Study into Facial Recognition

Robert Alex (ralex@gmu.edu)
December 9, 2021
G01287617
No Teammate

1) Problem Definition

There are plenty of articles and papers on the web that focus on the basic, surface – level implementation of a facial recognition system that will work on a small set of facial classes. While these programs serve as a fine introduction into the topic, I was more interested in a deeper dive into the methods, parameters, and models that could be used to fine-tune a facial recognition system. I would then use this program to study the accuracy performance impact when greatly scaling the population pool.

The end goal was to create a user-friendly program that one could easily operate to set up their own facial recognition database and run small-scale recognition tests on user-selected training and testing images; but additionally test performance with different classification models, Primary Component Analysis (PCA) parameters, image per-processing methods, and the ability to shuffle test/training images within each facial class – all while the population size can scale dramatically upwards.

2) Motivation

Facial detection and recognition is a very hot topic in politics at the moment, with new articles coming out about uses for the technology in both nefarious and altruistic purposes on a daily basis. The ethics behind facial recognition in an open public is a fiercely debated topic online, with public backlash reaching the development of some ongoing projects. I encountered this effect myself, as I was searching for a database to use to expand my population of people. I came across a Microsoft database called “MS Celeb” that consisted of a data set of 10 million faces of famous celebrities (Works Cited 1). However, the project faced backlash when a journalist discovered their own face in the database and wrote an article reprimanding the project for its unsolicited use of faces.

The project was eventually shut down, but other threats still remain. A facial recognition system is being developed by China to track foreign journalists, and another to keep track of its own citizens. Dangerous programs like these make investigating the science behind facial recognition more important now than ever. These issues made me sure to obtain permission from all my friends and family to include their faces in my research (outside of Professor Rangwala, who I assumed would not mind). However, there was no way for me to manually create a database of faces with thousands of subjects on my own, so I used the University of Massachusetts at Amherst’s Labeled Faces in the Wild (LFW) publicly available database (Works Cited 2). These facial images were obtained purely for academic and research purposes, and are not meant to be used for private facial recognition commercial applications.

I also work as a weapons system software engineer as my profession, and the ethics behind facial detection as a means of target detection is fiercely debated in my field. While Department of Defense Directive (DODD) 3000.09 currently prohibits autonomous, target selecting and firing systems to defensive uses only, there is much discussion about relaxing these restrictions to allow for more fully autonomous behavior in the weapons space. With polarizing topics like this surely to manifest in the next few years of my career, I could not pass up the opportunity to try to create a facial recognition system myself and learn as much about the topic as possible.

3) Literature Search

As previously stated, it is not difficult to find articles and walk-throughs online about the basics of facial recognition. These articles usually focus on the "ease" at which it is to setup a very basic facial recognition system in python. However, these articles and walk-throughs barely scratch the surface of all the work it takes to develop a decent performing system on user obtained images and classes. The articles use the easiest approaches to distinguish between a small set of facial classes, with each face having hundreds of reference images to learn with (Works Cited 3). My project was different in not only would I dive much deeper into the different methods and parameters used to tune a system, but I would also collect my own user images and test how these slightly smaller sample size of training images affected the performance as facial class population grew larger.

A great research paper explaining the details behind PCA was done by Kyungnam Kim at the University of Maryland (Works Cited 4). This article gives great detail about the mathematics principles behind PCA, as well as look into the eigenfaces created by the dimensional reduction method. While this paper was useful into learning the science behind one of the main adjustable parameters in my own research, the project was significantly more limited in scope. This paper served as a very deep dive into one aspect of a good facial recognition system, while my project focused more broadly on a collection of topics used to hone a recognition program.

I was unable to find a article or paper that focused on the same collection of topics discussed in my research.

4) Background

From the beginning, I new that I wanted my program to be both flexible enough that a user could easily submit their own labeled training images and testing images and easily get results back – as an exploratory introduction. However, I also wanted to include the capabilities to do a detailed analysis on results and performance, with many aspects being user adjustable either at run-time through the command line, or by changing parameters within the code at labeled locations.

