Project Log on WGEML Kin Recognition

1 Related Work

1.1 Main Paper - WGEML

- The main parts of the paper are the face detection, the four face descriptors: LBP, HOG, SIFT, VGG, the penalty graphs and intrinsic graph and then using the graphs to figure out how the faces in the images are related.

2 Implementation Notes

2.1 Testing

- A folder for unit tests is made to correspond to each of the modules of the source code. This folder is under the src file and the test file is further subdivided into each source folder.

- Unit testing is done using a combination of pytest and coverage. A make command is used to run the coverage command which references a .coveragerc file which makes sure that none of the __init__.py files, the venv or test files are included in the coverage report.

2.2 Face Detection

- Firstly, OpenCV2 was used to create a base implementation to draw a rectangle around a person's face in an image. This was done using the pre-trained classifier in "haarcascade_frontalface_default.xml". This allowed us to take a file image and output another saved file image which was the original picture with a rectangle around each face. The next step is to output an image of just the face and nothing else with the same dimensions.

- We were able to save the face on its own to an external image and change the dimensions of the outputted picture as needed. The current dimensions of the output is 64×64 as that is what the paper specified.

2.3   Feature Vectors

2.3.1   LBPs

- First, it was necessary to read a paper on LBPs applied to faces (Face Description with Local Binary Patterns: Application to Face Recognition).

- From the paper, it was found that there were 59 labels that each pixel can belong to. It can either be uniform or non-uniform and we only cared about the uniform labels. These were values where there were only at most 2 bit transitions circularly. For example, 10000001 (2 transitions) was uniform but 10101000 (6 transitions) isn't.

- It was necessary to first get the LBP value for each pixel in the image. This was done by looking at the direct neighbors of the pixel and determining if they are greater than or less than the pixel. If they were greater than the pixel, the value of that cell would be 1, otherwise it would be 0. Then, the value of the pixel in question was determined by looking at the pixel to the left and going counterclockwise and creating the bit string. In the following example:

5	8	1
5	**4**	1
3	7	2

$\longrightarrow$

0	0	1
0		1
1	0	1
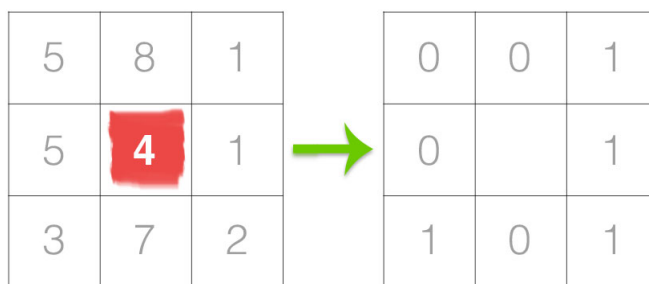
And so the value for the pixel becomes $01011100_2 = 92$. This value isn't uniform so this would have been marked as -1 in the process to mark it as non-uniform. This is done with every pixel in the image. For the pixels on the border, a neighborhood of size 3×3 was still taken but any "neighbors" that weren't in the image were assumed to be 0. So, for the top pixels, the 3 pixels above it were assumed to be 0, for example.

- After getting the LBP value for each pixel, the face image is split into 8×8 rectangular blocks and the vector is computed in each block. The vector is just a histogram of the values that each pixel could have been. Since there are 58 uniform values and 1 for any non-uniform values, there were 59 values that the pixels could have taken so the vector for each block would correspond to:

$$[\text{count}(-1), \text{count}(0), ..., \text{count}(255)]$$

Where the uniform values are ordered in ascending order.

The vectors for each block are then concatenated to each other where the top left block is first and then the block to the right of it and so on going row by row. This outputs the 3776 dimensional vector for each face image for a 64×64 image.

2.3.2   Histogram of Gradients

- First, to compute the gradients of the image, the Sobel operator was used. To approximate the gradient in the x direction, first the kernel:

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

  Was convolved on the image and then to get the gradient in the y direction, the kernel:

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

  Was convolved on the image. This let us get the gradients in both the x and y direction for each pixel and then that was converted into magnitude and angle. For each pixel, the maximum magnitude and maximum angle for the 3 channels was taken to be the magnitude and angle for that pixel. For example, if a pixel had magnitudes $(1, 2, 3)$ for the (R, G, B) channels, then 3 would be the magnitude of that pixel. We also required that the angles be unsigned, so they must be between $0°$ and $180°$.

