the training set. This straight line does not have the flexibility to accurately replicate the arc in the "true" relationship. No matter how it is fit the line, it won't curve. That inability to capture the "true" relationship is called bias. As the straight line cannot be curved, it has a relatively large amount of bias. On the other hand, using a polynomial function, it fits a squiggly line to the training dataset. By increasing the order of the polynomial function it fits very well on the training dataset. Hence it has very little bias. The squiggly line can fit the training data so well that the least square error is 0 or really near 0. It is true only for the training dataset but when predicting the test dataset, everything changes. That is, the least square error for a testing dataset with the squiggly line might be terrible compared to the straight line. If the order is too large then the algorithm is *overfitting* the data. Thus the least square error will be large when measuring the error on the test dataset. The difference in fits between training and testing dataset is called variance. It is hard to predict how well the squiggly line will perform with feature data sets. It might do well sometimes, and other times it might do terribly. All in all, the straight line has high bias and relatively low variance because sums of square error are very similar for different datasets. The ideal algorithm has low bias and can accurately model the true relationship. It should have low variance by producing consistent predictions across different datasets as well. It can be done by finding the sweet spot between a simple model and a complex model. There are three common methods for finding that sweet spot, namely, Regularization, boosting, and bagging.

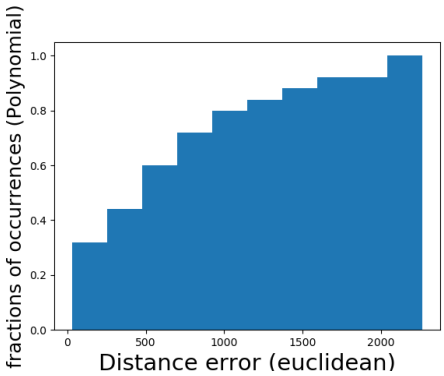
Figure 2.6: **Histograms of distance error for gaze (Polynomial, degree=2)**



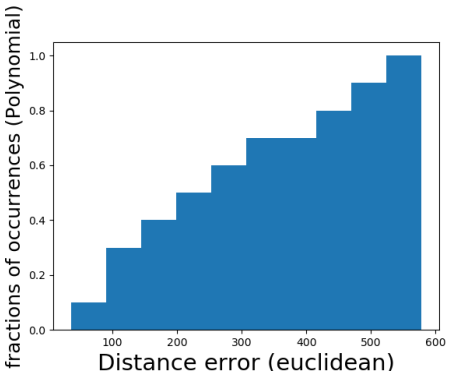
(a) pattern0



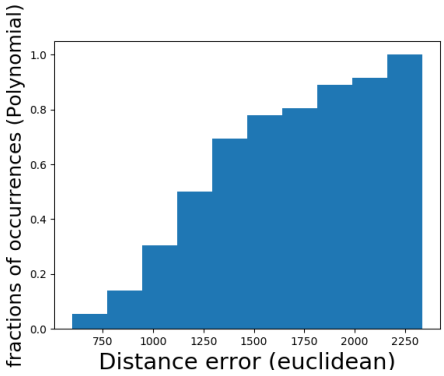
(b) pattern1



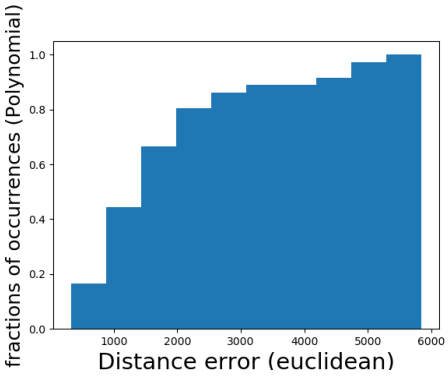
(c) pattern2



(d) pattern3



(e) moving medium



(f) moving hard

Chapter 3

Assignment Three

3.1 Introduction

Given the *FashionMNIST* dataset, I am supposed to implement several models to train and test the data. The models are as follows. Logistic regression, Support vector machine (linear kernel), Support vector machine (polynomial kernel), K nearest neighbors, multilayer perceptron (MLP), and convolutional neural network (CNN). The dataset is consisting of 28 x 28 pixel grayscale images associated with one of ten classes of clothing articles. A total of **60,000 training** samples and **10,000 test** samples are provided.

3.2 Background

There is a lot of machine learning methods. They can be categorized within three learning styles, Supervised Learning, Unsupervised Learning, and Semi-Supervised Learning. In Supervised Learning, training data has a known label or result such as spam/not-spam or a stock price at a time. In Unsupervised Learning, training data has not labeled and does not have a known result. In Semi-Supervised Learning, training data is a mixture of labeled and unlabelled examples. In this assignment, my focus is on Supervised Learning algorithms.

Logistic Regression is similar to the Linear Regression except Logistic Regression predicts whether something is True or False instead of predicting something like height based on weight. Instead of fitting a line to the data Logistic Regression fits an S-shaped logistic function. The curve goes from 0 to 1. It finds the S-shaped line using maximum likelihood[1]. A modification of the Logistic Regression uses the *softmax* function instead of the *sigmoid*

function (S-shaped) to predict a target variable having more than 2 classes. e.g among 10 different classes, it predicts an image belongs to which class.