This motivation led to me creating a file system in the src directory that can easily be used to upload and get results from user defined images. Simply add your labeled photos (first_last_xxxx.jpg) into whatever population size folder you would like to test in (all – 8000 people, big_test – 800 people, Friends_Family – 22 people, but these can all be changed), then add an image you would like to test into the TEST_IMAGES folder (first_last_0.jpg)(multiple test people can be added to the TEST_IMAGES folder, but please only test one picture of each individual at a time). The program will automatically detect your training images (as long as they are named the same) and match them with the test images, and facial detection results will be given at the end, complete with probabilities for the n highest probability matches. The faces detected from the training data will be stored in the heads folder (this is useful to check if the facial detection system is detecting the head in your images properly), and the Test_output folder holds the greyscale faces detected from the TEST_IMAGES folder. The heads folder can get pretty large when it is holding all heads detected from all images in the training data, so the "clean" function can be adjusted in the code to automatically wipe the folder at the start of each new execution of the program. The test_pics folder is simply a redundant place to keep my test images, so that I can easily swap them in and out of code execution in the TEST_IMAGES folder.

The first primary step of the code is to detect faces from the selected folder of training images. The facial detection system can sometimes be tricky, so I have pointed out in the code where to change values to best fit whatever images you are adding to the folders (however, sometimes pre-cropping an image to focus mainly on a subject's face will greatly help the detection.) Then, PCA is performed on

the training set, and fit to both the training and testing images. The component count used in the PCA analysis is user-adjustable, and greatly explored in my analysis. Once the dimensionality reduction of PCA is completed, the user-selected classification model classifies the test images and prints the results for the predictions. If the user is simply testing the images selected in the TEST_IMAGES folder (shuffle is off), then the program only runs once and the predictions are displayed.

However, if shuffle is turned on, this signifies that the user does not care about classifying a test image and instead wants to measure the accuracy of the program by shuffling the training and test images within each class. This performs a pseudo-kfold like process that randomly selects one test image within each class of face defined in TEST_IMAGES, then uses the rest of the images as training images. When using this function, the user can input how many test runs they would like performed. This creates an average across multiple randomly selected test and training images within each class and can more accurately display how well the program is learning each class of face. Obviously, accuracy may drop when using shuffle, as every image may predict differently even when it is the same person. Also, accuracy drops as sample population size goes up, as there are more faces in the system to learn, muddying the waters of the programs recognizing ability.

5) Program Development

Approaching the development for this program was difficult, as I knew I wanted a file-based system that a more novice user could easily operate, but also a program that a research analyst could use to run tests. I had also never used images as data source before, and had no idea where to begin my research. I eventually came across the openCV toolkit for python that I could use to import my images. I knew that I wanted to focus on the facial detection aspect of the program, so I could use openCV's built in functionality to detect faces and pre-process the images quickly. Unfortunately, this proved to sometimes be difficult, as misreads of faces in images were common. Eventually, after tuning, I landed on the parameter set that is in the code now, although some cropping of the initial images I used was done to help the function hone in on faces.

After I knew I had a detection system implemented, I set up the file system in a way that I could add images as I see fit, and detected faces from test and training images would be stored for review. Getting the program and file system working up to the point before adding the PCA capability took much longer than I had anticipated. However, once I had all of my training data and testing data stored in numpy arrays of arrays of pixel data, I was able to quickly implement PCA using the sklearn.decomposition library. My initial idea was to perhaps implement a PCA system on my own, but as the program grew – so did the scope of parameters being included in my analysis, and I decided that the sklearn PCA system was enough for my project.

Before I merged the PCA dimension-reduced data with a classification model, I decided to quickly implement a Multi-Layer Perceptron (MLP) neural net using all of the image data – no PCA used. My basic dimensions stored for each image is 512x512, so even with the limited data set I was using at the time, the algorithm was long, slow, and extremely inaccurate. This makes sense, as PCA is used to find the values that most-show the differences between the data, so training a neural net on a large set of data that includes useless pixels would obviously not perform well. I did keep this initial neural net in my code, however as my data set expanded, it became unusable due to resource limitations on my systems, so testing analysis was very limited (it performed terribly anyway).

After this experiment, I refocused on the PCA to classifier connection. I again made a MLP classifier, and began testing using the PCA reduction with varying component sizes and MLP parameters. Performance, was immediately far more accurate than before and speed and resource efficiency improved as well. I then added in the prediction classification ability with probability printouts, and I had a working classifier.