- Once the magnitudes and the unsigned angles are obtained for the image, the image is split up into blocks of size $n \times n$, in which in our case, it is first 16×16 then 8×8. For each block, a 9-dimensional vector is obtained which is the histogram of angles for that block. The labels of the histogram are the angles:

$$[0, 20, 40, 60, 80, 100, 120, 140, 160]$$

  So if a gradient has angle $0°$, then it would count towards the first bin. Given a pixel with gradient with magnitude m and angle θ, if θ is one of the labels, then you would add m to the bar with label θ. For example, if $\theta = 0$, then vec[0] += m. If θ is between labels ϕ_1 and ϕ_2, then vec[ϕ_1] += $\dfrac{\phi_2 - \theta}{20} \cdot m$ and vec[ϕ_2] += $\dfrac{\theta - \phi_1}{20} \cdot m$. In other words, the amount that goes to each label is weighted with respect to the magnitude of the gradient and how close to the labels the angle of the gradient is.

- The vectors for each of the 256 blocks are created for when there are 16×16 blocks and then for the 64 blocks for when there are 8×8 blocks. The paper then doesn't seem to normalize the vectors so **that is a potential improvement on the algorithm** as normalization tends to improve performance.

- For a 64×64 face image, the vector that will come out of it will be a 2880-dimensional vector. The vectors for the 16×16 blocks are concatenated first and then the ones for the 8×8 blocks.

3

- Much of the information from this comes from the paper "Histograms of Oriented Gradients for Human Detection"

2.3.3 SIFT

- Read the paper "Distinctive Image Features from Scale-Invariant Keypoints" which was what introduced the SIFT algorithm.

- The first thing to do is to create octaves for the given image. An octave is a set of the given image being blurred multiple times. For example, in the first octave, you'll have the original image as the first image, and then you'll blur it a bit for the next image and then that image will be blurred for the next image in the octave, etc. In the second octave, the image is halved in size and the same blurring happens. So, if the original image was of size 64×64, then the images in the second octave will be 32×32 and in the third it would be 16×16 and so on. A specified number of octaves and blurred images are used for the SIFT algorithm. The way that the image is blurred is as follows:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$

Where x, y is the coordinate in the image, I is the function mapping coordinates to the value of the image at that coordinate, σ is the amount of blurring, $*$ represents convolving G on the image and:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Which is the Gaussian blur.

- From here, for each octave, a difference of Gaussians is created to help find the keypoints of the image. The difference of Gaussians is just the octave[i] - octave[i - 1]:
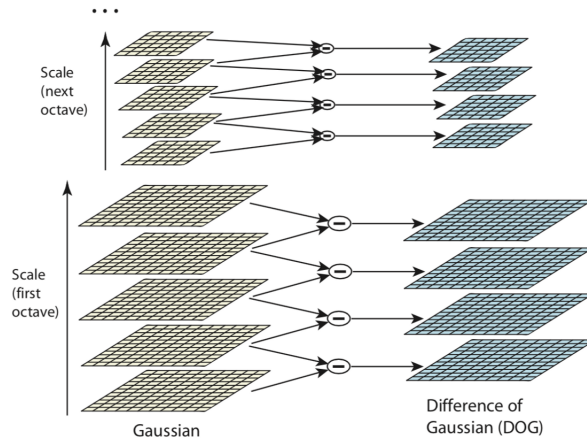
This is used to approximate the Laplacian of Gaussian as the LoG helps find the edges of the image by blurring the image a bit and then finding the second order derivatives.

It is first blurred as taking the Laplacian straight away would be sensitive to noise. However, this is computationally expensive. The Difference of Gaussians is a good approximation of the scale invariant Laplacian, $\sigma^2 \nabla^2 G$.