Support Vector Machine (SVM) finds a line or a hyper-plane (if the target variable has more than 2 classes) that distinctly classifies the data points [2]. The most important characteristic of the SVM classifier is, it tries to achieve a good margin. The shortest distance between the observation and the threshold (line) is called margin. It aims to find a threshold that gives the largest margin to make classifications, it is called Maximum Margin. It is sensitive to outliers in the training dataset. To overcome this problem, SVM should be allowed misclassification. The linear and polynomial kernel are simply different in case of making the hyperplane decision boundary between the classes. The kernel functions are used to map the original dataset (linear/nonlinear) into a higher dimensional space with a view to making it linear dataset[3].

k-nearest neighbor is another classification method that is done in a super simple way. When a data point is going to be classified, the algorithm will look at its nearest labeled neighbors and classify that as the category of its neighbors. The k refers to the number of neighbors that are going to be considered as neighbors. To select the K that's right for the data, one should run the KNN algorithm several times with different values of K. And choose the K that reduces the number of errors observed while maintaining the algorithm's ability to accurately make predictions when it's given data that it hasn't seen before.[4].

Multi-layer Perceptron (MLP) has an input layer and an output layer and some arbitrary number of hidden layers (at least one). Each node in hidden layers and output layer is a neuron that uses a nonlinear activation function e.g *relu, softmax,* and *cross-entropy.* It utilizes a supervised learning technique called *backpropagation* for training. It has the characteristic of fully connected layers[5].

Convolutional Neural Network (CNN) is a powerful artificial neural network technique. There is three main part of a CNN model. The first part is the convolution which is the first layer to extract features from an input image. Convolution preserves the relationship between pixels by learning image features using small squares of input data. The second part is applying non-linearity and the third part is the pooling operation which allows downsampling the special resolution of the image[6].

3.3 Experiments

Although neural networks are generally more time consuming than the other discussed model, They do a better job. Especially if they are used on image data. It took about two and a half hours to train the data on my machine for a CNN model with the neural network architecture as follows.

```
class CNN_Net1(nn.Module):
 def __init__(self):
  super(CNN_Net1, self).__init__()
  self.conv1= nn.Conv2d(in_channels=1,out_channels=32,
                                    kernel_size=5)
  self.conv2= nn.Conv2d(in_channels=32,out_channels=64,
                                    kernel_size=5)
  self.fc1= nn.Linear(in_features=64*4*4*1,out_features=128)
  self.fc2= nn.Linear(in_features=128, out_features=10)
```

Neural networks are parametric models. That is, unlike the models already introduced, they take some parameters e.g learning rate. **Neural network** models require more effort to set up than already discussed models. It is a complex task to calculate the new image size after a *convolution* and the output of the *max-pool* operation. There is no such complex task in other models (discussed models). When training the data, one should be careful about how many iterations the model should be done (epochs). The model might **underfit** with setting little epochs while setting large epochs might cause **overfitting**. However, one can utilize a regularisation technique to avoid overfitting. To do so, I added three dropout layers to the CNN network and implement early stopping for the training algorithm. I named it *("CNN_Net0_Req_Patience=3")*. It is a naive early stopping implementation. I kept track of the loss function values and whenever the certain number (patience) of last loss values were increasing, I broke the loop. I believe it would be much better if I could use *pytorchtools* for implementing the early stopping [7]. The CNN is best suited to this particular task (in general image classification) because of its high accuracy. see Figure 3.1. Nevertheless, the early stopping implementation reduced training time and it makes sense because dropout layers ignore nodes (i.e. neurons) during the training phase of certain set of neurons which is chosen at random. By "ignore", I mean these units are not considered during a particular forward or backward pass. i.e individual nodes are either dropped out of the net with probability 1-p or kept with probability p. I set the p to **0.5**.

3.4 Results and evaluation

Before evaluating the results, I want to mention that I set the epochs parameter to **111** for all the neural networks models. Besides, the parameters, *transform, optimizer, lr, and batch_size* have been set with the same arguments so they were in the same condition.
All the **Accuracy, Precision, and Recall bar-graphs** for all the models in this assignment are shown in Figure 3.1, Figure 3.2, Figure 3.3, respectively to make the comparison easier. Additionally, I provide an example of Confusion Matrix in Figure 3.4 and an example of Image grid in Figure 3.5. Looking at Figure 3.1 and ignoring the neural networks, it seems the best model is the Support Vector Machine (SVM) with polynomial kernel. But the k-nearest neighbor model and even the Logistic Regression model have almost the same accuracy as (SVM polynomial kernel) model. So it is not so wise to apt the model only based on its accuracy value. One should use the Precision and Recall metric to pick the best model as well. However, it is hard to see the trade off for Precision and Recall, dealing with 10 classes. It is beneficial to calculate the weighted average of Precision and Recall which is called, F_1 **score** [8]. On the other hand, the *MLP_0* model is the loser in this comparison but *MLP_1* is better the other mosels (not neural networks model). Both *MLP_0* and *MLP_1* are a MLP model but with different architecture. blow is their architectures.

```
class MLP_Net0(nn.Module):
  def __init__(self):
    super(MLP_Net0, self).__init__()
    self.linear1=nn.Linear(in_features=784,out_features=21)
    self.linear2= nn.Linear(in_features=21,out_features=10)
```

```
class MLP_Net1(nn.Module):
  def __init__(self):
    super(MLP_Net1, self).__init__()
    self.linear1 = nn.Linear(in_features=784,out_features=85)
    self.linear2 = nn.Linear(in_features=85,out_features=10)
```