At this point I had only used the limited data set of around 5 friends that I had added, with 3-5 pictures of each person. I focused on expanding my data sets. I increased my user-added Friends_Family repository to 22 people with around 113 total images. This was a laborious process, and getting the facial detection system to perfectly detect each face on all images was not easy. I will note here, that I added Seth MacFarlane and Elon Musk because people often say I look like them. I did find some light correlation in predictions with Elon Musk and myself, but not enough to say anything for certain. It was at this point that I also found the LFW database, so I added the 13000 image database into a new folder, and added my user-added images in as well. It immediately became clear that pre-processing, storing, training, and predicting on 13000 images was very slow and resource intensive. I experimented with using incremental PCA to lighten the burden on my ram usage during PCA (I have 16gb on my laptop, but PCA was often requesting over 24 gb for all of the data), but I ended up reducing image resolution to 200x200 when using all images. I also created the big_test folder that includes a reduced 800 images, this was quicker and easier to test my progress with.

After I had created a fairly robust training and testing system using the MLP classifier and PCA, I then added the support vector machine (SVM) classifier and the K nearest neighbor (KNN) classifier. These additions were simple, and I chose them because they represented a good variety of classifier types. However, as expected, and as my analysis will later show, these classification methods paled in comparison to the MLP neural network.

I then attempted to add a k-fold validation setup into my code, but realized that shuffling and splitting the entire data set would not work, since I am classifying thousands of faces, some splits might not include any training data for a tested class. This led me to a breakthrough. This whole time I had been testing and perfecting my classifier only for test images that I selected. This allowed my to fine-tune my algorithm for these test images, creating better accuracy scores, but not accurately representing the generalization learning for each class. I then embarked on created the shuffle function of my program, the most powerful tool to test learning performance.

Shuffle takes all testing classes described in TEST_IMAGES, matches each class with all images from the training set, then shuffles them together. After the test images are intertwined with the training images, it randomly selects a new image from each class to serve as that class's test image. Prediction results are recorded and aggregated, then the process can be completed as many times as the user likes – creating a average accuracy score for the random selections. While this addition did cut the accuracy scores of my algorithm greatly, I was pleased knowing that I was now testing a truly generalizing learned algorithm, not fine tuning to my images. This process also made me realize how many photos it takes to truly learn a face. I have 113 images split between 22 classes in Friends_Family, but I think it would take many more images to get truly great generalization accuracy out of this program.

The last major addition I made was the validation capability. This was simply a way for me to test that my algorithms were learning the correct data. Validation simply learns a model on the training data, then all the training data is passed back in as testing data. This produces extremely accurate results (usually perfect results), proving that my algorithms were accurately learning the training data.

The last mission to accomplish was testing. I did a huge amount of testing. Results will be shown in the following sections of this paper, but these results only show a fraction of the testing done to achieve them. Even on my 24 core, 64gb ram personal desktop computer, running tests on the 13000 image database took hours when shuffle was enabled with multiple averaging runs. On my personal laptop, a single 1 loop execution of Friends_Family takes about a minute, big_test can take 5 minutes, and "all" can take 30 minutes. Of course, all of these times change depending on the number of averaging shuffle runs, but also the classification method, PCA components, other aspects, such as how

many iterations in the MLP net are allowed or your tolerance figure. Overall, testing was thorough, laborious, and time consuming – but the summary produced the results in the following sections.

6) Experimental Evaluation

No Shuffle, Optimizing for Test Images Results:

These results focused on optimizing my algorithm for the predictions on the test images only in the TEST_IMAGES folder. This means that there was no shuffling of the training and testing data, so I could focus on optimizing my parameter for optimal performance in each algorithm for these test images.