- From here we look for the keypoints in the image using the Difference of Gaussians. This is done by finding the local maxima and minima of the DoG. Once you have the extrema of the difference of Gaussians, we need to refine the approximation of the keypoint because the actual keypoint is probably between pixels. Thus, we can use a Taylor expansion around the proposed keypoint of the scale-space function $D(\sigma, x, y)$ where σ is the blur level in the DoG. This Taylor expansion looks like:

$$D(\mathbf{x}) = D + \frac{\partial D}{\partial \mathbf{x}}^T \mathbf{x} + \frac{1}{2}\mathbf{x^T}\frac{\partial^2 D}{\partial \mathbf{x}^2}\mathbf{x}$$

Where D is the value of D at the proposed keypoint, $\mathbf{x} = (\sigma, x, y)^T$, $\frac{\partial D}{\partial \mathbf{x}}^T = (\frac{\partial D}{\partial \sigma}, \frac{\partial D}{\partial x}, \frac{\partial D}{\partial y})$. Letting this equal 0, we get that the offset from our keypoint is:

$$\hat{\mathbf{x}} = -\frac{\partial^2 D}{\partial \mathbf{x}^2}^{-1}\frac{\partial D}{\partial \mathbf{x}}$$

Each of these were calculated using the numpy gradient function which is an approximation of the actual gradient of D. If our offset is greater than 0.5 in any dimension, then we want to try again since that means it's closer to another sample point. We keep trying again until we get an offset which is close to the sample point. We then find the value at the subpixel extrema:

$$D(\hat{\mathbf{x}}) = D + \frac{1}{2}\frac{\partial D}{\partial \mathbf{x}}^T \hat{\mathbf{x}}$$

And if the value of the extrema is less than 0.03×255, then we throw it out since it is an unstable extrema and has low contrast.

- Furthermore, we also eliminate any keypoints that are potentially on an edge. We can do this by looking at the hessian of the keypoint. We have that the Hessian is:

$$\mathbf{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{yx} & D_{yy} \end{bmatrix}$$

To determine whether something is a corner, we care about the eigenvalues of $\mathbf{H}$, or more specifically, the ratio between the eigenvalues. Using the trace and determinant of $\mathbf{H}$, we can find that:

$$\frac{\text{Tr}(\mathbf{H})^2}{\text{Det}(\mathbf{H})} = \frac{(r+1)^2}{r}$$

Where r is the ratio of the eigenvalues. Thus, letting the maximum ratio that the eigenvalues are allowed to be at be $r_0 = 10$, we just need to find if:

$$\frac{\text{Tr}(\mathbf{H})^2}{\text{Det}(\mathbf{H})} < \frac{(r_0 + 1)^2}{r_0}$$

If it is, then this is a proper keypoint. If it isn't then this is an edge so we can discard it.

- The actual paper that I was implementing didn't require the keypoints at all unfortunately. This was only realized after the keypoint orientation function was implemented. What the paper did was take 7×7 overlapping patches and, with each patch, obtained a SIFT-like feature vector. This just involved getting the gradient at each point in the patch and getting the magnitude and angle of the gradients. Then, the patch was split into a 4×4 grid and for each square in that grid, an 8 dimensional vector was obtained by doing something similar to HoG where a histogram of the angles was obtained but with only 8 bins and not having the angles being unsigned. This length 8 vector is then weighted by the Gaussian function where the center is from the center of the patch and each of these weighted 8-dimensional vectors are appended together and normalized. This is then appended to the overall feature vector. This is done for each patch in the face image and this gets a 6072-dimensional feature vector if the face image is of size 64×64.

Licensing

1. The licensing is given here `https://github.com/wiseman/sift/blob/master/LICENSE.ubc`

2. The US version of the patent is here `https://patents.google.com/patent/US6711293B1/en`

2.3.4 VGG

- Firstly, the original VGG16 architecture was looked at before looking at the one used for faces specifically. The main differences were in the number of neurons in the softmax layer and that there was a Dropout layer after the first 2 Dense layers. The paper used was (Parkhi et. al, "Deep Face Recognition").

- At first, I was going to train the network using the data but then was informed that this could take days and is very time-consuming.