Therefore, a low layer configuration leads to a poor result. see Figure 3.1 Nevertheless, the best overall model in this assignment with respect to accuracy metric is, *CNN_1* with the layer configuration illustrated in section Experiments. The other CNN with a lower layer configuration, *CNN_0*, is less accurate than *CNN_1*. Again it reveals the importance of implementing a reasonable network for the model. This can be a tough pill to swallow for beginners to the field of machine learning, looking for an analytical way to calculate the optimal number of layers and nodes, or easy rules of thumb to follow [9].

Finally, *CNN_Net0_Reg_patience=3* model which is a CNN model with early stopping implementation and dropout layers. Here is its architecture.

```
class CNN_Net_Regularisation(nn.Module):
  def __init__(self):
    super(CNN_Net_Regularisation, self).__init__()
    self.conv1= nn.Conv2d(in_channels=1,out_channels=8,
                                    kernel_size=5)
    self.conv2= nn.Conv2d(in_channels=8,out_channels=16,
                                    kernel_size=5)
    self.fc1= nn.Linear(in_features=16*4*4*1, out_features=32
                                    )     self.fc2= nn.Linear(
                                    in_features=32,
                                    out_features=10)
    self.dropout= nn.Dropout(p=0.5)
    self.dropout2d= nn.Dropout2d(p=0.5)

  def forward(self, x):
    x = F.relu(self.conv1(x))
    x = F.max_pool2d(x, kernel_size=2, stride=2)
    x = self.dropout2d(x)
    x = F.relu(self.conv2(x))
    x = F.max_pool2d(x, kernel_size=2, stride=2)
    x = torch.flatten(x, start_dim=1)
    x = F.relu(self.fc1(x))
    x = self.dropout(x)
    return F.relu(self.fc2(x))
```

I set the *patience* to 3. It seems that this model is lees accurate than the other two CNN models. There could be many reasons for that e.g the naive early stopping implementation or the low layer configuration. Anyway, with an optimistic view, this model might be able to do a better classification with a new dataset, **which is the primary goal of this model**.