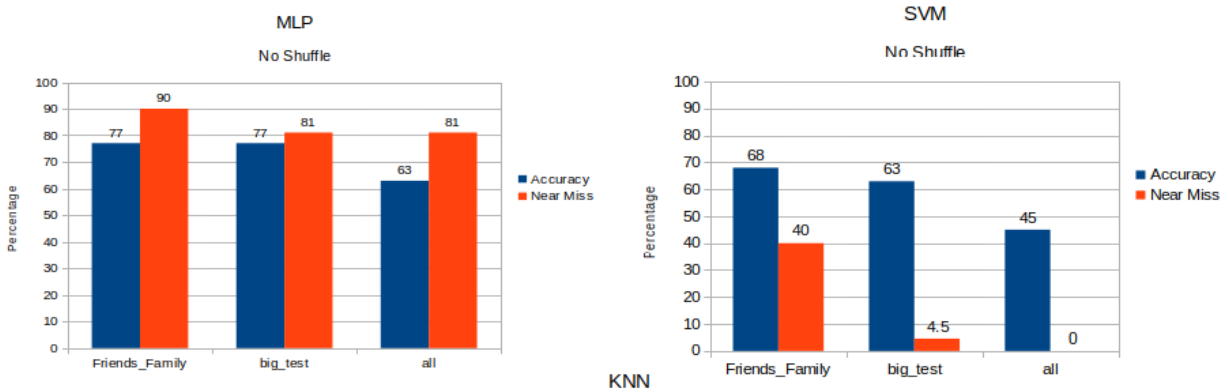
When I started building the program, I included only 7 different facial classes (people) in my training and testing data. Even though I was using only 3 to 4 photos per person, the MLP classifier was still able to get 100% prediction accuracy on these test images. As my program matured, I eventually expanded the basic database (Friends_Family) to 22 classes of people with 113 training images and 1 testing image per class. Expanding this database degraded my accuracy to 77% at the best. This was frustrating, because it seemed that algorithm would never learn some people, no matter what combination of testing and training images I used. Even more confusingly, when using MLP, some classes of faces would not be predicted correctly when using the smaller Friends_Family database, but would be predicted correct when using a larger database like big_test. I can only assume that this behavior is due to the increased number of iterations it takes to train the MLP model as the training set population increases – allowing for algorithm to converge differently then with the smaller data set.

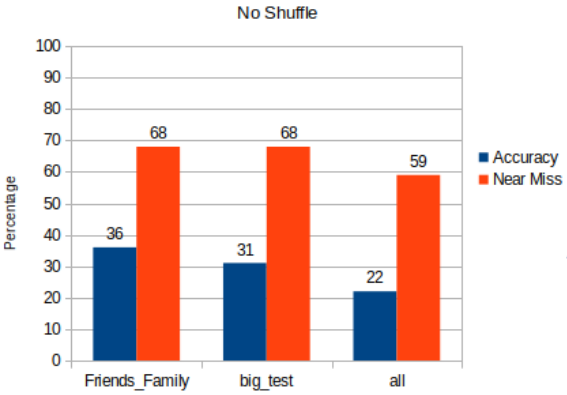
As previously stated, the testing consisted of finding the maximum accuracy performance available for each classifier model included in my program (MLP, SVM, KNN), then comparing performances across the scaling population sizes. For all classifier models, parameters adjusted during testing included image pre-processing methods and of course, PCA component count. These parameters were included in our range of testing along with the following parameters per classification model:

- MLP: Max iteration count, nodes / layer configurations, tolerance metric for early stopping
- SVM: C parameter (penalty parameter), Gamma
- KNN: k (nearest neighbor count)

I also added a secondary performance metric called “near miss”. This metric can be adjusted and will count as a positive hit if the correct class of the test images was given in the top ‘n’ closest predictions. For my testing, I used a correct answer in the top 5 predictions to count as a near miss. This metric obviously does not completely count accuracy, but it does signify that algorithm was close to predicting the right class and that if there were maybe more training images of that person included, the prediction might have been correct. Note that SVM does not natural support giving percent certainty data, however, it can be simulated. Unfortunately, the simulation did not work correctly for most of my testing, with the correct class not being given in the top 5, even though the prediction was correct. This results in near miss numbers that are much lower than the accuracy data, which is impossible. I have decided to include the data anyway, but please ignore the near miss data for the SVM classifier.