- The alternatives that came from this were to either fine tune the network using Keras's built-in VGG function which was already trained on ImageNet and fine tune it to the needed dataset or to take the weights from `https://www.robots.ox.ac.uk/~vgg/software/vgg_face/`.

- The weights in the url were only in the forms of a matconvnet, torch, or caffe formats so if this was to be used, it would need to be transformed into something usable by Keras. This was done with the matconvnet format and using loadmat from scipy. The main difference from this and my version of the architecture, however, was that they had Convolutional layers instead of Dense layers but this was easily solved by

reshaping the weights from the Convolutional layers to what was required for the Dense layers.

- In the end, the VGG face descriptor is obtained by creating the VGG network, downloading the weights if it isn't already available, loading the weights into the network after reshaping it, removing the last softmax layer and then using this model to "predict" the face descriptor of the image.

2.4 Main Algorithm of The Paper

2.4.1 Notes on the Paper

- The paper does this algorithm for each relationship.

- They construct an intrinsic graph to characterize intraclass compactness and 2 penalty graphs to characterize interclass separability.

- "In the intrinsic graph, each pair with kinship relation is connected. In the penalty graphs, pairs without kinship relation and the K-nearest neighbors of their matching samples are connected accordingly." (The Paper, 2019)

- They aim to deal with the problem of high intraclass variance and low interclass variance. They also extract multiple features and fuse them together using the weighted graph embedding framework.

2.4.2 Notes on the Implementation

- Overall is a fairly straightforward algorithm in terms of the calculations needed to be done. The first steps were to create a function that would return w_p and U_p for each face descriptor p.

- The values of K and r and β were taken from the paper where they said that it was set as 5, 5 and 0.5 respectively.

- A few decisions were to be made. The first of which was whether I should implement k-nearest neighbors by myself or just use the sklearn version. I opted to use the sklearn version since implementing KNN isn't too interesting anyway and it's better to use the more optimized version anyway.

  The second thing I needed to figure out was how to solve the generalized eigenvalue problem:
  $$(\frac{1}{2}(D_{1p} + D_{2p}) + D_p)u = \lambda S_p u$$

  Since the top d, where $d \ll D$ and where D is the dimension of the feature vector, eigenvectors need to be obtained to get U_p. $U_p = [u_1, u_2, ..., u_d]$ where the corresponding eigenvalues follow $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_d$. In the end, I went with using scipy's

version of finding eigenvalues and vectors as it had a function to solve the problem. The main problem currently is that I'm getting all of the vectors and values and then sorting the values list to get the top d vectors.

- There is, at this point in the notes, confusion on what d actually will end up being when the entire algorithm is run.

- Unit tests were difficult to run on the function getting w_p and U_p since calculating these values by hand prove to be fairly tedious so I ended up using the functions I wrote to generate the outputs and unit tested them in depth. In these unit tests, $d = 2$ and $D = 4$ which meant that d wasn't much less than D and since I found that $U_p^T U_p \neq I$, I chalked it up to this reason as it seemed very close but a few entries in the resulting matrix were wrong.

2.5 Data Preparation

- It was at this point that the data started to be obtained from KinFaceW-I and KinFaceW-II. As we are using the datasets, **it is necessary to cite the original authors** "Citation Jiwen Lu, Junlin Hu, Xiuzhuang Zhou, Yuanyuan Shang, Yap-Peng Tan and Gang Wang. Neighborhood Repulsed Metric Learning for Kinship Verification, IEEE International Conference on Computer Vision and Pattern Recognition (CVPR'12), 2012." **in the final dissertation**.

- Functions were written to split up the data into their specified folds and to separate the positive pairs from the negative pairs.

- Next, a script was written that would take the name of the dataset and create maps of the path to each image in the dataset to the corresponding face descriptor. So, an LBP, HOG, SIFT, and VGG map was created for the dataset and then, using pickle, this was saved on disk for use later.

- Another script was then written which would take in the dataset, relationship and whether we were using the image-restricted or unrestricted setting. The restricted setting is just a setting as to whether or not we should give the algorithm negative pairs of the relationship to learn on, where unrestricted gives it to the algorithm and restricted doesn't. This script would then create the 5 training sets using the splits in the dataset and then create the w vector and U matrices for the training set and save it on disk.