Figure 3.1: **Accuracy bar-graph**

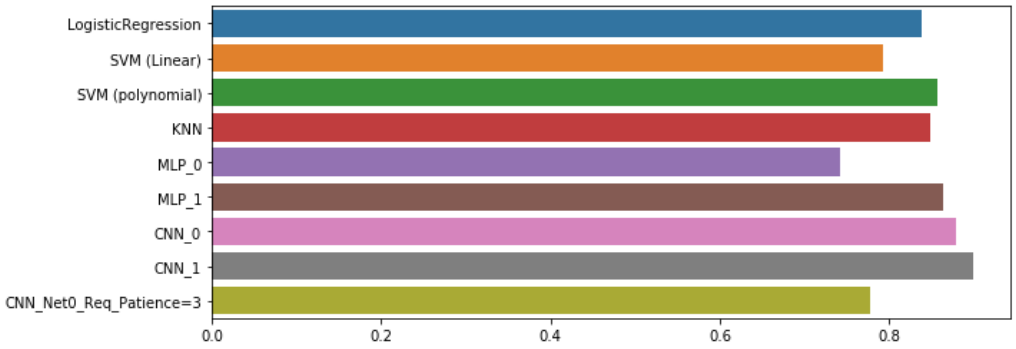


Figure 3.2: **Precision bar-graph**

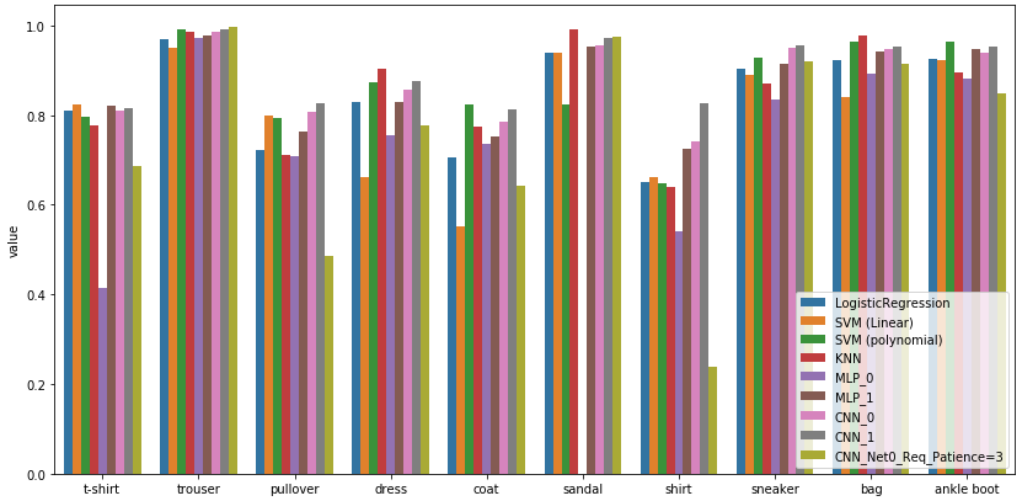


Figure 3.3: **Recall bar-graph**

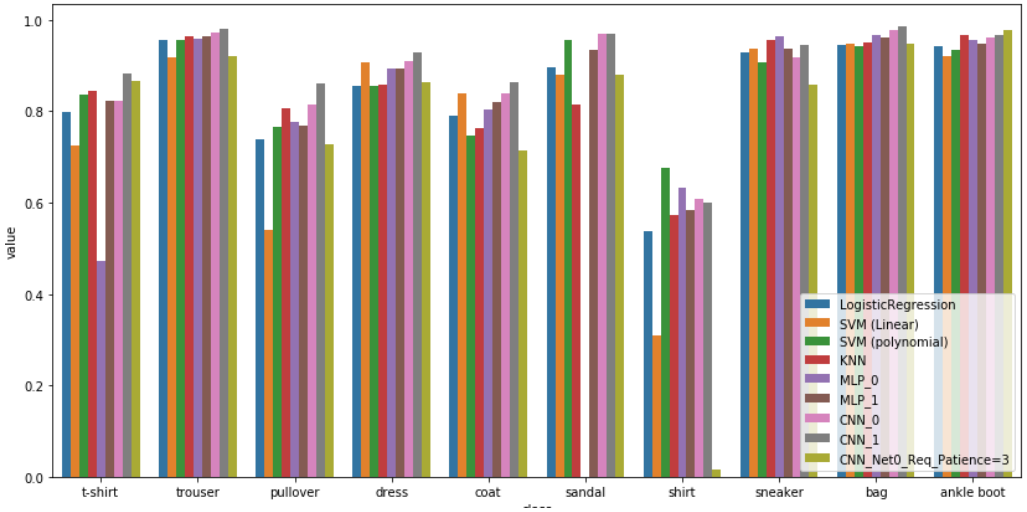


Figure 3.4: **Confusion matrix**

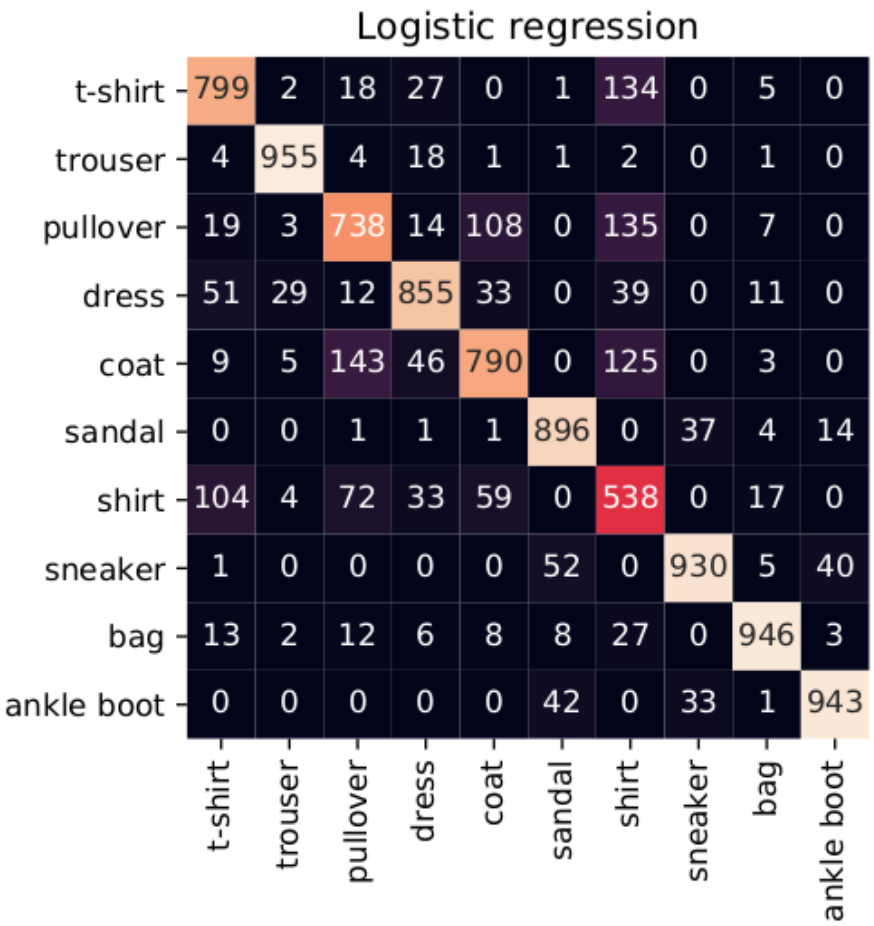
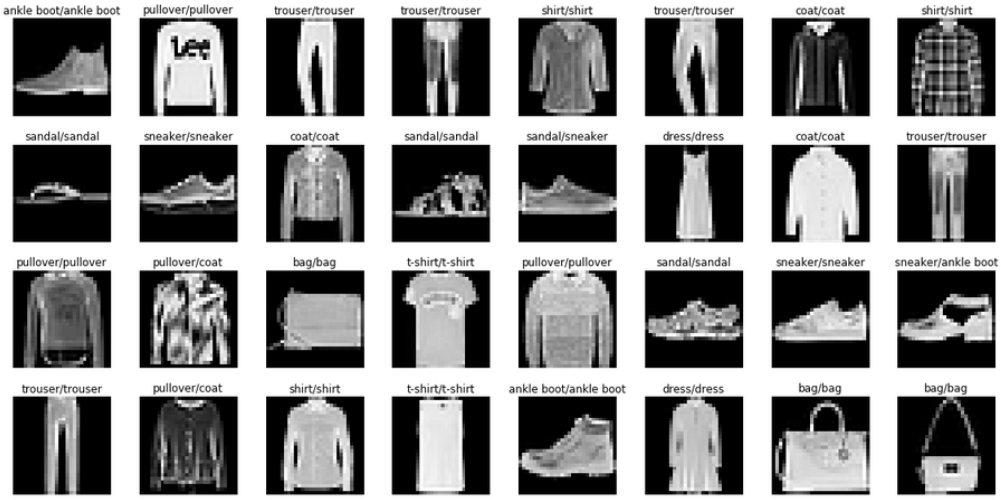