The graphs below show the optimal performance achieved for each classification model across the different database sizes, with shuffling set to off:





Best parameters displayed in these results:

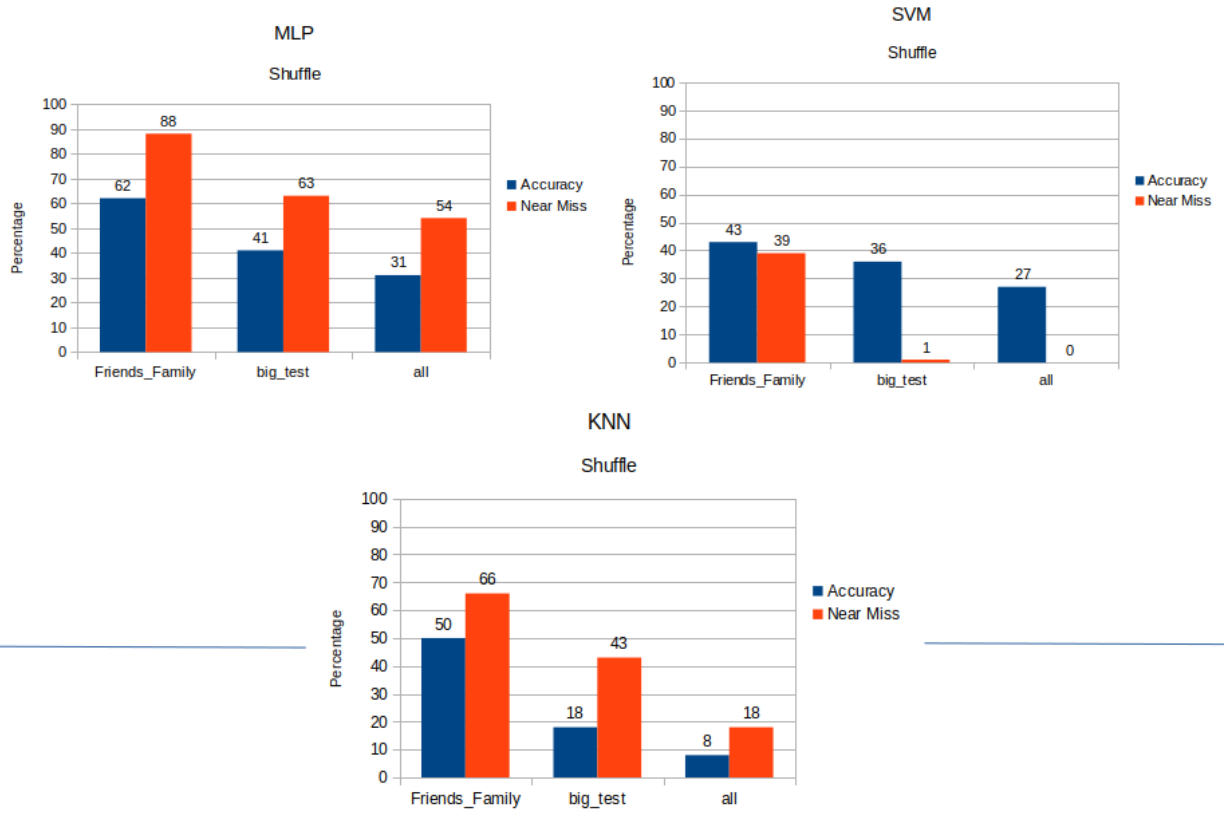
- MLP – Friends_Family: PCA 50, 1000 iter, nodes (1024,), tol 1e-4
- MLP – big_test: PCA 50, iter 1000, nodes (1024, 512, 256), tol 1e-4
- MLP – all: PCA 50, iter 1000, nodes (1024,) tol 1e-4
- SVM – Friends_Family: PCA 50, c 5, gamma .01
- SVM – big_test: PCA 50, c 5, gamma .01
- SVM – all: PCA 50, c 5, gamma .01
- KNN – Friends_Family: PCA 50, k = 9
- KNN – bit_test: PCA 50, k = 9
- KNN – all: PCA 50, k = 9

Shuffle on, Generalization Tests

As described earlier, the shuffle tests are used to measure the generalization performance of each algorithm. This is done by shuffling the testing and training images data for each class together, then randomly selecting a testing image for each class, then using the remaining images as the training data. This ensures that a different photo will be selected at each run, so fine tuning an algorithm to fit a certain test image is impossible.

The testing parameters for the shuffle tests remain the same as those listed above, except with the addition of the shuffle rounds parameter. This is a user input number that describes how many shuffle, selection, train, prediction, loops are completed before displaying the aggregate data. This can be an extremely time consuming process for the larger databases, so a rounds count of 10 was used in the Friends_Family and big_test database, while 5 was used for any “all” tests.