- A problem that was obtained with this was that my implementation of WGEML was very slow when it had to calculate D_{1p} and D_{2p} and when it had to calculate the eigenvalues and eigenvectors. While not much could be done with the eigenvalue problem, to calculate the matrices, numpy was used more instead of 2 nested for

loops. For the eigenvalue problem, I ended up using the scipy function for getting eigenvalues but instead of having it solve:

$$A\mathbf{x} = \lambda B\mathbf{x}$$

I had it solve:

$$(B^{-1}A)\mathbf{x} = \lambda\mathbf{x}$$

Which made the function run faster, although it did give slightly different results and it wasn't as fast as I had hoped.

- Next, I was able to find the TSKinFace Dataset with help from my supervisor which was a different format than the KinFaceW datasets were so I had to do extra work to integrate it into my scripts which just involved creating the pairs arrays myself along with using sklearn to do get the splits.

- Next, since PCA was mentioned to have been used on each of the feature descriptors, PCA was implemented in to project the feature descriptors to a 200-dimensional space and then truncate it to the first 100 as this seemed to be what was written in the paper although there is confusion since the sentence was "we use PCA to project each feature representation to a 200-dimensional space and then set the reduced dimension as 100". This was later changed to just project it to a 200-dimensional space and then have the dimension of U be 200×100.

2.6 Prediction

- Prediction using the weights outputted from WGEML wasn't explicitly stated in the paper except for one figure in the beginning which stated:

$$sim(x, y) = \sum_p \frac{w_p}{2}\left(\frac{x_p^T U_p^T U_p y}{\sqrt{x_p^T U_p^T U_p x}\sqrt{y_p^T U_p^T U_p y}} + 1\right)$$

However, this similarity function didn't end up making sense as U_p had the dimension $D \times d$ so this would force x and y to be vectors of dimension d when D was stated to be the dimension of the vectors. Along with this, U_p was created such that $U_p^T U_p$ was the identity matrix which would make this similarity function, essentially, just the cosine similarity between the vectors in Euclidean space. Thus, instead of using $U_p^T U_p$, the matrix $A_p = U_p U_p^T$ was used such that:

$$sim(x, y) = \sum_p \frac{w_p}{2}\left(\frac{x_p^T A_p y}{\sqrt{x_p^T A_p x}\sqrt{y_p^T A_p y}} + 1\right)$$

was the similarity function used. Furthermore, the value that this needed to be compared to in order to get whether the images corresponding to x and y were of a kin relationship wasn't explicitly stated in the paper. Thus, this required to mess around with the value, θ.

2.7   Experiments

- All of the code was uploaded to an external GPU using scp. From there, a venv was set up and all of the necessary dependencies were installed in the venv. There were a few problems with how Tensorflow was running on the CPU instead of the GPU but that was fixed by upgrading the CUDA version and cuDNN versions.

- From there, the code was run end-to-end on KinFaceW-I on an unrestricted setting before looking at any other configurations. Predictably, the accuracies obtained weren't what were wanted as, at first, it was 0% accuracy for every relationship. This was because $\theta = 0.9$ in my implementation so lowering the number made proper accuracies appear. However, this still wasn't resembling the accuracies obtained in the paper. The negative pairs were added to the test set since, before, it was just the positive pairs, however that didn't change much at all.

- With a bit of testing, the theta value that maxes the accuracies was around $\theta = 0.63$ for the KinFaceW-I dataset on the unrestricted setting. However, the accuracies were still not right as some were way too high or low. For example, "ms" was at 0.75 when it should've been 0.806 with $\theta = 0.62$. Similarly, for 0.63, fs was at accuracy 0.824 when it should've been 0.785.

- When $\theta = 0.6$ and $d = 10$ for the U matrices, we get accuracies that are within 5% of the accuracies in the paper. However, these values weren't in the paper and I have a bad feeling about how my restricted implementation worked and since I got those values purely experimentally, work is still continued on fixing it up.