Figure 3.5: **Image grid**



Chapter 4

Assignment Four

4.1 Theoretical questions

4.1.1 Question 1

a)

$$A^4 = A*A*A*A = \begin{bmatrix} a_1 & 0 & 0 \\ 0 & a_2 & 0 \\ 0 & 0 & a_3 \end{bmatrix} * \begin{bmatrix} a_1 & 0 & 0 \\ 0 & a_2 & 0 \\ 0 & 0 & a_3 \end{bmatrix} * \begin{bmatrix} a_1 & 0 & 0 \\ 0 & a_2 & 0 \\ 0 & 0 & a_3 \end{bmatrix} * \begin{bmatrix} a_1 & 0 & 0 \\ 0 & a_2 & 0 \\ 0 & 0 & a_3 \end{bmatrix} =$$

$$\begin{bmatrix} a_1 * a_1 * a_1 * a_1 & 0 & 0 \\ 0 & a_2 * a_2 * a_2 * a_2 & 0 \\ 0 & 0 & a_3 * a_3 * a_3 * a_3 \end{bmatrix} = \begin{bmatrix} a_1^4 & 0 & 0 \\ 0 & a_2^4 & 0 \\ 0 & 0 & a_4 \end{bmatrix}$$

A is a diagonal matrix. Multiplying diagonal matrices just require multiplying corresponding diagonal elements. Therefore, if I want to raise the matrix A to the power of n, I just need raising each element in the main diagonal to the power of n.

b)

$$A\mathbf{v} = \begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} a_1 * v_1 + a_2 * v_2 \\ a_3 * v_1 + a_4 * v_2 \end{bmatrix}$$

4.1.2 Question 2

a) They are linearly independent since it has not a none-zero solution.

$$0 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + 0 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + 0 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The set of all linear combinations of set a is called the span. Then the span of set a is,

$$c_1 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + c_2 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + c_3 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

where c_i can be any real number i.e

$$7 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + 10 \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} + 12 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 7 \\ 10 \\ 12 \end{bmatrix}$$

They are a basis of $\mathbb{R}^3$ as they are linearly independent.

b) They are linearly dependent since it has a none-zero solution.

$$-2 \begin{bmatrix} 1.5 \\ 2 \\ 0 \end{bmatrix} + 1 \begin{bmatrix} 3 \\ 4 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The set of all linear combinations of set b is called the span. Then the span of set b is,

$$c_1 \begin{bmatrix} 1.5 \\ 2 \\ 0 \end{bmatrix} + c_2 \begin{bmatrix} 3 \\ 4 \\ 0 \end{bmatrix}$$

where c_i can be any real number i.e

$$4 \begin{bmatrix} 1.5 \\ 2 \\ 0 \end{bmatrix} + 5 \begin{bmatrix} 3 \\ 4 \\ 0 \end{bmatrix} = \begin{bmatrix} 17 \\ 28 \\ 0 \end{bmatrix}$$

They are not a basis of $\mathbb{R}^3$ as they are linearly dependent.

c) They are linearly independent since it has not a none-zero solution.

$$0 \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} + 0 \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} + 0 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The set of all linear combinations of set c is called the span. Then the span of set c is,

$$c_1 \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} + c_2 \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix} + c_3 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

where c_i can be any real number i.e

$$-3\begin{bmatrix}2\\1\\0\end{bmatrix}+7\begin{bmatrix}1\\2\\0\end{bmatrix}+2\begin{bmatrix}0\\0\\1\end{bmatrix}=\begin{bmatrix}1\\11\\2\end{bmatrix}$$

They are a basis of $\mathbb{R}^3$ as they are linearly independent.

d) They are linearly dependent since it has a none-zero solution.