Best results for shuffle are displayed in the graphs below:



Best parameters displayed in these results:

- MLP – Friends_Family: PCA 50, iter 100, nodes (1024,), tol 1e-4
- MLP – big_test: PCA 50, iter 100 (1024,), tol 1e-4
- MLP – all: PCA 50, iter 1000 (1024,), tol 1e-4
- SVM – Friends_Family: PCA 50, c 5, gamma .01
- SVM – big_test: PCA 50, c 5, gamma .01
- SVM – all: PCA 50, c 5, gamma .01
- KNN – Friends_Family: PCA 50, k = 9
- KNN – big_test: PCA 50, k = 9
- KNN – all: PCA 50, k = 9

7) Conclusions

With the extreme amount of testing performed, there are hundreds of interesting observations and takeaways that can be made from the data, but I will limit this section to only the major analysis of the results as well as the most interesting observations only.

In no shuffle tests, the MLP neural net performed 13% better on average than the SVM, and 42% better than KNN. In shuffle tests, the MLP beat SBM by 9% and beat KNN by 19%. From MLP, this behavior was expected. Neural nets seem to be the most common choice in industry for racial recognition systems, so their performance was expected to be good. SVM was disappointing here, I was hoping for results with this classifier. Perhaps a more robust SVM system could be developed with better performance, but the results in this test were significantly worse than the neural net.

In no shuffle tests, increasing population size from 21 in Friends_Family to 800 in big_test decreased performance by 3.3%, and increasing population from 800 to 8000 in “all” decreased performance by 13.6% on average. In shuffle tests, Friends_Family to big_test decreased accuracy by 20%, and big_test to “all” decreased performance by 29% on average. I was actually pleased by these results, especially in the no shuffle tests. A total decrease of 16% on accuracy when increasing population size from 21 to 8000 is surprisingly impressive.

These accuracy results may seem disappointing overall, however, I think these results are actually more impressive than the numbers let on. Using MLP, this algorithm is learning 8000 people’s faces, and is still able to correctly guess the test image 31% of the time (and 54% of the time is a near miss) with a randomly selected image. Remember that many of these classes only include 3-6 images of each person to train with. I firmly believe that with an adequate count of training faces per class (say, 100), this algorithm could reach very impressive accuracy numbers. It is also worth noting that I included 10 images of myself for the algorithm to learn, and the MLP classifies me correctly nearly every time, no matter the population size. This implies that I either have a very distinguishable face to a computer (no), or that I increasing training data size dramatically improves the prediction capability of that class. Some classes, like “nicholas_studebaker”, were almost never accurately predicted at all, while other such as “scott_flood”, was predicted almost every time, even though they both only include 3 training images. This shows that certain selections of training and testing images can have dramatic impacts on results, especially when training image count is low.

Another curious observation is that almost all best-performing results had PCA = 50. This was curious, as I tried many different values for PCA in all test. I would assume that PCA would increase, especially as population count increased, but apparently combinations of 50 components is enough to predict for every classifier and population size. In MLP, I was hoping to find that different node configurations would improve performance as population size increased, but one layer of 1024 nodes ended up being almost unanimously the best performer. While iteration count needed to adequately converge the model scaled upwards with population size (unexpectedly), no change in node and layer configuration made a significant positive impact.

Overall, I am very pleased with the results and state of my program in my Study into Facial Recognition. I created a versatile platform for facial recognition testing and modeling. The no shuffle tests prove that fine-tuning can be done for specific test images to achieve impressive results, even with small training data counts for each class and massively scaling population counts. The shuffle tests decreased performance significantly on average, but the near-miss results proved that the algorithm was still close to predicting correctly for many classes, and more training data per class would significantly improve performance across all population sizes. Future testing could focus on training image counts per class, and see how scaling up training data would improve performance in both the shuffle and non-shuffled tests. This program can be used to gather large amounts of interesting data about facial recognition, and I encourage people to use it for whatever research purposes they desire.

Works Cited

1) https://exposing.ai/msceleb/

2) http://vis-www.cs.umass.edu/lfw/

3) https://pythonmachinelearning.pro/face-recognition-with-eigenfaces/

4) http://staff.ustc.edu.cn/~zwp/teach/MVA/pcaface.pdf