- When I varied d while keeping θ constant on the KinFaceW-I dataset on the unrestricted setting, I found that the minimum average difference between the results I get and the results from the paper occurs when $d = 21$. However, when tested with the rest of the datasets, the average difference was fairly low until it got to TSKin-Face where it was about 11.3% which is much higher than the difference for when $d = 10$. The lowest for TSKinFace was when $d = 2$ at a difference of 1.29% but that is too low for the rest of the datasets. It seems to be that when $d = 10$ is a good compromise between TSKinFace and the KinFaceW datasets.

- I initially ran the script end to end while having theta, the value that we compare the similarity of two images value to, be set to 0.9, the dimension of U set to 100 and PCA reducing the dimensions of the feature vectors to 200 dimensions. This ended up with me getting an accuracy of 0% though so I had to tweak each of those values. I first made the dimension of U to be 10 instead of 100 as I believed that 100 wasn't significantly less than 200. I was able to see a slight increase in accuracy but I ended up making theta lower to 0.5 and some good accuracies were obtained, although I don't remember what they were. Another thing I realized around here was that I was only testing the positive pairs and none of the pre-made negative pairs in the KinFaceW datasets so I made sure to include pairs that weren't of the relationship

that was being tested in the test set for KinFaceW. There were no pre-made pairs like that for TSKinFace so I ended up just using the positive pairs in my test set.

At this point, I tried varying theta and seeing at what value of theta would the maximum accuracy be for each dataset, restricted/unrestricted configuration and relationship, aside from TSKinFace as that would just say that making $\theta = 0$ would give the maximum accuracy since I only had positive pairs in the test set for TSKinFace so accepting every pair as a positive pair would make the accuracy 100% for that dataset. What I found for the KinFaceW datasets was that the accuracy was maximized at around $\theta = 0.59$ to $\theta = 0.65$ so I ended up settling on $\theta = 0.6$ as my boundary.

Once I had $\theta = 0.6$, I tried messing around with the PCA part of the script and found that there wasn't any significant difference in accuracies between having PCA reduce the feature vectors to 200 and running WGEML and reducing it to 200 and then truncating the vector to be a 100-dimensional vector. However, this seems to mimic what the paper said as they had, in figure 6, found that the accuracy stays fairly steady after the amount of dimensions in the feature vectors was 100. I also tried just not using PCA to see if that would improve it at all but that was a lot more computationally expensive than I realized as WGEML had to find eigenvalues and eigenvectors for matrices of the size $D \times d$, where D is the feature vector size, and it would have to do this for each feature vector for each fold in the cross-validation set and for each relationship and it was taking way too long for just one feature vector and one fold in one relationship so I didn't end up finishing that experiment. As such, I decided to leave it as a 100-dimensional vector that was truncated.

Finally, I tried to vary the dimension of the U matrices, as it is a matrix of size $D \times d$ where the value of d wasn't explained anywhere. I had a similar methodology as I did when I varied theta, which was just to try every value it could be and see where accuracy was maximized. I found that for the KinFaceW datasets, it was around 19-24 whereas for TSKinFace it was at 2. As such, I tried $d = 21$ but, as you can see from the attachment, although it did well on the KinFaceW datasets, it was terrible with the TSKinFace dataset. Similarly, when $d = 2$, TSKinFace does well but the KinFaceW datasets suffer. It seems like, by chance, I happened to pick a good compromise between the two datasets at the beginning and I stuck with $d = 10$ as my dimension size.

- I then added negative pairs to the TSKinFace dataset and added negative training and test pairs. When I added negative test pairs, the accuracy went down for the TSKinFace dataset for all relationships which leads me to believe that the original paper only tested it on positive pairs and not negative ones. However, there was negligible difference when I added the negative training pairs which I found to be the case for each dataset.

- Ablation studies were done by using all of the different combinations of the face descriptors for each of the dataset/restricted configurations. For example, using only

the VGG and HOG vectors for prediction was done. One thing that was found was the most obvious thing which is that adding more face descriptors makes the accuracy better in every case. However, a slightly more interesting thing is that the marginal accuracy gained for each face descriptor decreases every time a new face descriptor is added. My high-level guess as to why this is so is that as you add more face descriptors, the features that each face descriptor capture are more likely to overlap with each other, compared to when there's only one face descriptor, then another one could add a lot more information. However, if there is already 3 face descriptors, another face descriptor might share a lot of the same information the others already have and only add a tiny bit more.