$$-1\begin{bmatrix}1\\0\\0\end{bmatrix}+-1\begin{bmatrix}0\\1\\0\end{bmatrix}+-1\begin{bmatrix}0\\0\\1\end{bmatrix}+1\begin{bmatrix}1\\1\\1\end{bmatrix}=\begin{bmatrix}0\\0\\0\end{bmatrix}$$

The set of all linear combinations of set d is called the span. Then the span of set d is,

$$c_1\begin{bmatrix}1\\0\\0\end{bmatrix}+c_2\begin{bmatrix}0\\1\\0\end{bmatrix}+c_3\begin{bmatrix}0\\0\\1\end{bmatrix}+c_4\begin{bmatrix}1\\1\\1\end{bmatrix}$$

where c_i can be any real number i.e

$$1.5\begin{bmatrix}1\\0\\0\end{bmatrix}+2\begin{bmatrix}0\\1\\0\end{bmatrix}+0\begin{bmatrix}0\\0\\1\end{bmatrix}+1\begin{bmatrix}1\\1\\1\end{bmatrix}=\begin{bmatrix}2.5\\3\\1\end{bmatrix}$$

They are not a basis of $\mathbb{R}^3$ as they are linearly dependent.

4.1.3 Question 3

a) Translation, Scaling(Stretching, Squeezing), Rotation, Shearing ,and Reflection.

b) Its system of linear equations has four unknowns and thus requires four equations to have a unique solution. Therefore, it needs two points.

c) $x\prime = t_1x + t_2y$
$y\prime = t_3x + t_4y$

$$\begin{bmatrix}x & y & 0 & 0\\0 & 0 & x & y\end{bmatrix}\begin{bmatrix}t_1\\t_2\\t_3\\t_4\end{bmatrix}=\begin{bmatrix}x\prime\\y\prime\end{bmatrix}$$

4.1.4 Question 4

a)

$$\begin{bmatrix} 1 & 1 & 2 \\ 1 & 1 & 1 \\ 3 & 2 & 3 \end{bmatrix}$$

b)

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

4.2 Approach (Dimensionality reduction)

I implemented a function that takes the face shapes as an NxM matrix and returns the sorted principle components of the shapes and their corresponding ordered eigenvalues. I find the *mean* of the input to subtract it from the input (to center the samples). Then I calculate the *covariance matrix* of the centered data and used it to find **eigenvalues and eigenvectors**. I sorted the the eigenvectors by their associated eigenvalues to return them as the principle components. I also implemented two functions to transform from feature space to principal component space and from principal component space to feature space (*inverse_transform*). To do the former, I first centered the input features like I did before and then returned the principal component space by multiplying the input PCA (W) by the centered samples. To do the latter, I multiplied the transpose of the input PCA (W) by the vector input (principal component space) to get the feature space. Then returned the feature space plus the input mean. Here are their declarations.

```
def features_to_components(features, W, mu)
def components_to_features(vector, W, mu)
```

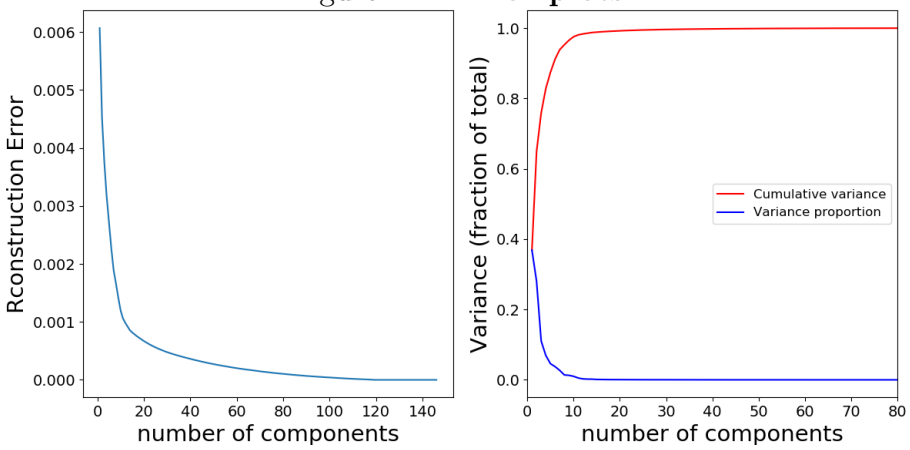
Having that three functions ready, I implemented another function to calculate the *reconstruction error*. Inside the function I called *features_to_components* function to get the principal component space and then passed it to *components_to_features* function to get back the features (**which have some error now**). Finally, I returned the root square mean of the initial features and the transformed back features as the *reconstruction error*. Additionally, I implemented two functions to derive *proportional_variance* and *cumulative_proportional_variance* given an arbitrary number of **sorted eigenvalues**. The first one sums all the eigenvalues and divide each eigenvalue by

the total sum and then returns the results. The second one follows the exact approach as the first one except it returns the **cumulative sum** of the results.

4.2.1 Results/Evaluation

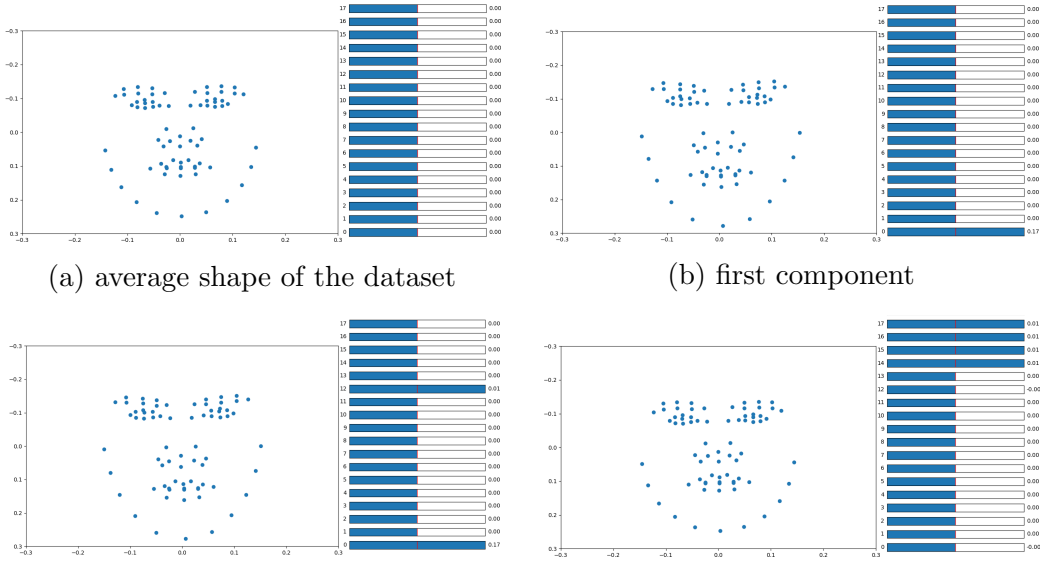
First and foremost, Let's see the plotted *reconstruction error*, *proportional*, and *cumulative proportional variance* in one figure.

Figure 4.1: **Error plots**



The left plot of Figure 4.1 indicates that even using a single eigenvector derived from a dataset, one can reconstruct a sample that was transformed to a lower dimension, with only 0.006 error (in this study).
The right plot of Figure 4.1 illustrates *Variance proportion* and *Cumulative variance*. In this plot, **eigenvalues were used instead of eigenvector.** The *Cumulative variance* approaches one mush earlier than the *reconstruction error* approaches 0. Since with a few eigenvalues it is possible to reconstruct all the variations in data.
Figure 4.2 shows four variation of the number of components to generate a new sample similar to the training data. I reconstruted a various different samples only using the first component. see Figure 4.2 (b). Using the first and thirteen components (c) it produced almost the same sample as in last sample (b). Moreover, using 15th, 16th, 17th, and 18th components did not produce a much different sample. That is, it is almost the same as the average shape of the dataset. Compare (a) and (d) in Figure 4.2. It makes sense because the first component is the most important component. It is the eigenvector with corresponding largest eigenvalue. Therefore, adding another component has a trivial impact on the result. It makes more sense if

Figure 4.2: **Varying the number of components**



(a) average shape of the dataset

(b) first component

(c) first and thirteen component

(d) 15th, 16th, 17th, 18th component

you look at Figure 4.1.

4.3 Approach (Clustering)

I extracted a list of scaled images of the face images dataset using the *read_image_files* from *dset*. I set its scale parameter to 0.1 since it took much time to fit the data, particularly when training data with *meanshift* on my machine. I looped over the list to get image *patches/windows* for each scaled face image. Then I merged all the windows into a single data matrix. Afterward, I instantiated three clustering algorithms, *KMeans, AgglomerativeClustering,* and *MeanShift* from *sklearn.cluster*. After that, I fitted the data into them to get the models. *KMeans* and *AgglomerativeClustering* are **parametric models**. The developer should provide the number of clusters to be formed (default is 8 and 2 respectively). I **randomly** set them to **20** since I did not have any clue about the number of cluster in data! Next, for all models I predicted a cluster for each window. Subsequently I used the array of the cluster indices to create a histogram for each image. Eventually, I set up *NearestNeighbors* (for all models) to fit calculated histograms for finding the **k closest neighbors** for a certain histogram using its *kneighbors* method.

4.3.1 Results/Evaluation

The number of clusters found by the *MeanShift* model is 2 and for the others are 20 as I set them when initializing them. This is the big advantage of *MeanShift* over the the two models. That is, it is able to automatically **determine** the number of **cluster** in data whereas those two need predetermined number of clusters which in many cases is impossible. Two plots of the cluster centers found by *KMeans* and *MeanShift* models are shown in Figure 4.1. All the three sample nearest neighbor outputs are as follows.

id	kmeans	Agglomerative	meanshift
0	0	0	0
1	2	0	0
2	2	0	0
3	2	0	0
4	2.44948974	0	0
5	2.44948974	1.41421356	0
6	3.16227766	3.16227766	0
7	3.46410162	4.24264069	0
8	3.46410162	4.24264069	0
9	3.46410162	4.24264069	0
10	3.46410162	4.69041576	0
11	3.46410162	5.29150262	0

Table 4.1: Distances

id	kmeans	Agglomerative	meanshift
0	7	2	0
1	2	1	47
2	8	7	8
3	6	6	11
4	1	8	73
5	58	0	7
6	101	58	44
7	63	61	45
8	60	63	113
9	62	60	76
10	59	9	1
11	66	59	72

Table 4.2: Indices

For the three models I found 12 nearest neighbors of the seventh histogram and plotted them together with their eleventh nearest neighbor as well as all their differences. The green histograms are the seventh histogram, the blue ones the eleventh nearest neighbors and finally, the red ones are their differences. Observations are illustrated in next page.