In very few cases were there instances where a smaller set of face descriptors would have a higher accuracy than all 4. Many of these cases occurred in the KinFaceW-I restricted configuration and most of those in this configuration tended to have the VGG face descriptor and, most of the time, the SIFT face descriptor. There were 24 instances of (dataset, restricted, face descriptors used, kin relationship) configurations where the accuracy was higher than when all of the face descriptors for the same relationship and dataset configuration was used. Of these 24 instances, only 1 didn't use VGG which was the (KinFaceW-II, Unrestricted, [HOG, LBP], fd) configuration.

3   Extension 1: Videos

- The way to get face descriptors from a video is taken from here `https://www.kinfacew.com/papers/KFVW.pdf` in which 100 random frames are taken from a person's video feed and the average of the face descriptors of that is used.

- I needed to request the dataset via email but so far, I haven't obtained it. In the meanwhile, I went to the second extension.

4   Extension 2: Messing around with this shit

- KinFaceW-II results were wrong since I used a 200-dimensional vector instead of a truncated 100-dimensional one, which was used for the rest of the datasets. For consistency, I recalculated the results and the accuracies stayed similar but changed by about 1-2%.

- I started using different datasets for the training and testing sets, for example using KinFaceW-I for training and KinFaceW-II for testing.

4.1   Replacing VGG with a different Network

- I replaced the VGG Face Descriptor with the network described here `https://github.com/deep-fry/mayo/blob/master/models/cifarnet.yaml`.

- This network was implemented with Keras and trained on the CIFAR-10 dataset. At first the learning rate was constant but it would only hit an accuracy of around 86%. However, by making the learning rate a bit more adaptive, I was able to train the model to a 91% accuracy. The model then takes about 30 minutes to train on the CIFAR-10 dataset.

- Then, the softmax layer is taken out, just like in VGG, and this allows us to create a 192-dimensional vector for each image, which is then PCA'd to 100-dimensions. The images have to be downsampled to 32×32 since that's what the network takes.

- The results are in the spreadsheet.

- I essentially found that replacing VGG with CFN does worse in the KinFaceW-I dataset, it does better in the KinFaceW-II dataset and it doesn't have a huge effect on the TSKinFace dataset (with the biggest difference being a 1% difference in accuracy). When grouped by relationship, I found that the father-son relationship had the least change in accuracy with an average 0.092% accuracy difference. This is the original face descriptor accuracies minus the LBP, HOG, SIFT and CFN face descriptor accuracies (so a positive number implies that the original face descriptors did better). We see that there is more variation in the other relationships, ignoring fms and fmd as there's only one dataset for this, than there is in the father-son relationship, and this is reflected in the standard deviations for each relationship. Generally, VGG did better than CFN for KinFaceW-I and did worse for KFW-II. In terms of any other things I thought might be interesting to note, the maximum difference in accuracies was 5.4% which occurred in the KFW-II unrestricted setting with the mother-daughter relationship in which CFN made the accuracy increase by 5.4%. The maximum decrease in accuracy was 4.7% with the KFW-I Restricted dataset for the mother-son relationship. We also got that the KFW-II Unrestricted setting with the mother-son relationship had no difference whatsoever in accuracies between the face descriptors, though I think this is more likely to have happened by chance than having any underlying reason.

  With respect to why KFW-II did better with the CFN face descriptor, I'm not entirely sure but I think it goes back to the fact that the pairs of images are taken from the same image in KFW-II but not in KFW-I. My guess is that, since the input to the CFN network is a 32 x 32 image, potentially it could be generalizing the lighting and other non-face related features more which could be exacerbating the problem we saw earlier with the bias. The main reason I'm unsure about this is since I would've imagined something similar would have occurred with TSKinFace if this was the case but it doesn't seem to have happened, or at least not with the same magnitude of effect.