The visual word method is scalable and accurate. This method serves in natural language processing with some improvements. Anyway, I do believe this method is **not accurate enough** for finding images of the same person since the **objects** (face images) **belong to the same category**. It is accurate in the essence of object detection e.g predict whether an image contains a dog.

Figure 4.1:

Figure 4.2: **Kmeans**

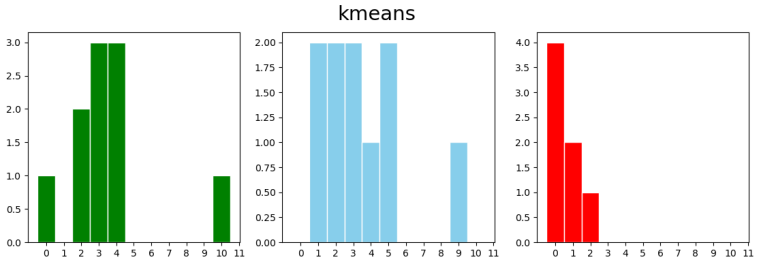


Figure 4.3: **Agglomerative**

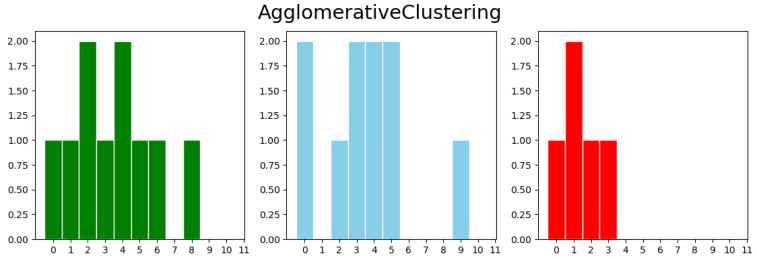
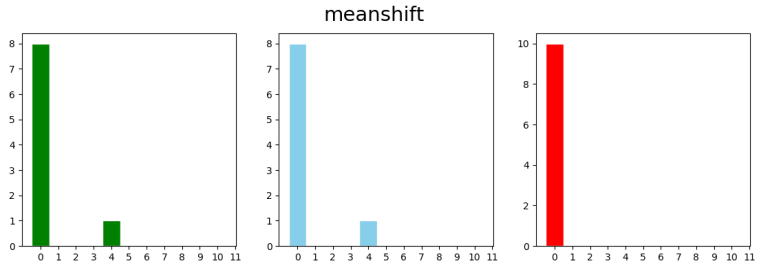


Figure 4.4: **Meanshift**



In conclusion, the *KMeans* and *Agglomerative* algorithms have less complexity than *MeanShift* which is **NP-hard**. *Agglomerative* model that is a **Hierarchical Clustering model**, does not seem to be a good candidate in this analyze. Apparently, it does not provide any API to predicate the new data. However, the **MeanShift** model is the victor at this stage. Not only because it is a **non-parametric model** but also a **better result** than other models.

References

[1] *StatQuest: Logistic Regression*. 2018. URL: https://www.youtube.com/watch?v=yIYKR4sgzI8&t=290s.

[2] Rohith Gandhi. *Support Vector Machine — Introduction to Machine Learning Algorithms*. 2018. URL: https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47.

[3] *Diffference between SVM Linear, polynmial and RBF kernel*. URL: https://www.researchgate.net/post/Diffference_between_SVM_Linear_polynmial_and_RBF_kernel.

[4] Onel Harrison. *Machine Learning Basics with the K-Nearest Neighbors Algorithm*. 2018. URL: https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761.

[5] Nitin Kumar Kain. *Understanding of Multilayer perceptron (MLP)*. 2017. URL: https://medium.com/@AI_with_Kain/understanding-of-multilayer-perceptron-mlp-8f179c4a135f.

[6] Alexander Amini. *Convolutional Neural Networks — MIT 6.S191*. 2020. URL: https://youtu.be/iaSUYvmCekI.

[7] *Early-stopping-pytorch*. URL: https://github.com/Bjarten/early-stopping-pytorch/blob/master/MNIST_Early_Stopping_example.ipynb.

[8] Andrew Ng. *Machine Learning System Design — Trading Off Precision And Recall*. 2017. URL: https://www.youtube.com/watch?v=W5meQnGACGo.

[9] Jason Brownlee. *How to Configure the Number of Layers and Nodes in a Neural Network*. 2018. URL: https://machinelearningmastery.com/how-to-configure-the-number-of-layers-and-nodes-in-a-neural-network/.